Oracle® JavaScript Extension Toolkit (Oracle JET)

Developing Oracle JET Apps Using MVVM Architecture

18.1.0
G26356-02
May 2025

Oracle JavaScript Extension Toolkit (Oracle JET) Developing Oracle JET Apps Using MVVM Architecture, 18.1.0

G26356-02

Primary Author: Oracle Corporation

Contents

8 Work with Oracle JET Web Components

12 Develop Accessible Oracle JET Apps

13 Internationalize and Localize Oracle JET Apps

14 Use CSS and Themes in Oracle JET Apps

Preface

Developing Oracle JET Apps Using MVVM Architecture describes how to build responsive web apps using Oracle JET.

Topics:

Audience

Developing Oracle JET Apps Using MVVM Architecture is intended for intermediate to advanced front-end developers who want to create client-side, responsive web, or progressive web apps based on JavaScript, TypeScript, HTML, and CSS.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Resources

For more information, see these Oracle resources:

- Oracle JET Web Site
- API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET)
- Oracle® JavaScript Extension Toolkit (JET) Keyboard and Touch Reference
- Oracle® JavaScript Extension Toolkit (JET) Styling Reference

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Get Started with Oracle JavaScript Extension Toolkit (JET)

Oracle JET is a collection of Oracle and open source JavaScript libraries engineered to make it as simple and efficient as possible to build client-side web apps based on JavaScript, HTML5, and CSS.

To begin using Oracle JET, you do not need more than the basics of JavaScript, HTML, and CSS. Many developers learn about these related technologies in the process of learning Oracle JET.

Oracle JET is designed to meet the following app needs:

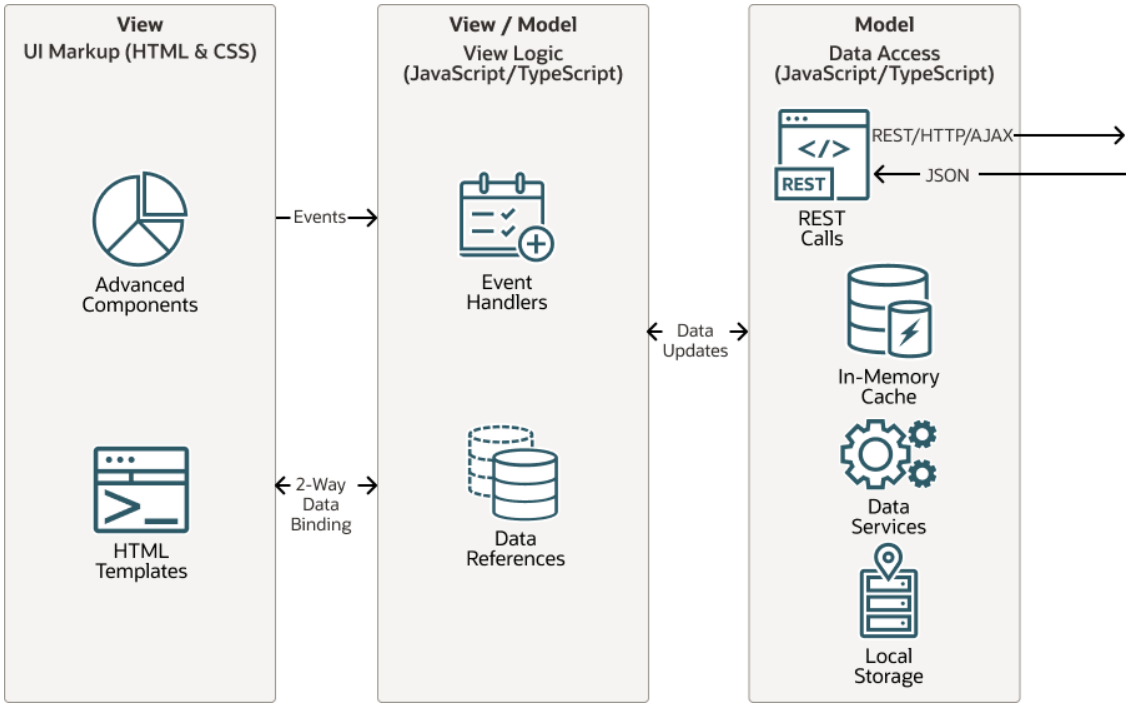
- Add interactivity to an existing page.
- Create a new end-to-end client-side web app using JavaScript, HTML5, CSS, and best practices for responsive design.

View videos that provide an introduction to Oracle JET in the Oracle JET Videos collection.

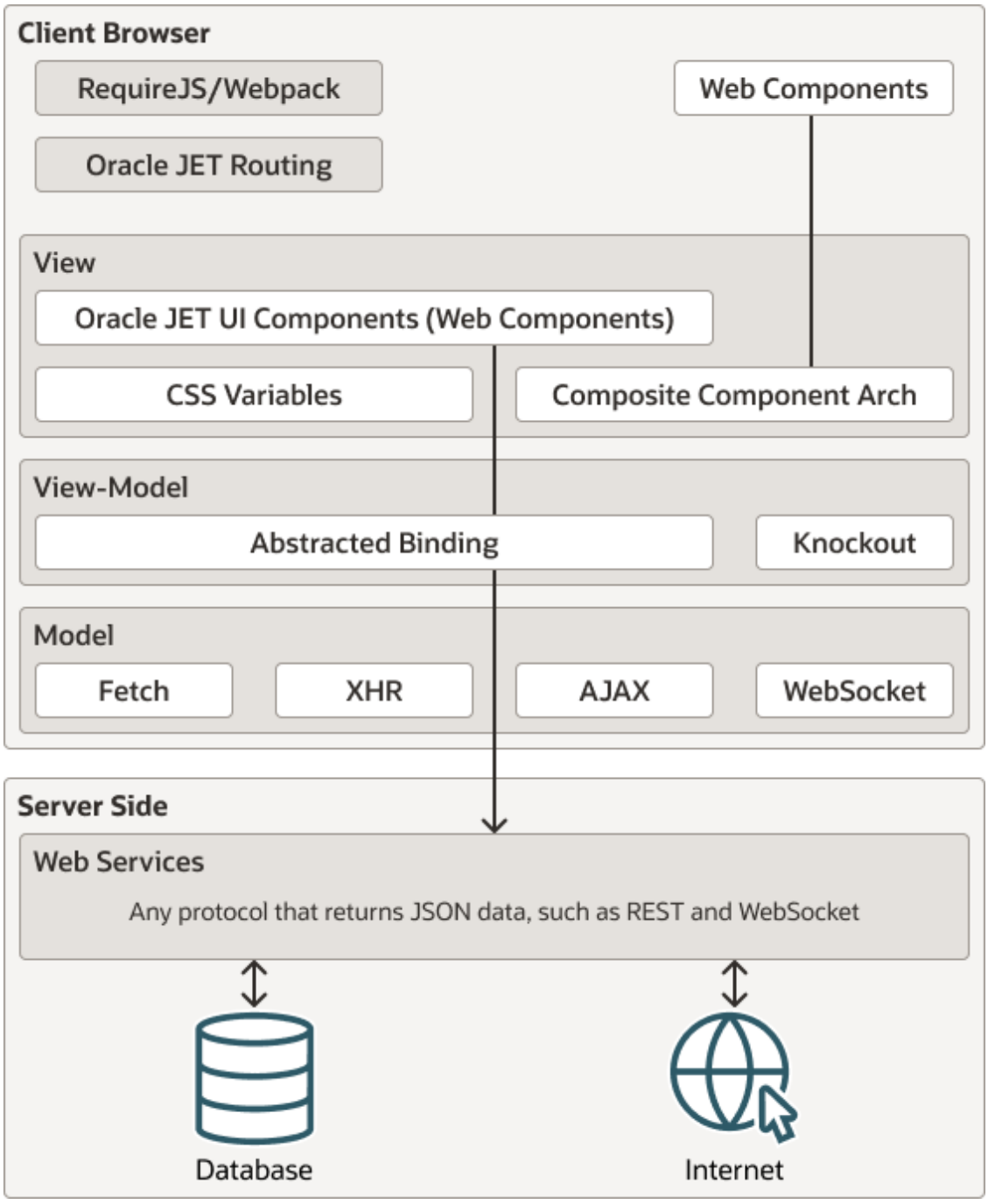
The Oracle JET Model-View-ViewModel Architecture

Oracle JET supports the Model-View-ViewModel (MVVM) architectural design pattern.

In MVVM, the Model represents the app data, and the View is the presentation of the data. The ViewModel exposes data from the Model to the view and maintains the app's state.



To support the MVVM design, Oracle JET is built upon a modular framework that includes a collection of third-party libraries and Oracle-provided files, scripts, and libraries.



To implement the View layer, Oracle JET provides a collection of UI components implemented as HTML5 custom elements, ranging from basic buttons to advanced data visualization components such as charts and data grids.

Knockout.js implements the ViewModel and provides two-way data binding between the view and model layers.

Oracle JET Features

Oracle JET features include:

- Messaging and event services for both Model and View layers
- Validation framework that provides UI element and component validation and data converters
- Caching services at the Model layer for performance optimization of pagination and virtual scrolling
- Filtering and sorting services provided at the Model layer
- Connection to data sources through Web services, such as Representational State Transfer (REST) or WebSocket
- Management of URL and browser history using Oracle JET `CoreRouter` and `oj-module` components
- Integrated authorization through OAuth 2.0 for data models retrieved from REST Services
- Resource management provided by RequireJS
- A `RESTDataProvider` API to represent data from JSON-based REST services
- JavaScript logging
- Popup UI handling

Oracle JET Visual Component Features

Oracle JET visual components include the following features and standards compliance:

- Compliance with Oracle National Language Support (NLS) standards (i18n) for numeric, currency, and date/time formatting
- Built-in theming supporting the Oracle Redwood theme style specifications and implementing the Oracle Redwood Design System
- Support for Oracle software localization standards, l10n, including:
  - Lazy loading of localized resource strings at run time
  - Oracle translation services formats
  - Bidirectional locales (left-to-right, right-to-left)
- Web Content Accessibility Guidelines (WCAG) 2.1. In addition, components provide support for high contrast and keyboard-only input environments.
- Gesture functionality by touch, mouse, and pointer events where appropriate
- Support for Oracle test automation tooling
- Responsive layout framework

What's Included in Oracle JET

The Oracle JET zip distribution includes Oracle JET libraries and all third party libraries that the toolkit uses.

Specifically, Oracle JET includes the following files and libraries:

- CSS and CSS files for the Redwood theme

- Minified and debug versions of the Oracle JET libraries
- Data Visualization Tools (DVT) CSS and JavaScript
- jQuery libraries
- RequireJS, RequireJS `text` plugin, and RequireJS CSS plugin
- js-signals
- Hammer.js

  Oracle JET components use Hammer.js internally for gesture support. Do not add to Oracle JET components or their associated DOM nodes.
- Oracle JET `dnd-polyfill` HTML5 drag and drop polyfill
- proj4js library
- webcomponentsjs polyfill

Third Party Libraries Used by Oracle JET

To begin using Oracle JET, you do not need to understand more than the basics of JavaScript, HTML, and CSS or the third party libraries and technologies that Oracle JET uses. In fact, many developers learn about these related technologies in the process of learning Oracle JET.

Name	Description	More Information
CSS	Cascading Style Sheets	http://www.w3.org/Style/CSS
HTML5	Hypertext Markup Language 5	http://www.w3.org/TR/html5
JavaScript	Programming language	https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript
TypeScript	Typed superset of JavaScript that enables you to support typechecking against the TypeScript API of JET elements and non-element classes.	http://www.typescriptlang.org
jQuery	JavaScript library designed for HTML document traversal and manipulation, event handling, animation, and Ajax. jQuery includes an API that works across most browsers.	http://jquery.com
Knockout	JavaScript library that provides support for two-way data binding	http://www.knockoutjs.com
RequireJS	JavaScript file and module loader used for managing library references and lazy loading of resources. RequireJS implements the Asynchronous Module Definition (AMD) API.	RequireJS: http://www.requirejs.org AMD API: http://requirejs.org/docs/whyamd.html
SASS	SASS (Syntactically Awesome Style Sheets) extends CSS3 and enables you to use variables, nested rules, mixins, and inline imports to customize your app's themes. Oracle JET uses the SCSS (Sasy CSS) syntax of SASS.	http://www.sass-lang.com

If you will be using Oracle JET tooling, you may also want to familiarize yourself with the following technology.

Name	Description	More Information
Node.js	Open source, cross-platform runtime environment for developing server-side web apps, used by Oracle JET for package management. Node.js includes the `npm` command line tool.	https://nodejs.org

Create a Development Environment for Oracle JET

You can decide what development environment you want to use before you start developing Oracle JET apps. If you will use Oracle JET tooling to develop web apps, you must install the Oracle JET packages.

Choose a Development Editor

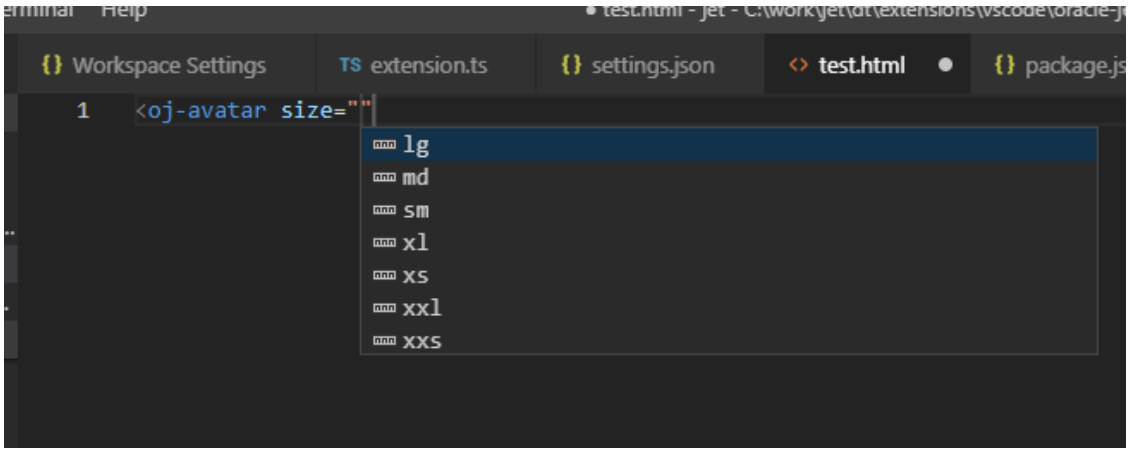
You can develop Oracle JET apps in virtually any integrated development environment (IDE) that supports JavaScript (or TypeScript), HTML5, and CSS3. However, an IDE is not required for developing Oracle JET apps, and you can use any text editor to develop your app.

You can use an IDE in conjunction with the Oracle JET command-line tooling, where you scaffold web apps by using one of the provided starter templates. You can proceed to develop the scaffolded app in the IDE of your choice by opening the project that was created using the JET tooling, in that IDE. After saving changes your app files in the IDE, you use the JET tooling to build and run the JET app.

If you are using Microsoft Visual Studio Code (VS Code) as your editor, you can add the Visual Studio Code Extension of Oracle JET Core to support developing Oracle JET apps. Specifically, the Oracle JET extension for VS Code improves developer productivity for creating clientside JavaScript or TypeScript web apps by providing:

- Code completion against the JET API and JET component metadata.
- Ability to work with code snippets for the most commonly used Oracle JET components.
- Capability to diagnose app source (JavaScript, HTML, CSS, and JSON files) by running Oracle JET audit reports.

This custom HTML data support for JET components support means that when you are editing HTML files, VS Code will prompt you with Oracle JET tags and attributes. As you start typing your Oracle JET HTML tag, a dropdown will show a list of matching choices:



For more examples of Oracle JET support for VS Code, visit the Oracle JET Core Extension download page in the Visual Studio Marketplace.

Install Oracle JET Tooling

You must install Node.js to use Oracle JET tooling to develop Oracle JET apps. You'll also need the Oracle JET CLI, `ojet-cli`. You can use the `ojet-cli` through the Node.js package runner (`npx`), or you can install it on your development platform.

If you already have Oracle JET tooling installed on your development platform, check that you are using the minimum versions supported by Oracle JET and upgrade as needed. For the list of minimum supported versions, see Oracle JET Support.

Install Node.js

Install Node.js on your development machine.

From a web browser, download and install one of the installers appropriate for your OS from the Node.js download page. Oracle JET recommends that you install the latest LTS version. Node.js is pre-installed on macOS, but is likely an old version, so upgrade to the latest LTS version if necessary.

After you complete installation and setup, you can enter `npm` commands from a command prompt to verify that your installation succeeded. For example, enter `npm config list` to show config settings for Node.js.

If your computer is connected to a network, such as your company's, that requires you to use a proxy server, run the following commands so that your npm installation can work successfully. This task is only required if your network requires you to use a proxy server. If, for example, you connect to the internet from your home, you may not need to perform this task.

```
npm config set proxy http-proxy-server-URL:proxy-port
npm config set https-proxy https-proxy-server-URL:proxy-port
```

Include the complete URL in the command. For example:

```
npm config set proxy http://my.proxyserver.com:80
npm config set https-proxy http://my.proxyserver.com:80
```

Use the npx Node.js Package Runner

We recommend that you use the `npx` Node.js package runner to create and manage Oracle JET apps. With the `npx` Node.js package runner, you won't need to uninstall and reinstall the NPM packages that deliver the ojet-cli if you frequently change releases of the `ojet-cli`.

To use `npx`, you must install Node.js and you must uninstall any globally installed instances of the Oracle JET CLI from your computer. To list globally-installed packages, run the `npm list --depth=0 -g` command in a terminal window. To uninstall a globally installed instance of the Oracle JET CLI, run the `npm -g un @oracle/ojet-cli` command.

Once you have installed Node.js, you can use `npx` and the version of the Oracle JET CLI NPM package that you want to use, plus the appropriate command. The following examples

demonstrate how you create Oracle JET apps using different releases of the CLI and then serve them on your local development computer.

```
// Create and serve an Oracle JET 15.0.0 app
$ npx @oracle/ojet-cli@15.0.0 create myJET15app --template=navdrawer
$ cd myJET15app
$ npx @oracle/ojet-cli@15.0.0 serve

// Create and serve an Oracle JET 18.1.0 app
$ npx @oracle/ojet-cli@18.1.0 create myJETapp --template=navdrawer
$ cd myJETapp
$ npx @oracle/ojet-cli@18.1.0 serve
```

You can use all the Oracle JET CLI commands (`create`, `build`, `serve`, `strip`, `restore`, and so on) by following the syntax shown in the previous examples (`npx` *`package command`*).

The `npx` package runner fetches the necessary package (for example, `@oracle/ojet-cli@18.1.0`) from the NPM registry and runs it. The package is installed in a temporary cache directory, as in the following example for a Windows computer:

```
C:\Users\JDOE\AppData\Roaming\npm-cache\_npx
```

No NPM packages for the releases of the Oracle JET CLI shown in the previous examples are installed on your computer, as you will see if you run the command to list globally installed NPM packages:

```
$ npm list --depth=0 -g
C:\Users\JDOE\AppData\Roaming\npm
+-- json-server@0.16.3
+-- node-gyp@9.0.0
+-- typescript@4.2.3
`-- yarn@1.22.18
```

Use the following command to clear the temporary cache directory:

```
npx clear-npx-cache
```

To learn more about `npx`, see the Node.js documentation. For more information about the commands that the Oracle JET CLI provides, see Understand the Web App Workflow.

Install the Oracle JET Command-Line Interface

Use npm to install the Oracle JET command-line interface (`ojet-cli`).

- At the command prompt of your development machine, enter the following command as Administrator on Windows or use `sudo` on Macintosh and Linux machines:

  ```
  [sudo] npm install -g @oracle/ojet-cli
  ```

  It may not be obvious that the installation succeeded. Enter `ojet help` to verify that the installation succeeded. If you do not see the available Oracle JET commands, scroll through the install command output to locate the source of the failure.

  - If you receive an error related to a network failure, verify that you have set up your proxy correctly if needed.

- If you receive an error that your version of `npm` is outdated, type the following to update the version: `[sudo] npm install -g npm`.

You can also verify the Oracle JET version with `ojet --version` to display the current version of the Oracle JET command-line interface. If the current version is not displayed, please reinstall by using the `npm install` command for your platform.

Yarn Package Manager

Oracle JET CLI supports usage of the Yarn package manager.

You must install Node.js as Oracle JET uses the npm package manager by default. However, if you install Yarn, you can use it instead of the default npm package manager by specifying the `--installer=yarn` parameter option when you invoke an Oracle JET command.

The `--installer=yarn` parameter can be used with the following Oracle JET commands:

- `ojet create --installer=yarn`
- `ojet build --installer=yarn`
- `ojet serve --installer=yarn`
- `ojet strip --installer=yarn`

Enter `ojet help` at a terminal prompt to get additional help with the Oracle JET CLI.

As an alternative to specifying the `--installer=yarn` parameter option for each command, add `"installer": "yarn"` to your Oracle JET app's `oraclejetconfig.json` file, as follows:

```
{
. . .
  "generatorVersion": "18.1.0",
  "installer": "yarn"
}
```

The Oracle JET CLI then uses Yarn as the default package manager for the Oracle JET app.

For more information about the Yarn package manager, including how to install it, see Yarn's website.

Configure Oracle JET Apps for TypeScript Development

If you plan to build an Oracle JET app or Oracle JET Web Component in TypeScript, your app project requires the TypeScript type definitions that Oracle bundles with the Oracle JET NPM package.

When you install Oracle JET from NPM, the TypeScript type definitions for version 5.7.2 get installed with the JET bundle and are available for use when you develop apps. To begin app development using TypeScript, Oracle JET tooling supports scaffolding your app by using a variety of Oracle JET Starter Templates that have been optimized for TypeScript development, with the default ES6 implementation. For details, see Scaffold a Web App.

If you have already created an app and you want to switch to developing with TypeScript, you can use the Oracle JET tooling to add support for type definitions and compiler configuration.

To add TypeScript version 5.7.2 to an existing app, use `ojet add typescript` from your app root.

```
ojet add typescript
```

When you add TypeScript support to an existing app, Oracle JET tooling installs TypeScript locally with an NPM install. The tooling also creates the `tsconfig.json` compiler configuration file at your app root. You can relocate the `tsconfig.json` file within your project by setting the optional `paths.source.tsconfig` subproperty in the `oraclejetconfig.json` file. See Properties in the oraclejetconfig.json File.

When you begin development with TypeScript, you can import TypeScript definition modules for Oracle JET custom elements, as well as non-element classes, namespaces, and interfaces. For example, in this Oracle JET app, the oj-chart import statement supports typechecking against the element's TypeScript API.

```
import ArrayDataProvider = require("ojs/ojarraydataprovider");
import { ojChart } from "ojs/ojchart";
import "ojs/ojchart";

interface ChartItem {
  id: number;
  series: string;
  group: string;
  value: number;
}

class ViewModel {
  private readonly data: Array<ChartItem> = [
    {
      "id": 0,
      "series": "Series 1",
```

And, your editor can leverage the definition files for imported modules to display type information.

```
import ArrayDataProvider = require("ojs/ojarraydataprovider");
import { ojChart } from "ojs/ojchart";
import "ojs/ojchart";

interface ChartItem {
  id: number;
  series: string;
  group: string;
  value: number;
}

class ViewModel {
  private readonly data: Array<ChartItem> = [
    {
      "id": 0,
      "series": "Series 1",
      "group": "Group A",
      "value": 42
    },
    {
      "id": 2,
      "series": "Series 2",
      "group": "Group A",
      "value": 55
    }
  ];
  readonly dataProvider = new ArrayDataProvider<ChartItem["id"], ChartItem>(this.data, {
    keyAttributes: "id"
  })
  setType = () =>
    const chartEl
    chartElement.type = "pie";
  }
}
export = ViewModel;
```

```
(property) ojChart<number, ChartItem, null, null>.type: "line" | "area" | "funn
el" | "pyramid" | "bar"
```

Note that the naming convention for JET custom element types is changing. The type name that you specify within your TypeScript project to import a JET component's exported interface will follow one of these two naming conventions:

- *componentName* + `Element` (new "suffix" naming convention)

  For example, `oj-input-search` and the `oj-stream-list` have the type name `InputSearchElement` and `StreamListElement`, respectively.

  or

- `oj` + *componentName* ("oj" prefix naming convention of not yet migrated components)

  For example, `oj-chart` and `oj-table` continue to adhere to the old-style type naming with "oj" prefix: `ojChart` and `ojTable`, respectively.

Until all JET component interface type names have been migrated to follow the new standard, suffix naming convention, some JET core components will continue to follow the old "oj" prefix naming convention (without the "Element" suffix). To find out the type name to specify in your TypeScript project, view the Module Usage section of the API documentation for the component.

For more information about working with TypeScript in JET, see API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) - JET In Typescript Overview.

Work with the Oracle JET Starter Templates

The Oracle JET Starter Templates provide everything you need to start working with code immediately. Use them as the starting point for your own app or to familiarize yourself with the JET components and basic structure of an Oracle JET app.

Each template is designed to work with the Oracle JET Cookbook examples and follows current best practice for app design.

You can also view a video that shows how to work with the Oracle JET Starter Templates in the Oracle JET Videos collection.

About the Starter Templates

Each template in the Starter Template collection is a single page app that is structured for modular development. The collection of available Starter Templates supports JavaScript or TypeScript development and will depend on the template type you add to your app.

Instead of storing all the app markup in the `index.html` file, the app uses the `oj-module` component to bind either a view template containing the HTML markup for the section or both the view template and JavaScript or TypeScript file that contains the viewModel for any components defined in the section.

The following code shows a portion of the `index.html` file in the Web Nav Drawer Starter Template that highlights the `oj-module` component definition. For the sake of brevity, most of the code and comments are omitted. Comments describe the purpose of each section, and you should review the full source code for accessibility and usage tips.

```
<!DOCTYPE html>
<html lang="en-us">
  <head>
    <title>Oracle JET Starter Template - Web Nav Drawer</title>
       ... contents omitted
  </head>
  <body class="oj-web-applayout-body">

    ... contents omitted

     <oj-module role="main" class="oj-web-applayout-max-width oj-web-applayout-content"
config="[[moduleAdapter.koObservableConfig]]">
     </oj-module>

     ... contents omitted

    <!-- This injects script tags for the main javascript files -->
        <!-- injector:scripts -->
        <!-- endinjector -->
  </body>
</html>
```

The main page's content area uses the Oracle JET `oj-web-applayout-*` CSS classes to manage the responsive layout. The main page's content uses the HTML `oj-module` element with its role defined as main (`role="main"`) for accessibility.

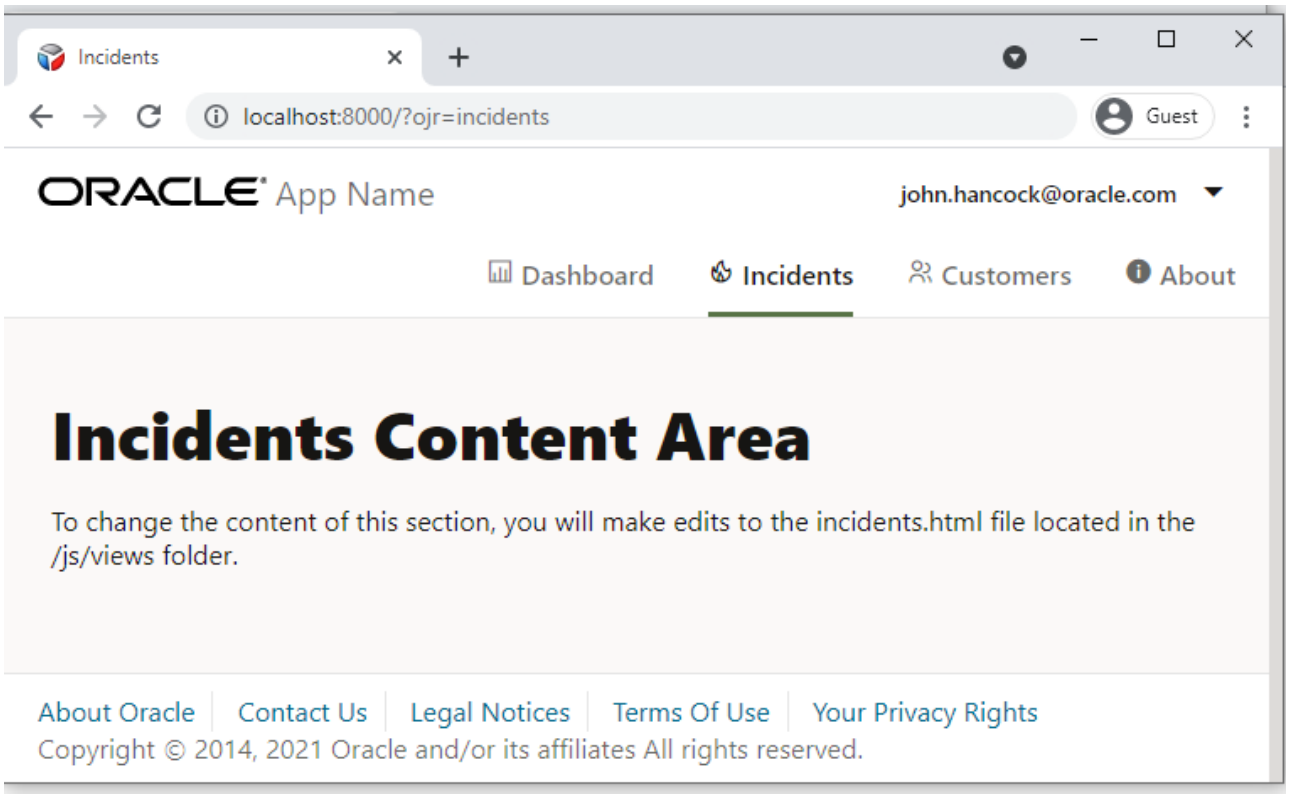
The `oj-module` component's `config.view` attribute tells Oracle JET that the section is only defining a view template, and the view will be bound to the existing viewModel. When the `oj-module` element's `config.view-model` attribute is defined, the app will load both the viewModel

and view template with the name corresponding to the value of the `config.view-model` attribute.

When the `oj-module` element's `view` and `view-model` attributes are missing, as in this example, the behavior will depend on the parameter specified in the `config` attribute's definition.

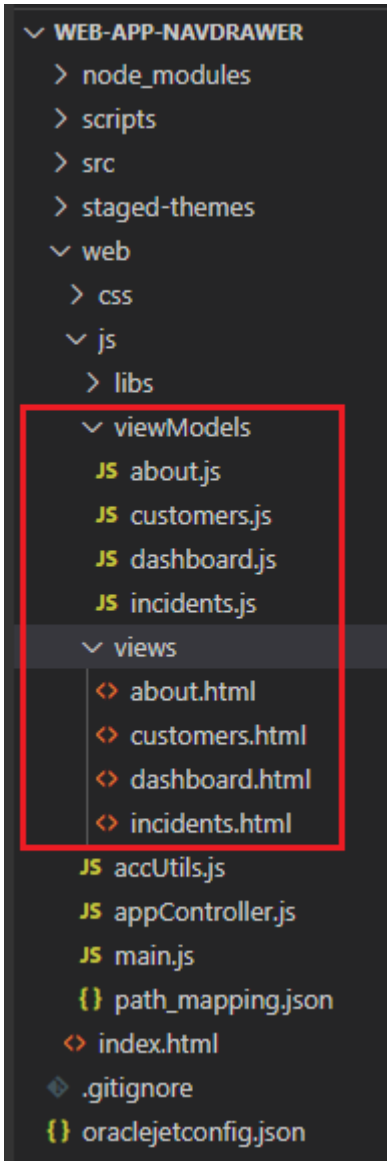
- If the parameter is an Oracle JET router's `moduleConfig` object as in the above example, then `oj-module` will automatically load and render the content of the viewModel script and view template corresponding to the router's current state.

  The Web Nav Drawer Starter Template uses `CoreRouter` to manage navigation when the user clicks one of the app's navigation items. The routes include `dashboard`, `incidents`. `customers`, and `about`. If the user clicks **Incidents**, for example, the main content area changes to display the content in the `incidents` view template.



- If the parameter is a Knockout observable containing the name of the viewModel, the app will load both the viewModel and view template with the indicated name.

The `/js/views` folder contains the view templates for the app and the `/js/viewModels` contains the viewModel scripts. The image below shows the Web Nav Drawer Starter Template file structure.



For additional information about working with single page apps, `oj-module`, `CoreRouter`, and Knockout templates, see Create Single-Page Apps.

For details about the `oj-web-applayout-*` CSS classes, see Web Application Patterns. For additional information about working with responsive design, see Design Responsive Apps.

About Modifying Starter Templates

The Starter Template is the starting point for creating your apps. You can modify any Oracle JET starter template to provide a customized starting point.

You can obtain the Starter Template from the Oracle JET app that you create when you Scaffold a Web App. Load the starter template into your favorite IDE, or extract the zip file into a development folder.

> **Tip:**
>
> If you used the command line tooling to scaffold your app, you can still use an IDE like Visual Studio Code for editing. For example, in Visual Studio Code, choose **File –> Open Folder** and select the folder containing the app you created. Edit your app as needed, but use the tooling commands in a terminal window to build and serve your app.

To modify the template you can remove unneeded content and add new content. Content that you add can be your own or you can reuse content from Oracle JET Cookbook samples. When you copy markup from a Cookbook sample, you copy the desired HTML and the supporting JavaScript.

Included in the code you add will be the RequireJS module dependency for the code. The app's `main.js` file contains the list of RequireJS modules currently included in the app. If you are using the Cookbook sample, you can determine modules that you need to add by comparing list of libraries in the app's `main.js` file to the list in the Cookbook sample. You will add any missing modules to the `define()` function in the JavaScript file for your app. For example, to add the `oj-input-date-time` component from the Cookbook, you would need to add the `ojs/ojdatetimepicker` module to the `dashboard.js` viewModel file since it's not already defined in `dashboard.js`.

To familiarize yourself with the RequireJS module to add for a Cookbook sample or for your own code, see the table at About Oracle JET Module Organization.

If you add content to a section that changes its role, then be sure to change the role associated with that section. Oracle JET uses the role definitions for accessibility, specifically WAI-ARIA landmarks. For additional information about Oracle JET and accessibility, see Develop Accessible Oracle JET Apps.

Modify Starter Template Content

To add content, modify the appropriate view template and ViewModel script (if it exists) for the section that you want to update. Add any needed RequireJS modules to the ViewModel's `define()` definition, along with functions to define your ViewModel.

The example below uses the Web Nav Drawer Starter Template, but you can use the same process on any of the Starter Templates.

Before you Begin:

- See the Date and Time Pickers demo in the Oracle JET Cookbook. This task uses code from this sample.

To modify the Starter Template content:

1. In your app's `index.html` file, locate the `oj-module` element for the section you want to modify and identify the template and optional ViewModel script.

   In the Web Nav Drawer Starter Template, the `oj-module` element is using the `config` attribute. The following code sample shows the `mainContent` HTML `oj-module` definition in `index.html`, where the `moduleAdapter` observable, a `ModuleAdapterClass` object, obtains the configuration from its `koObservableConfig` field.

   ```
   <oj-module role="main" class="oj-web-applayout-max-width oj-web-applayout-content"
              config="[[moduleAdapter.koObservableConfig]]">
   </oj-module>
   ```

The return value of the `[[moduleAdapter.koObservableConfig]]` observable is set to the current state of the `CoreRouter` object. The `CoreRouter` object is defined with an initial value of `dashboard` in the app's `appController.js` script, where the page initially loads and no path is yet specified, as shown in the `navData` array below for the empty `path` case. The `router` object is created from the array and then passed to the `moduleAdapter` declaration.

```
let navData = [
   { path: '', redirect: 'dashboard' },
   { path: 'dashboard', detail: { label: 'Dashboard', iconClass: 'oj-ux-ico-bar-
chart' } },
   { path: 'incidents', detail: { label: 'Incidents', iconClass: 'oj-ux-ico-
fire' } },
   { path: 'customers', detail: { label: 'Customers', iconClass: 'oj-ux-ico-contact-
group' } },
   { path: 'about', detail: { label: 'About', iconClass: 'oj-ux-ico-information-
s' } }
];

// Router setup
let router = new CoreRouter(navData, {
  urlAdapter: new UrlParamAdapter()
});
router.sync();

this.moduleAdapter = new ModuleRouterAdapter(router);
this.selection = new KnockoutRouterAdapter(router);
```

The navigation data provider for `oj-navigation-list` element is created as an `ArrayDataProvider` object that associates the available `navData` routes by using the `slice(1)` function to remove the first path definition in the `navdata` array that specifically handles the "empty path" case.

```
// Setup the navDataProvider with the routes, excluding the first redirected route.
this.navDataProvider = new ArrayDataProvider(navData.slice(1), {keyAttributes:
"path"});
```

To modify the starter templates, for example, the Dashboard Content Area, you will modify both `dashboard.html` and `dashboard.js`.

2. To modify the view template, remove unneeded content, and add the new content to the view template file.

   For example, if you are working with an Oracle JET Cookbook sample, you can copy the markup into the view template you identified in the previous step (`dashboard.html`). Replace everything after the `<h1>Dashboard Content Area</h1>` markup in the template with the markup from the sample.

   The following code shows the modified markup if you replace the existing content with a portion of the content from the Date and Time Pickers demo.

```
<div id="div1">
  <oj-label for="dateTime">Default</oj-label>
  <oj-input-date-time id="dateTime" value='{{value}}'>
  </oj-input-date-time>

<br/><br/>

  <span class="oj-label">Current component value is:</span>
  <span><oj-bind-text value="[[value]]"></oj-bind-text></span>
```

```
</div>
```

3. To modify the ViewModel, remove unneeded content, and add the new content as needed. Include any additional RequireJS modules that your new content may need.

   The app's `main.js` file contains the list of RequireJS modules currently included in the app. Compare the list of libraries with the list you need for your app, and add any missing modules to your `define()` function in the ViewModel script. For example, to use the `oj-input-date-time` element shown in the demo and to use the `IntlConverterUtils` namespace API, add `ojs/ojdatetimepicker` and add `ojs/ojconverterutils-i18n` modules to the `dashboard.js` ViewModel script since it's not already defined in `dashboard.js`.

   The sample below shows a portion of the modified `dashboard.js` file, with the changes highlighted in bold.

```
define(['knockout', 'ojs/ojconverterutils-i18n', 'ojs/ojknockout', 'ojs/
ojdatetimepicker', 'ojs/ojlabel'
   ], function(ko, ConverterUtilsI18n) {
   /**
    * The view model for the main content view template
    */
  function DashboardViewModel() {
    var self = this;
    self.value =
ko.observable(ConverterUtilsI18n.IntlConverterUtils.dateToLocalIso(new Date(2020, 0,
1)));
  }

  return DashboardViewModel;
});
```

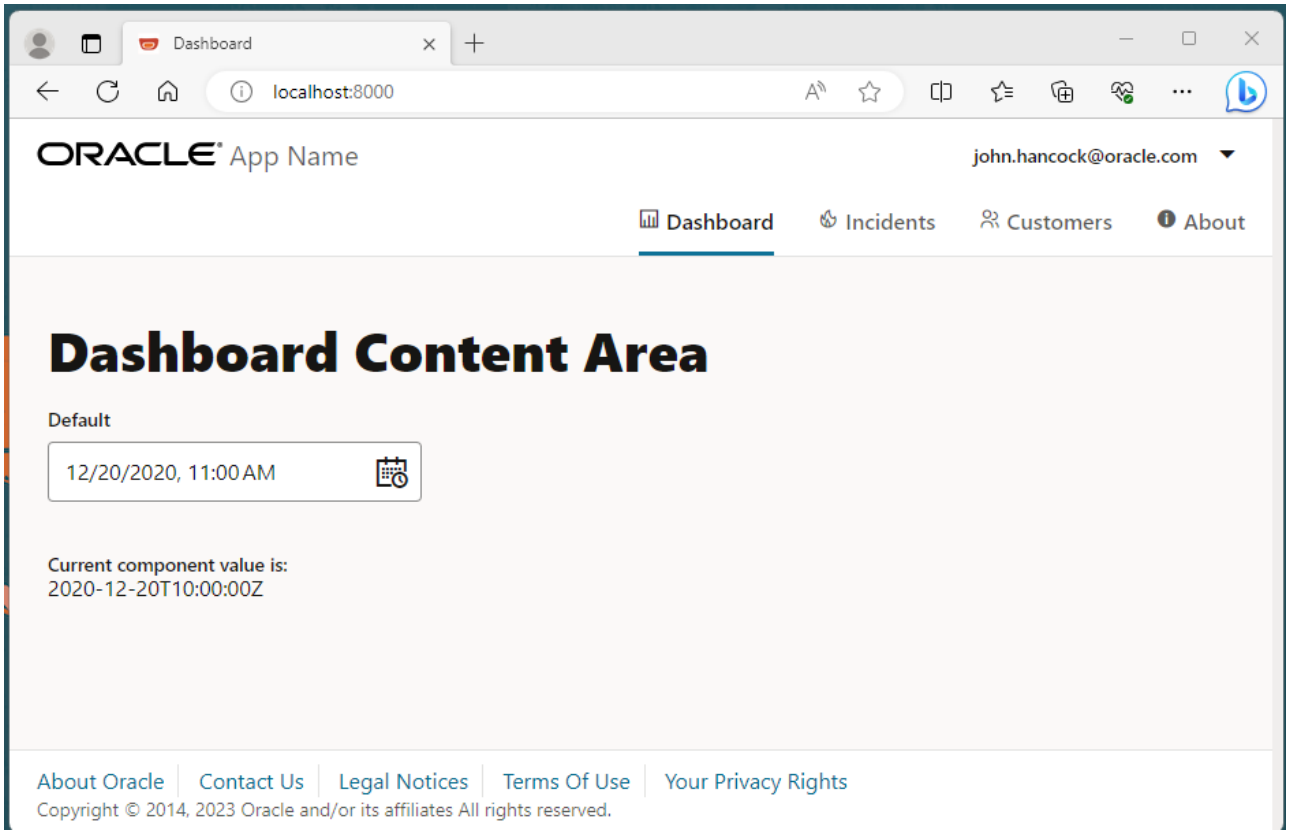
> **Note:**
>
> In this example, you are not copying the entire code section. The Cookbook uses a `require()` call to load and use the needed libraries in a single bootstrap file. The Starter Template that you are pasting uses `define()` to create a RequireJS module that can be used by other parts of your app.

4. If you want to add, change, or delete modules or templates in the app, modify the `main.js` RequireJS bootstrap file and `appController.js` file as needed.

   The `appController.js` file also contains the event handler that responds when a user clicks one of the navigation items. Depending upon your modifications, you may need to update this method as well.

5. Verify the changes in your favorite browser.

   The following image shows the runtime view of the Web Nav Drawer Starter Template with the new Dashboard Content Area content showing `oj-input-date-time` with its current value.



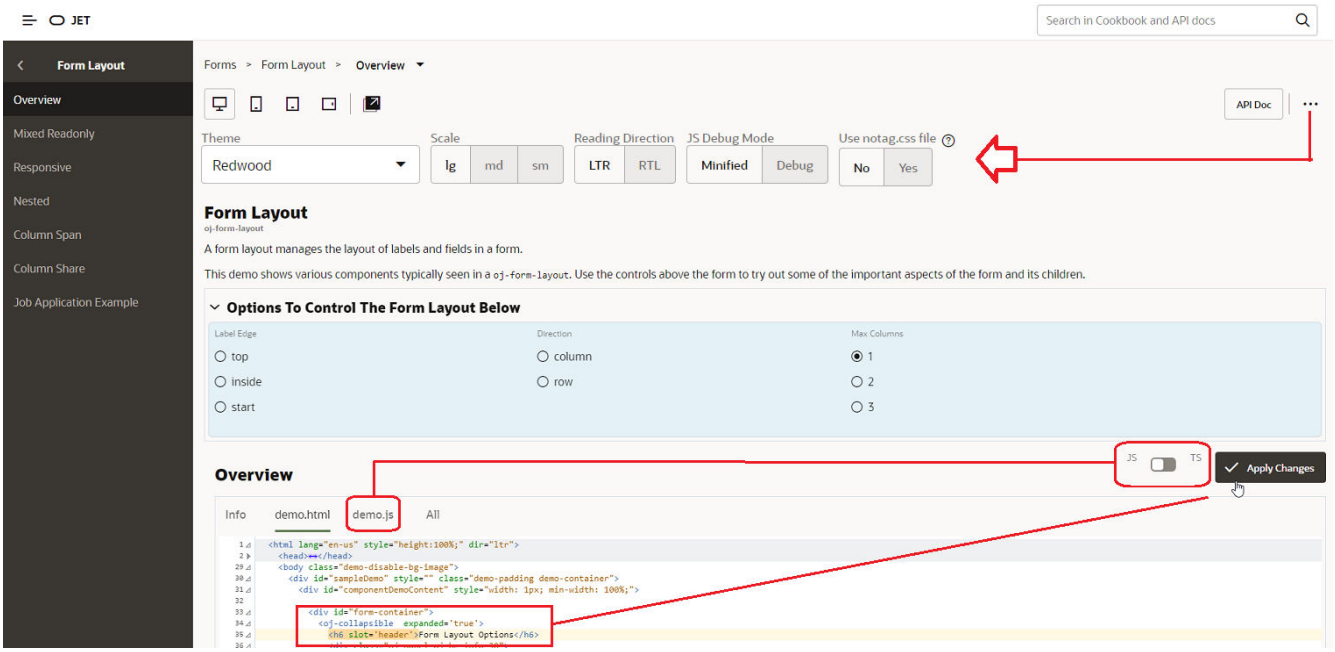
Work with the Oracle JET Cookbook

The Oracle JET Cookbook is a valuable resource for you to use as you develop apps. It includes an implementation of each JET component, along with samples that demonstrate how to implement common usage patterns using one or more of the JET components.

Each Oracle JET Cookbook demo includes a brief introduction, the demo implementation, and, following the implementation, tabs (Info, demo.html, and so on) that describe how to implement the demo, show the source HTML, JavaScript/TypeScript, and, where applicable, the CSS and JSON.

Effective use of the cookbook will make your development task easier. You can, for example, copy code from the cookbook to your own apps, or you can modify code directly in your browser and click **Apply Changes** to preview changes in your browser.

- **API Doc**: Navigates to the API doc for the components that are used in the demo.
- ••• (Setting button): Displays additional controls that allow you to change the theme, the font size, reading direction, and debug mode of the demo code in the browser. Note that these changes affect only the portion of the browser that renders the demo implementation.
- **JS/TS**: Toggle this button to view the demo implementation in JavaScript or TypeScript.
- **Apply Changes**: You can edit the HTML and JS/TS of the demo in the browser and click **Apply Changes** to see your changes in the browser. This is useful if you want to test the behavior or a component when you change an attribute value, for example.



Optimize App Startup Using Oracle CDN and Oracle JET Libraries

You can configure the Oracle JET app to minimize the network load at app startup through the use of Oracle Content Delivery Network (CDN) and the Oracle JET distributions that the CDN supports.

When your production app supports users who access the app from diverse geographical locations, you can perform a significant performance optimization by configuring the Oracle JET app to access Oracle CDN as its source for loading the required Oracle JET libraries and modules. Oracle maintains its CDN with the libraries and modules that are specific to a given Oracle JET release. The CDN support for each release is analogous to the way Oracle JET tooling also supports copying these files into the local `src` folder of the app for a particular release. In both cases, access to the appropriate libraries and modules is automated for the app developer. You configure the app to determine where you want the app to load the libraries and modules from.

After you create your app, the app is configured by default to load the needed libraries and modules from the local `src` folder. This allows you to create the app without the requirement for network access. Then, when you are ready to test in a staging environment or to move to production, you can configure the Oracle JET app to use CDN server replication to reduce the network load that occurs when users access the app at the start of a browser session. When the user initially starts the app in their browser, Oracle CDN ensures a distributed server closest to the geographic location of the user is used to deliver the app's needed third party libraries and Oracle JET modules to the user's browser.

Configuring the app to load from CDN offers these advantages over loading from the app `src` folder:

- Once loaded from a CDN distribution server, the required libraries and modules will be available to other apps that the user may run in the same browser session.

- Enables the option to load bundled libraries and modules using a bundles configuration file that Oracle maintains on CDN. The bundles configuration file groups the most commonly accessed libraries and modules into content packages that are specific to the release and makes them available for delivery to the app as a bundle.

> **Tip:**
>
> Configuring your app to reference the bundles configuration on Oracle CDN is recommended because Oracle maintains the configuration for each release. By pointing your app to the current bundles configuration, you will ensure that your app runs with the latest supported library and module versions. For information about how to enable this bundle loading optimization, see About Configuring the App for Oracle CDN Optimization.

2

Understand the Web App Workflow

Developing client-side web apps with Oracle JET is designed to be simple and efficient using the development environment of your choice and starter templates to ease the development process.

Oracle JET supports creating web apps from a command-line interface:

- Before you can create your first Oracle JET web app using the CLI, you must install the prerequisite packages if you haven't already done so. For details, see Install Oracle JET Tooling.
- Then, use the Oracle JET command-line interface package (`ojet-cli`) to scaffold a web app containing either a blank template or a complete pre-configured app that you can modify as needed.
- After you have scaffolded the app, use the `ojet-cli` to build the app, serve it in a local web browser, and create a package ready for deployment.

You must not use more than one version of Oracle JET to add components to the same HTML document of your web app. Oracle JET does not support running multiple versions of Oracle JET components in the same web page.

Scaffold a Web App

Use the Oracle JET command-line interface (CLI) to scaffold an app that contains a blank template or one pre-configured Starter Template with a basic layout, navigation bar, or navigation drawer. Each Starter Template is optimized for responsive web apps. Additionally, Starter Templates support TypeScript development should you wish to create your app in TypeScript. After scaffolding, you can modify the app as needed.

Before you can create your first Oracle JET web app using the CLI, you must also install the prerequisite packages if you haven't already done so. For details, see Install Oracle JET Tooling.

To scaffold an Oracle JET web app:

1. At a command prompt, enter `ojet create` with optional arguments to create the Oracle JET app and scaffolding.

```
ojet create [directory]
            [--template={template-name:[web]|template-url|template-file}]
            [--typescript]
            [--use-global-tooling]
            [--help]
```

> **Tip:**
>
> You can enter `ojet help` at a terminal prompt to get additional help with the Oracle JET CLI.

For example, the following command creates a web app in the `my-web-app` directory using the web version of the `navbar` template:

```
ojet create my-web-app --template=navbar
```

To scaffold the web app using the same Starter Template but with support for TypeScript version 5.7.2 development, add the `--typescript` argument to the command:

```
ojet create my-web-app --template=navbar --typescript
```

To scaffold the web app that will use the globally-installed `@oracle/oraclejet-tooling` rather than install it locally in the app directory, enter the following command:

```
ojet create my-web-app --use-global-tooling
```

2. Wait for confirmation.

   The scaffolding will take some time to complete. When successful, the console displays:

   ```
   Oracle JET: Your app is ready! Change to your new app directory my-web-app
   and try ojet build and serve...
   ```

3. In your development environment, update the code for your app.

> **Tip:**
>
> If you selected the blank template during scaffolding, you can still follow the same process to add cookbook samples or other content to your app. However, it will be up to you to create the appropriate view templates or viewModel scripts.

About ojet create Command Options for Web Apps

Use `ojet create` with optional arguments to create the Oracle JET web app and scaffolding.

The following table describes the available `ojet create` command options and provides examples for their use.

Option	Description
directory	App location. If not specified, the app is created in the current directory. The directory will be created during scaffolding if it doesn't already exist.

Option	Description

`template`	Template to use for the app. Specify one of the following:
	• *`template-name`*
	Predefined template. You can enter `blank`, `basic`, `navbar` or `navdrawer` . Defaults to `blank` if not specified.
	• *`template-URL`*
	URL to zip file containing the name of a zipped app: `http://`*`path-to-app/app-name`*`.zip`.
	• *`template-file`*
	Path to zip file on your local file system containing the name of a zipped app: "*`path-to-app/app-name`*`.zip`". For example:
	`--template="C:\Users\SomeUser\app.zip"`
	`--template="/home/users/SomeUser/app.zip"`
	`--template="~/projects/app.zip"`
	If the `src` folder is present in the zip file, then all content will be placed under the `src` directory of the app, except for the `script` folder which remains in the root. If no `src` folder is present, the contents of the zip file will be placed at the root of the new app.

`use-global-tooling`	If not specified, the Oracle JET CLI installs the Oracle JET tooling in `appRootDir/node_modules/@oracle` and the following dev dependency appears in the app's `package.json` file.
	`"devDependencies": {`
	`"@oracle/oraclejet-tooling": "https://.../ojet-dev-local/oracle-oraclejet-tooling-18.1.0.tgz"`
	`},`
	If you work with JET apps that use different versions of JET (11.0.0, 10.1.0, and so on), we recommend that you install the JET tooling locally in app.
	If you scaffold an app using `ojet create my-web-app --use-global-tooling`, the scaffolded app uses the globally-installed tooling. On a Windows computer, the globally-installed tooling is in a directory similar to `C:\Users\...\AppData\Roaming\npm\node_modules\@oracle\ojet-cli\node_modules\@oracle\oraclejet-tooling`.

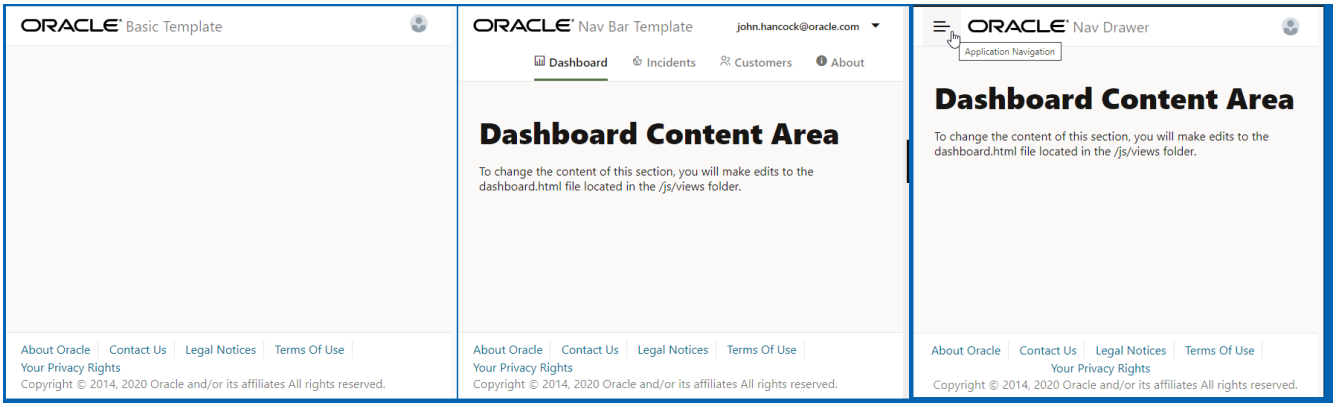
`webpack`	Specify `--webpack` if you want to scaffold an app that uses Webpack. See Use Webpack in Oracle JET App Development.
`installer`	Specify `--installer=yarn` if you want to scaffold an app using the Yarn package manager rather than the default Node package manager (npm). See Yarn Package Manager.
`help`	Displays a man page for the `ojet create` command, including usage and options: `ojet create --help`.

About Scaffolding a Web App

Scaffolding is the process you use in the Oracle JET command-line interface (CLI) to create an app that contains a blank template or one pre-configured with a basic layout, navigation bar, or navigation drawer. Each pre-configured template is optimized for responsive web apps. After scaffolding, you can modify the app as needed.

The following image shows the differences between the pre-configured Starter Templates. The blank template contains an `index.html` file but no UI features. The `basic:web` template is

similar to the blank template but adds responsive styling that will adjust the display when the screen size changes. The `navbar:web` and `navdrawer:web` templates contain sample content and follow best practices for layout, navigation, and styling that you can also modify as needed.



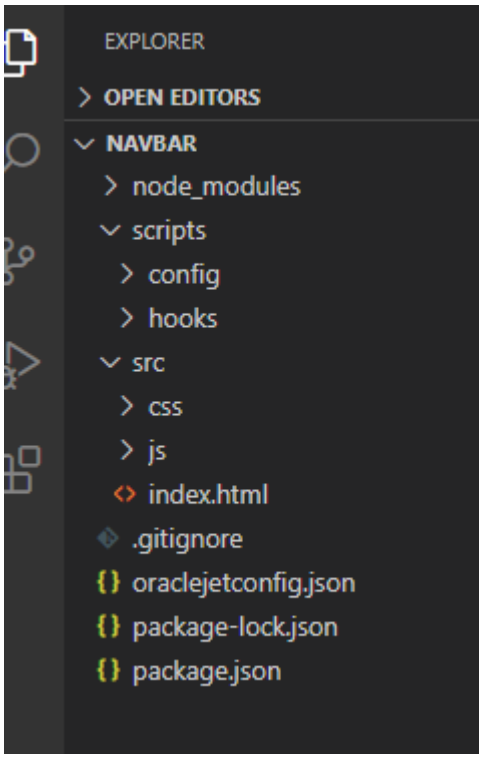
After scaffolding, you can perform the following tasks to customize your app:

- Modify Starter Template Content
- Modify the Web App's File Structure

About the Web App File Structure

The Oracle JET scaffolding process creates files and folders that you modify as needed for your app.

The new app will have a directory structure similar to the one shown in the following image.



The app folders contain the app and configuration files that you will modify as needed for your own app.

Directory or File	Description
`node_modules`	Contains the Node.js modules used by the tooling.
`scripts`	Contains template hook scripts that you can modify to define new build and serve steps for your app. See Customize the Web App Tooling Workflow
`src`	Site root for your app. Contains the app files that you can modify as needed for your own app and should be committed to source control. The content will vary, depending upon your choice of template. Each template, even the blank one, will contain an `index.html` file and a `main.js` RequireJS bootstrap file. Other templates may contain view templates and viewModel scripts pre-populated with content. For example, if you specified the navbar template during creation, the `js/views` and `js/viewModels` folders will contain the templates and scripts for a web app that uses a nav bar for navigation.
`.gitignore`	Defines rules for app folders to ignore when using a GIT repository to check in app source. Users who do not use a GIT repository can use `ojet strip` to avoid checking in content that Oracle JET always regenerates. Note this file must not be deleted since the `ojet strip` command depends on it.
`oraclejetconfig.json`	Contains the default source and staging file paths that you can modify if you need to change your app's file structure.
`package.json`	Defines npm dependencies and project metadata.

After scaffolding, you can perform the following tasks to customize your app:

- Modify Starter Template Content
- Modify the Web App's File Structure

Modify the Web App's File Structure

You can modify your scaffolded app's file structure if the default structure doesn't meet your needs.

The `oraclejetconfig.json` file in your app's top level directory contains the default source and staging file paths that you can modify.

```
{
  "paths": {
    "source": {
      "common": "src",
      "web": "src-web",
      "hybrid": "src-hybrid",
      "javascript": "js",
      "typescript": "ts",
      "styles": "css",
      "themes": "themes"
    },
    "staging": {
```

```
      "web": "web",
      "hybrid": "hybrid",
      "themes": "themes"
    }
  },
  "defaultBrowser": "chrome",
  "sassVer": "1.80.5",
  "defaultTheme": "redwood",
  "architecture": "mvvm",
  "generatorVersion": "18.1.0"
}
```

Other entries in the `oraclejetconfig.json` file, such as the value of the `architecture` property are used by Oracle JET tooling to process the app source appropriately for the app architecture. You should not modify this property's value.

To change the web app's file structure:

1. In your app's top level directory, open `oraclejetconfig.json` for editing.
2. In `oraclejetconfig.json`, change the paths as needed and save the file.

   For example, if you want to change the default `styles` path from `css` to `app-css`, edit the following line in `oraclejetconfig.json`:

   ```
   "styles": "app-css"
   ```

3. Rename the directories as needed for your app, making sure to change only the paths listed in `oraclejetconfig.json`.

   For example, if you changed `styles` to `app-css` in `oraclejetconfig.json`, change the app's `css` directory to `app-css`.

4. Update your app files as needed to reference the changed path.

   For example, if you modified the path to the CSS for your app to `app-css`, update the links appropriately in your app's `index.html`.

   ```
   <link rel="icon" href="app-css/images/favicon.ico" type="image/x-icon" />

       <!-- This is the main css file for the default theme -->
       <!-- injector:theme -->
       <link rel="stylesheet" href="app-css/libs/oj/v18.1.0/redwood/oj-
   redwood-min.css" type="text/css"/>
       <!-- endinjector -->
   ```

5. At the command prompt, from the app root directory, build your app to use the new paths.

   ```
   ojet build
   ```

Add Progressive Web App Support to Web Apps

Add Progressive Web App (PWA) support to your JET web app if you want to give users a native-like mobile app experience on the device where they access your JET web app.

Using the `ojet add pwa` command, you add both a service worker script and a web manifest to your JET web app. You can customize these artifacts to determine how your JET web app behaves when accessed as a PWA.

Using the `ojet add pwa` command, you add both a service worker script, an `assets` folder with a series of image files for app launcher icons and splash screens, plus a web manifest file to your JET web app. You can customize these artifacts to determine how your JET web app behaves when accessed as a PWA.

To add PWA support to your web app:

- At a terminal prompt, in your app's top level directory, enter the following command to add PWA support to your JET web app:

  ```
  ojet add pwa
  ```

When you run the command, Oracle JET tooling makes the following changes to your JET web app:

- Adds the following two files to the app's `src` folder:
  - `assets` folder

    The `assets` folder contains an additional two sub-folders, `icons` and `splashscreens`, that contain image files of various dimensions to use as app launcher icons and splash screens on the devices where you install the JET web app.

  - `manifest.json`

    This file tells the browser about the PWA support in your JET web app, and how it should behave when installed on a user's desktop or mobile device. Use this file to specify the app name to appear on a user's device, plus device-specific icons. For information about the properties that you can specify in this file, see Add a web app manifest.

  - `swinit.js`

    This is the initialization file for the service worker script (`sw.js`).

  - `sw.js`

    This is the service worker script that the browser runs in the background. Use this file to specify any additional resources from your JET app that you want to cache on a user's device if the PWA service worker is installed. By default, JET specifies the following resources to cache:

    ```
    const resourcesToCache = [
      'index.html',
      'manifest.json',
      'js/',
      'css/',
      'assets/',
    ];
    ```

- Registers the splash screen files, the manifest file, and the service worker script in the JET web app's `./src/index.html` file:

```
<html lang="en-us">
  <head>
  . . .
   <meta name="apple-mobile-web-app-title" content="Oracle JET" />
    <meta name="theme-color" content="#000000">
    <!-- Splash screens -->
    <link rel="apple-touch-startup-image" href="assets/splashscreens/
splash-640x1136.jpg" . . .">
  . . .
  <link rel="manifest" href="manifest.json">
</head>
 . . .
<script src="swinit.js"></script>
</body>
```

With these changes, a user on a mobile device, such as an Android phone, can initially access your JET web app through its URL using the Chrome browser, add it to the Home screen of the device, and subsequently launch it like any other app on the phone. Note that browser and platform support for PWA is not uniform. To ensure an optimal experience, test your PWA-enabled JET web app on your users' target platforms (Android, iOS, and so on) and the browsers (Chrome, Safari, and so on).

PWA-enabled JET web apps and service workers require HTTPS. The production environment where you deploy your PWA-enabled JET web app will serve the app over HTTPS. If, during development, you want to serve your JET web app to a HTTPS-enabled server, see Serve a Web App to a HTTPS Server Using a Self-signed Certificate.

Build a Web App

Use the Oracle JET command-line interface (CLI) to build a development version of your web app before serving it to a browser. This step is optional.

Change to the app's root directory and use the `ojet build` command to build your app.

```
ojet build [--cssvars=enabled|disabled
            --theme=themename
            --themes=theme1,theme2,...
            --sass]
```

> **Tip:**
>
> You can enter `ojet help` at a terminal prompt to get help for specific Oracle JET CLI options.

The command will take some time to complete. If it's successful, you'll see the following message:

```
Done.
```

The command will also create a web folder in your app's root to contain the built content.

> **Note:**
>
> You can also use the `ojet build` command with the `--release` option to build a release-ready version of your app. For information, see Package Web Apps.

About ojet build Command Options for Web Apps

Use the `ojet build` command with optional arguments to build a development version of your web app before serving it to a browser.

The following table describes the available options and provides examples for their use.

Option	Description
`--theme`	Theme to use for the app. The theme defaults to `redwood`. You can also enter a different *`themename`* for a custom theme as described in About CSS Variables and Custom Themes in Oracle JET.
`--themes`	Themes to include in the app, separated by commas. If you don't specify the `--theme` flag as described above, Oracle JET will use the first element that you specify in `--themes` as the default theme.
`--cssvars`	Injects a Redwood theme CSS file that supports working with CSS custom properties when you want to override CSS variables to customize the Redwood theme, as described in About CSS Variables and Custom Themes in Oracle JET.
`--sass`	Manages Sass compilation. If you add Sass and specify the `--theme` or `--themes` option, Sass compilation occurs by default and you can use `--sass=false` or `--no-sass` to turn it off. If you add Sass and do not specify a theme option, Sass compilation will not occur by default, and you must specify `--sass=true` or `--sass` to turn it on.

Serve a Web App

Use `ojet serve` to run your web app in a local web server for testing and debugging. By default, the Oracle JET live reload option is enabled which lets you make changes to your app code that are immediately reflected in the browser.

To run your web app from a terminal prompt:

1. At a terminal prompt, change to the app's root directory and use the `ojet serve` command with optional arguments to launch the app.

```
ojet serve [--server-port=server-port-number --livereload-port=live-reload-
port-number
            --livereload
            --watch-files
            --sass
```

```
 --build
 --cssvars=enabled|disabled
 --theme=themename --themes=theme1,theme2,...
 --server-only
 --server-url=server-url
]
```

For example, the following command launches your app in the default web browser with live reload enabled.

```
ojet serve
```

2. Make any desired code change in the `src` folder, and the browser will update automatically to reflect the change unless you set the `--no-livereload` flag.

   While the app is running, the terminal window remains open, and the watch task waits for any changes to the app. For example, if you change the content in `src/js/views/dashboard.html` , the watch task will reflect the change in the terminal as shown below on a Windows desktop.

```
. . .
Listening on port 35729.
Starting watcher.
Watching files.
Watching Interval: 1000.
Watcher: sass is ready.
Watcher: sourceFiles is ready.
Watcher: themes is ready.
Changed: C:\jetMVVMapp\src\ts\views\dashboard.html
Running before_watch hook.
Running after_watch hook.
Page reloaded resume watching.
```

3. To terminate the process, close the app and press Ctrl+C at the terminal prompt. You may need to enter Ctrl+C a few times before the process terminates.

The `ojet serve` command supports a variety of optional arguments that you can use to run the app for specific platforms and with custom themes. Also, when you finish development of your JET app, you use the `ojet serve` command with the `--release` option to serve a release-ready version of your app. See Package Web Apps.

To get additional help for the supported `ojet serve` options, enter `ojet serve --help` at a terminal prompt.

About ojet serve Command Options and Express Middleware Functions

Use `ojet serve` to run your web app in a local web server for testing and debugging.

The following table describes the available `ojet serve` options and provides examples for their use.

Oracle JET tooling uses Express, a Node.js web app framework, to set up and host the web app when you run `ojet serve`. If the ready-to-use `ojet serve` options do not meet your requirements, you can add Express configuration options or write Express middleware functions in Oracle JET's `before_serve.js` hook point. For an example that demonstrates how

to add Express configuration options, see Serve a Web App to a HTTPS Server Using a Self-signed Certificate.

The `before_serve` hook point provides options to determine whether to replace the existing middleware or instead prepend and append a middleware function to the existing middleware. Typically, you'll prepend a middleware function (`preMiddleware`) that you write if you want live reload to continue to work after you serve your web app. Live reload is the first middleware that Oracle JET tooling uses. You must use the `next` function as an argument to any middleware function that you write if you want subsequent middleware functions, such as live reload, to be invoked by the Express instance. In summary, use one of the following arguments to determine when your Express middleware function executes:

- `preMiddleware`: Execute before the default Oracle JET tooling middleware. The default Oracle JET tooling middleware consists of `connect-livereload`, `serve-static`, and `serve-index`, and executes in that order.
- `postMiddleware`: Execute after the default Oracle JET tooling middleware.
- `middleware`: Replaces the default Oracle JET tooling middleware. Use if you need strict control of the order in which middleware runs. Note that you will need to redefine all the default middleware that was previously added by Oracle JET tooling.

Option	Description
`server-port`	Server port number. If not specified, defaults to 8000.
`livereload-port`	Live reload port number. If not specified, defaults to 35729.
`watch-files`	Enable the watch files feature. Watch files is enabled by default (`--watch-files=true`). Use `--watch-files=false` or `--no-watch-files` to disable the watch files feature. Disabling watch files also disables the live reload feature. Configure the interval at which the watch files feature polls the Oracle JET project for updates by configuring a value for the `watchInterval` property in the `oraclejetconfig.json` file. The default value is `1000` milliseconds.
`livereload`	Enable the live reload feature. Live reload is enabled by default (`--livereload=true`). Use `--livereload=false` or `--no-livereload` to disable the live reload feature. Disabling live reload can be helpful if you're working in an IDE and want to use that IDE's mechanism for loading updated apps. The interval at which the live reload feature polls the Oracle JET project depends on the `watch-files` option.
`build`	Build the app before you serve it. By default, an app is built before you serve it (`--build=true`). Use `--build=false` or `--no-build` to suppress the build if you've already built the app and just want to serve it.
`theme`	Theme to use for the app. The theme defaults to `redwood`.
`themes`	Themes to use for the app, separated by commas. If you don't specify the `--theme` flag as described above, Oracle JET will use the first element that you specify in `--themes` as the default theme. Otherwise Oracle JET will serve the app with the theme specified in `--theme`.
`server-only`	Serves the app, as if to a browser, but does not launch a browser. Use this option in cloud-based development environments so that you can attach your browser to the app served by the development machine.
`server-url`	Specify the server URL to serve the Oracle JET app from. For example, `ojet serve --server-url=https://www.example.com/jet`. If not specified, defaults to `localhost`.

Serve a Web App to a HTTPS Server Using a Self-signed Certificate

You can customize the JET CLI tooling to serve your web app to a HTTPS server instead of the default HTTP server that the Oracle JET `ojet serve` command uses.

Do this if, for example, you want to approximate your development environment more closely to a production environment where your web app will eventually be deployed. Requests to your web app when it is deployed to a production environment will be served from an SSL-enabled HTTP server (HTTPS).

To implement this behavior, you'll need to install a certificate in your web app directory. You'll also need to configure the `before_serve.js` hook to do the following:

- Create an instance of Express to host the served web app.
- Set up HTTPS on the Express instance that you've created. You specify the HTTPS protocol, identify the location of the self-signed certificate that you placed in the app directory, and specify a password.
- Pass the modified Express instance and the SSL-enabled server to the JET tooling so that `ojet serve` uses your middleware configuration rather than the ready-to-use middleware configuration provided by the Oracle JET tooling.
- To ensure that live reloads works when your web app is served to the HTTPS server, you'll also create an instance of the live reload server and configure it to use SSL.

If you can't use a certificate issued by a certificate authority, you can create your own certificate (a self-signed certificate). Tools such as OpenSSL, Keychain Access on Mac, and the Java Development Kit's `keytool` utility can be used to perform this task for you. For example, using the Git Bash shell that comes with Git for Windows, you can run the following command to create a self-signed certificate with the OpenSSL tool:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes
```

Once you've obtained the self-signed certificate that you want to use, install it in your app's directory. For example, place the two files generated by the previous command in your app's root directory:

```
...
.gitignore
cert.pem
key.pem
node_modules
...
```

Once you have installed the self-signed certificates in your app, you configure the script for the `before_serve` hook point. To do this, open the `AppRootDir/scripts/hooks/before_serve.js` with your editor and configure it as described by the comments in the following example configuration.

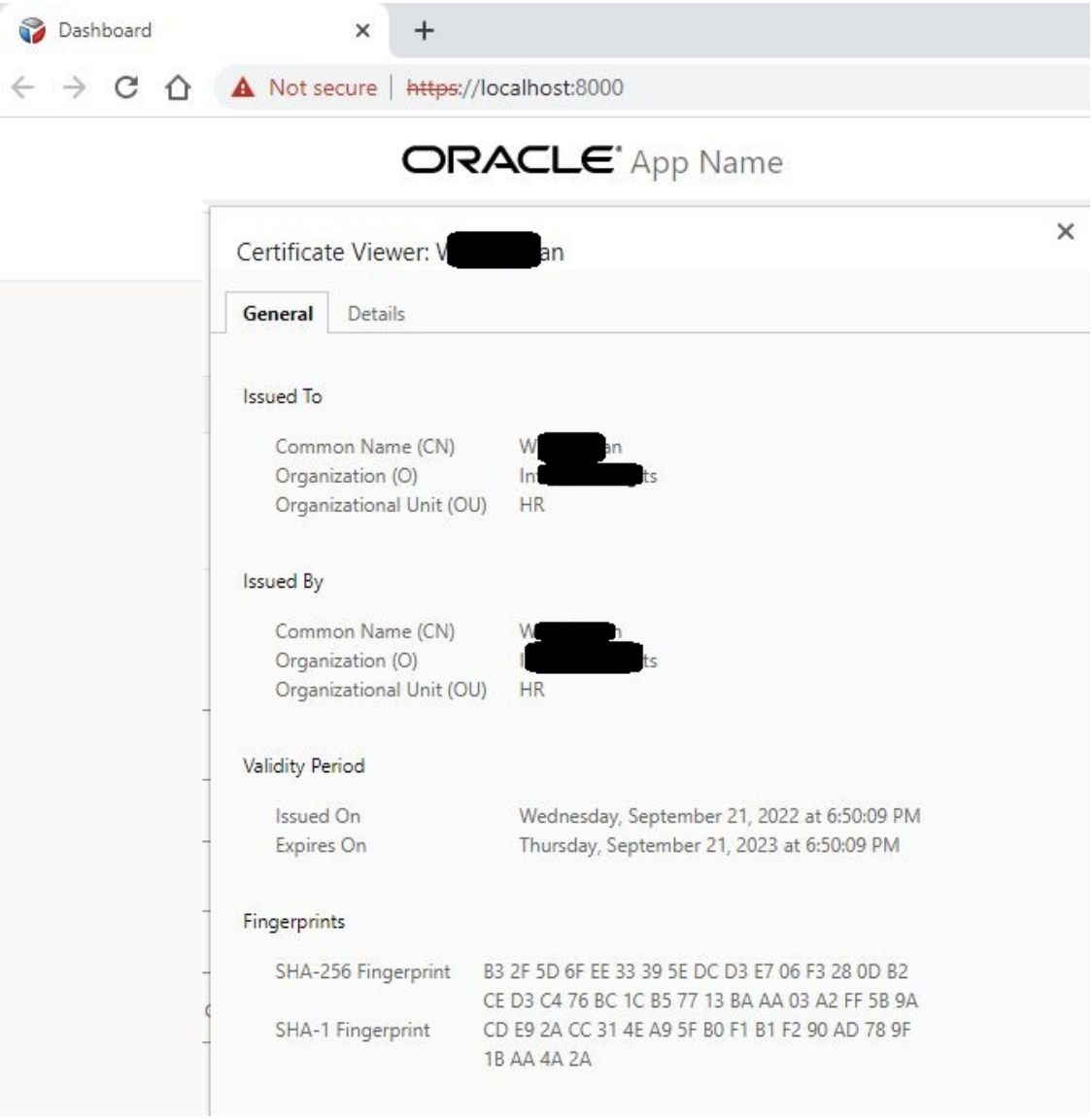
```
'use strict';

module.exports = function (configObj) {
  return new Promise((resolve, reject) => {
    console.log("Running before_serve hook.");
```

```
    // Create an instance of Express, the Node.js web app framework that
Oracle
    // JET tooling uses to host the web apps that you serve using ojet serve
    const express = require("express");

    // Set up HTTPS
    const fs = require("fs");
    const https = require("https");

    // Specify the self-signed certificates. In our example, these files
exist
    // in the root directory of our project.
    const key = fs.readFileSync("./key.pem");
    const cert = fs.readFileSync("./cert.pem");
    // If the self-signed certificate that you created or use requires a
    // password, specify it here:
    const passphrase = "1234";

    const app = express();

    // Pass the modified Express instance and the SSL-enabled server to the
Oracle JET tooling
    configObj['express'] = app;
    configObj['urlPrefix'] = 'https';
    configObj['server'] = https.createServer({
      key: key,
      cert: cert,
      passphrase: passphrase
    }, app);

    // Enable the Oracle JET live reload option using its default port number
so that
    // any changes you make to app code are immediately reflected in the
browser after you
    // serve it
    const tinylr = require("tiny-lr");
    const lrPort = "35729";

    // Configure the live reload server to also use SSL
    configObj["liveReloadServer"] = tinylr({ lrPort, key, cert, passphrase });

    resolve(configObj);
  });
};
```

Once you have completed these configuration steps, run the series of commands (`ojet build` and `ojet serve`, for example) that you typically run to build and serve your web app. As the certificate that you are using is a self-signed certificate rather than a certificate issued by a certificate authority, the browser that you use to access the web app displays a warning the first time that you access the web app. Acknowledge the warning and click the options that allow you to access your web app. On Google Chrome, for example, you click **Advanced** and **Proceed to localhost (unsafe)** if your web app is being served to `https://localhost:8000/`.

Once your web app opens in the browser, you'll see that the HTTPS protocol is used and an indicator that the connection is not secure, because you are not using a certificate from a certificate authority. You can also view the certificate information to confirm that it is the self-

signed certificate that you created. In Google Chrome, click **Not secure** and **Certificate** to view the certificate information.



The `before_serve` hook point is one of a series of script hook points that you can use to customize the tooling workflow for Oracle JET apps. See Customize the Web App Tooling Workflow.

Serve a Web App Using Path-based Routing

Oracle JET apps use parameter-based routing by default. With a couple of changes, you can use path-based routing instead.

With parameter-based routing, the URLs that appear in a user's browser may not be descriptive or easy to remember. For example, a JET app that uses the `navbar` template displays the following URLs in the browser when served locally:

- `http://localhost:8000/?ojr=dashboard`
- `http://localhost:8000/?ojr=incidents`

By way of contrast, the same app configured to use path-based routing uses the following URLs when the app displays the Dashboard or Incidents page:

- `http://localhost:8000/dashboard`
- `http://localhost:8000/incidents`

To implement this behavior, you need to configure the JET app so that when it receives a request from the client for a page, it rewrites the URL before it serves the request. Specifically, you'll need to do the following:

- Update the `appController.js` file so that your app uses path-based routing by creating an instance of the `UrlPathAdapter` class that takes the base URL from which the app is served as a parameter.
- Configure the `before_serve.js` hook to do the following:
  - Write an Express function to inspect the path of the requested URL. If the request is for any of the typical file extensions (`.js`, `.ts`, and so on), the Express instance handles these requests while other requests are passed to the app's `index.html` file for JET's `CoreRouter` to manage.
  - Invoke the Express middleware function that you write using the `before_serve.js` hook point's `configObj['preMiddleware']` option so that the new Express middleware function is invoked before the default middleware used by Oracle JET tooling, such as live reload.

To update the `appController.js` file so that your app uses path-based routing, open the `AppRootDir/src/js/appController.js` with your editor and configure it as described by the comments in the following example configuration.

```
// Replace the entries that the JET tooling generates for 'ojs/
ojurlparamadapter' and UrlParamAdapter
// with entries for 'ojs/ojurlpathadapter' and UrlPathAdapter
define(... 'ojs/ojurlpathadapter', 'ojs/ojarraydataprovider', 'ojs/
ojknockouttemplateutils', 'ojs/ojmodule-element', 'ojs/ojknockout'],
  function(... UrlPathAdapter, ArrayDataProvider, KnockoutTemplateUtils) {
....
let baseUrl = "/";
       let router = new CoreRouter(navData, {
        urlAdapter: new UrlPathAdapter(baseUrl)
      });
      router.sync();
....
```

To configure the script for the `before_serve` hook point, open the `AppRootDir/scripts/hooks/before_serve.js` with your editor and configure it as described by the comments in the following example configuration.

```
'use strict';

module.exports = function (configObj) {

    /* Write an Express middleware function to inspect the path of the
requested
    URL. If the request is for any of the extensions (js, ts, and so on),
```

```
    the Express instance handles these requests while other requests are
    passed to the app's index.html file for the JET CoreRouter to manage.
    */
  function urlRewriteMiddleware(req, res, next) {
    const matchStaticFiles = req.url.match(/\/(js|css)\/.*/);
    req.url = matchStaticFiles ? matchStaticFiles[0] : '/index.html';
    next();
  }
  return new Promise((resolve, reject) => {
    /* Call the Express middleware function that inspects the URL to rewrite
       and prepend it to JET's default middleware so that other options
provided
       by JET's default middleware, such as live reload continue to work.
    */
    configObj['preMiddleware'] = [urlRewriteMiddleware]
    resolve(configObj);
  });
};
```

Customize the Web App Tooling Workflow

Hook points that Oracle JET tooling defines let you customize the behavior of the JET build and serve processes when you want to define new steps to execute during the tooling workflow using script code that you write.

When you create an app, Oracle JET tooling generates script templates in the `/scripts/hooks` app subfolder. To customize the Oracle JET tooling workflow, you can edit the generated templates with the script code that you want the tooling to execute for specific hook points during the build and serve processes. If you do not create a custom hook point script, Oracle JET tooling ignores the script templates and follows the default workflow for the build and serve processes.

To customize the workflow for the build or serve processes, you edit the generated script template file named for a specific hook point. For example, to trigger a script at the start of the tooling's build process, you would edit the `before_build.js` script named for the hook point triggered before the build begins. That hook point is named `before_build`.

Therefore, customization of the build and serve processes that you enforce on Oracle JET tooling workflow requires that you know the following details before you can write a customization script.

- The Oracle JET build or serve mode that you want to customize:
  - Debug — The default mode when you build or serve your app, which produces the source code in the built app.
  - Release — The mode when you build the app with the `--release` option, which produces minified and bundled code in a release-ready app.
- The appropriate hook point to trigger the customization.
- The location of the default hook script template to customize.

About the Script Hook Points for Web Apps

The Oracle JET hooks system defines various script trigger points, also called hook points, that allow you to customize the create, build, serve, package, and restore workflow across the

various command-line interface processes. Customization relies on script files and the script code that you want to trigger for a particular hook point.

The following table identifies the hook points and the workflow customizations they support in the Oracle JET tooling create, build, serve, and restore processes. Unless noted, hook points for the build and serve processes support both debug and release mode.

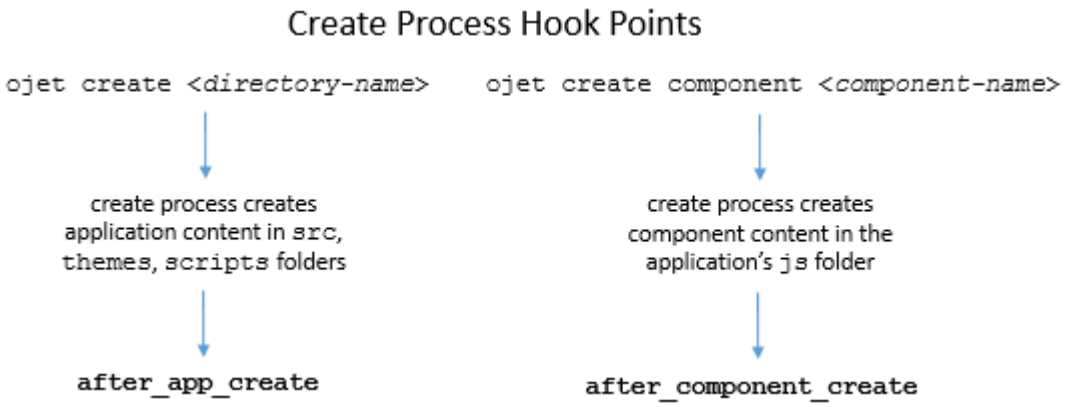
Hook Point	Supported Tooling Process	Description
`after_app_create`	create	This hook point triggers the script with the default name `after_app_create.js` immediately after the tooling concludes the create app process.
`after_app_restore`	restore	This hook point triggers the script with the default name `after_app_restore.js` immediately after the tooling concludes the restore app process.
`before_build`	build	This hook point triggers the script with the default name `before_build.js` immediately before the tooling initiates the build process.
`before_release_build`	build (release mode only)	This hook point triggers the script with the default name `before_release_build.js` before the minification step and the requirejs bundling step occur.
`before_app_typescript`	build / serve	This hook point triggers the script with the default name `before_app_typescript.js` after the build process or serve process steps occur and before the app is transpiled. Use the hook to update, add or remove TypeScript compiler options defined by your app's `tsconfig.json` compiler configuration file. The hook system passes your reference to the modified `tsconfig` object to the TypeScript compiler. A script for this hook point can only be used with a TypeScript app.
`after_app_typescript`	build / serve	This hook point triggers the script with the default name `after_app_typescript.js` after the build process or serve process steps occur and immediately after the `before_app_typescript` hook point executes. This hook provides an entry point for apps that require further processing, such as compiling generated `.jsx` output using babel. A script for this hook point can only be used with a TypeScript app.
`before_component_typescript`	build / serve	This hook point triggers the script with the default name `before_component_typescript.js` after the build process or serve process steps occur and before the component is transpiled. Use the hook to update, add or remove TypeScript compiler options defined by your app's `tsconfig.json` compiler configuration file. The hook system passes your reference to the modified `tsconfig` object to the TypeScript compiler. A script for this hook point can only be used with a TypeScript app.
`after_component_typescript`	build / serve	This hook point triggers the script with the default name `after_component_typescript.js` after the build process or serve process steps occur and immediately after the `before_component_typescript` hook point executes. This hook provides an entry point for apps that require further processing, such as compiling generated `.jsx` output using babel. A script for this hook point can only be used with a TypeScript app.

Hook Point	Supported Tooling Process	Description
`before_injection`	build / serve	This hook point triggers the script with the default name `before_injection.js` after the tooling concludes the before build process and before Oracle JET injects the correct path mappings into your application and performs the tasks to insert the CSS theme into the app. In other words, this hook point controls which files and which markers are patched by Oracle JET itself.
`before_optimize`	build / serve (release mode only)	This hook point triggers the script with the default name `before_optimize.js` before the release mode build/serve process minifies the content.
`before_component_optimize`	build / serve	This hook point triggers the script with the default name `before_component_optimize.js` before the build/serve process minifies the content. A script for this hook point can be used to modify the build process specifically for a project that defines a Web Component.
`after_build`	build	This hook point triggers the script with the default name `after_build.js` immediately after the tooling concludes the build process.
`after_component_create`	build	This hook point triggers the script with the default name `after_component_create.js` immediately after the tooling concludes the create Web Component process. A script for this hook point can be used to modify the build process specifically for a project that defines a Web Component.
`after_component_build`	build (debug mode only)	This hook point triggers the script with the default name `after_component_build.js` immediately after the tooling concludes the Web Component build process. A script for this hook point can be used to modify the build process specifically for a project that defines a Web Component.
`before_serve`	serve	This hook point triggers the script with the default name `before_serve.js` before the web serve process connects to and watches the app.
`after_serve`	serve	This hook point triggers the script with the default name `after_serve.js` after all build process steps complete and the tooling serves the app.
`before_watch`	serve	This hook point triggers the script with the default name `before_watch.js` after the tooling serves the app and before the tooling starts watching for app changes.
`after_watch`	serve	This hook point triggers the script with the default name `after_watch.js` after the tooling starts the watch and after the tooling detects a change to the app.
`after_component_package`	package	This hook point triggers the script with the default name `after_component_package.js` immediately after the tooling concludes the component package process.
`before_component_package`	package	This hook point triggers the script with the default name `before_component_package.js` immediately before the tooling initiates the component package process.

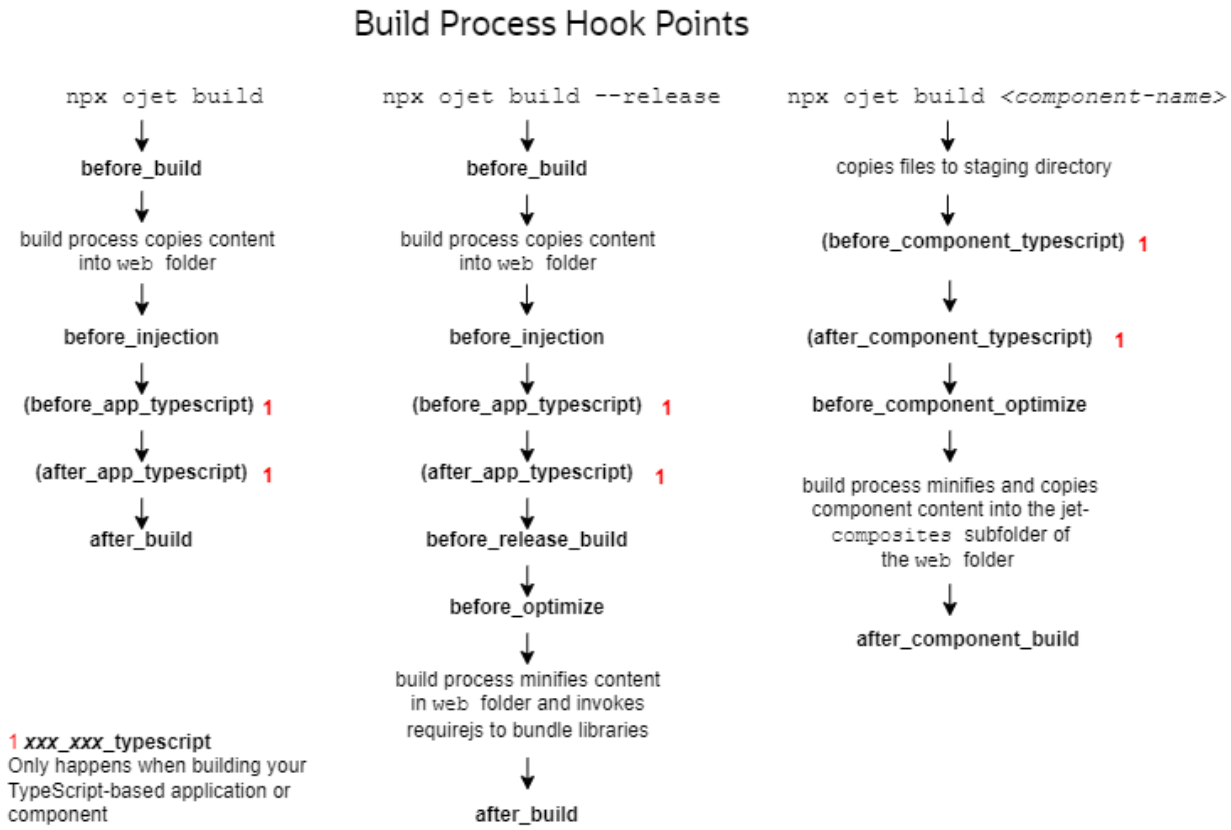
About the Process Flow of Script Hook Points

The Oracle JET hooks system defines various script trigger points, also called hook points, that allow you to customize the create, build, serve, and restore workflow across the various command-line interface processes.

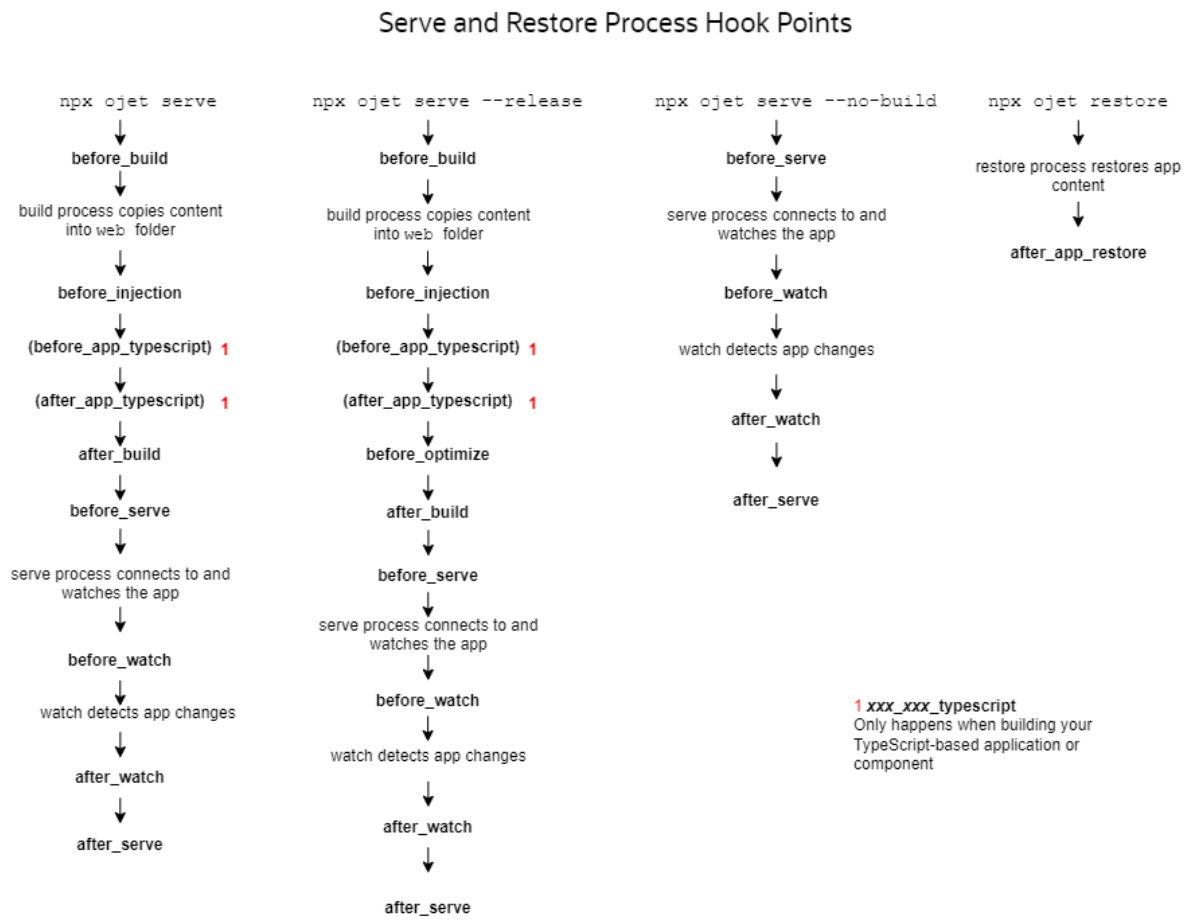
The following diagram shows the script hook point flow for the create process.



The following diagram shows the script hook point flow for the build process.



The following diagram shows the script hook point flow for the serve and restore processes.



Change the Hooks Subfolder Location

When you create an app, Oracle JET tooling generates script templates in the `/scripts/hooks` app subfolder. Your development effort may require you to relocate hook scripts to a common location, for example to support team development.

By default, the hooks system locates the scripts in the `hooks` subfolder using a generated JSON file (`hooks.json`) that specifies the script paths. When the tooling reaches the hook point, it executes the corresponding script which it locates using the `hooks.json` file. If you relocate hook script(s) to a common location, you must edit the `hooks.json` file to specify the new location for the hook scripts that you relocated, as illustrated by the following example.

```
{
    "description": "OJET-CLI hooks configuration file",
    "hooks": {
        "after_app_create": "scripts/hooks/after_app_create.js",
    ...
        "after_serve": "http://example.com/cdn/common/scripts/hooks/
after_serve.js "
    }
}
```

Create a Hook Script for Web Apps

You can create a hook point script to define a new command-line interface process step for your web app. To create a hook script, you edit the hook script template associated with a specific hook point in the tooling build and serve workflow.

The Oracle JET hooks system defines various script trigger points, also called hook points, that allow you to customize the build and serve workflow across the various build and serve modes. Customization relies on script files and the script code that you want to trigger for a particular hook point. Note that the generated script templates that you modify with your script code are named for their corresponding hook point.

To customize the workflow for the build or serve processes, you edit the generated script template file named for a specific hook point. For example, to trigger a script at the start of the tooling's build process, you would edit the `before_build.js` script named for the hook point triggered before the build begins. That hook point is named `before_build`.

A basic example illustrates a simple customization using the `before_optimize` hook, which allows you to control the RequireJS properties shown in bold to modify the app's bundling configuration.

```
requirejs.config(
{
  baseUrl: "web/js",
  name: "main-temp",
  paths: {
  // injector:mainReleasePaths
    "knockout":"libs/knockout/knockout-3.x.x.debug",
    "jquery":"libs/jquery/jquery-3.x.x",
    "jqueryui-amd":"libs/jquery/jqueryui-amd-1.x.x",
     ...
  }
  // endinjector
  out: "web/js/main.js"
}
...
```

A script for this hook point might add one line to the `before_optimize` script template, as shown below. When you build the app with this customization script file in the default location, the tooling triggers the script before calling `requirejs.out()` and changes the `out` property setting to a custom directory path. The result is that the app-generated `main.js` is created in the named directory instead of the default `web/js/main.js` location.

```
module.exports = function (configObj) {
  return new Promise((resolve, reject) => {
    console.log("Running before_optimize hook.");
    configObj.requirejs.out = 'myweb/js/main.js';
    resolve(configObj);
  });
};
```

You can retrieve more information about the definition of `configObj` that is passed into many script hook points as a parameter by making the following modification in one of the build-

related hook points and then running `ojet build`. For example, the `before_build.js` hook point can be modified as follows:

```
module.exports = function (configObj) {
  return new Promise((resolve, reject) => {
      console.log("Running before_build hook.", configObj);
      resolve(configObj);
   });
};
```

The console from where you run the `ojet build` command then displays the available options that you can customize in `configObj`.

```
Cleaning staging path.
Running before_build hook {
  buildType: 'dev',
  opts: {
    stagingPath: 'web',
    injectPaths: {
      startTag: '// injector:mainReleasePaths',
. . .
```

Elsewhere, read examples that illustrate how to use the `configObj` to customize a hook point to, for example, add Express configuration options or write Express middleware functions in the `before_serve.js` hook point if the ready-to-use `ojet serve` options do not meet your requirements. See Serve a Web App to a HTTPS Server Using a Self-signed Certificate and Serve a Web App Using Path-based Routing.

> **Tip:**
>
> If you want to change app path mappings, it is recommended to always edit the `path_mappings.json` file. An exception might be when you want app runtime path mappings to be different from the mappings used by the bundling process, then you might use a `before_optimize` hook script to change the `requirejs.config` `paths` property.

The following example illustrates a more complex build customization using the `after_build` hook. This hook script adds a customize task after the build finishes.

```
'use strict';

const fs = require('fs');
const archiver = require('archiver');

module.exports = function (configObj) {
  return new Promise((resolve, reject) => {
   console.log("Running after_build hook.");

    //Set up the archive
    const output = fs.createWriteStream('my-archive.war');
    const archive = archiver('zip');
```

```
    //Callbacks for the archiver
    output.on('close', () => {
      console.log('Files were successfully archived.');
      resolve();
    });

    archive.on('warning', (error) => {
      console.warn(error);
    });

    archive.on('error', (error) => {
      reject(error);
    });

    //Archive the web folder and close the file
    archive.pipe(output);
    archive.directory('web', false);
    archive.finalize();
  });
};
```

In this example, assume the script templates reside in the default folder generated when you created the app. The goal is to package the app into a ZIP file. Because packaging occurs after the app build process completes, this script is triggered for the `after_build` hook point. For this hook point, the modified script template `after_build.js` will contain the script code to ZIP the app, and because the `.js` file resides in the default location, no hooks system configuration changes are required.

> **Tip:**
>
> Oracle JET tooling reports when hook points are executed in the message log for the build and serve process. You can examine the log in the console to understand the tooling workflow and determine exactly when the tooling triggers a hook point script.

Pass Arguments to a Hook Script for Web Apps

You can pass extra values to a hook script from the command-line interface when you build or serve the web app. The hook script that you create can use these values to perform some workflow action, such as creating an archive file from the contents of the `web` folder.

You can add the `--user-options` flag to the command-line interface for Oracle JET to define user input for the hook system when you build or serve the web app. The `--user-options` flag can be appended to the build or serve commands and takes as arguments one or more space-separated, string values:

```
ojet build --user-options="some string1" "some string2" "some stringx"
```

For example, you might write a hook script that archives a copy of the build output after the build finishes. The developer might pass the user-defined parameter `archive-file` set to the archive file name by using the `--user-options` flag on the Oracle JET command line.

```
ojet build web --user-options="archive-file=deploy.zip"
```

If the flag is appended and the appropriate input is passed, the hook script code may write a ZIP file to the `/deploy` directory in the root of the project. The following example illustrates this build customization using the `after_build` hook. The script code parses the user input for the value of the user defined `archive-file` flag with a promise to archive the app after the build finishes by calling the NodeJS function `fs.createWriteStream()`. This hook script is an example of taking user input from the command-line interface and processing it to achieve a build workflow customization.

```
'use strict';
const fs = require('fs');
const archiver = require('archiver');
const path = require('path');

module.exports = function (configObj) {
  return new Promise((resolve, reject) => {
    console.log("Running after_build hook.");

    //Check to see if the user set the flag
    //In this case we're only expecting one possible user defined
    //argument so the parsing can be simple
    const options = configObj.userOptions;
    if (options){
      const userArgs = options.split('=');
      if (userArgs.length > 1 && userArgs[0] === 'archive-file'){
        const deployRoot = 'deploy';
        const outputArchive = path.join(deployRoot,userArgs[1]);

        //Ensure the output folder exists
        if (!fs.existsSync(deployRoot)) {
          fs.mkdirSync(deployRoot);
        }

        //Set up the archive
        const output = fs.createWriteStream(outputArchive);
        const archive = archiver('zip');

        //callbacks for the archiver
        output.on('close', () => {
          console.log(`Archive file ${outputArchive} successfully created.`);
          resolve();
        });

        archive.on('error', (error) => {
          console.error(`Error creating archive ${outputArchive}`);
          reject(error);
        });

        //Archive the web folder and close the file
        archive.pipe(output);
```

```
        archive.directory('web', false);
        archive.finalize();
      }
      else {
        //Unexpected input - fail with information message
        reject(`Unexpected flags in user-options: ${options}`);
      }
    }
    else {
      //nothing to do
      resolve();
    }
  });
};
```

Use Webpack in Oracle JET App Development

You can use Webpack to manage your Oracle JET app, as well as the build and serve tasks.

If you decide to use Webpack, Oracle JET passes responsibility to Webpack to build and serve the source files of your Oracle JET project. Before you decide to use Webpack, note that it is not possible to use Webpack with Oracle JET apps that need to build, package or publish web components, or to test apps using Oracle JET's Component WebElements UI automation library (TestAdapters).

If you decide to use Webpack in your Oracle JET app, you can specify it as a command-line argument when you scaffold the project, as demonstrated by the following example command:

```
ojet create <app-name> --template=basic --webpack
```

To build and serve the app with Webpack, simply run `ojet build` and `ojet serve` respectively. You cannot use the `ojet serve --release` command. To run a release build from your local development environment, use the `ojet build --release` command, and then use a static server of your choice (for example, http-server) from the `/web` folder.

To add Webpack to an existing Oracle JET app, run the following command from the root directory of your Oracle JET project:

```
ojet add webpack
```

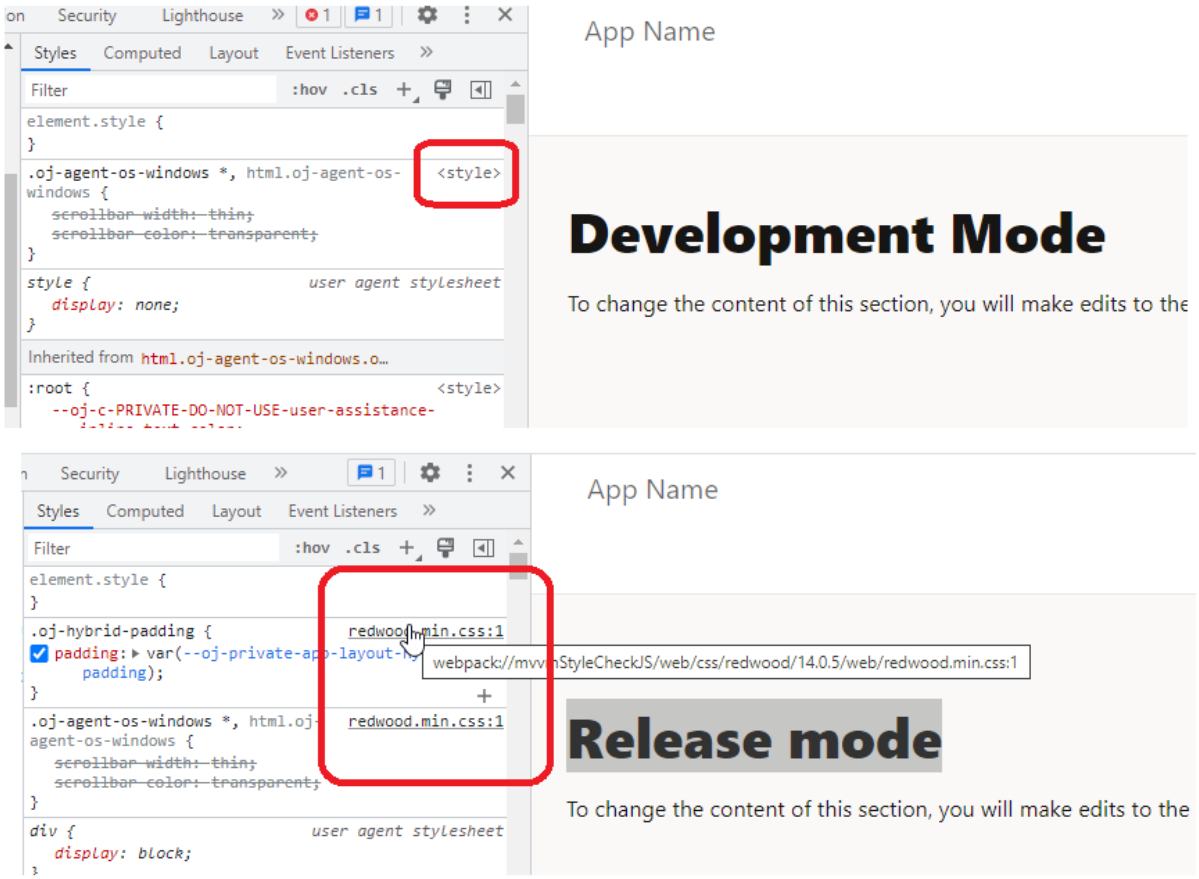
The files and directories in an Oracle JET project that uses Webpack differ to a project an app generated without specifying the `--webpack` argument. The following table describes the differences that result from use of the `--webpack` argument.

Option	Description
`ojet.config.js`	This file manages Oracle JET's default webpack configuration. For more detail about this file and how to configure it, see Configure Oracle JET's Default Webpack Configuration.
`tsconfig.json`	In constrast to the `tsconfig.json` generated for apps without Webpack, the `esModuleInterop` and `resolveJsonModule` flags are set to `true`. The `esModuleInterop` flag allows apps to standardize on default imports for all module types. Calls such as `import * as <importName> from "path/to/import"` should be written as `import <importName> from "path/to/import"`. The `resolveJsonModule` flag allows apps to directly import JSON files. Paired with Webpack's automatic support for resolving JSON file imports, you do not have to use the RequireJs `text!` plugin followed by `JSON.parse` to consume JSON files in the Oracle JET app.

As mentioned at the start of this topic, it is not possible to use Webpack in Oracle JET projects that build, package or publish web components, or projects that need use JET theming. In other words, this means that you cannot use the following commands from the Oracle JET CLI:

- `ojet build (component|pack)`
- `ojet package (component|pack)`
- `ojet publish (component|pack)`

One other thing to note is that when you serve your Oracle JET app in development mode (the default), the Oracle JET app loads styles from memory and styles appear in the `<styles>` tag in the HTML of your browser. In contrast, when you serve an Oracle JET app that you have built in release mode (`ojet build --release`), styles come from the CSS file link that is included in the HTML file. In the following image, with an Oracle JET app instance that runs in development mode and release mode instance, you can see the different entries using the browser's developer tools.



Configure Oracle JET's Default Webpack Configuration

You can configure the default Webpack configuration generated by the Oracle JET CLI through the `webpack` function in the `ojet.config.js` file.

The `webpack` function receives an object with the Oracle JET build context (`context`) and Webpack configuration (`config`). Note the `buildType` property which indicates whether Webpack executes in development or release mode. As for `config`, the default Webpack configuration generated by Oracle JET, you can customize it to fit your needs.

To view the default options in the `ojet.config.js` file, add a console log statement to the `ojet.config.js` file, as demonstrated by the following examples:

```
. . .
webpack: ({ context, config }) => {
  if (context.buildType === "release") {
    // update config with release / production options
  } else {
    // Print out the default webpack configuration options
    // as a JSON string
    console.log(JSON.stringify(config));
    // Or let your terminal console determine how to
    // present the configuration
    console.log(config);
  }
. . .
```

Then build your Oracle JET project using the following command to render the default configuration in the terminal console:

```
ojet build
```

> **Tip:**
>
> Create a JSON-formatted file in Visual Studio Code to view the default configuration in a more readable form to that returned by the terminal.

The default Webpack configuration for an Oracle JET app built in development mode includes the following top-level nodes:

```
{
  "entry": { },
  "output": { },
  "module": {},
  "resolve": {},
  "resolveLoader": {    },
  "plugins": [],
  "mode": "development",
  "devServer": { }
}
```

Once you have identified the configuration setting in Oracle JET's default Webpack configuration that you want to change, you add the alternative value in the `ojet.config.js` file. The following example illustrates how to change the port number when you serve your Oracle JET app in development mode using Webpack.

```
module.exports = {
```

```
  webpack: ({ context, config }) => {
    if (context.buildType === 'release') {
      // update config with release / production options
    } else {

      // update config with development options. In the following example, we
specify
      // a different server port number to the default of 8000
      config.devServer.port = 3000;

     // Print out the default webpack configuration options as a JSON string
     console.log(JSON.stringify(config));
     // Or let your terminal console determine how to present the
configuration
     console.log(config);
    }
    return config;
  }
};
```

3

Design Responsive Apps

Oracle JET includes classes that support a flexbox–based layout, 12-column responsive grid system, responsive form layout, and responsive JavaScript that you can use to design responsive web apps.

Oracle JET and Responsive Design

Responsive design describes a design concept that uses fluid grids, scalable images, and media queries to present alternative layouts based on the media type. With responsive design, you can configure Oracle JET apps to be visually appealing on a wide range of devices, ranging from small phones to wide-screen desktops.

Oracle JET includes classes that support a flexible box layout. In a flex layout, you can lay out the children of a flex container in any direction, and the children will grow to fill unused space or shrink to avoid overflowing the parent. You can also nest boxes (for example, horizontal inside vertical or vertical inside horizontal) to build layouts in two dimensions.

Oracle JET also provides a 12-column grid system and form layout classes that include styles for small, medium, large, and extra large screens or devices that you can use in conjunction with the flex layout classes to achieve finer control of your app's layout. The grid system and form classes use media queries to set the style based on the width of the screen or device, and you can use them to customize your page layout based on your users' needs.

In addition, media queries form the basis for responsive helper classes that show or hide content, align text, or float content based on screen width. They are also the basis for responsive JavaScript that loads content conditionally or sets a component's option based on screen width.

Media Queries

CSS3 media queries use the `@media` at-rule, media type, and expressions that evaluate to true or false to define the cases for which the corresponding style rules will be applied. Media queries form the basis for Oracle JET's responsive classes.

```
<style>
@media media_types (expression){
  /* media-specific rules */
}
</style>
```

The CSS3 specification defines several media types, but specific browsers may not implement all media types. The media type is optional and applies to all types if not specified. The following media query will display a sidebar only when the screen is wider than 767 pixels.

```
@media (max-width: 767px){
 .facet_sidebar {
    display: none;
  }
}
```

Oracle JET defines CSS3 media queries and style class custom properties to define screen widths for the themes included with Oracle JET.

Width and Custom Property Name	Redwood Theme: Default Range in Pixels	Device Examples
small `$screenSmallRange`	0-599	phones
medium `$screenMediumRange`	600-1023	tablet portrait
large `$screenLargeRange`	1024-1439	tablet landscape, desktop
extra large `$screenXlargeRange`	1440 and up	large desktop

For printing, Oracle JET uses the large screen layout for printing in landscape mode and the medium screen layout for printing in portrait mode.

Oracle JET's size defaults and media queries are defined in the Sass variables contained in *site_root*`/scss/oj/18.1.0/common/_oj.common.variables.scss` and are used in the grid, form, and responsive helper style classes. The following code sample shows the responsive screen width variables and a subset of the responsive media queries. In most cases the defaults are sufficient, but be sure to check the file for additional comments that show how you might modify the variables for your app if needed.

```
// responsive screen widths
$screenSmallRange:  0, 767px !default;
$screenMediumRange: 768px, 1023px !default;
$screenLargeRange:  1024px, 1280px !default;
$screenXlargeRange: 1281px, null !default;

// responsive media queries
$responsiveQuerySmallUp:    "print, screen" !default;
$responsiveQuerySmallOnly:  "screen and (max-width: #{upper-
bound($screenSmallRange)})" !default;

$responsiveQueryMediumUp:   "print, screen and (min-width: #{lower-
bound($screenMediumRange)})" !default;
$responsiveQueryMediumOnly: "print and (orientation: portrait), screen and
(min-width: #{lower-bound($screenMediumRange)}) and (max-width: #{upper-
bound($screenMediumRange)})" !default;
$responsiveQueryMediumDown: "print and (orientation: portrait), screen and
(max-width: #{upper-bound($screenMediumRange)})" !default;

$responsiveQueryLargeUp:    "print and (orientation: landscape), screen and
(min-width: #{lower-bound($screenLargeRange)})" !default;
$responsiveQueryLargeOnly:  "print and (orientation: landscape), screen and
(min-width: #{lower-bound($screenLargeRange)}) and (max-width: #{upper-
bound($screenLargeRange)})" !default;
$responsiveQueryLargeDown:  "print and (orientation: landscape), screen and
(max-width: #{upper-bound($screenLargeRange)})" !default;

$responsiveQueryXlargeUp:   "screen and (min-width: #{lower-
bound($screenXlargeRange)})" !default;
```

```
$responsiveQueryXlargeOnly: null !default;
$responsiveQueryXlargeDown: null !default;

$responsiveQueryXXlargeUp:  null !default;

$responsiveQueryPrint:      null !default;
```

Responsive media queries are based on the screen widths defined in the `$screen{size}Range` variables and a range qualifier. For example:

- `$responsiveQuerySmallUp` applies to all screens in the `$screenSmallRange` or wider.
- `$responsiveQuerySmallOnly` applies only to screens in the `$screenSmallRange`.
- `$responsiveQueryXlargeDown` applies to all screens in the `$screenXlargeRange` and narrower.

For additional information about Oracle JET's use of Sass and theming, see Use CSS and Themes in Oracle JET Apps.

For additional information about CSS3 media queries, see https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Media_queries and http://www.w3.org/TR/css3-mediaqueries.

Oracle JET Flex, Grid, Form, and Responsive Helper Class Naming Convention

The Oracle JET flex, form, grid, and responsive style classes use the same naming convention which can help you identify the style size, function, and number of columns the class represents.

Each class follows the same format as shown below:

`oj-size-function-[1-12]columns`

Size can be one of `sm`, `md`, `lg`, `xl`, and `print` and are based on the media queries described in Media Queries. Oracle JET will apply the style to the size specified and any larger sizes unless *function* is defined as `only`. For example:

- `oj-lg-hide` hides content on large and extra-large screens.
- `oj-md-only-hide` hides content on medium screens. The style has no effect on other screen sizes.

You can find a summary of the classes available to you for responsive design in Oracle JET in the Oracle® JavaScript Extension Toolkit (JET) Styling Reference.

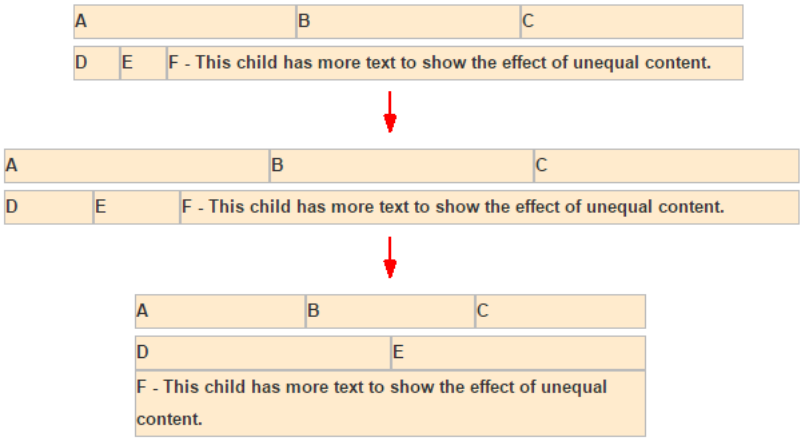
Oracle JET Flex Layouts

Use the Oracle JET `oj-flex` and `oj-flex-item` classes to create flexible box layouts that are based on the CSS flexible box layout model.

In the flex layout model, you create flex containers with children that you can lay out in any direction or order. As the available unused space grows or shrinks, the children grow to fill the unused space or shrink to avoid overflowing the parent.

To create a basic flex layout, add the `oj-flex` class to a container element (HTML `div`, for example) and then add the `oj-flex-item` class to each of the container's children.

The following image shows an example of a default flex layout using the Oracle JET flex box styles. The sample contains two flex containers, each with three children. As the screen size widens, the flex container allocates unused space to each of the children. As the screen size shrinks below the width of one of the flex items, the flex container will wrap the content in that item as needed to no wider than the maximum display width. In this example, this has the effect of causing the F child to wrap to the next row.



The markup for this flex layout is shown below, with the flex layout classes highlighted in bold. The demo-flex-display class sets the color, font weight, height, and border around each flex item in the layout.

```
<div id="container">
  <div class="demo-flex-display">
    <div class="oj-flex">
      <div class="oj-flex-item">A</div>
      <div class="oj-flex-item">B</div>
      <div class="oj-flex-item">C</div>
    </div>

    <div class="oj-flex">
      <div class="oj-flex-item">D</div>
      <div class="oj-flex-item">E</div>
      <div class="oj-flex-item">F - This child has more text to show the
effect of unequal content.</div>
    </div>
  </div>
</div>
```

You can customize the flex layout using styles detailed in Flex Layout Styling and described below.

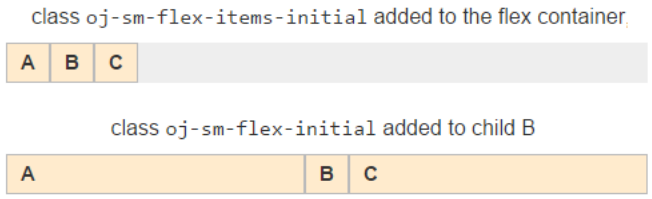
About Modifying the `flex` Property

Oracle JET provides layout classes that you use to modify the properties of your responsive layout.

The Oracle JET layout classes are based on the CSS Flexible Box Layout Module which defines CSS flex common values for flex item sizing. By default, Oracle JET's flex layout

defaults to the `auto` CSS flex property which allows a flex item to shrink or grow as needed for responsive layouts. However, the CSS model sets the default `flex` property to `initial` , which allows a flex item to shrink but will not allow it to grow.

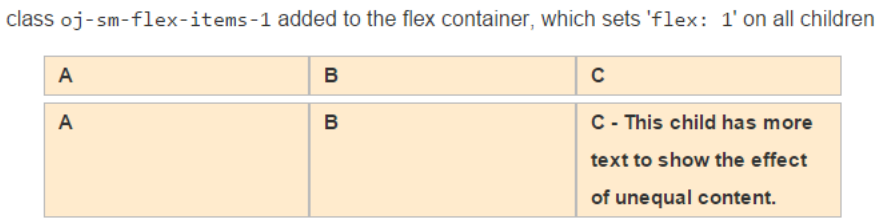
You can achieve the same effect by adding the `oj-sm-flex-items-initial` class to the flex container to set the flex property to `initial` for all child flex items, or add the `oj-sm-flex-initial` class to an individual flex item to set its property to `initial`. The following image shows the effect.



The code sample below shows the markup. In this example, padding is also added to the content using the `oj-flex-items-pad` class on the parent container.

```
<div id="container">
  <div class="demo-flex-display oj-flex-items-pad">
    <div class="oj-flex oj-sm-flex-items-initial">
      <div class="oj-flex-item">A</div>
      <div class="oj-flex-item">B</div>
      <div class="oj-flex-item">C</div>
    </div>

    <div class="oj-flex">
      <div class="oj-flex-item">A</div>
      <div class="oj-sm-flex-initial oj-flex-item">B</div>
      <div class="oj-flex-item">C</div>
    </div>
  </div>
</div>
```

You can also override the default `auto` flex property by using the `oj-size-flex-items-1` class on the flex container. This class sets the `flex` property to `1`, and all flex items in the flex container with a screen size of *size* or higher will have the same width, regardless of the items' content.



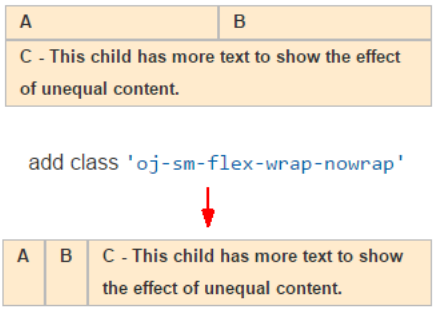
To set the flex property to 1 on an individual flex item, add `oj-sm-flex-1` to the flex item. The Flex Layouts section in the Oracle JET Cookbook includes the examples used in this section that you can use and modify to observe the flex layout's responsive behavior.

About Wrapping Content with Flex Layouts

You can set the `flex-wrap` property to `nowrap` by adding `oj-sm-flex-nowrap` to the `oj-flex` container.

By default, Oracle JET sets the CSS `flex-wrap` property to `wrap`, which sets the flex container to multi-line. Child flex items will wrap content to additional lines when the screen width shrinks to less than the width of the flex item's content. However, the CSS model sets the `flex-wrap` property to `nowrap`, which sets the flex container to single-line. When a child item's content is too wide to fit on the screen, the content will wrap within the child.

The following image shows the effect of changing the `flex-wrap` property to `nowrap`.



About Customizing Flex Layouts

You can customize an Oracle JET flex layout by adding the appropriate style to the flex container or child. The flex layout classes support some commonly-used values.

- `flex-direction`
- `align-items`
- `align-self`
- `justify-content`
- `order`

The Oracle JET Cookbook includes examples for customizing your flex layout at: Flex Layouts.

Oracle JET Grids

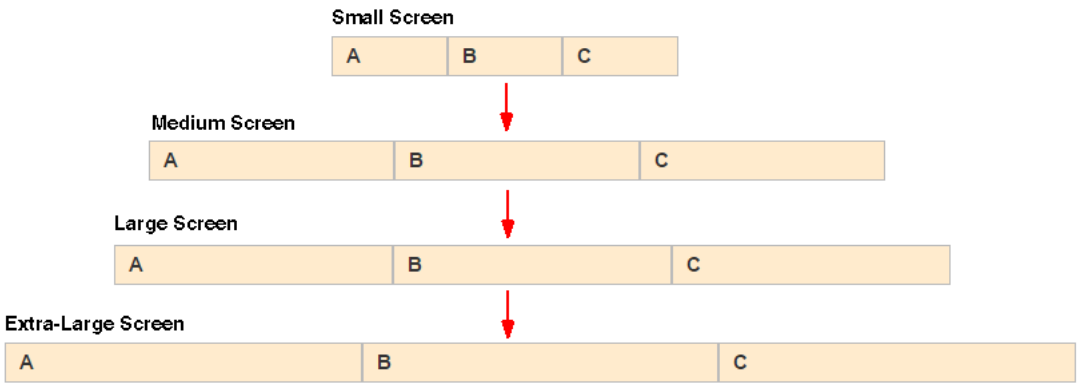
Use the Oracle JET grid classes with flex layouts to create grids that vary the number and width of columns based on the width of the user's screen.

The Responsive Grids section in the Oracle JET Cookbook provides several examples and recipes for using the Oracle JET grid system, and you should review them to get accustomed to the grid system.

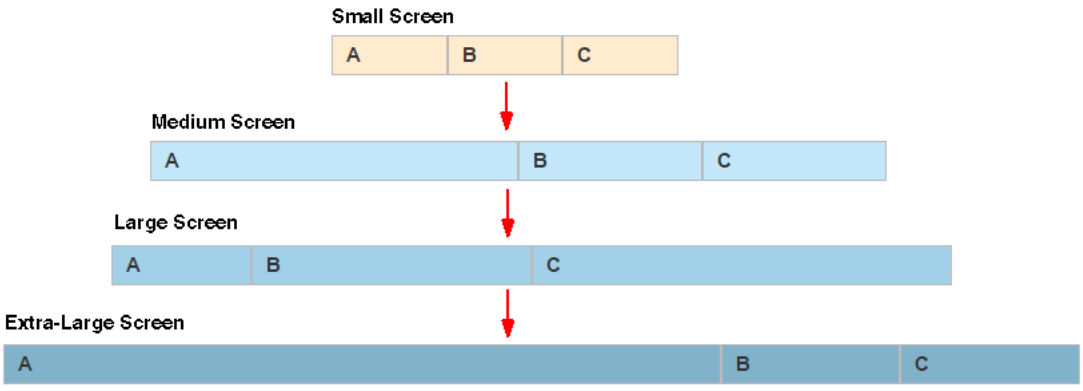
About the Grid System

Oracle JET provides a 12-column responsive mobile-first grid system that you can use for responsive design. The grid builds upon the Oracle JET flex layout and allows you to specify the widths of each flex item using sizing classes for small, medium, large, and extra-large screens.

For example, you can use the grid classes to change the default display in the Flex Layouts Auto example to use different widths for the flex items when the screen size changes. As shown in the image below, the flex layout by default will allocate the unused space evenly to the three flex items regardless of the screen size.



When the defaults are not sufficient, you can specify relative widths for the flex items when the screen size changes. In the following image, the flex layout is using grid classes to define different widths when the screen size changes to medium, large, and extra large.



The grid classes follow the Oracle JET Flex, Grid, Form, and Responsive Helper Class Naming Convention. Use `oj-`*size*`-`*numberofcolumns* to set the width to the specified *numberofcolumns* when the screen is the specified *size* or larger. For example:

- `oj-sm-6` works on all screen sizes and sets the width to 6 columns.
- `oj-lg-3` sets the width to 3 columns on large and extra-large screens.

- `oj-sm-6` and `oj-lg-3` on the same flex item sets the width to 6 columns wide on small and medium screens and 3 columns wide on large and extra-large screens.

Design for the smallest screen size first and then customize for larger screens as needed. You can further customize the grid by adding one of the Grid Convenience Classes or by using one of the responsive helper classes described in Use the Responsive Helper Classes.

The following code sample shows the markup for the modified Flex Auto Layout display, with grid classes defined for medium, large, and extra-large screens.

```
<div class="oj-flex">
  <div class="oj-md-6 oj-lg-2 oj-xl-8 oj-flex-item">A</div>
  <div class="oj-md-3 oj-lg-4 oj-xl-2 oj-flex-item">B</div>
  <div class="oj-md-3 oj-lg-6 oj-xl-2 oj-flex-item">C</div>
</div>
```

When the screen size is small, the flex layout default styles are used, and each item uses the same amount of space. When the screen size is medium, the A flex item will use 6 columns, and the B and C flex items will each use 3 columns. When the screen size is large, The A flex item will use 2 columns, the B flex item will use 4 columns, and the C flex item will use 6 columns. Finally, when the screen size is extra large, the A flex item will use 8 columns, and the B and C flex items will each use 2 columns.

For a complete example that illustrates working with the grid system, see Responsive Grids.

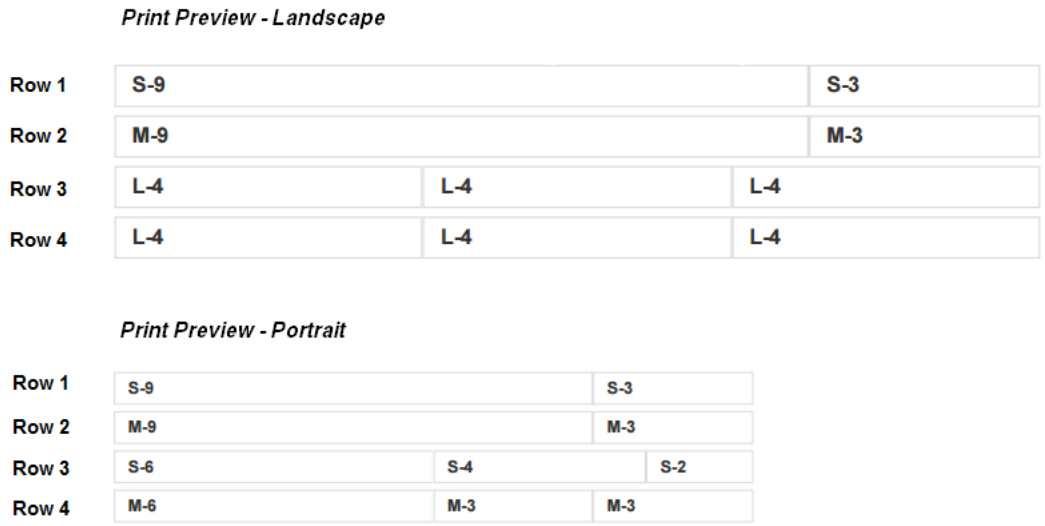
The Grid System and Printing

The Oracle JET grid system applies the large styles for printing in landscape mode and the medium style for printing in portrait mode if they are defined. You can use the defaults or customize printing using the print style classes.

In the grid example below, Row 2 and Row 4 include the `oj-md-*` style classes. Row 3 and Row 4 include the `oj-lg-4` style for all columns in the row.

```
<div class="demo-grid-sizes demo-flex-display">
  <div class="oj-flex oj-flex-items-pad">
    <div class="oj-sm-9 oj-flex-item"></div>
    <div class="oj-sm-3 oj-flex-item"></div>
  </div>
  <div class="oj-flex oj-flex-items-pad">
    <div class="oj-sm-6 oj-md-9 oj-flex-item"></div>
    <div class="oj-sm-6 oj-md-3 oj-flex-item"></div>
  </div>
  <div class="oj-flex oj-flex-items-pad">
    <div class="oj-sm-6 oj-lg-4 oj-flex-item"></div>
    <div class="oj-sm-4 oj-lg-4 oj-flex-item"></div>
    <div class="oj-sm-2 oj-lg-4 oj-flex-item"></div>
  </div>
  <div class="oj-flex oj-flex-items-pad ">
    <div class="oj-sm-8 oj-md-6 oj-lg-4 oj-xl-2 oj-flex-item"></div>
    <div class="oj-sm-2 oj-md-3 oj-lg-4 oj-xl-8 oj-flex-item"></div>
    <div class="oj-sm-2 oj-md-3 oj-lg-4 oj-xl-2 oj-flex-item"></div>
  </div>
</div>
```

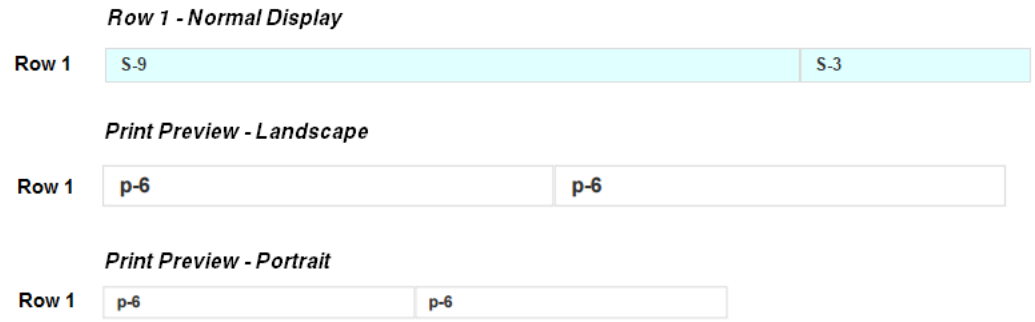
As shown in the following print preview, when you print this grid in landscape mode, the `oj-lg-4` style classes will be applied on Row 3 and Row 4. When you print the grid in portrait mode, the `oj-md-*` style classes apply on Row 2 and Row 4.



If you want to change the printing default, you can set the Sass `$responsiveQueryPrint` variable to `print` in a custom settings file. After you enable the print classes, you can add the `oj-print-numberofcolumns` style class to the column definition. This has the effect of changing the column sizes for printing purposes only. In the following example, Row 1 includes the `oj-print-6` class for each column in the row.

```
<div class="oj-flex oj-flex-items-pad">
  <div class="oj-sm-9 oj-print-6 oj-flex-item"></div>
  <div class="oj-sm-3 oj-print-6 oj-flex-item"></div>
</div>
```

In normal mode, Row 1 contains two columns, one column with a size of 9 and one column with a size of 3, regardless of screen size. If you do a print preview, however, you'll see that Row 1 will print with two columns in portrait and landscape mode, both with a size of 6.

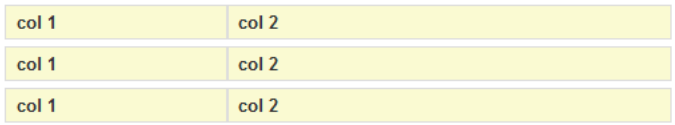


Grid Convenience Classes

Oracle JET's grid system includes convenience classes that make it easier to create two- and four- column layouts with specified widths.

- `oj-size-odd-cols-numberofcolumns`: Use this in a 2-column layout. Instead of putting sizing classes on every column, you can put a single class on the flex parent. The number of columns specifies how many of the 12 columns the odd-numbered columns can use. In a 2-column layout, the even-numbered columns will take up the remainder of the columns.

For example, setting `oj-md-odd-cols-4` on the flex parent will have the effect of setting the odd column (`col1`) width to 4 and the even column (`col2`) width to 8 for all rows in the grid on medium-size screens and higher.



The code sample below shows the grid configuration used to render the figure. The example also sets `oj-sm-odd-cols-12` which will set the odd column width to 12 on small screens, displaying `col2` on a new row.

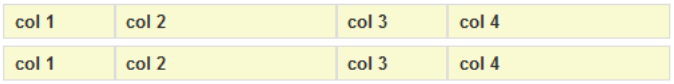
```
<div class="oj-md-odd-cols-4 oj-flex-items-pad">
  <div class="oj-flex">
    <div class="oj-flex-item">col 1</div>
    <div class="oj-flex-item">col 2</div>
  </div>
  <div class="oj-flex">
    <div class="oj-flex-item">col 1</div>
    <div class="oj-flex-item">col 2</div>
  </div>
  <div class="oj-flex">
    <div class="oj-flex-item">col 1</div>
    <div class="oj-flex-item">col 2</div>
  </div>
</div>
```

You could achieve the same effect by defining `oj-md-4` for the first column's width and `oj-md-8` for the second column's width on each flex item.

```
<div class="oj-flex-items-pad"
  <div class="oj-flex">
    <div class="oj-sm-12 oj-md-4 oj-flex-item">col 1</div>
    <div class="oj-sm-12 oj-md-8 oj-flex-item">col 2</div>
  </div>
  <div class="oj-flex">
    <div class="oj-sm-12 oj-md-4 oj-flex-item">col 1</div>
    <div class="oj-sm-12 oj-md-8 oj-flex-item">col 2</div>
  </div>
  <div class="oj-flex">
    <div class="oj-sm-12 oj-md-4 oj-flex-item">col 1</div>
    <div class="oj-sm-12 oj-md-8 oj-flex-item">col 2</div>
  </div>
</div>
```

`oj-`*`size`*`-even-cols-`*`numberofcolumns`*: Use in a 4-column layout. In this layout, you must use both the `odd-cols` class to control the width of odd-numbered columns and the `even-cols` class to control the width of the even columns.

For example, setting `oj-md-odd-cols-2` and `oj-md-even-cols-4` on the flex parent has the effect of setting the first and third column widths to 2, and the second and fourth column widths to 4.



The code sample below shows the grid configuration used to render the figure.

```
<div class="oj-sm-odd-cols-12 oj-md-odd-cols-2 oj-md-even-cols-4 oj-flex-items-pad">
  <div class="oj-flex">
    <div class="oj-flex-item">col 1</div>
    <div class="oj-flex-item">col 2</div>
    <div class="oj-flex-item">col 3</div>
    <div class="oj-flex-item">col 4</div>
  </div>
  <div class="oj-flex">
    <div class="oj-flex-item">col 1</div>
    <div class="oj-flex-item">col 2</div>
    <div class="oj-flex-item">col 3</div>
    <div class="oj-flex-item">col 4</div>
  </div>
</div>
```

If you don't use the convenience classes, you must define the size classes on every column in every row as shown below.

```
<div class="oj-flex-items-pad">
  <div class="oj-flex">
    <div class="oj-sm-odd-cols-12 oj-md-2 oj-flex-item">col 1</div>
    <div class="oj-sm-odd-cols-12 oj-md-4 oj-flex-item">col 2</div>
    <div class="oj-sm-odd-cols-12 oj-md-2 oj-flex-item">col 3</div>
    <div class="oj-sm-odd-cols-12 oj-md-4 oj-flex-item">col 4</div>
  </div>
  <div class="oj-flex">
    <div class="oj-sm-odd-cols-12 oj-md-2 oj-flex-item">col 1</div>
    <div class="oj-sm-odd-cols-12 oj-md-4 oj-flex-item">col 2</div>
    <div class="oj-sm-odd-cols-12 oj-md-2 oj-flex-item">col 3</div>
    <div class="oj-sm-odd-cols-12 oj-md-4 oj-flex-item">col 4</div>
  </div>
</div>
```

Responsive Form Layouts

Oracle JET provides the `oj-form-layout` component that you can use to create form layouts that adjust to the size of the user's screen. Use the `oj-input-text` and `oj-text-area` custom elements within the `oj-form-layout` component to create an organized layout.

For more information on `oj-form-layout` component, see Form Layouts.

Add Responsive Design to Your App

To create your responsive app using Oracle JET, design for the smallest device first and then customize as needed for larger devices. Add the applicable app, flex, grid, form, and responsive classes to implement the design.

To design a responsive app using Oracle JET classes:

1. Design the app content's flex layout.

   For help, see Oracle JET Flex Layouts.

2. If the flex layout defaults are not sufficient and you need to specify column widths when the screen size increases, add the appropriate responsive grid classes to your flex items.

   For help, see Oracle JET Grids.

3. If you're adding a form to your page, add the appropriate form style classes.

4. Customize your design as needed.

   For additional information, see:

   - Use Responsive JavaScript
   - Use the Responsive Helper Classes
   - Create Responsive Images
   - Change Default Font Size

For the list of responsive design classes and their behavior, see the Responsive* classes listed in the Oracle® JavaScript Extension Toolkit (JET) Styling Reference.

For Oracle JET Cookbook examples that implement responsive design, see:

- Flex Layouts
- Responsive Grids
- Form Layouts
- Responsive Behaviors

Use Responsive JavaScript

Oracle JET includes the `ResponsiveUtils` and `ResponsiveKnockoutUtils` utility classes that leverage media queries to change a component's `value` option or load content and images based on the user's screen size or device type.

The Responsive JavaScript Classes

The `ResponsiveUtils` and `ResponsiveKnockoutUtils` responsive JavaScript classes provide methods that you can use in your app's JavaScript to obtain the current screen size and use the results to perform actions based on that screen size. In addition, the `ResponsiveUtils` provides a method that you can use to compare two screen sizes, useful for performing actions when the screen size changes.

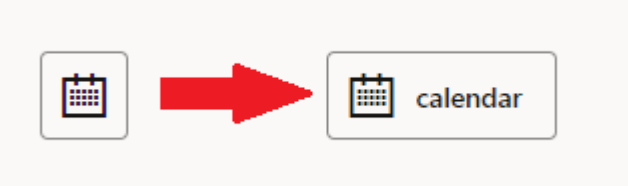
JavaScript Class	Methods	Description
`responsiveUtils`	`compare(size1, size2)`	Compares two screen size constants. Returns a negative integer if the first argument is less than the second, a zero if the two are equal, and a positive integer if the first argument is more than the second. The screen size constants identify the screen size range media queries. For example, the `ResponsiveUtils.SCREEN_RANGE.SM` constant corresponds to the Sass `$screenSmallRange` variable and applies to screen sizes smaller than 768 pixels in width.
`responsiveUtils`	`getFrameworkQuery(frameworkQueryKey)`	Returns the media query to use for the framework query key parameter. The framework query key constant corresponds to a Sass responsive query variable. For example, the `ResponsiveUtils.FRAMEWORK_QUERY_KEY.SM_UP` constant corresponds to the $`responsiveQuerySmallUp` responsive query which returns a match when the screen size is small and up.

JavaScript Class	Methods	Description
`responsiveKnockoutUtils`	`createMediaQueryObservable(queryString)`	Creates a Knockout observable that returns true or false based on a media query string. For example, the following code will return true if the screen size is 400 pixels wide or larger. `var customQuery = ResponsiveKnockoutUtils.createMediaQueryObservable( '(min-width: 400px)');`
`responsiveKnockoutUtils`	`createScreenRangeObservable()`	Creates a computed Knockout observable, the value of which is one of the `ResponsiveUtils.SCREEN_RANGE` constants. For example, on a small screen (0 - 767 pixels), the following code will create a Knockout observable that returns `ResponsiveUtils.SCREEN_RANGE.SM`. `self.screenRange = ResponsiveKnockoutUtils.createScreenRangeObservable();`

For additional detail about `responsiveUtils`, see the ResponsiveUtils API documentation. For more information about `responsiveKnockoutUtils`, see ResponsiveKnockoutUtils.

Change a Custom Element's Attribute Based on Screen Size

You can set the value for a custom element's attribute based on screen size using the responsive JavaScript classes. For example, you may want to add text to a button label when the screen size increases using the `oj-button` element's `display` attribute.



In this example, the `oj-button` element's `display` attribute is defined for icons. The code sample below shows the markup for the button.

```
<div id="optioncontainer">
  <oj-button display="[[large() ? 'all' : 'icons']]">
    <span slot='startIcon' class="oj-fwk-icon oj-fwk-icon-calendar"></span>
    calendar
  </oj-button>
</div>
```

The code sample also sets the `oj-button` display attribute to `all`, which displays both the label and icon when the `large()` method returns `true`, and icons only when the `large()` method returns false.

The code sample below shows the code that sets the value for `large()` and completes the knockout binding. In this example, `lgQuery` is set to the `LG_UP` framework query key which applies when the screen size is large or up. `this.large` is initially set to `true` as the result of the call to `ResponsiveKnockoutUtils.createMediaQueryObservable(lgQuery)`. When the

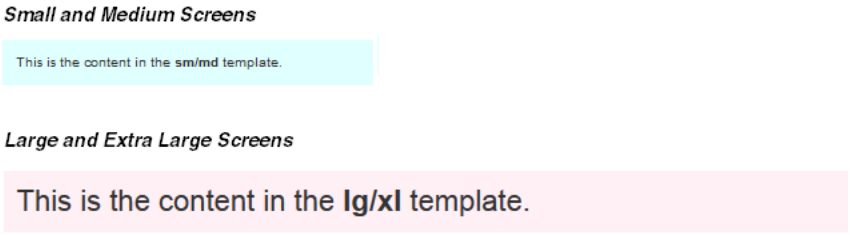
screen changes to a smaller size, the `self.large` value changes to `false`, and the `display` attribute value becomes `icons`.

```
require(['knockout', 'ojs/ojbootstrap', 'ojs/ojresponsiveutils', 'ojs/
ojresponsiveknockoututils', 'ojs/ojknockout', 'ojs/ojbutton'],
  function(ko, Bootstrap, ResponsiveUtils, ResponsiveKnockoutUtils)
  {
    function MyModel(){
      // observable for large screens
      var lgQuery = ResponsiveUtils.getFrameworkQuery(
                        ResponsiveUtils.FRAMEWORK_QUERY_KEY.LG_UP);

      this.large = ResponsiveKnockoutUtils.createMediaQueryObservable(lgQuery);
    }

    Bootstrap.whenDocumentReady().then(
       function ()
       {
         ko.applyBindings(new MyModel(), document.getElementById('optioncontainer'));
       }
    );
  });
```

The Oracle JET Cookbook contains the complete code for this example which includes a demo that shows a computed observable used to change the button label's text depending on the screen size. You can also find examples that show how to use custom media queries and Knockout computed observables. For details, see Responsive JavaScript Framework Queries.

Conditionally Load Content Based on Screen Size

You can change the HTML content based on screen size using the responsive JavaScript classes. For example, you might want to use a larger font or a different background color when the screen size is large.



In this example, the HTML content is defined in Knockout templates. The markup uses Knockout's data-bind utility to display a template whose name depends on the value returned by the `large()` call. If the screen is small or medium, the app will use the `sm_md_template`. If the screen is large or larger, the app will use the `lg_xl_template`.

```
<div id="sample_container">

  <!-- large template -->
  <script type="text/html" id="lg_xl_template">
    <div id="lg_xl"
         style="background-color:lavenderblush;
                padding: 10px; font-size: 22px" >
     This is the content in the <strong>lg/xl</strong> template.
    </div>
```

```
  </script>

  <!-- small template -->
  <script type="text/html" id="sm_md_template">
    <div id="sm_md"
         style="background-color:lightcyan;
                padding: 10px; font-size: 10px" >
      This is the content in the <strong>sm/md</strong> template.
    </div>
  </script>


  <!-- display template -->
  <div data-bind="template: {name: large() ? 'lg_xl_template' :
                                             'sm_md_template'}"></div>
</div>
```

The code that sets the value for `large()` is identical to the code used for setting component option changes. For details, see Change a Custom Element's Attribute Based on Screen Size.

For the complete code used in this example, see the Responsive Loading with JavaScript demo in the Oracle JET Cookbook.

Create Responsive Images

You can use the responsive JavaScript classes to load a different image when the screen size changes. For example, you may want to load a larger image when the screen size changes from small and medium to large and up.

In this example, the image is defined in a HTML `img` element. The markup uses Oracle JET's attribute binding to display a larger image when the `large()` call returns `true`.

```
<div id="container">
    <p>current width:
      <strong>
  <span>
    <oj-bind-text value="[[large() ? 'large' : 'not large']]"></oj-bind-text>
  </span>
      </strong>
    </p>

    <img alt="puzzle" id="puzzle"
 :src="[[large() ? 'images/responsive/puzzle.png' : 'images/responsive/
puzzle_small.png']]"
 />
</div>
```

The code that set the value for `large()` is identical to the code used for setting component option changes. For details, see Change a Custom Element's Attribute Based on Screen Size.

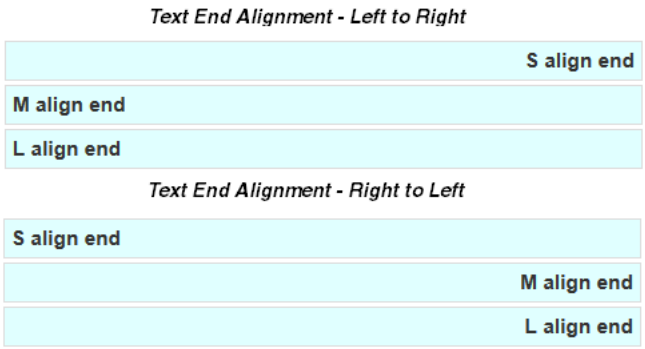
> **Note:**
>
> The image will not begin to load until the JavaScript is loaded. This could be an issue on devices with slower connections. If performance is an issue, you can use responsive CSS as described in Create Responsive CSS Images. You could also use the HTML `picture` element which supports responsive images without CSS or JavaScript. However, browser support is limited and may not be an option for your environment.

For the complete code used in this example, see the Oracle JET Cookbook Responsive Images demos.

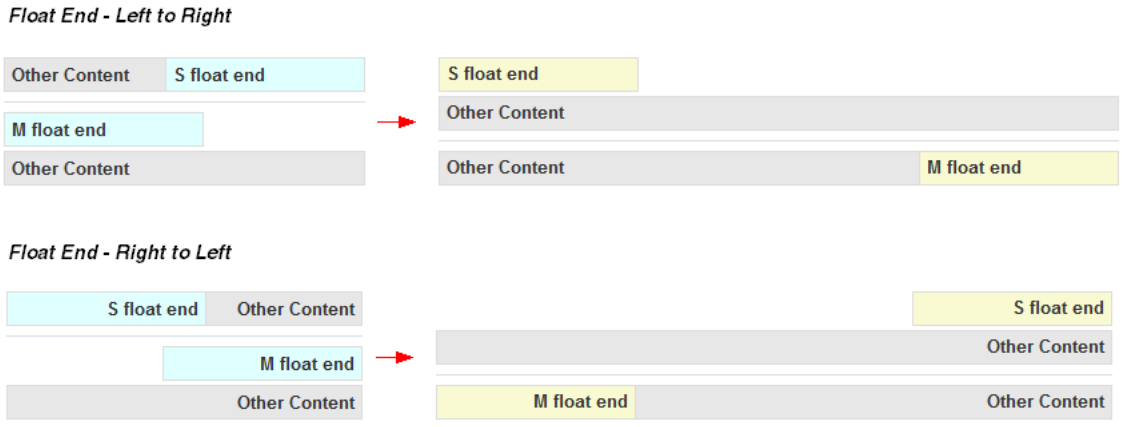
Use the Responsive Helper Classes

Use the Oracle JET generic responsive utility classes to hide content, end align text, and set float in your grid.

- `oj-size-hide`: Hide content at the specified *size*.
- `oj-size-text-align-end`: In left-to-right languages, set `text-align` to right. In right-to-left languages, set `text-align` to left.



- `oj-size-float-end`: In left-to-right languages, set `float` to right. In right-to-left languages, set `float` to left.



- `oj-size-float-start`: In left-to-right languages, set `float` to left. In right-to-left languages, set `float` to right.

To see examples that implement the responsive helper classes, consult the Oracle JET Cookbook at Responsive Helpers.

Create Responsive CSS Images

Use CSS generated from Sass media query variables and responsive queries to use a different, larger image when the screen width changes from small to large.

The code below shows the markup that defines the image. In this example, `bulletlist` is a CSS class generated from the Sass responsive variables and media queries.

```
<div role="img" class="oj-icon bulletlist" title="bulleted list image"></div>
```

The following image shows the `bulletlist` CSS class. When the screen is small or medium size, the `icon_small.png` image loads. When the screen increases to large or larger, or to print, the `icon.png` loads instead.

```
.bulletlist {
  width: 24px;
  height: 24px; }
.bulletlist:before {
  content: url("images/hiResContrast/icon_small.png"); }
@media print and (orientation: landscape), screen and (min-width: 1024px) {
  .bulletlist {
    width: 48px;
    height: 48px; }
  .bulletlist:before {
    content: url("images/hiResContrast/icon.png"); } }
```

The Oracle JET Cookbook includes the Sass variables and queries used to generate the `bulletlist` CSS class. You can also find a Sass mixin that makes generating the CSS easier, but you are not required to use SCSS to create responsive CSS images.

In addition, the Oracle JET Cookbook includes examples that show high resolution images, sprites, button images, and more. For details, see Responsive CSS Images.

> **Note:**
>
> You can also use responsive JavaScript to change the images based on screen size. For details, see Create Responsive Images.

Change Default Font Size

By default, Oracle JET includes the Redwood theme, starting in JET release 9.0.0, that set a default font size of 1em (16px) on the root (`html`) element. This font size is optimized for

visibility and touchability on mobile devices, but you can customize the size as needed for your app.

Change Default Font Size Across the App

The browser's font size is defined in the Sass `$rootFontSize` variable and included in the generated CSS `html` class. You can use Sass to change the variable or override the generated CSS.

To change the browser default font size across your app, do *one* of the following:

- In a custom Sass settings file, modify the Sass `$rootFontSize` variable, and regenerate the CSS.
- In your app-specific CSS, override the font-size setting for the `html` class.

  For example, to set the browser's default font size to 12 pixels, add the following to your app-specific CSS:

```
html {
  font-size: 12px;
}
```

Change Default Font Size Based on Device Type

You can change the default font size based on device type by detecting the device type used to access the browser and then setting the appropriate style on the `html` element.

To change the browser default font size based the user's device type:

1. Use whatever means you like to detect that the browser is running on the specified device.

   For example, you may want to change the browser's default font size on a desktop device. Use your own code or a third party tool to detect the device type.
2. When your app detects that the user is accessing the browser with the specified device, on the `html` element in your markup, set `style="font-size: `*xx*`px`". Substitute the desired pixel size for *xx*.

   For example, to set the font size to 12 pixels when the app detects the specified device, add logic to your app to add the highlighted code to your markup.

```
<html style="font-size: 12px">
   ... contents omitted
</html>
```

   Perform this step before initializing components to avoid issues with some Oracle JET components that measure themselves.

Control the Size and Generation of the CSS

You can change the size of the CSS content automatically generated by Oracle JET so that unused classes or particular types of classes are compressed, removed, excluded, or not generated.

When you use the responsive framework classes, Oracle JET generates a large number of classes that you may not need. Here are some steps you can take to control the size and generation of the CSS.

- Use compression.

  The responsive classes are often repetitive and compress well. For details about compressing your CSS, see Optimize Performance of Oracle JET Apps.

- Remove unused classes.

  By default, Oracle JET generates responsive classes small, medium, large, and xlarge screens. If you know that your app will not use some of these classes, you can set the associated `$responsiveQuery*` variables to `none`.

  ```
  // If you don't want xlarge classes, you could set:
  $screenXlargeRange:         none;
  $responsiveQueryLargeOnly:  none;
  $responsiveQueryXlargeUp:   none;
  ```

- Exclude unused classes from the app layout, flex, grid, form layout, and responsive helper groups.

  You can use the following variables to exclude classes from these groups altogether:

  - `$includeAppLayoutClasses`
  - `$includeAppLayoutWebClasses`
  - `$includeFlexClasses`
  - `$includeGridClasses`
  - `$includeFormLayoutClasses`
  - `$includeResponsiveHelperClasses`

- Stop generation of a particular responsive helper class.

  For finer-grained control, there are additional variables that you can set to `false` to generation of a particular type of class.

Variable	Description
`$responsiveGenerateHide`	Generate hide classes like `.oj-md-hide`.
`$responsiveGenerateTextAlignEnd`	Generate text-align end classes like `.oj-md-text-align-end`.
`$responsiveGenerateFloatStart`	Generate float start classes like `.oj-md-float-start`.
`$responsiveGenerateFloatEnd`	Generate float end classes like `.oj-md-float-end`.

4

Use RequireJS for Modular Development

Oracle JET includes RequireJS, a third party JavaScript library that you can use in your app to load only the Oracle JET libraries you need. Using RequireJS, you can also implement lazy loading of modules or create JavaScript partitions that contain more than one module.

About Oracle JET and RequireJS

RequireJS is a JavaScript file and module loader that simplifies managing library references and is designed to improve the speed and quality of your code.

RequireJS implements the Asynchronous Module Definition (AMD) API which provides a mechanism for asynchronously loading a module and its dependencies.

Oracle JET's modular organization enables app developers to load a subset of needed features without having to execute `require()` calls for each referenced object. Each Oracle JET module represents one functional area of the toolkit, and it typically defines more than one JavaScript object.

You do not have to use RequireJS to reference Oracle JET libraries, but it is required if you plan to use Oracle JET's internationalization or data visualization components in your app. The Oracle JET download includes the RequireJS library, and it is used by default in the Oracle JET Starter Templates and Cookbook examples.

For more information about RequireJS, see http://requirejs.org.

About Oracle JET Module Organization

The Oracle JET modules are listed in the following table with description and usage tips. Use this table to determine which modules you must load in your app. Where your app can directly interact with a module API, the available objects that the module returns also appear in the table. Your app would typically call functions on the returned object or instantiate new objects via the constructor function. For more information about module loading in Oracle JET apps, see API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) - JET Module Loading Overview.

> **Note:**
>
> Certain functionality that had been previously available from the now deprecated `ojcore`, `ojvalidation-base`, `ojvalidation-datetime`, and `ojvalidation-number` modules is provided by refactored modules that return their own object. The table indicates which modules have been refactored. You should explicitly import needed modules in the dependency list of your require function to use its classes.

Oracle JET Module	Refactored	Available Objects	Description	When to Use

`ojs/ojmodel`	No	`Collection`
		`Events`
		`Model`
		`OAuth`
		`URLError`
			Classes of the JET Common Model	Other than `OAuth`, all Oracle JET Common Model classes should be replaced by the Oracle JET `RESTDataProvider` class.

`ojs/ojknockout-model`	No	`KnockoutUtils`	Utilities for integrating Oracle JET's Common Model into Knockout.js	Deprecated. Use the Oracle JET `RESTDataProvider` class.
`ojs/`***`ojcomponent`***	No	Varies by component	Oracle JET component modules. Examples include • `ojs/ojbutton` • `ojs/ojtoolbar` • `ojs/ojtabs` Most Oracle JET components have their own module with the same name in lowercase and without hyphens as shown above, except for the following components: • `oj-buttonset-*`: `ojs/ojbutton` • `oj-input-password`: `ojs/ojinputtext` • `oj-text-area`: `ojs/ojinputtext` • `oj-combobox-*`: `ojs/ojselectcombobox` • `oj-select-*`: `ojs/ojselectcombobox` • `oj-spark-chart`: `ojs/ojchart` • `oj-*-gauge`: `ojs/ojgauge`	Use component modules that correspond to any Oracle JET component in your app.
`ojs/ojknockout`	No	`ComponentBinding`	Oracle JET data binding for global attributes of any HTML element in the user interface	Use when your app includes HTML elements (JET custom elements included) with databound global attributes (ones that use the : (colon) prefix) or that use the [[..]]/{{..}} syntax (for global attributes of custom elements).
`ojs/ojcorerouter`	No	`urlParamAdapter`, `urlPathAdapter` `CoreRouter`, `CoreRouterState`	Class for managing routing in single page apps	Use if your single page app needs to manage routing.
`ojs/ojmodule`	No	`ModuleBinding`	Binding that implements navigation within a region of a single page app	Use if your single page app needs to manage navigation within a page region.

Oracle JET Module	Refactored	Available Objects	Description	When to Use
ojs/ojmodule-element	No	ModuleElementAnimation	Component that implements navigation within a region of a single page app	Use if your single page app needs to manage navigation within a page region.
ojs/ojmoduleanimations	No	ModuleAnimations	Used in conjunction with ojs/ojmodule-element. Adds animation support via CSS animation effects.	Use if your app adds animation effects.
ojs/ojcontext	Yes, from ojcore	BusyContext, Context	Class that exposes the BusyContext that keeps track of components that are currently animating or fetching data.	Use if your app needs to query the busy state of components on the page.
ojs/ojconfig	Yes, from ojcore	Config	Class for setting and retrieving configuration options.	Use if your app needs to set or retrieve app configuration details.
ojs/ojlogger	Yes, from ojcore	Logger	Utilities for setting up a logger and collecting logging information	Use if your app needs to define logger callback functions.
ojs/ojresponsiveutils	Yes, from ojcore	ResponsiveUtils	Utilities for working with responsive screen widths and ranges. Often used in conjunction with ojs/ojresponsiveknockoututils to create knockout observables that can be used to drive responsive page behavior.	Use if your app needs to work with responsive page design.
ojs/ojthemeutils	Yes, from ojcore	ThemeUtils	Utilities for getting theme information	Use if your app needs to know about the theme's fonts or the target platform.
ojs/ojtimeutils	Yes, from ojcore	TimeUtils	Utilities for time information	Use if your app needs to calculate the position of a given time point within a range.
ojs/ojtranslation	Yes, from ojcore	Translations	Service for retrieving translated resources	Use if your app needs to work with translated resources.
ojs/ojattributegrouphandler	No	AttributeGroupHandler, ColorAttributeGroupHandler, ShapeAttributeGroupHandler	Classes for managing attribute groups	Use if your app needs to generate attribute values from data set values (key value pairs).
ojs/ojknockouttemplateutils	No	KnockoutTemplateUtils	Utilities for converting Knockout templates to a renderer function that can be used in JET component renderer APIs	Use if your app needs to work with Knockout templates.
ojs/ojresponsiveknockoututils	No	ResponsiveKnockoutUtils	Utilities for creating Knockout observables to implement responsive page design	Use if your app needs to create observables for responsive page design.

Oracle JET Module	Refactored	Available Objects	Description	When to Use
ojs/ojswipetoreveal	No	SwipeToRevealUtils	Utilities for setting up and handling swipe to reveal on an offcanvas element	Use if your app needs to support the swipe gesture.
ojs/ojkeyset	No	KeySet, KeySetImpl, AllKeySetImpl	Class for working with selection items in ojTable, ojListView, and ojDataGrid components	Use if your app needs to work with selections as a set.
ojs/ojdiagram-utils	No	DiagramUtils	Utilities for working with a JSON object to support the ojDiagram component	Use if your app creates an ojDiagram component from a JSON object.
ojs/ojoffcanvas	No	OffcanvasUtils	Class for controlling off-canvas regions	Use if your app needs to manage off-canvas regions.
ojs/ojcube	No	Cube, CubeAggType, CubeAxis, CubeAxisValue, CubeCellSet, CubeDataValue, CubeHeaderSet, CubeLevel, DataColumnCube, DataValueAttributeCube	Classes for aggregating data values in ojDataGrid	Use if your app renders aggregated cubic data in an ojDataGrid component.
ojs/oj***type***dataprovider	No	ArrayDataProvider, ArrayTreeDataProvider, CollectionDataProvider, DeferredDataProvider, FlattenedTreeDataProvider, IndexerModelTreeDataProvider, ListDataProviderView, PagingDataProviderView, TreeDataProviderView FilterFactory	Data provider modules. Examples include: • ojs/ojarraydataprovider • ojs/ojcollectiondataprovider • ojs/ojtreedataprovider	Use if your app includes an Oracle JET component, and its data source is defined in one of the *DataProvider classes.
ojs/ojtimezonedata	No	no public class available	Time zone data	Use if you want to add time zone support to oj-input-date-time, oj-input-time, or converters.
ojconverter-color, ojconverter-datetime, ojconverter-number	Yes, from ojvalidation-base, ojvalidation-datetime, or ojvalidation-number	ColorConverter, converterDateTime, converterColor	Color, date, and time conversion services.	Use if your app needs to support conversion services.

Oracle JET Module	Refactored	Available Objects	Description	When to Use
ojvalidator-daterestriction, ojvalidator-datetimerange, ojvalidator-length, ojvalidator-numberrange, ojvalidator-regexp, ojvalidator-required	Yes, from ojvalidation-base, ojvalidation-datetime, or ojvalidation-number	DateRestrictionValidator, DateTimeRangeValidator, LengthValidator, NumberRangeValidator, RegExpValidator, RequiredValidator	Date and number validation services.	Use if your app needs to support validation services.
ojasyncvalidator-daterestriction, ojasyncvalidator-datetimerange, ojasyncvalidator-length, ojasyncvalidator-numberrange, ojasyncvalidator-regexp, ojasyncvalidator-required	Yes, from ojvalidation-base, ojvalidation-datetime, or ojvalidation-number	AsyncDateRestrictionValidator, AsyncDateTimeRangeValidator, AsyncLengthValidator, AsyncNumberRangeValidator, AsyncRegExpValidator,	Async date and number validation services.	Use if your app needs to support async validation services.

About RequireJS in an Oracle JET App

Oracle JET includes the RequireJS library and sample bootstrap file in the Oracle JET download.

The code below shows excerpts of the `main-template.js` bootstrap file distributed with the Oracle JET base distribution in the `appRootDir\node_modules\@oracle\oraclejet\dist\js\libs\oj\` directory. Typically, you place the bootstrap file in your app's `js` directory and rename it to `main.js` . The comments in the code describe the purpose of each section. The sections that you normally edit are highlighted in bold.

```
/**
 * Example of Require.js boostrap javascript
 */
(function () {
    requirejs.config({
      // Path mappings for the logical module names
      paths: {

      },

      // This section configures the i18n plugin. It is merging the Oracle
JET built-in translation
      // resources with a custom translation file.
      // Any resource file added, must be placed under a directory named
"nls". You can use
      // a path mapping or you can define a path that is relative to the
location
      // of this main.js file.
      config: {
        ojL10n: {
```

```
          merge: {
            // 'ojtranslations/nls/ojtranslations': 'resources/nls/
myTranslations'
          }
        },
        text: {
          // Override for the requirejs text plugin XHR call for loading text
          // resources on CORS configured servers
          // eslint-disable-next-line no-unused-vars
          useXhr: function (url, protocol, hostname, port) {
            // Override function for determining if XHR should be used.
            // content omitted for brevity
            // Return true or false. true means "use xhr", false
            // means "fetch the .js version of this resource".
            return true;
          }
        }
      }
    });
  }());

  /**
   * A top-level require call executed by the app.
   * Although 'ojcore' and 'knockout' would be loaded in any case (they are
specified as dependencies
   * by the modules themselves), we are listing them explicitly to get the
references to the 'oj' and 'ko'
   * objects in the callback.
   *
   * For a listing of which JET component modules are required for each
component, see the specific component
   * demo pages in the JET cookbook.
   */
  require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/
ojbutton', 'ojs/ojtoolbar', 'ojs/ojmenu'],
     // add additional JET component modules as needed
    // eslint-disable-next-line no-unused-vars
    function (oj, ko, $) { // this callback gets executed when all required
modules are loaded
        // add any startup code that you want here
    }
  );
```

You can use RequireJS in a regular app as shown, or you can use RequireJS with `oj-module` element to define view templates and viewModels for page sections in a single-page app. For example, the Oracle JET Starter Templates use the `oj-module` element with RequireJS to use a different view and viewModel when the user clicks one of the navigation buttons.

For additional information about the Oracle JET Starter Templates, see About the Starter Templates. For more information about using `ojModule` and templates, see Create Single-Page Apps.

Use RequireJS in an Oracle JET App

To use RequireJS in your app, edit the bootstrap file to add the Oracle JET modules you need. You can also add your own modules as needed for your app code.

If needed, install Oracle JET and install RequireJS at http://requirejs.org.

To use RequireJS in an Oracle JET app:

1. In the bootstrap file or your app scripts, in the `require()` definition, add additional Oracle JET modules as needed.
2. Add any scripts that your app uses to the `require()` definition and update the `function(ko)` definition to include the script.
3. Add any app startup code to the callback function.
4. If your app includes resource bundles, enter the path to the bundle in the `merge` section.

Here's an example of the steps in order.

```
oj-dialog

require(['knockout', 'ojs/ojmodel', 'ojs/ojknockout-model','ojs/ojdialog'],
  function(ko) // obtaining a reference to the oj namespace
  {
  }
);
```

Then, to use a script named `myapp.js`, add the highlighted code to your `require()` definition.

```
require(['myapp', 'knockout', 'ojs/ojmodel', 'ojs/ojknockout-model', 'ojs/ojdialog'],
  function(myapp, ko) // obtaining a reference to the oj namespace
  {
  }
);
```

Next, you have a Knockout binding call for an element named `dialogWrapper`. Add that to the callback function.

```
require(['myapp', 'knockout', 'ojs/ojmodel', 'ojs/ojknockout-model', 'ojs/ojdialog'],
  function(myapp, ko) // obtaining a reference to the oj namespace
  {
    ko.applyBindings(new app()/*View Model instance*/,
                     document.getElementById('dialogWrapper'));
  }
);
```

Finally, you have a translations bundle, which you add to the `merge` section.

```
config: {
  ojL10n: {
    merge: {
      'ojtranslations/nls/ojtranslations': 'resources/nls/myTranslations'
    }
  }
}
```

For more information about module loading in Oracle JET apps, see API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) - JET Module Loading Overview.

Add Third-Party Tools or Libraries to Your Oracle JET App

You can add third-party tools or libraries to your Oracle JET app. The steps to do this vary depending on the method you used to create your app.

If you used the Oracle JET command-line tooling to scaffold your app, you install the library and make modifications to `appRootDir/src/js/path_mapping.json`. If you created your app using any other method and are using RequireJS, you add the library to your app and update the RequireJS bootstrap file, typically `main.js`.

> **Note:**
>
> This process is provided as a convenience for Oracle JET developers. Oracle JET does not support the additional tools or libraries and cannot guarantee that they will work correctly with other Oracle JET components or toolkit features.

To add a third-party tool or library to your Oracle JET app, complete one of the following procedures.

- If you created your app with command-line tooling, perform the following steps:
  1. In your app's root directory, enter the following command in a terminal window to install the library using NPM:

     ```
     npm install library-name --save
     ```

  2. Add the new library to the path mapping configuration file.
     a. Open `appRootDir/src/js/path_mapping.json` for editing.

        A portion of the file is shown below.

        ```
        {
         "baseUrl": "js",
          "use": "local",
          "cdns": {
            "jet": "https://static.oracle.com/cdn/jet/18.1.0/default/js",
            "css": "https://static.oracle.com/cdn/jet/18.1.0/default/css",
            "config": "bundles-config.js"
          },
          "3rdparty": "https://static.oracle.com/cdn/jet/18.1.0/3rdparty"
        },
        "libs": {
          "knockout": {
              ...
          },
        ...
        ```

        Oracle JET's CLI helps manage third-party libraries for your app by adding entries to its local `path_mapping.json` file.

        For each library listed in the `libs` map in the `path_mapping.json` file, the following two actions happen at build time. First, one or more files are copied from

somewhere (usually `appRootDir/node_modules`) into the `appRootDir/web` output folder. Second, a `requireJS` path value is created for you to represent the path to the library; it is injected into the built `main.js` file or optimized bundle. If you use TypeScript, then this should be the same path name that you use at design time for imports from the library.

The following example is an existing path mapping entry for the `persist` library, which we analyze below to describe what each attribute does and how it relates to the two build-time actions defined above.

```
1  | "persist": {
2  |   "cdn": "3rdparty",
3  |   "cwd": "node_modules/@oracle/oraclejet/dist/js/libs/persist",
4  |   "debug": {
5  |     "cwd": "debug",
6  |     "src": [
7  |       "**"
8  |     ],
9  |     "path": "libs/persist/debug",
10 |     "cdnPath": "persist/debug"
11 |   },
12 |   "release": {
13 |     "cwd": "min",
14 |     "src": [
15 |       "**"
16 |     ],
17 |     "path": "libs/persist/min",
18 |     "cdnPath": "persist/min"
19 |   }
20 | },
```

- Line 1: This is the name of the `libs` map entry ("`persist`", in this case). This name is used as the name of the `requireJS` path entry that is created; therefore, it is also the import path. The same name is used for the folder that gets created under the `appRootDir/web/js/libs` folder in the built application.

  > **Note:**
  >
  > If your app uses TypeScript, then you'll also need a `tsconfig.json` file `paths` entry with the same name.

- Line 2 (Optional): The `cdn` attribute sets the name of the CDN root for the location of this library. In this case, it is set to the value `3rdparty`, which maps to a particular location in the Oracle CDN. Any third-party libraries that you add yourself (that is, those that are not present on the Oracle CDN) should not have a `cdn` or `cdnPath` value at all, unless you maintain your own CDN infrastructure where you can place the library.
- Line 3: The first role of each path mapping entry at build time is to copy files. The `cwd` attribute on this line tells the tooling where to start copying from, and subsequent paths are relative to this root. This is generally some folder under `appRootDir/node_modules/<library>`.
- Lines 4 through 12: The `debug` and `release` sections here are the same, with the exception that if you do a normal build, then the specifications in the `debug`

section are followed, and if you do a `--release` build, then the specifications in the `release` section are followed. For example, the latter case can allow the copying of only optimized assets for release.

- Line 5 (Optional): In the context of the debug or release build, this second `cwd` attribute further refines the copy root for everything that follows. For example, when running a normal (debug) build for the `persist` library, the copy root is `appRootDir/node_modules/@oracle/oraclejet/dist/js/libs/persist/debug`, where the debug `cwd` value is appended to the top-level `cwd` value. Additionally, this folder name is used in the output; the copied files end up in the `appRootDir/web/js/libs/persist/debug` directory. This attribute is optional; if omitted, then all copies are copied from paths relative to the root `cwd` location and placed in the root of the destination folder.
- Line 6: The `src` attribute holds an array of all the files that you want to copy from your root location into the built app (that is, all the files that are actually needed at runtime). The paths you add here can either be specific file names, or you can use globs to match multiple files at once. In the example above, the glob `**` is used, which results in copying everything from the `/debug` folder, including subfolders. Folder structure is preserved in the copied output under the `/web` directory.
- Line 9: The `path` attribute is used for the second role performed by each path mapping entry; it defines the value of the `requireJS` path that gets injected into the `main.js` file. For the above example, the `requireJS` path mapping is `"persist":"libs/persist/min"`, which is relative to the `requireJS` load root, typically `appRootDir/web/js`.
- Line 10 (Optional): The `cdnPath` attribute is optional and should only be used if you have actually put a copy of the library onto a CDN. If that is the case, then if the `path_mapping.json` file has the `use` attribute set to `"cdn"` rather than `"local"`, then the generated `requireJS` path statement uses this `cdnPath` value rather than the `path` value (line 9). The path here is relative to the CDN root defined by the alias allocated to the CDN and used in the `cdn` attribute for that library. If `"use"` is set to `"cdn"` but a library does not include CDN information, then the build falls back to using the local copy of the library and set up the `requireJS` path accordingly.

b. Copy one of the existing library entries in the `"libs"` map and modify as needed for your library.

The code sample below shows modifications for `my-library`, a library that contains both minified and debug versions.

```
...
 "libs": {
    "my-library": {
      "cwd": "node_modules/my-library/dist",
      "debug": {
        "src": "my-library.debug.js",
        "path": "libs/my-library/my-library.debug.js"
      },
      "release": {
        "src": "my-library.js",
        "path": "libs/my-library/my-library.js"
      }
```

```
    },
...
```

In this example, the `cwd` attribute points to the location where NPM installed the library, the `src` attribute points to a path or array of paths containing the files that are copied during a build, and the `path` attribute points to the destination that contains the built version.

When defining your own library entry in the `path_mapping.json` file, you should test it out by running the `ojet build` command and then confirm that all the expected files have been copied into the `appRootDir/web/js/libs` directory and that the `requireJS` path mapping has been injected in the built `appRootDir/web/js/main.js` file.

> **Note:**
>
> If the existing library entry that you copy to modify includes `"cdn": "3rdparty",` remove that line from the newly-created entry for your library. This line references the Oracle JET third-party area on the content distribution network (CDN) managed by Oracle. Your library won't be hosted there, and keeping this line causes a release build to fail at runtime by mapping your library's path to a non-existent URL.
>
> If you use a CDN, add the URL to the CDN in the entry for the `cdnPath` attribute. For information on working with libraries loaded from CDNs, see Understand the Path Mapping Script File and Configuration Options.

3. If your project uses TypeScript, then you also need to define a `paths` entry in your `tsconfig.json` file that allows you to import the library using the same path name at design time as you'll use at runtime. If the library in question also provides some types for you to use, then the path should point to these to allow your editor to provide TypeScript support for your usage of that library. Here is an example of an existing `paths` entry in an Oracle JET app created with the command-line tooling.

```
"paths": {
  "ojs/*": [
    "./node_modules/@oracle/oraclejet/dist/types/*"
],
...
```

- If you didn't use command-line tooling to create your app, perform the following steps to add the tool or library.

1. In the app's `js/libs` directory, create a new directory and add the new library and any accompanying files to it.

   For example, for a library named `my-library`, create the `my-library` directory and add the `my-library.js` file and any needed files to it. Be sure to add the minified version if available.

2. In your RequireJS bootstrap file, typically `main.js`, add a path for the library file in the path mapping section of the `requirejs.config()` definition.

For example, add the highlighted code below to your bootstrap file to use a library named `my-library`.

```
requirejs.config({
  // Path mappings for the logical module names
  paths:
  {
    'knockout': 'libs/knockout/knockout-3.x.x',
    'jquery': 'libs/jquery/jquery-3.x.x.min',
    ...
    'text': 'libs/require/text',
    'my-library': 'libs/my-library/my-library
  },
  require(['knockout', 'my-library'],
  // this callback gets executed when all
  // required modules are loaded
  function(ko)
  {
      // Add any start-up code that you want here
  }
);
```

For additional information about using RequireJS to manage your app's modules, see Use RequireJS for Modular Development.

3. If your project uses TypeScript, then you also need to define a `paths` entry in your `tsconfig.json` file that allows you to import the library using the same path name at design time as you'll use at runtime. If the library in question also provides some types for you to use, then the path should point to these to allow your editor to provide TypeScript support for your usage of that library. Here is an example of an existing `paths` entry in an Oracle JET app created with the command-line tooling.

```
"paths": {
  "ojs/*": [
    "./node_modules/@oracle/oraclejet/dist/types/*"
 ],
...
```

Troubleshoot the Addition of Third-Party Tools and Libraries

In most cases, when adding a third-party tool or library to your Oracle JET app, failures manifest in one of three ways:

1. A 404 error in your browser-tools network panel.
   If this occurs, you may have omitted a file from the set that you copied, or the library that you have selected has downstream dependencies and you need to define additional third-party libraries.
2. An error dump on the browser console.
   This is often an indication that the library is not in the correct format; see the following list of common problems for more information.
3. Your usage of the library fails.

When diagnosing integration issues with third-party libraries, the following problems are frequently encountered:

- The library is not a browser library.
  Sometimes developers choose to use libraries that are intended for use in Node.js, rather than in a browser. Read the library's documentation to ensure that it is a browser library.
- The library is in the wrong format.
  There are multiple library formats in the JavaScript ecosystem, some of which are intended for use in browsers and others that are not. You may be unable to load a library that you found because it is in an incorrect format.

  Check that your path mapping entry uses the correct distribution of the library out of its possible format options. For runtime use, you need modules that are in either AMD or UMD format. Alternatively, you can convert from one module format to another using external tools.
- The library has runtime dependencies.
  The library might require you to include multiple dependencies into your `path_mapping.json` file so that all the required paths and modules are available at runtime. When loading a library at runtime, a 404 error is often a result of a missing downstream dependency.

  Read the library's documentation, examine the `package.json` file for the library itself, and review the 404 error in detail to help figure out the error.
- The library expects specific RequireJS path names.
  The library that you are consuming may expect to be loaded from a specific path name, or it may expect a dependency to be set up in the same way. This exhibits a 404 error when trying to load a resource.

  From the path submitted in the GET request, you should be able to figure out what is going on. Remember that you can configure the `path_mapping.json` file with the name of the path that must be created, so you should be able to correct this in the metadata to get everything working.
- The library needs a script tag to load code.

  Most libraries are compatible with a module loading system and, assuming they are AMD compatible, will work with an import statement or `define()` and `require()` methods. However, in some cases, you need to use a `<script>` tag in your page to load the code, rather than have a module loader do it for you.

  A case where you may need to use a `<script>` tag is when the JavaScript library that you want to use is only available in ESM format. If you don't tranform it to AMD format, you import it into the root level of your project (the `appRootDir/src/index.html` file) using a `<script>` tag.

  Be cautious with code that requires a `<script>` tag like this, as it is probably likely to interact with the global JavaScript context in a way that is incompatible with modern module development. In cases like this, the library's documentation should help point you in the right direction.

Troubleshoot RequireJS in an Oracle JET App

RequireJS issues are often related to modules used but not defined.

Use the following tips when troubleshooting issues with your Oracle JET app that you suspect may be due to RequireJS:

- Check the JavaScript console for errors and warnings. If a certain object in the `oj` namespace is undefined, locate the module that contains it based on the information in

About Oracle JET Module Organization or the Oracle JET Cookbook and add it to your app.

- If the components you specified using Knockout.js binding are not displayed and you are not seeing any errors or warnings, verify that you have added the `ojs/ojknockout` module to your app.

About JavaScript Partitions and RequireJS in an Oracle JET App

RequireJS supports JavaScript partitions that contain more than one module.

You must name all modules using the RequireJS `bundles` option and supply a path mapping with the configuration options.

```
requirejs.config(
  {
    bundles:
    {
      'commonComponents': ['ojL10n', 'ojtranslations/nls/ojtranslations',
                           'ojs/ojknockout', 'ojs/ojcomponentcore',
                           'ojs/ojbutton', 'ojs/ojpopup'],
      'tabs': ['ojs/ojtabs', 'ojs/ojconveyorbelt']    }
  }
);
```

In this example, two partition bundles are defined: `commonComponents` and `tabs`.

RequireJS ships with its own Optimizer tool for creating partitions and minifying JavaScript code. The tool is designed to be used at build time with a complete project that is already configured to use RequireJS. It analyzes all static dependencies and creates partitions out of modules that are always loaded together. The Oracle JET team recommends that you use an optimizer to minimize the number of HTTP requests needed to download the modules.

For additional information about the RequireJS Optimizer tool, see http://requirejs.org/docs/optimization.html.

For additional information about optimizing performance of Oracle JET apps, see Optimize Performance of Oracle JET Apps.

5

Create Single-Page Apps

Oracle JET includes the `oj-module` component and `CoreRouter` framework class that you can use to create single-page apps that simulate the look and feel of desktop apps.

Design Single-Page Apps Using Oracle JET

Oracle JET includes Knockout for separating the model layer from the view layer and managing the interaction between them. Using Knockout, the Oracle JET `oj-module` component, and the Oracle JET `CoreRouter` framework class, you can create single-page apps that look and feel like a standalone desktop app.

Understand Oracle JET Support for Single-Page Apps

Single-page apps (SPAs) are typically used to simulate the look and feel of a standalone desktop app. Rather than using multiple web pages with links between them for navigation, the app uses a single web page that is loaded only once. If the page changes because of the user's interaction, only the portion of the page that changed is redrawn.

Oracle JET includes support for single page apps using the `CoreRouter` class for virtual navigation in the page, the `oj-module` component for managing view templates and viewModel scripts, and Knockout for separating the model layer from the view layer and managing the binding between them. In the Oracle JET Cookbook, you can view a number of implementations that use the `CoreRouter` class. These implementations range from the simple, that switch tabs, to more complex examples that use parameters and child routers. See the CoreRouter demo in the Oracle JET Cookbook.

When routing a single-page app, the page doesn't reload from scratch but the content of the page changes dynamically. In order to be part of the browser history and provide bookmarkable content, the Oracle JET `CoreRouter` class emulates the act of navigating using the HTML5 history push state feature. The `CoreRouter` also controls the URL to look like traditional page URLs. However, there are no resources at those URLs, and you must set up the HTML server. This is done using a simple rule for a rewrite engine, like mod rewrite module for Apache HTTP server or a rewrite filter like UrlRewriteFilter for servlets.

In general, use query parameters when your app contains only a few views that the user will bookmark and that are not associated with a complex state. Use path segments to display simpler URLs, especially for nested paths such as `customers/cust/orders`.

The Oracle JET Cookbook uses the Oracle JET `oj-module` feature to manage the Knockout binding. With `oj-module`, you can store your HTML content for a page section in an HTML fragment or template file and the JavaScript functions that contain your viewModel in a viewModel file.

When `oj-module` and `CoreRouter` are used in conjunction, you can configure an `oj-module` object where the module name is the router state. When the router changes state, `oj-module` will automatically load and render the content of the module name specified in the value of the current `RouterState` object.

Create a Single-Page App in Oracle JET

The Oracle JET Cookbook includes complete examples and recipes for creating a single-page app using path segments and query parameters for routing and examples that use routing with the `oj-module` component. Regardless of the routing method you use, the process to create the app is similar.

To create a single-page app in Oracle JET:

If needed, create the app that will house your main HTML5 page and supporting JavaScript. For additional information, see Understand the Web App Workflow.

1. Design the app's structure and identify the templates and ViewModels that your app will require.
2. Add code to your app's main script that defines the states that the router can take, and add the `ojs/ojcorerouter` module to your `require()` list.
3. Add code to the markup that triggers the state transition and displays the content of the current state.

   When the user clicks one of the buttons in the header, the content is loaded according to the router's current state.

   For additional information about creating templates and ViewModels, see Use the oj-module Element.
4. To manage routing within a module, add a child router using `CoreRouter.createChildRouter()`.
5. Add any remaining code needed to complete the content or display.

See the CoreRouter demo in the Oracle JET Cookbook that implements `CoreRouter` and provides a link to the API documentation.

Use the oj-module Element

With the `oj-module` element, you can store your HTML content for a page section in an HTML fragment or template file and the JavaScript functions that contain your viewModel in a viewModel file.

Many of the Oracle JET Cookbook and sample apps use `oj-module` to manage the Knockout binding.

To use `oj-module` in your Oracle JET app:

If needed, create the app that will house your main HTML5 page and supporting JavaScript. See Understand the Web App Workflow. Oracle JET apps are built with default views and viewModels folders under *`app_folder`*`/src/js`.

1. In your RequireJS bootstrap file (typically `main.js`) add `ojs/ojmodule-element` to the list of RequireJS modules, along with any other modules that your app uses.

```
require(['knockout', 'ojs/ojmodule-element-utils', 'ojs/ojcorerouter',
'ojs/ojlogger',
  'ojs/ojresponsiveknockoututils', 'ojs/ojarraydataprovider', 'ojs/
ojoffcanvas',
  'ojs/ojknockouttemplateutils', 'ojs/ojmodule-element', 'ojs/ojknockout',
```

```
'ojs/ojbutton',
  'ojs/ojmenu', 'ojs/ojcheckboxset', 'ojs/ojswitch']
```

2. Create your view templates and add them to the `views` folder as the default location.
3. Create your viewModel scripts and add them to the `viewModels` folder as the default location.
4. Add code to the app's HTML page to reference the view template or viewModel in the `oj-module` element. To obtain the router configuration, set the `config` attribute of the `oj-module` element to the `koObservableConfig` observable created by the `ModuleRouterAdapter`.

```
<oj-module role="main" class="oj-panel" style="padding-bottom:30px"
           config="[[moduleAdapter.koObservableConfig]]"></oj-module>
```

For more information about `CoreRouter` and `oj-module`, see the Oracle JET CoreRouter and oj-module API documentation.

> **Tip:**
>
> `oj-module` is not specific to single-page apps, and you can also use it to reuse content in multi-page apps. However, if you plan to reuse or share your content across multiple apps, consider creating Oracle JET Web Components instead. Web Components are reusable components that follow the HTML5 Web Component specification. They have the following benefits:
>
> - Web Components have a contract. The API for a Web Component is well defined by its `component.json`, which describes its supported properties, methods, and events in a standard, universal, and self-documenting way. Providing a standardized contract makes it easier for external tools or other apps to consume these components.
> - Web Components include version and dependency metadata, making it clear which versions of Oracle JET they support and what other components they may require for operation.
> - Web Components are self-contained. A Web Component definition can contain all the libraries, styles, images, and translations that it needs to work.
>
> To learn more about Web Component features, see Work with Oracle JET Web Components.

Work with oj-module's ViewModel Lifecycle

The `oj-module` element provides lifecycle listeners that allow you to specify actions to take place at defined places in the ViewModel's lifecycle.

For example, you can specify actions to take place when the ViewModel is about to be used for the View transition, after its associated View is inserted into the document DOM, and after its View and ViewModel are inactive.

The following table lists the available methods with a description of their usage.

Method Name	Description
`connected()`	The optional method will be invoked after the View is inserted into the DOM. This method might be called multiple times: • after the View is created and inserted into the DOM • after the View is reconnected after being disconnected • after a parent element (oj-module) with attached View is reconnected to the DOM
`transitionCompleted()`	This optional method will be invoked after transition to the new View is complete. This includes any possible animation between the old and the new View.
`disconnected()`	This optional method will be invoked when the View is disconnected from the DOM. This method might be called multiple times: • after the View is disconnected from the DOM • after a parent element (oj-module) with attached View is disconnected from the DOM

You can also find stub methods for using the `oj-module` lifecycle methods in some of the Oracle JET templates. For example, the `navbar` template, available as a template when you Scaffold a Web App, defines stub methods for `connected()`, `disconnected()`, and `transitionCompleted()`. Comments describe the expected parameters and use cases.

- TypeScript
- JavaScript

TypeScript

```
import * as AccUtils from "../accUtils";
class DashboardViewModel {

  constructor() {

  }

  /**
   * Optional ViewModel method invoked after the View is inserted into the
   * document DOM.  The application can put logic that requires the DOM being
   * attached here.
   * This method might be called multiple times - after the View is created
   * and inserted into the DOM and after the View is reconnected
   * after being disconnected.
   */
  connected(): void {
    AccUtils.announce("Dashboard page loaded.");
    document.title = "Dashboard";
    // implement further logic if needed
  }

  /**
   * Optional ViewModel method invoked after the View is disconnected from
the DOM.
```

```
   */
  disconnected(): void {
    // implement if needed
  }

  /**
   * Optional ViewModel method invoked after transition to the new View is
complete.
   * That includes any possible animation between the old and the new View.
   */
  transitionCompleted(): void {
    // implement if needed
  }
}

export = DashboardViewModel;
```

JavaScript

```
/*
 * Your dashboard ViewModel code goes here
 */
define(['../accUtils'], function (accUtils) {
  function DashboardViewModel() {
    // Below are a set of the ViewModel methods invoked by the oj-module
component.
    // Please reference the oj-module jsDoc for additional information.

    /**
     * Optional ViewModel method invoked after the View is inserted into the
     * document DOM.  The application can put logic that requires the DOM
being
     * attached here.
     * This method might be called multiple times - after the View is created
     * and inserted into the DOM and after the View is reconnected
     * after being disconnected.
     */
    this.connected = () => {
      accUtils.announce('Dashboard page loaded.', 'assertive');
      document.title = 'Dashboard';
      // Implement further logic if needed
    };

    /**
     * Optional ViewModel method invoked after the View is disconnected from
the DOM.
     */
    this.disconnected = () => {
      // Implement if needed
    };

    /**
     * Optional ViewModel method invoked after transition to the new View is
complete.
     * That includes any possible animation between the old and the new View.
```

```
     */
    this.transitionCompleted = () => {
      // Implement if needed
    };
  }

  /*
   * Returns an instance of the ViewModel providing one instance of the
ViewModel. If needed,
   * return a constructor for the ViewModel so that the ViewModel is
constructed
   * each time the view is displayed.
   */
  return DashboardViewModel;
});
```

For more, see the oj-module API documentation.

6

Understand Oracle JET User Interface Basics

Oracle JET User Interface (UI) components extend the `htmlElement` prototype to implement the World Wide Web Consortium (W3C) web component specification for custom elements. Custom elements provide a more declarative way of working with Oracle JET components and allow you to access properties and methods directly on the DOM layer.

About the Oracle JET User Interface

Oracle JET includes components, patterns, and utilities to use in your app. The toolkit also includes an API specification (if applicable) and one or more code examples in the Oracle JET Cookbook.

Identify Oracle JET UI Components, Patterns, and Utilities

The Oracle JET Cookbook lists all the components, design patterns, and utilities available for your use. By default, the cookbook is organized into functional sections, but you can also click **Sort** to arrange the contents alphabetically.

The Cookbook contains samples that you can edit online and see the effects of your changes immediately. You'll also find links to the API documentation, if applicable.

About Common Functionality in Oracle JET Components

All Oracle JET components are implemented as custom HTML elements, and programmatic access to these components is similar to interacting with any HTML element.

Custom Element Structure

Oracle JET custom element names start with `oj-`, or `oj-c-` for the newer Core Pack components, and you can add them to your page the same way you would add any other HTML element. In the following example, the `oj-label` and `oj-input-date-time` custom elements are added as child elements to a standard HTML `div` element.

```
<div id="div1">
  <oj-label for="dateTime">Default</oj-label>
  <oj-input-date-time id="dateTime" value='{{value}}'>
  </oj-input-date-time>
</div>
```

Each custom element can contain one or more of the following:

- Attributes: Modifiers that affect the functionality of the element.

  String literals will be parsed and coerced to the property type. Oracle JET supports the following string literal type coercions: boolean, number, string, Object, Array, and any. The any type, if used by an element, is marked with an asterisk (*) in the element's API documentation and coerced to Objects, Arrays, or strings.

In the `oj-input-date-time` element defined above, `value` is an attribute that contains a Date object. It is defined using a binding expression that indicates whether the element's ViewModel should be updated if the attribute's value is updated.

```
<oj-input-date-time id="dateTime" value='{{value}}'>
```

The `{{...}}` syntax indicates that the element's `value` property will be updated in the element's ViewModel if it's changed. To prevent the attribute's value from updating the corresponding ViewModel, use the `[[...]]` syntax.

- Methods: Supported methods

  Each custom element's supported method is documented in its API.

- Events: Supported events

  Events specific to the custom element are documented in its API. Define the listener's method in the element's ViewModel.

```
var listener = function( event )
{
  // Check if this is the end of "inline-open" animation for inline message
  if (event.detail.action == "inline-open") {
    // Add any processing here
  }
};
```

  Reference the listener using the custom element's DOM attribute, JavaScript property, or the `addEventListener()`.

  – Use the DOM attribute.

    Declare a listener on the custom element using `on-event-name` syntax.

```
<oj-input-date-time on-oj-animate-start='[[listener]]'</oj-input-date-time>
```

    Note that in this example the listener is declared using the `[[...]]` syntax since its value is not expected to change.

  – Use the JavaScript property.

    Specify a listener in your ViewModel for the `.onEventName` property.

```
myInputDateTime.onOjAnimateEnd = listener
```

    Note that the JavaScript property uses camelCase for the `onOjAnimateEnd` property. The camelCased properties are mapped to attribute names by inserting a dash before the uppercase letter and converting that letter to lower case, for example, `on-oj-animate-end`.

  – Use the `addEventListener()` API.

```
myInputDateTime.addEventListener('ojAnimateEnd', listener);
```

  By default, JET components will also fire `propertyChanged` custom events whenever a property is updated, for example, `valueChanged`. You can define and add a listener using

any of the three methods above. When referencing a *property*`Changed` event declaratively, use `on-`*property*`-changed` syntax.

```
<oj-input-date-time value="{{currentValue}}" on-value-
changed="{{valueChangedListener}}" </oj-input-date-time>
```

- Slots

  Oracle JET elements can have two types of child content that determine the content's placement within the element.

  – Any child element with a supported slot attribute will be moved into that named slot. All supported named slots are documented in the element's API. Child elements with unsupported named slots will be removed from the DOM.

  ```
  <oj-table>
    <div slot='bottom'<oj-paging-control></oj-paging-control></div>
  </oj-table>
  ```

  – Any child element lacking a slot attribute will be moved to the default slot, also known as a regular child.

A custom element will be recognized only after its module is loaded by the app. Once the element is recognized, Oracle JET will register a busy state for the element and will begin the process of upgrading the element from a normal element to a custom element. The element will not be ready for interaction until the upgrade process is complete. The app should listen to either the page-level or element-scoped `BusyContext` before attempting to interact with any JET custom elements. However, property setting (but not property getting) is allowed before the `BusyContext` is initialized. See the BusyContext API documentation on how BusyContexts can be scoped.

The upgrade of custom elements relies on a binding provider which manages the data binding. The binding provider is responsible for setting and updating attribute expressions. Any custom elements within its managed subtree will not finish upgrading until the provider applies bindings on that subtree. By default, there is a single binding provider for a page, but subtree specific binding providers can be added by using the `data-oj-binding-provider` attribute with values of `none` and `knockout`. The default binding provider is `knockout`, but if a page or DOM subtree does not use any expression syntax or knockout, the app can set `data-oj-binding-provider="none"` on that element so that its dependent JET custom elements do not wait for bindings to be applied to finish upgrading.

Other Common Functionality

Oracle JET custom elements also have the following functionality in common:

- Context menus

  Custom elements support the `slot` attribute to add context menus to Oracle JET custom elements, described in each element's API documentation.

```
<oj-some-element>
  <-- use the contextMenu slot to designate this as the context menu for
this component -->
  <oj-menu slot="contextMenu" style="display:none" aria-label="Some
element's context menu"
    ...
  </oj-menu>
</oj-some-element>
```

- Keyboard navigation and other accessibility features

  Oracle JET components that support keyboard navigation list the end user information in their API documentation. For additional information about Oracle JET components and accessibility, see Develop Accessible Oracle JET Apps.

- Drag and drop

  Oracle JET includes support for standard HTML5 drag and drop and provides the `dnd-polyfill` library to extend HTML5 drag and drop behavior to supported mobile and desktop browsers. In addition, some Oracle JET custom elements such as `oj-table` support drag and drop behavior through the `dnd` attribute. For specific details, see the component's API documentation and cookbook examples. To learn more about HTML5 drag and drop, see http://www.w3schools.com/html/html5_draganddrop.asp.

- Deferred rendering

  Many Oracle JET custom elements support the ability to defer rendering until the content shown using `oj-defer`. To use `oj-defer`, wrap it around the custom element.

```
<oj-collapsible id="defer">
  <h4 id="hd" slot="header">Deferred Content</h4>
    <oj-defer>
      <oj-module config='[[deferredRendering/content]]'>
      </oj-module>
    </oj-defer>
</oj-collapsible>
```

  Add the deferred content to the app's view and ViewModel, `content.html` and `content.js`, as specified in the `oj-module` definition. For the complete code example, see Collapsibles - Deferred Rendering.

  For a list of custom elements that support `oj-defer`, see oj-defer.

Custom Element Examples and References

The Oracle JET Cookbook and API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) provide examples that illustrate how to work with custom elements. In addition, the Cookbook provides demos with editing capability that allow you to modify the sample code directly and view the results without having to download the sample.

To learn more about the World Wide Web Consortium (W3C) web component specification for custom elements, see Custom Elements.

About Oracle JET Reserved Namespaces and Prefixes

Oracle JET reserves the `oj` namespace and prefixes for the original set of UI components that predate the introduction of the Oracle JET Core Pack components in release 14.0.0. This includes, but is not limited to component names, namespaces, pseudo-selectors, public event prefixes, CSS styles, Knockout binding keys, and so on. Oracle JET also reserves the `oj-c` namespace for the newer Core Pack components.

About Binding and Control Flow

Oracle JET includes components and expressions to easily bind dynamic elements to a page in your app using Knockout.

Use oj-bind-text to Bind Text Nodes

Oracle JET supports binding text nodes to variables using the `oj-bind-text` element and by importing the `ojknockout` module.

The `oj-bind-text` element is removed from the DOM after binding is applied. For example, the following code sample shows an `oj-input-text` and an `oj-button` with a text node that are both bound to the `buttonLabel` variable. When the input text is updated, the button text is automatically updated as well.

```
<div id='button-container'>
   <oj-button id='button1'>
      <span><oj-bind-text value="[[buttonLabel]]"></oj-bind-text></span>
   </oj-button>
   <br><br>
   <oj-label for="text-input">Update Button Label:</oj-label>
   <oj-input-text id="text-input" value="{{buttonLabel}}"></oj-input-text>
</div>
```

The script to create the viewModel for this example is shown below.

- TypeScript
- JavaScript

TypeScript

```
import * as ko from "knockout";
import "ojs/ojinputtext";
import "ojs/ojlabel";
import "ojs/ojbutton";

class ButtonModel {
  buttonLabel: ko.Observable<string>;

  constructor() {
    this.buttonLabel = ko.observable("My Button");
  }
}

export = ButtonModel;
```

JavaScript

```
define(["knockout", "ojs/ojknockout", "ojs/ojbutton",
                       "ojs/ojinputtext", "ojs/ojlabel"],

  function (ko) {

    function ButtonModel() {
      this.buttonLabel = ko.observable("My Button");
```

```
    }

    return ButtonModel;

  });
```

The figure below shows the output for the code sample.

The Oracle JET Cookbook contains the complete example used in this section. See Text Binding.

Bind HTML Attributes

Oracle JET supports one-way attribute data binding for attributes on any HTML element by prefixing ":" to the attribute name and by importing the `ojknockout` module.

To use an HTML attribute in an HTML or JET element, prefix a colon to it. JET component–specific attributes do not need the prefix.

The following code sample shows two JET elements and two HTML elements that use both the prefixed and non-prefixed syntax. Since the label and input elements are native HTML elements, all their data bound attributes should use the colon prefixing. The `oj-label` and `oj-input-text` elements use the prefix only for native HTML element attributes and the non-prefixed syntax for component-specific attributes.

```
<div id="demo-container">
  <oj-label for="[[inputId1]]">oj-input-text element</oj-label>
  <oj-input-text :id="[[inputId1]]" value="{{value}}"></oj-input-text>
  <br><br>
  <label :for="[[inputId2]]">HTML input element</label>
  <br>
  <input :id="[[inputId2]]" :value="[[value]]" style="width:100%;max-width:18em"/>
</div>
```

The script to create the viewModel for this example is shown below.

- TypeScript
- JavaScript

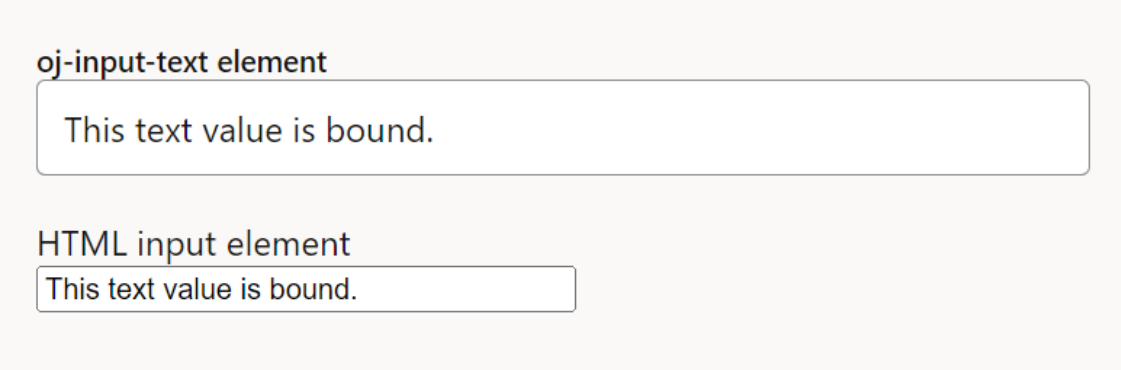
TypeScript

```
import * as ko from "knockout";
import "ojs/ojinputtext";
import "ojs/ojlabel";
import 'ojs/ojknockout';

class ViewModel {
  inputId1: ko.Observable<string>;
  inputId2: ko.Observable<string>;
  value: ko.Observable<string>;

  constructor() {
    this.inputId1 = ko.observable("text-input1");
    this.inputId2 = ko.observable("text-input2");
    this.value = ko.observable("This text value is bound.");
  }
}

export = ViewModel;
```

JavaScript

```
define(["knockout", "ojs/ojknockout", "ojs/ojinputtext", "ojs/ojlabel"],

  function (ko) {

    function ViewModel() {
      this.inputId1 = ko.observable("text-input1");
      this.inputId2 = ko.observable("text-input2");
      this.value = ko.observable("This text value is bound.");
    }

    return ViewModel;

  });
```

The figure below shows the output for the code sample.



The Oracle JET Cookbook contains the complete example used in this section. See Attribute Binding.

Use oj-bind-if to Process Conditionals

Oracle JET supports conditional rendering of elements by using the `oj-bind-if` element and importing the `ojknockout` module.

The `oj-bind-if` element is removed from the DOM after binding is applied, and must be wrapped in another element such as a `div` if it is used for slotting. The slot attribute has to be applied to the wrapper since `oj-bind-if` does not support it. For example, the following code sample shows an image that is conditionally rendered based on the option chosen in an `oj-buttonset`.

```
<div id="demo-container">
<oj-buttonset-one class="oj-buttonset-width-auto" value="{{buttonValue}}">
  <oj-option id="onOption" value="on">On</oj-option>
  <oj-option id="offOption" value="off">Off</oj-option>
</oj-buttonset-one>
<br><br>
<div>Image will be rendered if the button is on:</div>
```

```
<oj-bind-if test="[[buttonValue() === 'on']]">
  <oj-avatar role="img" aria-label="Avatar of Amy Bartlet" size="md"
initials="AB"
    src="../css/images/avatar-image.jpg" class="oj-avatar-image"></oj-avatar>
</oj-bind-if>
</div>
```

In the above example, the `oj-avatar` element is an icon which can display a custom or placeholder image. See oj-avatar.

The script to create the view model for this example is shown below.

- TypeScript
- JavaScript

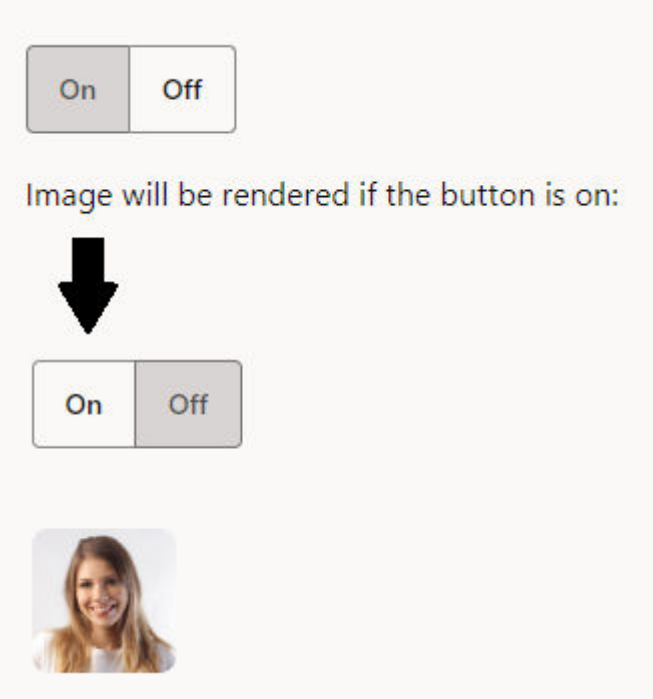
TypeScript

```
import * as ko from "knockout";
import "ojs/ojknockout";
import "ojs/ojbutton";
import "ojs/ojavatar";

class ViewModel {
  buttonValue = ko.observable("off");
}
export = ViewModel;
```

JavaScript

```
define(['knockout', 'ojs/ojknockout', 'ojs/ojbutton', 'ojs/ojavatar'],
  function (ko) {
    function ViewModel() {
      this.buttonValue = ko.observable("off");
    }
    return ViewModel;
  }
);
```

The figure below shows the output for the code sample. When the `oj-buttonset` is set to 'on', the `oj-avatar` element is rendered and displayed.



The Oracle JET Cookbook contains the complete example used in this section. See If Binding.

Use oj-bind-for-each to Process Loop Instructions

Oracle JET supports processing loop instructions, such as binding items from an array by using the `oj-bind-for-each` element and by importing the `ojknockout` module.

The `oj-bind-for-each` element only accepts a single `template` element as its direct child. Any markup to be duplicated, such as `li` tags, must be placed inside the `template` tag. For example, the following code sample shows an unordered list nested inside another unordered list. The list items are created using an `oj-bind-text` tag inside nested `oj-bind-for-each` elements.

```
<div id="form-container">
  <ul>
    <oj-bind-for-each data="[[categories]]">
      <template data-oj-as="category">
        <li>
          <ul>
            <oj-bind-for-each data="[[category.data.items]]">
              <template data-oj-as="item">
                <li>
                  <oj-bind-text value="[[category.data.name + ' : ' + item.data]]"></oj-
bind-text>
                </li>
              </template>
            </oj-bind-for-each>
          </ul>
        </li>
      </template>
    </oj-bind-for-each>
  </ul>
</div>
```

In the above example, the `data-oj-as` attribute provides an alias for the bound data. This alias is referenced in the nested `oj-bind-for-each` and `oj-bind-text` elements.

The script to create the viewModel for this example is shown below.

- TypeScript
- JavaScript

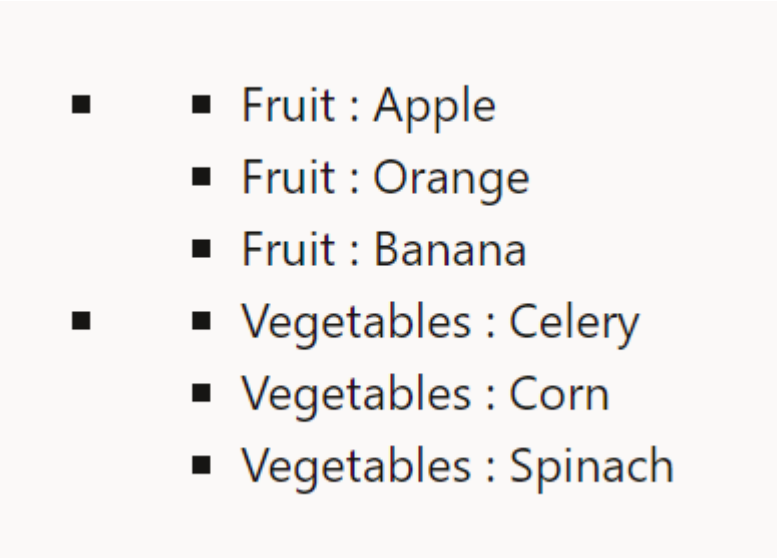
TypeScript

```
import 'ojs/ojknockout';
import "ojs/ojbutton";

class ViewModel {
  categories: Array<Object>;

  constructor() {
    this.categories =
      [{ name: "Fruit", items: ["Apple", "Orange", "Banana"] },
       { name: "Vegetables", items: ["Celery", "Corn", "Spinach"] }
      ];
  }
}

export = ViewModel;
```

JavaScript

```
define(["ojs/ojknockout", "ojs/ojbutton"],

  function () {

    function ViewModel() {

    this.categories =
      [{ name: "Fruit", items: ["Apple", "Orange", "Banana"] },
       { name: "Vegetables", items: ["Celery", "Corn", "Spinach"] }
      ];
    }

    return ViewModel;

  });
```

The figure below shows the output for the code sample.



The Oracle JET Cookbook contains the complete example used in this section. See Foreach Binding.

Bind Style Properties

The Oracle JET attribute binding syntax also supports style attributes, which can be passed as an object or set using dot notation. The `ojknockout` module must be imported.

The style attribute binding syntax accepts an object in which style properties should be referenced by their JavaScript names. Apps can also set style sub-properties using dot notation, which uses the CSS names for the properties. The code sample below shows two block elements with style attributes. The first element binds a `style` object, while the second binds properties directly to the defined style attributes.

```
<div id="demo-container">
  <div :style="[[style]]">Data bound style attribute</div>
  <br>
  <div :style.color="[[fontColor]]" :style.font-style="[[fontStyle]]">Data bound style
using dot notation</div>
</div>
```

The script to create the viewModel for this example is shown below. The `style` object referenced above is highlighted below.

- TypeScript
- JavaScript

TypeScript

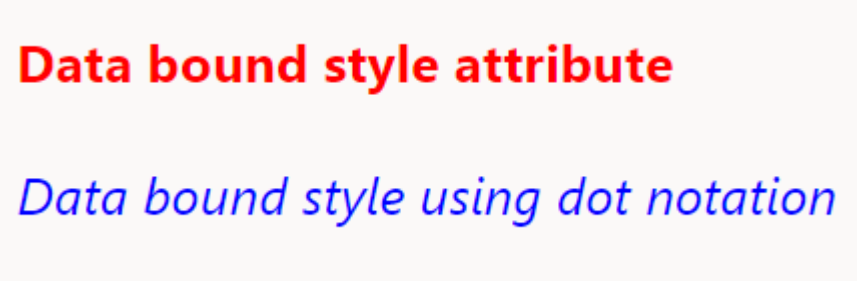
```
import 'ojs/ojknockout';
import 'ojs/ojlabel';
import 'ojs/ojinputtext';

class ViewModel {
  fontColor = 'blue';
  fontStyle = 'italic';
```

```
  style = { fontWeight: 'bold',
            color: 'red' };
}

export = ViewModel;
```

JavaScript

```
define(["ojs/ojknockout", "ojs/ojlabel", "ojs/ojinputtext" ],

  function () {

    function ViewModel() {

        this.fontColor = 'blue';
        this.fontStyle = 'italic';
        this.style = {
            fontWeight: 'bold',
            color: 'red'
          };
      }

   }

    return ViewModel;
});
```

The figure below shows the output for the code sample.



The Oracle JET Cookbook contains the complete example used in this section. See Style Binding.

Bind Event Listeners to JET and HTML Elements

Oracle JET provides one-way attribute data binding for event listeners on JET and HTML elements using the `on-[eventname]` syntax and by importing the `ojknockout` module.

Oracle JET event attributes provide two key advantages over native HTML event listeners. First, they provide three parameters to the listener:

- **`event`**: The DOM event, such as a click or mouseover.
- **`data`**: This parameter is equal to `bindingContext['$data']`. When used in iterations, such as in an `oj-bind-for-each`, this parameter is equal to `bindingContext['$current']`.

- **`bindingContext`**: The entire data-binding context (or scope) that is applied to the element.

Second, they have access to the model state and can access functions defined in the viewModel using the `data` and `bindingContext` parameters.

> **Note:**
>
> The `this` context is not directly available in the event listeners. This is the same behavior as native HTML event listeners.

For example, the following code sample shows an `oj-c-button` element that uses the `on-oj-action` event attribute and an HTML button element that uses the `on-click` event attribute to access custom functions defined in the viewModel shown below.

```
<div id="demo-container">
  <oj-label for="button1">oj-c-button element</oj-label>
  <oj-c-button id="button1" label="Click me!" on-oj-
action="[[clickListener1]]"></oj-c-button>
  <br><br>
  <label for="button2">HTML button element</label>
  <br>
  <button id="button2" on-click="[[clickListener2]]">Click me!</button>
  <br><br>
  <div style="font-weight:bold;color:#ea5b3f;">
    <oj-bind-text value="[[message]]"></oj-bind-text>
  </div></div>
```

> **Note:**
>
> HTML events use the prefix “on”, such as `onclick` and `onload`. JET events use the prefix “on-”, such as `on-click` and `on-load`.

The script to create the viewModel for this example is shown below. Note the usage of the `data` attribute to access the `message` parameter.

- TypeScript
- JavaScript

TypeScript

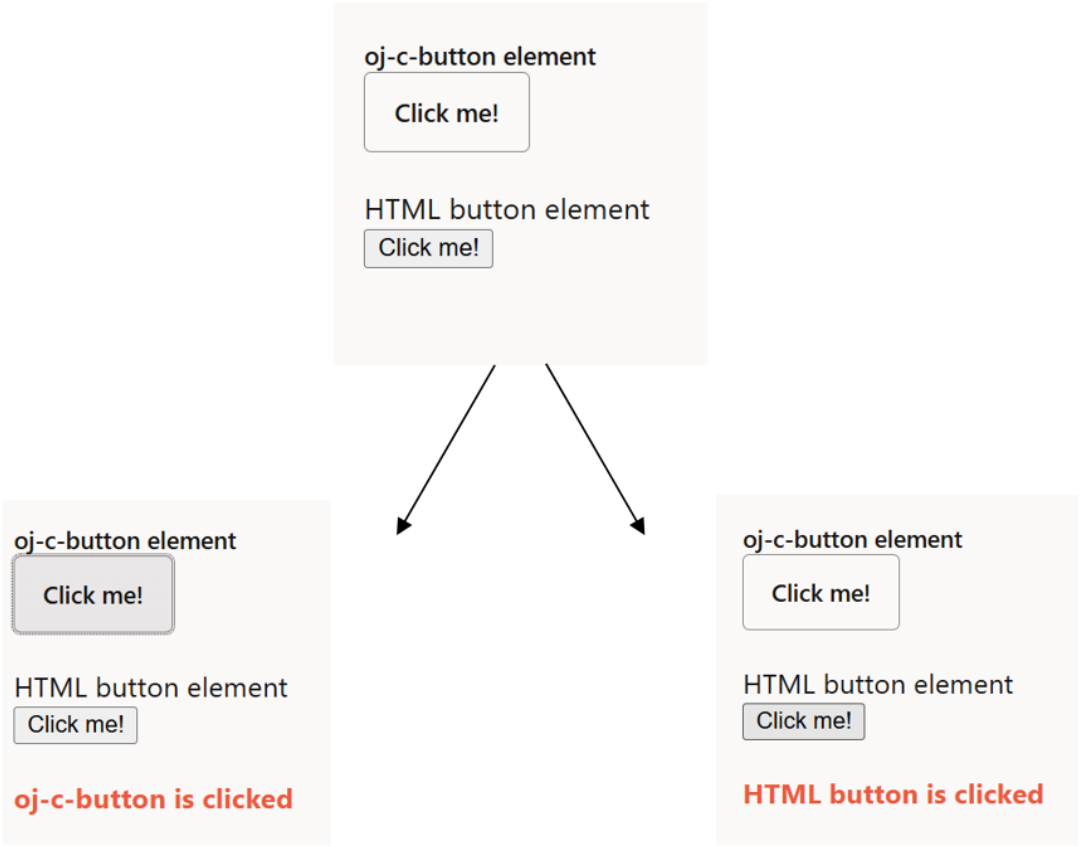
```
import * as ko from 'knockout';
import "oj-c/button";
import 'ojs/ojlabel';
import 'ojs/ojknockout';

class ViewModel {
  message = ko.observable();
  clickListener1 = (
```

```
    event: Event,
    data: { message: ko.Observable },
    bindingContext: ko.BindingContext
  ) => {
    data.message('oj-c-button is clicked');
  };
  clickListener2 = (
    event: Event,
    data: { message: ko.Observable },
    bindingContext: ko.BindingContext
  ) => {
    data.message('HTML button is clicked');
  };
}
```

JavaScript

```
define(["knockout", "ojs/ojknockout", "oj-c/button", "ojs/ojlabel"],

  function (ko) {
    function ViewModel() {
      this.message = ko.observable();
            this.clickListener1 = (event, data, bindingContext) => {
              data.message('oj-c-button is clicked');
            };
            this.clickListener2 = (event, data, bindingContext) => {
              data.message('HTML button is clicked');
            };
    }
    return ViewModel;
});
```

The figure below shows the output for the code sample.



The Oracle JET Cookbook contains the complete example used in this section. See Event Binding.

Bind Classes

The Oracle JET attribute binding syntax has enhanced support for the `class` attribute, which can be used to set classes on components. The attribute accepts a string, an object, or an array. The `ojknockout` module must be imported.

The `:class` attribute binding supports expressions that resolve to a space-delimited string, an array of classes, or an object whose keys are individual style classes and whose values are booleans or expressions used for adding and removing the classes in the DOM. Only object values can be used to toggle classes on and off, whereas array and string values are used only to set classes.

For example, the following code sample shows an `oj-input-text` and an HTML `input` that both use the `:class` attribute.

```
<oj-label for="input1">oj-input-text element</oj-label>
<oj-input-text
  id="input1"
  :class="[[{'oj-form-control-text-align-right': alignRight}]]"
  value="Text Content"></oj-input-text>
<br />
<br />
<oj-c-button
  id="button2"
  label="Toggle Alignment"
```

```
  on-oj-action="[[clickListener2]]">
</oj-c-button>
<br />
<br />
<br />
<label for="input2">HTML input element</label>
<br />
<input
  id="input2"
  :class="[[classArrayObs]]"
  value="Text Content"
  class="demo-width"/>
<br />
<br />
<oj-c-button
  id="button1"
  label="Add Class"
  on-oj-action="[[clickListener1]]">
</oj-c-button>
```

The script to create the viewModel for this example is shown below.

- TypeScript
- JavaScript

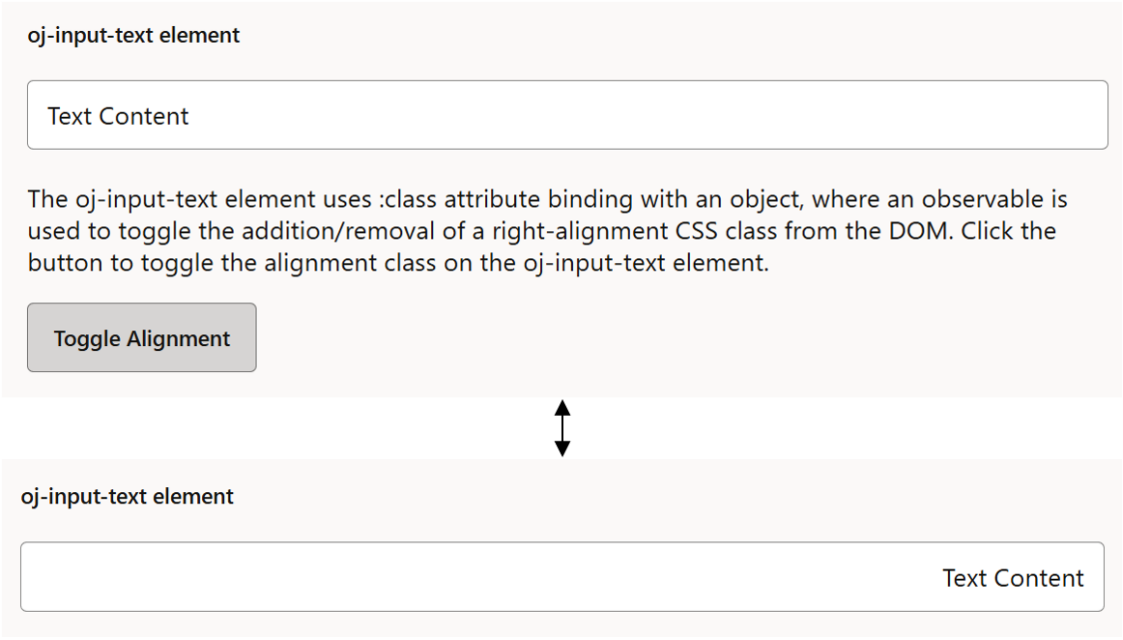
TypeScript

```
import * as ko from 'knockout';
import 'ojs/ojknockout';
import "oj-c/button";
import {CButtonElement } from "oj-c/button";
import 'ojs/ojlabel';
import 'ojs/ojinputtext';

class ViewModel {
  alignRight = ko.observable(false);
  classArrayObs = ko.observableArray();
  private classList = ['oj-text-color-danger', 'oj-typography-bold', 'demo-
italic'];
  clickListener1 = () => {
    this.classArrayObs.push(this.classList.pop()); // Disable the add button
once we're out of classes
    if (this.classList.length === 0) {
      (document.getElementById('button1') as CButtonElement).disabled = true;
    }
  };
  clickListener2 = () => {
    this.alignRight(!this.alignRight());
  };
}
export = ViewModel;
```

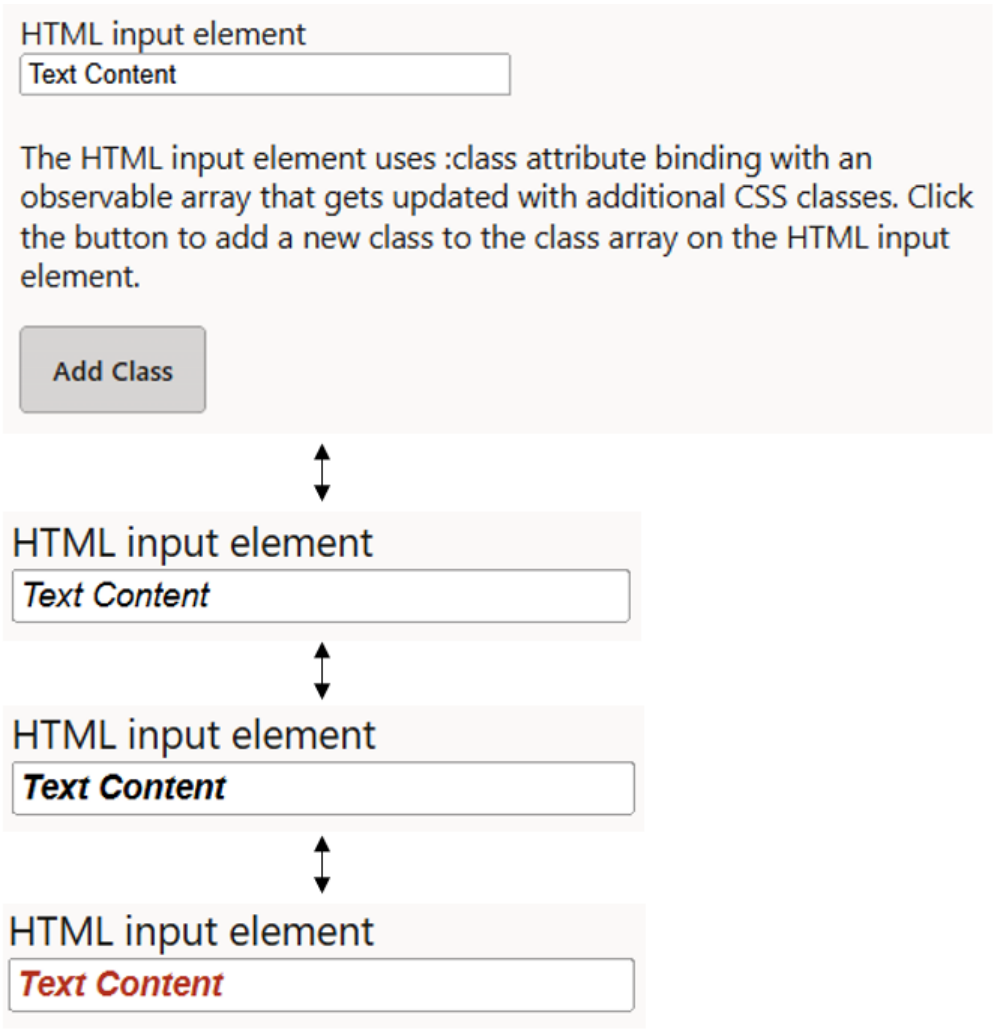
JavaScript

```
define(["knockout", "ojs/ojknockout", "oj-c/button", "ojs/ojlabel", "ojs/
ojinputtext"],

  function (ko) {
    function ViewModel() {
      this.alignRight = ko.observable(false);
      this.classArrayObs = ko.observableArray();
      this.classList = ["oj-text-color-danger", "oj-typography-bold", "demo-
italic",];
      this.clickListener1 = (event, data, bindingContext) => {
      this.classArrayObs.push(this.classList.pop());
        if (this.classList.length === 0) {
          document.getElementById("button1").disabled = true;
        }
      };
      this.clickListener2 = (event, data, bindingContext) => {
        this.alignRight(!this.alignRight());
      };
    }
    return ViewModel;
});
```

The figure below shows the first of the two outputs for the code sample. The button acts as a toggle to switch on and off the `oj-form-control-text-align-right` class property and so change the alignment of the text.



The figure below shows the second of the two outputs for the code sample. The button calls a function to take a predefined array of classes and add them to the input element. Each class has CSS modifications that come into effect when the class is added.



The Oracle JET Cookbook contains the complete example used in this section. See Class Binding.

Add an Oracle JET Component to Your Page

Use the Oracle JET Cookbook recipes and API documentation to locate examples that illustrate the specific element and functionality you want to add to your page.

If you haven't already, create the app that you will use for this exercise.

To add an Oracle JET custom element to your page:

1. Using the Oracle JET Cookbook, select the Oracle JET element that you want to add.
2. If you've set up your app using a Starter Template, or are using a page fragment, add the element to the `define` block.
3. Follow the example's recipe and add the markup to your HTML page. Modify the attributes to your need.

```
<div id="div1">
  <oj-label for="dateTime">Default</oj-label>
  <oj-input-date-time id="dateTime" value='{{value}}'>
  </oj-input-date-time>
```

```
  <br/><br/>

  <span class="oj-label">Current component value is:</span>
  <span><oj-bind-text value="[[value]]"></oj-bind-text></span>

</div>
```

In this example, the `oj-input-date-time` element is declared with its `value` attribute using `{{...}}` expression syntax, which indicates that changes to the value will also update the corresponding value in the ViewModel. Each Oracle JET custom element includes additional attributes that are defined in the custom element's API documentation.

4. Use the Oracle JET Cookbook for example scripts and the syntax to use for adding the custom element's Require module and ViewModel to your RequireJS bootstrap file or module.

   For example, the basic demo for `oj-input-date-time` includes the following script that you can use in your app.

```
require(['ojs/ojbootstrap, 'knockout', 'ojs/ojvalidation-base', 'ojs/ojknockout',
'ojs/ojdatetimepicker', 'ojs/ojtimezonedata', 'ojs/ojlabel'],
  require(['knockout', 'ojs/ojbootstrap',
           'ojs/ojconverterutils-i18n', 'ojs/ojknockout',
           'ojs/ojdatetimepicker', 'ojs/ojlabel'],

  function (Bootstrap, ko, ConverterUtilsI18n)
  {
    function SimpleModel()
    {
      this.value =
ko.observable(ConverterUtilsI18n.IntlConverterUtils.dateToLocalIso(new Date(2020, 0,
1)));
    }

    Bootstrap.whenDocumentReady().then(
      function ()
      {
        ko.applyBindings(new SimpleModel(), document.getElementById('div1'));
      }
    );
  });
```

   If you already have a RequireJS bootstrap file or module, compare your file with the Cookbook sample and merge in the differences. For details about working with RequireJS, see Use RequireJS for Modular Development.

The Cookbook sample used in this section is the Date and Time Pickers demo.

Add Animation Effects

You can use the `oj-module` component's `animation` property in conjunction with the `ModuleAnimations` namespace to configure animation effects when the user transitions between or drills into views. If you're not using `oj-module`, you can use the `AnimationUtils` namespace instead to add animation to Oracle JET components or HTML elements.

Adding Animation Effects Using the oj-module Component

The ModuleAnimations namespace includes pre-configured implementations that you can use to configure the following animation effects:

- `coverStart`: The new view slides in to cover the old view.
- `coverUp`: The new view slides up to cover the old view.
- `drillIn`: Animation effect is platform-dependent.
  - Web and iOS: `coverStart`
  - Android: `coverUp`
- `drillOut`: Animation effect is platform-dependent.
  - Web and iOS: `revealEnd`
  - Android: `revealDown`
- `fade`: The new view fades in and the old view fades out.
- `goLeft`: Navigate to sibling view on the left. Default effect is platform-dependent.
  - Web and iOS: none
  - Android: `pushRight`
- `goRight`: Navigate to sibling view on the right. Default effect is platform-dependent.
  - Web and iOS: none
  - Android: `pushLeft`
- `pushLeft`: The new view pushes the old view out to the left.
- `pushRight`: The new view pushes the old view out to the right.
- `revealDown`: The old view slides down to reveal the new view.
- `revealEnd`: The new view slides left or right to reveal the new view, depending on the locale.
- `zoomIn`: The new view zooms in.
- `zoomOut`: The old view zooms out.

For examples that illustrate how to add animation with the `oj-module` component, see Animation Effects with Module Component.

Adding Animation Effects Using AnimationUtils

The AnimationUtils namespace includes methods that you can use to configure the following animation effects on HTML elements and Oracle JET components:

- `collapse`: Use for collapsing the element
- `expand`: Use for expanding the element
- `fadeIn` and `fadeOut`: Use for fading the element into and out of view.
- `flipIn` and `flipOut`: Use for rotating the element in and out of view.
- `ripple`: Use for rippling the element.
- `slideIn` and `slideOut`: Use for sliding the element into and out of view.
- `zoomIn` and `zoomOut`: Using for zooming the element into and out of view.

Depending on the method's options, you can configure properties like `delay`, `duration`, and `direction`. For examples that illustrate how to configure animation using the `AnimationUtils` namespace, see Animation Effects.

Manage the Visibility of Added Component

Follow the recommended best practice when you programmatically manage the display of Oracle JET components in the DOM.

If you manage the display of collection components and other complex component by using the CSS `display` property values of `none` or `block`, you may also need to use `Components.subtreeHidden(node)` when you hide a component and `Components.subtreeShown(node)` when you display the component. The *`node`* parameter refers to the root of the subtree in the DOM for the component that you are hiding or displaying. You need to notify Oracle JET if you change the display status of these types of component to ensure that the component instance continues to work correctly in the app.

Not all components where you programmatically manage the display require you to notify Oracle JET when you change their display state. Components such as `oj-collapsible`, `oj-dialog`, and `oj-popup` are examples of components where you do not need to call `Components.subtreeHidden(node)` or `Components.subtreeShown(node)`. These components manage rendering when visibility is changed and also manage any components that they contain. Similarly, you do not need to use these methods in conjunction with the `oj-bind-if` component because Oracle JET rewrites the appropriate part of the DOM in response to the evaluation of the `oj-bind-if` component's `test` condition.

Failure to use the `subtreeHidden` and `subtreeShown` methods to notify Oracle JET when you hide or show a component can result in unexpected behavior for the component, such as failure to render data or failure to honor other component attribute settings. For more information about the `subtreeHidden` and `subtreeShown` methods, see API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET).

7

Work with Oracle JET User Interface Components

Oracle JET provides a variety of user interface (UI) components that you can configure for use in your app. The Oracle JET Cookbook includes an example of each component for working with collections, controls, forms, visualizations, and other features.

About Oracle JET User Interface Components

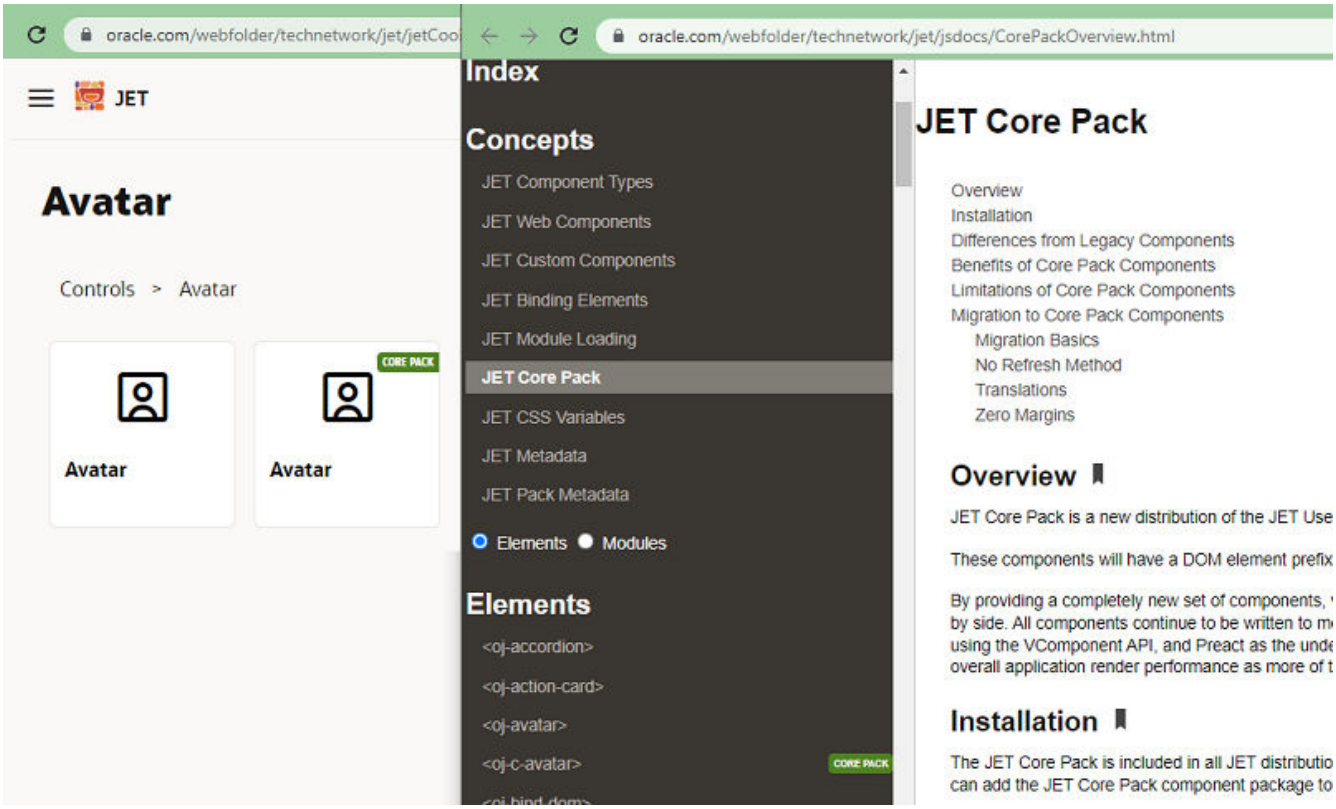
Oracle JET currently has two sets of UI components. The first set with components that use the `oj-` namespace are packaged in `@oracle/oraclejet` and date back to the initial releases of Oracle JET. The newer Core Pack components, introduced in January 2023 with release 14.0.0 of Oracle JET, use the `oj-c-` namespace and are packaged in `@oracle/oraclejet-core-pack`.

Core Pack components represent the future of JET. The JET team are rewriting all the existing JET UI components from scratch using Preact, a modern virtual DOM rendering library that uses React design and composition principles. The JET 14 release delivers the first set of these newer Core Pack components. This component set will grow over the coming years until all existing components have been re-created as Core Pack components. The JET team will also be introducing new UI components to the Core Pack component set during the same period.

JET's existing set of UI components, now referred to as *legacy components*, has served the JET community and its app developers well for the last 10 years. The legacy components, using the `oj-` namespace, will run side-by-side with the newer Core Pack components, using the `oj-c-` namespace in the same JET apps. The APIs are similar, if not identical in many cases. The newer Core Pack was created with the intention of allowing for updates and changes to APIs that don't make sense going forward, without disrupting your existing application code. As a developer, you can choose when you want to move to the new components as part of your regular development cycle.

You might ask why you should move to these components at all. The main reason is performance. Rendering performance is significantly improved with the Core Pack components. While a new component set will not automatically change the overall performance of your app, the render time can definitely make your apps look and feel more responsive to your customer. The second reason to use the Core Pack component set is that all future new components will be delivered as part of it. We will not be developing new legacy UI components (`oj-`) in the future.

Each new Core Pack component includes an example implementation in the Oracle JET Cookbook and an entry in the API documentation. A CORE PACK badge identifies these implementations and entries.



For details about how to work with the Core Pack components, any short-term limitations, and migration information, see Core Pack in the API documentation.

Work with Collections

Use Oracle JET data collection components to display data in tables, data grids, list views, or trees.

The Oracle JET data collection components include `oj-table`, `oj-data-grid`, `oj-tree`, and `oj-list-view`, and you can use them to display records of data. `oj-table` and `oj-data-grid` both display data in rows and columns, and your choice of component depends upon your use case. Use `oj-tree-view` to display hierarchical data such as a directory structure and `oj-list-view` to display a list of data. The toolkit also includes pagination and row expanders that display hierarchical data in a data grid or table row.

The Oracle JET Cookbook's Collections category and API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) include complete demos and examples for using the collection components.

> **Note:**
>
> If you programmatically control the display of a collection component, such as `oj-table`, review Manage the Visibility of Added Component.

Choose a Table, Data Grid, or List View

Oracle JET provides the `oj-table`, `oj-data-grid`, and `oj-list-view` components to display data in rows and columns. This section helps you decide which component to use in your app.

The `oj-table` component displays records of data on a row basis. It's best used when you have simple data that can be presented as rows of fields, and it should be your first choice as it provides a simpler layout to represent the data and also supports most of the common features, unless you require advanced features. A selection in the table provides you with the row of data and all of the fields in that row or record. The sizing of the table is based on the content itself. The height and width of the cells is adjusted for the content included. You can write templates using `oj-table` elements such as `tr`, `td`, `th`, and so on. Also consider making use of the `oj-table` component's `layout` attribute that enables you to change column sizing based on fixed values or content size. See the `oj-table` component's Column Layouts demo in the Oracle JET Cookbook.

The `oj-data-grid` is designed to provide grid functionality. It provides the ability to select individual or ranges of cells of data. It's best used when you need to display totals or tallies of data across columns or rows of data. The `oj-data-grid` is designed to be much more flexible in its layout and functionality than the `oj-table` component. It's a low-level component that you can shape in your app to how you want the data to be displayed. The overall size of the data grid is not determined by its content, and the developer specifies the exact height and width of the container. The data grid acts as a viewport to the contents, and unlike a table its size is not determined by the size of the columns and rows. With this custom HTML `oj-data-grid` element, you can host the template content inside it.

The `oj-list-view` element displays a list of data or a list of grouped data. It is best used when you need to display a list using an arbitrary layout or content. You can also use `oj-list-view` to display hierarchical data that contains nested lists within the root element.

The table below provides a feature comparison of the `oj-table`, `oj-data-grid`, and `oj-list-view` components.

Feature	oj-table	oj-data-grid	oj-list-view
Column/Row sizing	Controlled by content or fixed by CSS styles. All CSS sizing strings are supported for width and height.	Controlled by cell dimensions. Does not support percent values for width and height.	No
User-resizable column	Yes	Yes	No
User-resizable row	No	Yes	No
Row reordering	No	Yes	No
Column sorting	Yes	Yes	No
Column selection	Yes	Yes	No
Row sorting	No	Yes	No
Row selection	Yes	Yes	Yes
Cell selection	No	Yes	No
Marquee selection	No	Yes	No
Row header support	No	Yes	No

Feature	oj-table	oj-data-grid	oj-list-view
Freeze columns	Yes Use the `frozenEdge` property to freeze a column to the start or end of the table.	No	No
Pagination	Yes	Yes	Yes
Scrolling (high water mark / infinite scrolling)	Yes, when end of table reached (or document size). See note at the end about virtual scrolling.	Yes, when data grid column / row count reached (see note about virtual scrolling option)	Yes, when end of list reached (or document size) See note at the end about virtual scrolling.
Custom cell templates	Yes	Yes	No
Custom row templates	Yes	No	Yes
Custom cell renderers	Yes	Yes	No
Custom row renderers	Yes	No	Yes
Row expander support	Yes	No	No
Cell stamping	Yes	Yes	No
Cell merging	No	Yes	No
Render aggregated cubic data	No	Yes	No
Custom footer template	Yes (provides access to column data for passing to a JavaScript function)	Yes	No
Cell content editing	Yes	Yes	No
Content filtering	Yes	Yes	No
KeySet API support	Yes	No	Yes

Note:

True virtual scrolling is available as a feature of `oj-data-grid`. Modern design principles should be considered and implemented before implementing virtual scrolling. It is much more desirable to present the end user with a filtered list of data that will be more useful to them, than to display thousands of rows of data and expect them to scroll through it all. True virtual scrolling means that you can perform scrolling in both horizontal and vertical directions with data being added and removed from the underlying DOM. High water mark scrolling (with lazy loading) is the preferred method of scrolling, and you should use it as a first approach.

In the case of `oj-table` and `oj-list-view`, you can set `scroll-policy-options.scroller` to specify the document size as the maximum scroll position before the next data fetch occurs. This is particularly useful when your app runs in a mobile device where the table or list occupies the entire screen.

`KeySet` objects are used to represent the selected items in a component. The collection components generally uses an array for the selected items or an object that defines the range of selected items. You can modify these components to use a `KeySet` object that handles the

representation of selected items. Note that the Data Grid component does not currently support `KeySet` object referencing for the selected items. The `oj-list-view` component can use `KeySet` to determine the selected items. The `oj-table` component can use `KeySet` to determine the selected items for the rows or columns. If both values are specified, then row will take precedence and column will be reset to an empty `KeySet`.

About DataProvider Filter Operators

The DataProvider interface is used to get runtime data for JET components that display list of items. DataProvider implementations use filter operators for filtering.

You can specify two types of filters:

- Attribute Filter: Provides filters with the functionality of attribute operator filtering.

```
interface AttributeFilter<D> extends AttributeFilterDef<D>{
  filter(item: D, index?: number, array?: Array<D>): boolean;
}
type Filter<D> = AttributeFilter<D> | CompoundFilter<D>;

type RecursivePartial<T> = {
  [P in keyof T]?:
    T[P] extends (infer U)[] ? RecursivePartial<U>[] :
    T[P] extends object ? RecursivePartial<T[P]> :
    T[P];
};

interface AttributeFilterDef<D> {
  readonly op: AttributeFilterOperator.AttributeOperator;
  readonly attribute?: keyof D;
  readonly value: RecursivePartial<D>;
}

type AttributeFilterAttributeExpression = '*';
```

- Compound Filter: Provides filter operators for compound operations.

```
interface CompoundFilter<D> extends CompoundFilterDef<D>, BaseFilter<D> {
}

interface CompoundFilterDef<D> {
  readonly op: CompoundFilterOperator.CompoundOperator;
  readonly criteria: Array<AttributeFilterDef<D> | CompoundFilterDef<D>>;
}
```

Filter definition which filters on `DepartmentId` value 10:

```
{op: '$eq', value: {DepartmentId: 10}}
```

Filter definition which filters on `DepartmentId` value 10 and `DepartmentName` is Hello:

```
{op: '$eq', value: {DepartmentId: 10, DepartmentName: 'Hello'}}
```

Filter definition which filters on subobject `Location` where `State` is California and `DepartmentName` is Hello:

```
{op: '$eq', value: {DepartmentName: 'Hello', Location: {State: 'California'}}}
```

For additional detail and the complete list of operators that you can use with Attribute filters, see the AttributeFilterDef API documentation.

For additional detail and the complete list of operators that you can use with Compound filters, see the CompoundFilterDef API documentation.

Work with Controls

Oracle JET includes buttons, menus, and container elements to control user actions or display progress against a task. For HTML elements such as simple lists, you can use the standard HTML tags directly on your page, and Oracle JET will apply styling based on the app's chosen theme.

For example, you can use the `oj-button` element as a standalone element or include in `oj-buttonset` , `oj-menu`, and `oj-toolbar` container elements.

Navigation components such as `oj-conveyor-belt`, `oj-film-strip`, and `oj-train` use visual arrows or dots that the user can select to move backward or forward through data.

To show progress against a task in a horizontal meter, you can use the `oj-progress-bar` element. To show progress against a task in a circle, you can use the `oj-progress-circle` element.

The Oracle JET Cookbook's Controls category and API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) include complete demos and examples for using Oracle JET controls.

Work with Forms

Oracle JET includes classes to create responsive form layouts and components that you can add to your form to manage labels, form validation and messaging, input, and selection. The input components also include attributes to mark an input as disabled or read-only when appropriate.

The Oracle JET Cookbook's Forms category and API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) includes complete demos and examples for using forms.

> **Important:**
>
> When working with forms, use the HTML `div` element to surround any Oracle JET input components. Do not use the HTML `form` element because its postback behavior can cause unwanted page refreshes when the user submits or saves the form.

Work with Layout and Navigation

Use the Oracle JET `oj-accordian`, `oj-collapsible`, `oj-dialog`, `oj-drawer-*`, `oj-flex*`, `oj-navigation-list`, `oj-panel`, `oj-popup`, `oj-`*`size`*`*`, and `oj-tab-bar` components and patterns to control the initial data display and allow the user to access additional content by expanding sections, selecting tabs, or displaying dialogs and popups.

The Oracle JET Cookbook's Layout and Navigation category and API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) include complete demos and examples for using the layout and navigation components.

For information about the flex layout (`oj-flex*`) and responsive grid (`oj-size`) classes, see Design Responsive Apps.

Work with Visualizations

The Oracle JET visualization components include charts, gauges, and other components that you can use and customize to present flat or hierarchical data in a graphical display for data analysis.

Choose a Data Visualization Component for Your App

The visualization components include charts, gauges, and a variety of other visualizations including diagrams, timelines, thematic maps, and so on that you can use for displaying data. You may find the following usage suggestions helpful for determining which visualization to use in your app.

Charts

Charts show relationships among data and display values as lines, bars, and points within areas defined by one or more axes.

Chart Type	Image	Description	Usage Suggestions
Area		Displays series of data whose values are represented by filled-in areas. Areas can be stacked or unstacked. The axis is often labeled with time periods such as months.	Use to show cumulative trends over time, such as sales for the last 12 months. Area charts require at least two groups of data along an axis. If you are working with multiple series and want to display unstacked data, use line or line with area charts to prevent values from being obscured.
Bar		Displays data as a series of rectangular bars whose lengths are proportional to the data values. Bars display vertically or horizontally and can be stacked or unstacked.	Use to compare values across products or categories, or to view aggregated data broken out by a time period.
Box Plot		Displays the minimum, quartiles, median, and maximum values of groups of numerical data. Groups display vertically or horizontally. You can also vary the box width to make the width of the box proportional to the size of the group.	Use to analyze the distribution of data. Box plots are also called box and whisker diagrams.
Bubble		Displays three measures using data markers plotted on a two-dimensional plane. The location of the markers represents the first and second measures, and the size of the data markers represents the proportional values of the third measure.	Use to show correlations among three types of values, especially when you have a number of data items and you want to see the general relationships. For example, use a bubble chart to plot salaries (x-axis), years of experience (y-axis), and productivity (size of bubble) for your work force. Such a chart enables you to examine productivity relative to salary and experience.

Chart Type	Image	Description	Usage Suggestions
Combination		Displays series of data whose values are represented by a combination of bars, lines, or filled-in areas.	Combination charts are commonly configured as lines with bars for lines with stacked bars. For example, you can use a line to display team average rating with bars to represent individual team member ratings on a yearly basis.
Funnel		Visually represents data related to steps in a process as a three-dimensional chart that represents target and actual values, and levels by color. The steps appear as vertical slices across a horizontal cone-shaped section. As the actual value for a given step or slice approaches the quota for that slice, the slice fills.	Use to watch a process where the different sections of the funnel represent different stages in the process, such as a sales cycle. The funnel chart requires actual values and target values against a stage value, which might be time.
Line		Displays series of data whose values are represented by lines.	Use to compare items over the same time. Charts require data for at least two points for each member in a group. For example, a line chart over months requires at least two months. Typically a line of a specific color is associated with each group of data such as the Americas, Europe, and Asia. Lines should not be stacked which can obscure data. To display stacked data, use area or line with area charts.
Line with Area		Displays series of data whose values are represented as lines with filled-in areas.	Use for visualizing trends in a set of values over time and comparing those values across series.
Pie		Represents a set of data items as proportions of a total. The data items are displayed as sections of a circle causing the circle to look like a sliced pie.	Use to show relationship of parts to a whole such as how much revenue comes from each product line. Consider treemaps or sunbursts if you are working with hierarchical data or you want your visual to display two dimensions of data.
Polar		Displays series of data on a polar coordinate system. The polar coordinate system can be used for bar, line, area, combination, scatter, and bubble charts. Polygonal grid shape (commonly known as radar) is supported for polar line and area charts.	Use to display data with a cyclical x-axis, such as weather patterns over months of the year, or for data where the categories in the x-axis have no natural ordering, such as performance appraisal categories.

Chart Type	Image	Description	Usage Suggestions
Pyramid		Displays values as slices in a pyramid. The area of each slice represents its value as a percentage of the total value of all slices.	Use to display hierarchical, proportional and foundation-based relationships, process steps, organizational layers, or topics interconnections.
Range		Displays a series of data whose values are represented either as an area or bar proportional to the data values.	Use to display a range of temperatures for each day of a month for a city.
Scatter		Displays two measures using data markers plotted on a two-dimensional plane.	Use to show correlation between two different kinds of data values, such as sales and costs for top products. Scatter charts are especially useful when you want to see general relationships among a number of items.
Spark		Display trends or variations as a line, bar, floating bar, or area. Spark charts are simple and condensed.	Use to provide additional context to a data-dense display. Sparkcharts are often displayed in a table, dashboard, or inline with text.
Stock		Display stock prices and, optionally, the volume of trading for one or more stocks. When any stock or candlestick chart includes the volume of trading, the volume appears as bars in the lower part of the chart.	

Gauges

Gauges focus on a single value, displayed in relation to minimum, maximum, or threshold values.

Gauge Type	Image	Description	Usage Suggestions
LED		Graphically depicts a measurement, such as a key performance indicator (KPI). Several styles of shapes are available, including round or rectangular shapes that use color to indicate status, and triangles or arrows that point up, left, right, or down in addition to the color indicator.	Use to highlight a specific metric value in relation to its threshold.

Gauge Type	Image	Description	Usage Suggestions
Rating		Displays and optionally accepts input for a metric value.	Use to show ratings for products or services, such as the star rating for a movie.
Status Meter		Displays a metric value on a horizontal, vertical, or circular axis. An inner rectangle shows the current level of a measurement against the ranges marked on an outer rectangle. Optionally, status meters can display colors to indicate where the metric value falls within predefined thresholds.	

Other Data Visualizations

Other data visualizations include maps, timelines, Gantt charts and various other components that don't fit into the chart or gauge category.

Data Visualization Component	Image	Description	Usage Suggestions
Diagram		Models, represents, and visualizes information using a shape called a node to represent data, and links to represent relationships between nodes.	Use to highlight both the data objects and the relationships between them.
Gantt		Displays bars that indicate the start and end date of tasks.	Use to display project schedules.
Legend		Displays a panel which provides an explanation of the display data in symbol and label pairs.	Consider using the legend component when multiple visualizations on the same page are sharing a coloring scheme. For an example using `ojLegend` with a bubble chart, see Use Attribute Groups With Data Visualization Components.

Data Visualization Component	Image	Description	Usage Suggestions
NBox		Displays data items across two dimensions. Each dimension can be split into multiple ranges, whose intersections result in distinct cells representing data items.	Use to visualize and compare data across a two-dimensional grid, represented visually by rows and columns.
PictoChart		Uses stamped images to display discrete data as a visualization of an absolute number or the relative size of different parts of a population.	Common in infographics. Use when you want to use icons to: • visualize a discrete value, such as the number of people in a sample that meets a specified criteria. • highlight the relative sizes of the data, such as the number of people belonging to each age group in a population sample.
Sunburst		Displays quantitative hierarchical data across two dimensions, represented visually by size and color. Uses nodes to reference the data in the hierarchy. Nodes in a radial layout, with the top of the hierarchy at the center and deeper levels farther away from the center.	Use for identifying trends for large hierarchical data sets, where the proportional size of the nodes represents their importance compared to the whole. Color can also be used to represent an additional dimension of information. Use sunbursts to display the metrics for all levels in the hierarchy.
Tag Cloud		Displays textual data where font style and size emphasizes the importance of each data item.	Use for quickly identifying the most prominent terms to determine their relative importance.
Thematic Map		Displays data that is associated with a geographic location.	Use to show trends or patterns in data with a spatial element to it.
Time Axis		Displays a range of dates based on specified start and end dates and a time scale.	Use when you want to fulfil certain Gantt use cases. This component is intended to be stamped inside Table or Data Grid components.

Data Visualization Component	Image	Description	Usage Suggestions
Timeline		Displays a set of events in chronological order and offers rich support for graphical data rendering, scale manipulation, zooming, resizing, and objects grouping.	Use to display time specific events in chronological order.
Treemap		Displays quantitative hierarchical data across two dimensions, represented visually by size and color. Uses nodes to reference the data in the hierarchy. Nodes are displayed as a set of nested rectangles.	Use for identifying trends for large hierarchical data sets, where the proportional size of the nodes represents their importance compared to the whole. Color can also be used to represent an additional dimension of information Use treemaps if you are primarily interested in displaying two metrics of data using size and color at a single layer of the hierarchy.

For examples that implement visualization components, see the Oracle JET Cookbook at Data Visualizations.

> **Note:**
>
> To use an Oracle JET data visualization component, you must configure your app to use RequireJS. For details about adding RequireJS to your app, Use RequireJS in an Oracle JET App.

Use Attribute Groups With Data Visualization Components

Attribute groups allow you to provide stylistic values for color and shape that can be used as input for supported data visualization components, including bubble and scatter charts, sunbursts, thematic maps, and treemaps. In addition, you can share the attribute values across components, such as a thematic map and a legend, using an attribute group handler.

Using attribute groups is also one way that you can easily provide visual styling for data markers for a given data set. Instead of manually choosing a color for each unique property and setting a field in your data model, you can use an attribute group handler to get back a color or shape value given a data value. Once an attribute group handler retrieves a color or shape value given a data value, all subsequent calls that pass in the same data value will always return that color or shape.

Oracle JET provides the following classes that you can use for adding attribute groups to your data visualization components:

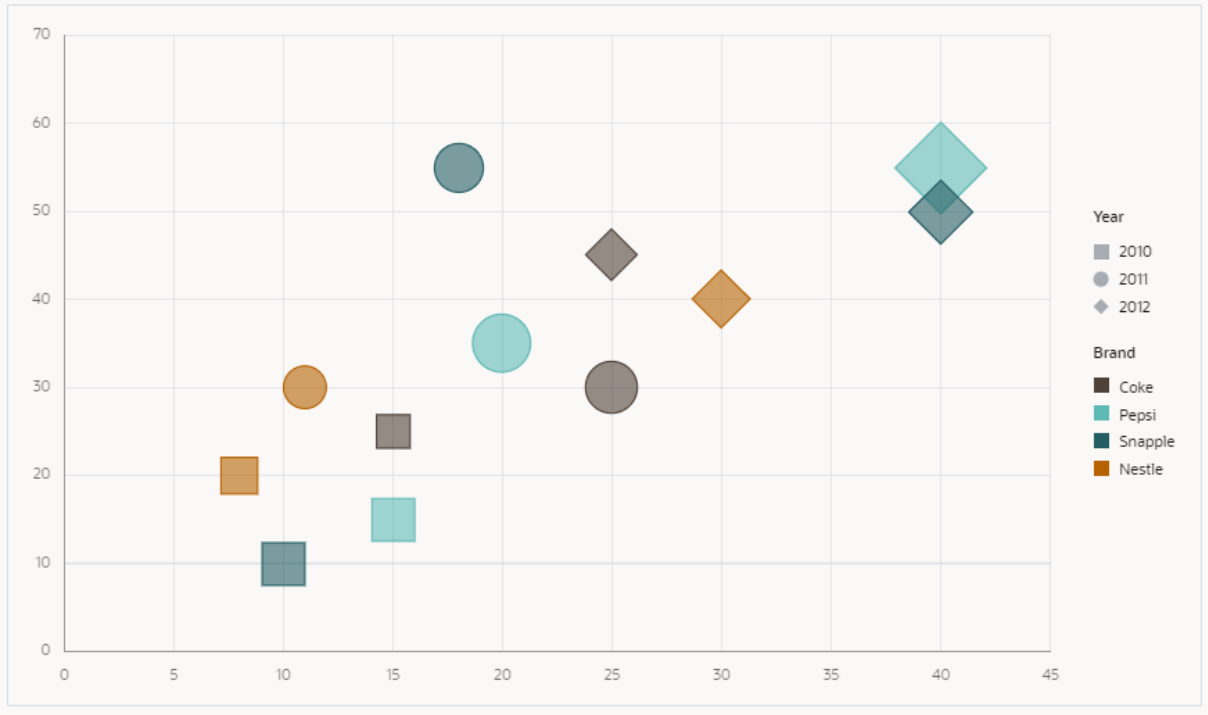
- `ColorAttributeGroupHandler`: Creates a color attribute group handler that will generate color attribute values.

  Colors are generated using the values in the `.oj-dvt-category-index*` tag selectors.

- `ShapeAttributeGroupHandler`: Creates a shape attribute group handler that will generate shape attribute values.

  Supported shapes include `square`, `circle`, `human`, `triangleUp`, `triangleDown`, `diamond`, and `plus`.

You can see the effect of applying attribute groups to a bubble chart in the following figure. In this example, the shape of the markers (`round`, `diamond`, and `square`) indicates the year for which the data applies. The color differentiates the brand. The example also uses the `Legend` data visualization component to provide a legend for the bubble chart.



The bubble chart's legend uses the same attribute group handlers for color and shape.

The Oracle JET Cookbook provides the complete code for implementing bubble charts at Bubble Charts.

You can also initialize an attribute group with match rules which consist of a map of key value pairs for categories and the matching attribute values. For example, if you wanted to specify colors for specific categories instead of using the default colors, you could define the color attribute group with match rules.

```
var colorHandler = new ColorAttributeGroupHandler({"soda":"#336699",
                                                   "water":"#CC3300",
                                                   "iced tea":"#F7C808"});
```

For detailed information about `ColorAttributeGroupHandler`, see the ColorAttributeGroupHandler API documentation. For more information about `ShapeAttributeGroupHandler`, see the ShapeAttributeGroupHandler API documentation.

8

Work with Oracle JET Web Components

Oracle JET Web Components are reusable pieces of user interface code that you can embed as custom HTML elements. Web Components can contain Oracle JET components, other Web Components, HTML, JavaScript, and CSS. You can create your own Web Component or add one to your page.

Design Custom Web Components

Oracle JET Web Components are custom components that include multiple component types. Web components that you create can be used in your app or they can be uploaded to Oracle Component Exchange to share with other developers.

The variety of component types supported by Oracle JET and Oracle Component Exchange are:

- **Standalone Web Components** are classic UI components with some kind of UI along with a defined set of properties, methods, events and slots. They can represent everything from a simple better-button type of widget all the way to a super complex whole page component such as a calendar.
- **Pack components**, also called a JET Pack, represents a versioned stripe of related components designed to be used together. When consumers pick up a component that is a member of a JET Pack they associate their app with the version of the pack as a whole, not the individual components within it. The JET Pack simplifies the setup of such projects and the dependency management as a whole.
- **Resource components** are re-usable libraries of assets used by Web Components contained in JET Packs. Resource components typically contain things like shared images, resource bundles, utility classes and so forth. A resource component has no hard and fast predefined structure so can contain anything that you want. However, it does not itself provide any UI components. Instead conventional standalone components would depend on one of these for shared resources. We only expect resource components to be used in concert with JET Packs.
- **Reference components** define a look-up reference to third party code. As such, reference components are a pointer to that code either as a NPM module and/or as a CDN location. Reference components don't actually include the third party libraries, they just point to it.

You can create standalone Web Components to support your specific app needs. You can also create sets of Web Components that you intend to be used together and assemble those in a JET Pack, or pack component type. The pack component contains the Web Components and configuration files that define the version stripe of each component in the pack. When Web Components are part of a pack, changes to their definition file are required to differentiate them from the same component used as a standalone component.

> **Tip:**
>
> When you assemble components into a small number of packs, from the consumer's point of view, it makes path setup and dependency management much simpler.

You can enhance JET Pack components by using resource components when you have re-usable libraries of assets that are themselves not specifically UI components. The resource component structure is flexible so you add anything that you want, such as shared images, resource bundles, utility classes and so forth.

If you need to reference third-party code in a Web Component, you can create a reference component. The reference component doesn't include any third-party libraries, but it can define a pointer to that code either as a NPM module and/or as a CDN location. Although it is possible to embed third party library code into the packaged distribution of a given component, by separating it out you get two particular benefits:

- The dependency is clear and declared up front. This is an important consideration for organizations that care about third party liability and license usage. It also makes reacting to security vulnerabilities in third party code much easier.
- You can maximize re-use of these libraries - particularly with common libraries, such as moment.js.

The only component type that is not allowed in packs are reference components.

When creating JET Packs it is important to think about how their components can evolve over time in relation to the consuming apps. A common mistake is to start out syncing the component version numbers to the version number of the primary consuming app. This relationship can break down, however, for example when a breaking API change in one of your components forces a major version change which now takes you out of sync. The best practice is to adopt Semantic Versioning (SemVer) of components from the start and not to sync versions with the consuming app. Take the time to understand how SemVer works, particularly in relation to re-release versions. For more information, see Version Numbering Standards.

You should maintain the source code for your component sets separately from the apps that will consume them. Remember that components will evolve over time at a separate rate from the consuming app so having the source code decoupled into a separate source code project and repository makes a huge amount of sense.

The recommended project layout for your component source project is based on the default project layout created by Oracle JET CLI tooling. The JET CLI supports the creation of TypeScript components as well as standard ES6 based components. If you follow adhere to the project layout generated by JET tooling, then you derive the following tooling benefits:

- Automatic creation of the correct requireJS path mappings for your components and their upstream dependencies when testing within the context of this component source project
- Support for live editing when using the `ojet serve` command (both JS and TS components)
- Auto transpilation of Typescript based components to ES6
- Automatic creation of both debug and minified components by the `ojet build` command
- Automatic creation of component bundles for JET Packs where bundling is specified
- Ability to directly publish to the Component Exchange using the `ojet publish` component command
- Ability to directly package your components into distributable zip files using the `ojet package` component command

About Web Components

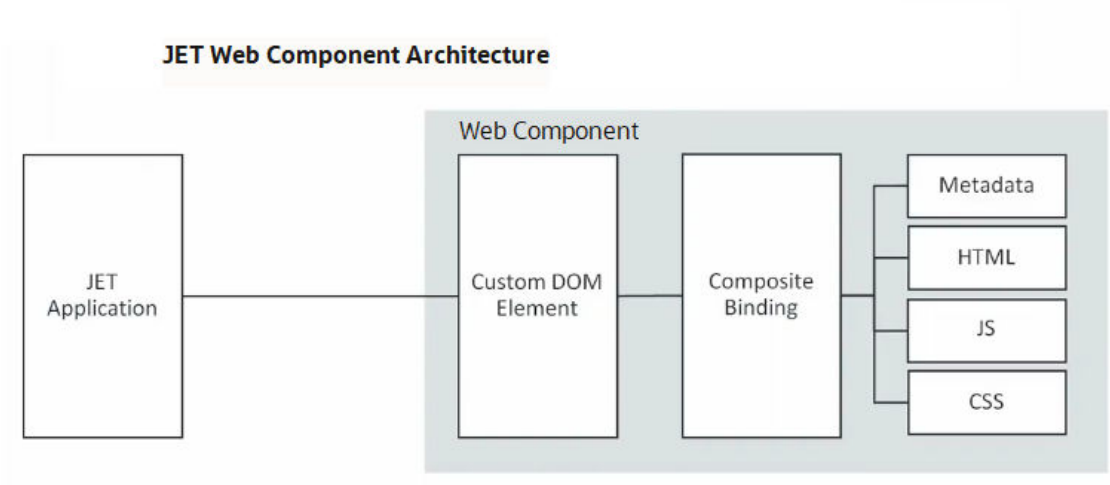
Oracle JET Web Components are packaged as standalone modules that your app can load using RequireJS. The framework supplies APIs that you can use to register Web Components.

Knockout currently provides one and two way data binding, template markup, and Web Component activation.

If you are new to Web Components and would like to learn more, visit this Oracle Blogs page for a series of articles that will introduce you to important concepts: https://blogs.oracle.com/groundside/cca.

Web Component Architecture

The following image shows a high-level view of the JET Web Component architecture. In this example, an Oracle JET app is consuming the Web Component, but you can add Web Components to other JavaScript or Oracle apps where supported.



Web Components contain:

- A custom DOM element: Named element that functions as a HTML element.

```
<my-web-component attribute1="value1" attribute2="value2" ...>
</my-web-component>
```

- Web Component binding definition: Knockout expression for setting Web Component attributes.

```
<my-web-component attribute1="value1" attribute2="[[value2]]"
attribute3="{{value3}}">
</my-web-component>
```

`attribute1`'s value can be a primitive JavaScript type (boolean, number, string) or a JSON object, `attribute2`'s value uses one way data binding, and `attribute3` 's value uses a two way binding. One way bindings on Web Components specify that the expression will not update the app's ViewModel if the value changes. In two way bindings, the expression will update and the value written back to the app's ViewModel.

In the following code sample, the Web Component is declared with three attributes: `type`, `data`, and `axis-labels`.

```
<my-chart type=bubble data="[[salesData]]" axis-
labels={{showAxisLabels}} ... </my-chart>
```

Because the `salesData` expression is declared using one way binding (`[[salesData]]`), it will not be written back to if the `data` property is updated by the Web Component's ViewModel. The `data` property will contain the current value, but the `salesData` expression will not be updated. Alternatively, if the `axisLabels` property is updated by the ViewModel, both the `axisLabel` property and the `{{showAxisLabels}}` expression will contain the updated value.

- Metadata: Data provided in JSON format which defines the Web Component's required properties: `name`, `version`, and `jetVersion`. Metadata may also define optional properties, including `description`, `displayName`, `dependencies`, `icon`, `methods`, `events`, and `slots`.

  Web Components support both runtime and design time metadata. Design time metadata isn't required at runtime and is useful for design time tools and property editors. Design time tools can define tools-specific metadata extensions to the Web Component's metadata. For any tool-specific metadata extensions, refer to the documentation for that specific tool. For additional information about metadata properties, see Composite in the API documentation.

  The following sample shows some of the available metadata fields with descriptions of their content and whether they are not used at run time. Required metadata are highlighted in bold.

```
{
  "name": "The component tag name",
  "version": "The component version. Note that changes to the metadata
even for minor updates like updating the jetVersion should result in at
least a minor Web Component version change, e.g. 1.0.0 -> 1.0.1.",
  "jetVersion": "The semantic version of the supported JET version(s). Web
Component authors should not specify a semantic version range that
includes unreleased JET major versions as major releases may contain non
backwards compatible changes. Authors should instead recertify Web
Components with each major release and update the component metadata or
release a new version that is compatible with the new release changes.",
  "description": "A high-level description for the component. Not used at
run time.",
  "displayName": "A user friendly, translatable name of the component. Not
used at run time.",

  "properties": {
    "property1": {
      "description": "A description for the property. Not used at run
time.",
      "displayName": "A user friendly, translatable name of the property.
Not used at run time.",
      "readOnly": "Boolean that determines whether a property can be
updated outside of the ViewModel. False by default.",
      "type": "The type of the property, following Google's Closure Compiler
syntax.",
      "value": "Object containing an optional default value for a
property.",
      "writeback": "Boolean that determines whether an expression bound to
this property should be written back to. False by default.",
      "enumValues": "An optional array of valid enum values for a string
property. An error is thrown if a property value does not match one of the
provided enumValues.",
      "properties": "A nested properties object for complex properties.
Subproperties exposed using nested properties objects in the metadata can
```

```
be set using dot notation in the attribute. See the Subproperties section for
more details on working with subproperties."
      },

    "property2": {
         ... contents omitted
      }
  },

  "methods": {
    "method1": {
      "description": "A description for the method. Not used at run time.",
      "displayName": "A user friendly, translatable name of the method.
Not used at run time.",
      "internalName": "An optional ViewModel method name that is different
from, but maps to this method.",
      "params": "An array of objects describing the method parameter . Not
used at run time.",
      "return": "The return type of the method, following Closure Compiler
syntax. Not used at run time."
     },
    "method2": {
     ... contents omitted
     }
  },

  "events": {
    "event1": {
      "bubbles": "Boolean that indicates whether the event bubbles up
through the DOM or not. Defaults to false. Not used at run time.",
      "cancelable": "Boolean that Indicates whether the event is
cancelable or not. Defaults to false. Not used at run time.",
      "description": "A description for the event. Not used at run time.",
      "displayName": "A user friendly, translatable name of the method.
Not used at run time.",
      "detail": {
        "field name": "Describes the properties available on the event's
detail property which contains data passed when initializing the event.
Not used at run time."
       }
     },
    "event2": {
     ... contents omitted
     }
   },

  "slots": {
    "slot1": {
      "description": "A description for the slot. Not used at run time.",
      "displayName": "A user friendly, translatable name of the method.
Not used at run time."
     }
   }
}
```

- HTML markup: (Required) Contains the View definition which describes how to render the Web Component. The Web Component's View has access to several `$` variables along with any public variables defined in the Web Component's ViewModel. Some of the variables that can be used in the Web Component's View are:

Variables	Description
`$properties`	A map of the Web Component's current property values
`$slotCounts`	A map of slot names containing a number of associated child nodes assigned to that slot
`$unique`	A unique string value provided for every component instance that can be used for unique ID generation
`$uniqueId`	The ID of the Web Component, if specified. Otherwise, it is the same as unique
`$props`	Deprecated since 5.0.0, use `$properties` instead
`$slotNodeCounts`	Deprecated since 5.0.0, use `$slotCounts` instead

- JavaScript: Optional script for defining the ViewModel and custom events.

  The ViewModel is also where you define callbacks for various stages of the Web Component's lifecycle. Web Components support the following optional lifecycle methods: `activated (context)`, `connected (context)`, `bindingsApplied (context)`, `propertyChanged (context)`, and `disconnected (element)`. For more information on lifecycle methods, see Composite - Lifecycle.

- CSS: Optional styling for the Web Component.

  CSS is not scoped to Web Components, and you must define styles appropriately.

- SCSS: Optional files containing Sass variables to generate the Web Component's CSS.

  If you're defining only a few styles for your component, then adding the CSS manually may be sufficient. However, there may be use cases where you want to use Sass variables to generate the CSS. In those cases, create and place the SCSS files in the Web Component's folder and use the tooling to add Dart Sass (`sass`) to your app. See Step 8 - Creating Web Components.

> **Important:**
>
> You must add the Sass files manually to the Web Component's folder. The tooling will compile any Sass files if they exist, but it will not create them for you.

Web Component Files

Web Components can contain CSS, HTML, JavaScript, and metadata files that you can modify according to your app requirements.

You can create a Web Component manually by creating a folder and adding the required files within the folder. You can also create a Web Component by using the Oracle JET CLI command `ojet create component <component-name>` that automatically generates the Web Component folder with the required files for your app.

When you create a Web Component manually, place the Web Component files in a folder with the same name as the Web Component tag. Typically, you place the folder within your app in a `jet-composites` folder: *app-path*`/jet-composites/my-web-component/`.

You can also place your Web Component in a different file location or reference a Web Component on a different server using RequireJS path mapping. For examples, see Composite - Packaging and Registration.

Each Web Component file should use the following naming convention to match the purpose:

- *my-web-component*`-view.html`: view template
- *my-web-component*`-viewModel.js`: ViewModel
- `component.json`: metadata
- *my-web-component*`-styles.css`: CSS styling
- *my-web-component*`-styles.scss`: Sass variables to generate CSS for Web Components
- `loader.js`: RequireJS module defining the dependencies for its metadata, View, ViewModel, and CSS This file should also include the Web Component registration.

Web Component Slotting

Slots are used as placeholders in a Web Component that users can fill in with their markup. Slot is defined in the component JSON file of your Web Component.

Use slotting to add child components (which can also be Web Components) that get slotted into specified locations within the Web Component's View markup. The following example contains a portion of the View markup for a Web Component named demo-columns.

```
<div class="oj-flex oj-flex-item-pad">
  <div role="group" :aria-label="[[$properties.headers[0]]]"
   class="oj-flex-item demo-columns-col-a oj-flex oj-sm-flex-direction-column
oj-sm-align-items-center">
    <h3>
            <oj-bind-text value="[[$properties.headers[0]]]"></oj-bind-text>
        </h3>
        <oj-bind-slot name="columnA">
        </oj-bind-slot>
     </div>
  ... content omitted
</div>
```

In this example, the demo-columns Web Component defines an `oj-bind-slot` named `columnA`. As shown below, a developer can specify a child component with a slot named `columnA` when adding the demo-columns Web Component to the page.

```
<demo-columns id="composite-container" headers='["Sales", "Human Resources",
"Support"]'>
  <!-- ko foreach: sales -->
    <demo-card slot="columnA" name="[[name]]" work-title="[[title]]"></demo-
card>
  <!-- /ko -->
  ... contents omitted
</demo-columns>
```

Web Component Template Slots

You can define placeholders in your template using template slots that can be filled with any markup fragment you want when the template element is used within a markup of your component.

When you need to reuse a stamped template with varying data, you can use a template slot to expose the additional data from the component's binding context.

You can define placeholders in your template using template slots that can be filled with any markup fragment you want when the template element is used within a markup of your component. When you need to reuse a stamped template with varying data, you can use a template slot to expose the additional data from the component's binding context.

Template slots for Web Components are used to define additional binding contexts for a slotted content within an app. To declaratively define a template slot, use the `oj-bind-template-slot` element in the Web Component's View markup for the slot that contains a stamped template DOM. The `oj-bind-template-slot` element is similar to the `oj-bind-slot` element, but its slotted content should be wrapped inside a `template` element within the app DOM.

In the below example, the demo-list Web Component defines an `oj-bind-template-slot` named `item`. This template slot provides the `data` attribute that exposes additional properties to the template DOM and an `data-oj-as` attribute that is used as an alias for the `$current` variable. Note that the `data-oj-as` attribute for `template` element can be referenced only inside a default template.

```
<table>
  <thead>
    <tr>
      <th>
        <oj-bind-text value="[[$properties.header]]"></oj-bind-text>
      </th>
    </tr>
  </thead>
  <tbody>
    <oj-bind-for-each data="{{$properties.data}}">
      <template>
        <tr>
          <td>
            <!-- Template slot for list items with default template and an
optional alias -->
            <oj-bind-template-slot name="item"
data="{{'value': $current.data}}">
              <!-- Default template -->
              <template data-oj-as="listItem">
                <span>
                 <oj-bind-text value='[[listItem.value]]'</oj-bind-text>
                </span>
              </template>
            </oj-bind-template-slot>
          </td>
        </tr>
      </template>
    </oj-bind-for-each>
  </tbody>
```

```
  ... contents omitted
</table>
```

The `oj-bind-template-slot` children are resolved when the Web Component View bindings are applied and are then resolved in the app's binding context extended with additional properties provided by the Web Component. These additional properties are available on the `$current` variable in the app provided template slot. The app can use an optional `data-oj-as` attribute as an alias in the template instead of the `$current` variable. The following example contains a portion of the app's markup named demo-list.

```
<demo-list data="{{groceryList}}" header="Groceries">
 <template slot="item" data-oj-as="groceryItem">
   <oj-checkboxset>
     <oj-option value="bought"><oj-bind-text
value='[[groceryItem.value]]'></oj-bind-text></oj-option>
   </oj-checkboxset>
 </template>
... contents omitted
</demo-list>
```

The Oracle JET Cookbook at Web Component - Template Slots includes complete examples for using template slots. oj-bind-template-slot API documentation describes the attributes and other template slot properties.

Web Component Events

Oracle JET Web Components can fire automatic property changed events that are mapped to the properties defined in the component metadata. These components will also fire custom events for the events delared in the component metadata.

Web Components can internally listen to the automatically generated `propertyChanged` events that are mapped to the properties in the component metadata using the `propertyChanged` lifecycle method. For example, a `propertyChanged` event is fired when a property is updated. This `propertyChanged` event contains the following properties:

- `property`: Name of the property that changed.
- `value`: Current value of the property.
- `previousValue`: Previous value of the property that changed.
- `updatedFrom`: The location from where the property was updated.
- `Subproperty`: An object holding information about the subproperty that changed.

When there is a need to declaratively define a custom event for a Web Component, you must declare the event in the component's metadata file. These events will only be fired if the code of the Web Component calls the `dispatchEvent()` method. The app can listen to these events by declaring the event listener attributes and property setters. These event listeners can be added declaratively or programmatically.

For the declarative specification of event listeners, use the `on-[event-name]` syntax for the attributes. For example, `on-click`, `on-value-changed`, and so on.

```
<oj-element-name value="{{currentValue}}" on-value-
changed="{{valueChangedListener}}"></oj-element-name>
```

The programmatic specification of event listeners may use the DOM `addEventListener` mechanism or by using the custom element property.

The DOM `addEventListener` mechanism uses the `elementName.addEventListener` method. For example:

```
elementName.addEventListener("valueChanged", function(event) {...});
```

The custom element property uses the `elementName.onEventName` syntax for the property setter. For example:

```
elementName.onValueChanged = function(event) {...};
```

For more information, see the Web Components - Events and Listeners API documentation.

Web Component Examples

Web Components can contain slots, data binding, template slots, nested Web Components, and events. You can use the examples provided in the Oracle JET Cookbook for these Web Component features.

The Oracle JET Cookbook contains complete examples for creating basic and advanced Web Components. You can also find examples that use slotting and data binding. For details, see Web Component - Basic.

For additional information about Web Component fields and methods, see Composite in the API documentation.

Best Practices for Web Component Creation

Best practices for creating Oracle JET Web Components include required and recommended patterns, configuration, coding practices, version numbering, and styling standards. Follow best practices to ensure interoperability with other Web Components and consuming frameworks.

Recommended Standard Patterns and Coding Practices

Recommended patterns and coding practices for Oracle JET Web Components include standards for configuration, versioning, coding, and archival.

Component Versioning

Your Web Component must be assigned a version number in semantic version format.

When assigning and incrementing the version number associated with your components, be sure to follow semantic version rules and update Major, Minor and Patch version numbers appropriately. By doing so, component consumers will have a clear understanding about the compatibility and costs of migrating between different versions of your component.

To assign a version number to your Web Component, see About semantic versioning.

JET Version Compatibility

You must use the semantic version rules to specify the `jetVersion` of the supported JET version(s). Web Component authors should not specify a semantic version range that includes

unreleased JET major versions as major releases may contain non backwards compatible changes. Authors should instead recertify Web Components with each major release and update the component metadata or release a new version that is compatible with the new release changes.

Translatable Resources

Developers who want to localize Web Component translatable resources now get a resource bundle (template) when they create their Web Component. These components should use the standard Oracle JET mechanism using the `ojL10n requireJS` plugin. You must store the translation bundles in the *`webcomponentname`*`/resources/nls` subdirectory that is a peer to the *`webcomponentname`*`/resources/nls/root` subdirectory with the resource strings for your Web Component's folder. You can declare the languages and locales that you support in the Web Component metadata.

Peer-to-Peer Communication

Components must prefer a shared observable provided by the consumer over any kind of secret signaling mechanism when you are dealing with a complex integration. For example, a filter component and a data display component. By using a shared observable you can pre-seed and programmatically interact with the components through the filter.

Alternatively, you can use events and public methods based on one of the following approaches being used:

- A hierarchical relationship between the source and receiver of the event.
- The identity of the source being passed to the receiver.

  Note that in some runtime platforms, the developer doing the wiring may not have access to component IDs to pass the relevant identity.
- Listeners attached by components at the document level.

  In this case, you are responsible for the cleanup of those listeners, management of duplicates, and so on. Also, such listeners should preferably be based on Web Component events, not common events such as click, which might be overridden by intermediate nodes.

> **Note:**
>
> Under the web-component standards (shadow DOM), events will be re-targeted as they transition the boundary between the component and the consuming view. That is, the apparent identity of the raising element might be changed, particularly in the case of Nested Web Component architecture where the event would get tagged with the element representing the outer Web Component rather than the inner Web Component. Therefore, you should not rely on the `event.target` attribute to identify the Web Component source when listening at the document level. Instead, the `event.deepPath` attribute can be used to understand the actual origin of the event.

Access to External Data

Web Components do not permit the usage of the knockout binding hierarchy to obtain data from outside the Web Component context, for example, `$root`, `$parent[1]`, and so on. All data transfer in and out of the component must be through the formal properties, methods, and events.

Object Scope

All properties and functions of Web Components should be confined to the scope of the view model. No window or global scope objects should be created. Similarly, the existence of window scope objects should not be assumed by the Web Component author. If a consumer Web Component defined externally at window or global level is required for read or write then that component must be passed in by the consuming view model through a formal property. Even if a well known global reference is needed from outside of the component, it should be formally injected using the require `define()` function and declared as a dependency in the Web Component metadata.

External References

If a Web Component must reference an external component, it should be part of the formal API of the component. The formal API passes the component reference through a property. For example, to allow the registration of a listener, the Web Component code requires a component reference defined externally. You must not allow Web Components to obtain IDs from hard-coded values, global storage, or walking the DOM.

Subcomponent IDs

Within the framework if any component needs a specific ID, use `context.unique` or `context.uniqueId` value to generate the ID. This ID is unique within the context of the page.

ID Storage

Any generated IDs should not be stored across invocation, such as in local storage or in cookies. The `context.unique` value for a particular Web Component may change each time a particular view is executed.

LocalStorage

It is difficult to consistently identify a unique instance of a Web Component within an app. So, it is advised not to allow a Web Component to utilize the local storage of a browser for persisting information that is specific to an instance of that Web Component. However, if the app provides a unique key through the public properties of the component you can then identify the unique instance of the component.

Additionally, do not use local storage as a secret signaling mechanism between composites. You cannot assure the availability of the capability and so it is recommended to exchange information through a shared JavaScript object or events as part of the public API for the component(s).

String Overrides

Web Components will often contain string resources internally to service their default needs for UI and messages. However, sometimes you may want to allow the consumer to override these strings. To do this, expose a property for this purpose on the component. By convention such a property would be called `translations`, and within it you can have sub-properties for each translatable string that relates to a required property on the component, for example `requiredHint`, `requiredMessageSummary`, and so on. These properties can then be set on the component tag using sub-property references. For example:

```
"properties" : {
  "translations": {
    "description": "Property to allow override of default messages",
    "writeback" : false,
```

```
    "properties" : {
      "requiredHint": {
        "description": "Change the text of the required hint",
        "type": "string"
         },
      "requiredMessageSummary": {
        "description": "...",
        "type": "string"
        },
      "requiredMessageDetail": {
          "description": "...",
          "type": "string"
        }
      }
     }
   }
}
```

Logging

Use `Logger` to write log messages from your code in preference to `console.log()`. The Web Components should respect the logging level of the consuming app and not change it. You should ideally prefix all log messages emitted from the component with an identification of the source Web Component. As a preference, you can use the Web Component name. The components should not override the writer defined by the consuming app.

Expensive Initialization

Web Components should carry out minimum work inside the constructor function. Expensive initialization should be deferred to the `activated` lifecycle method or later. The constructor of a Web Component is invoked even if the component is not actually added to the visible DOM. For example, if a constructor is invoked within a Knockout `if` block. The further lifecycle phases will only occur when the component is actually needed.

Service Classes

The use of global service classes, that is functionality shared across multiple Web Components, can constitute an invisible contract that the consumer of your Web Component has to know about. To avoid this, we recommend:

- Create the service as a module that every Web Component can explicitly set it as `require()` block, thus removing the need for the consumer to do this elsewhere.
- Consider the timing issues that might occur if your service class needs some time to initialize, for example fetching data from a remote service. In such cases, you should be returning promises to the service object so that the components can safely avoid trying to use the information before it is actually available.

Using ojModule

If you use `ojModule` in a Web Component and plan to distribute the Web Component outside of your app, you must take additional steps to ensure that the contained `ojModule` could be loaded from the location relative to the location of the Web Component. Unless the View and ViewModel instances are being passed to `ojModule` directly, you will need to provide the require function instance and the relative paths for views and view models. The require

function instance should be obtained by the component loader module by specifying `require` as a dependency.

```
<div data-bind="ojModule: {require: {instance: require_instance, viewPath:
"path_to_Web_Component_Views", modelPath:
"path_to_cWeb_Component_ViewModels"}}"></div>
```

`require` Option	**Type**	**Description**
`instance`	Function	Function defining the require instance
`viewPath`	String	String containing the path to the Web Component's Views
`modelPath`	String	String containing the path to the Web Component's ViewModels

For additional information about working with `ojModule`, see ojModule.

Archiving Web Components for Distribution

If you want to create a zip file for packaging, create an archive with the same name as the component itself. You may add version-identifying suffixes to the zip file name for operational reasons. The Web Component artifacts must be placed in the root of the zip file, and there should be no intermediate directory structure before reaching the files.

Using Lifecycle Methods

If a ViewModel is provided for a Web Component, the following optional callback methods can be defined on its ViewModel that will be called at each stage of the Web Component's lifecycle. Some of the callback methods that can be used are listed below:

- `activated(context)`: Invoked after the ViewModel is initialized.
- `connected(context)`: Invoked after the View is first inserted into the DOM and then each time the Web Component is reconnected to the DOM after being disconnected.
- `bindingsApplied(context)`: Invoked after the bindings are applied on the View.
- `propertyChanged(context)`: Invoked when properties are updated before the `[property]Changed` event is fired.
- `disconnected(element)`: Invoked when this Web Component is disconnected from the DOM.

For additional information on Web Component lifecycle methods, see Composite - Lifecycle.

Template Aliasing

JET components that support inline templates can now use an optional `data-oj-as` attribute to provide template specific aliases for `$current` variable at the app level. In the instances where the component must support multiple template slots as in the case of chart and table components, a single alias may not be sufficient. In such cases, you can use an optional `data-oj-as` attribute on the template element. For more information on the usage of this optional attribute with template slots, see oj-bind-template-slot API documentation.

CSS and Theming Standards

Oracle JET Web Components should comply with all recommended styling standards to ensure interoperability with other Web Components and consuming apps.

For information on the generic best practices for using CSS and Themes, see Best Practices for Using CSS and Themes.

Standard	Details	Example

Prevent flash of unstyled content	Oracle JET will add the `oj-complete` class to the Web Component DOM element after metadata properties have been resolved. To prevent a flash of unstyled content before the component properties have been setup, the component's CSS should include a rule to hide the component until the `oj-complete` class is set on the element.
	Note that this is an element selector, and there should *not* be a dot (.) before `acme-branding`.
		`acme-branding:not(.oj-complete) {`
		`  visibility: hidden;`
		`}`

Add scoping	Use an element selector to minimize the chance that one of your classes is used by someone outside of your component and becomes dependent on your internal implementation. In the example to the right, if someone tries to apply the class `acme-branding-header`, it will have no effect if it's not within an `acme-branding` tag.	`acme-branding .acme-branding-header {`
		`  color: white;`
		`  background: blue;`
		`}`
		
		IMPORTANT: If your component also includes a dialog, then when displayed, that dialog will be attached to the main document DOM tree and will not be a child of your component. Therefore, if you define a style to apply to a dialog defined by your component, you cannot scope it to the component name as that's not the actual container for the dialog when displayed.
		To resolve, use the component name as the prefix instead:
		
		`.acme-branding-dialog-background{`
		`  color: white;`
		`  background: blue;`
		`}`

Avoid element styling	The app will often style HTML tag elements like headers, links, and so on. In order to blend in with the app, avoid styling these elements in your Web Component.	⚠ Avoid styling on elements like headers.
		
		`acme-branding .acme-branding-header h3{`
		`  font-size: 45px;`
		`}`

Version Numbering Standards

All types of Web Components, including standalone, JET Pack, and Resource components, require a version number and that number should adhere to a semantic versioning (SemVer) scheme that ensures a standard for development teams to follow similar to the approach adopted by Oracle JET release versioning.

Reference components are a slightly special case as the version of the reference component will always match the version of the NPM library that it references.

All other Web Component types, rely on semantic versioning to designate a version of the component. Semantic versioning defines a version number which has three primary segments and an optional fourth segment. The first three segments are defined as MAJOR.MINOR.PATCH with these meanings:

- MAJOR version when you make incompatible API changes
- MINOR version when you add functionality in a backward compatible manner
- PATCH version when you make backward compatible bug fixes

> **Note:**
>
> For background on semantic versioning, visit https://semver.org.

When defining a version number for your Web Component, you must define all three of these core segments:

```
1.0.0
```

Additionally after the PATCH version, you can append an optional extension segment which can consist of two additional pieces of information:

- A segment delimited by a leading hyphen which allows you to assign a pre-release version
- A purely informational segment preceded by a plus sign (+) that you might use to hold a GIT commit hash or other control information. This segment plays no part in the comparison of component versions.

Here's an example of a fully-defined version number for a pre-release version of a component:

```
1.0.1-beta.1+332
```

In this case `beta.1` is a pre-release indicator and `332` is a build number.

The change in version number for a Web Component should indicate to consumers the risk level of consuming that new version. Consumers should know that they can drop in a new MINOR or PATCH release of your components without needing to revise their code. If you make changes to the Web Component source code that forces the consuming app to do more than just refresh the Web Component's directory or change a CDN reference, then you should revise the MAJOR version to indicate this.

The Web Component metadata file `component.json` lets you define the supported version of Oracle JET that the component can work with. This is the `jetVersion` attribute which can be

set to a specific version or version range, as defined by npm-semver. For example, the `jetVersion` attribute will be set to one of the following:

- Preferred: All MINOR and PATCH versions from the specified version forward until there is a change in MAJOR release number. For example: `"jetVersion:: "^9.1.0"`, , which indicates support for that release and all subsequent MINOR and PATCH versions, up to (but not including) the next MAJOR release and it is equivalent to `">=9.1.0 <10.0.0"`.
- The exact semantic version of exact version of Oracle JET that the Web Component supports. For example: `"jetVersion" : "9.1.0"`, which implies that this Web Component supports only Oracle JET 9.1.0.
- All PATCH versions of Oracle JET within a specific MAJOR.MINOR release. For example: `"jetVersion" : "9.0.x"`, which implies that this Web Component supports every release of JET between JET 9.0.0 and JET 9.1.0 (but specifically not JET 9.1.0 itself).
- A specific range of Oracle JET versions. For example: `"jetVersion" : "9.0.0 -9.1.0"`, which implies that this Web Component supports every release of JET between JET 9.0.0 and JET 9.1.0 inclusive (so not including, for example, JET 9.2.0 or JET 8.0.0).

Tip:

The Oracle JET recommended format is the first case defined similar to `"^9.1.0"`. Given that JET itself follows the semantic versioning rules, changes that occur in MINOR or PATCH versions ought not break your Web Component source code. This also means that you don't have to release an update to all your Web Components for every MINOR release of Oracle JET (unless you choose to make use of a new feature).

Note:

For background on npm-semver, visit npm documentation at this web site https://docs.npmjs.com/about-semantic-versioning.

In the case of JET Packs, you should pay attention to the `dependencies` attribute in the `component.json` file located in the pack. You should strive to require the various components bundled into the pack to be consumed together, and as such, you should define such dependencies with absolute version matches rather than version ranges to indicate this. For example, in this case the demo-memory-game component at version `1.0.2` will only expect to host a demo-memory-card at exactly version `1.0.2`.

```
{
  "name": "demo-memory-game",
  "version": "1.0.2",
  "type": "pack",
  "displayName": "JET Memory Game",
  "description": "A memory game element with a configurable card set and
attempt counter.",
  "dependencies": {
    "demo-memory-card": "1.0.2"
    }
}
```

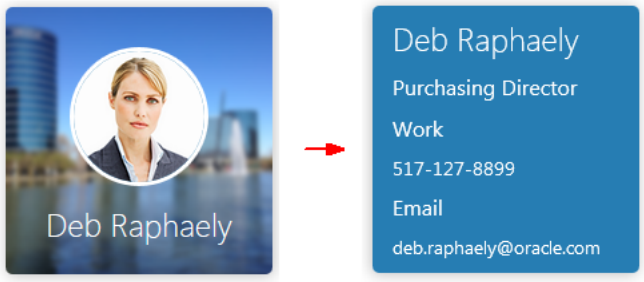
Create Web Components

Oracle JET supports a variety of custom Web Component component types. You can create standalone Web Components or you can create sets of Web Components that you intend to be used together and you can then assemble those in a JET Pack, or pack component type. You can enhance JET Packs by creating resource components when you have re-usable libraries of assets that are themselves not specifically UI components. And, if you need to reference third-party code in a standalone component, you can create reference components to define pointers to that code.

Create Standalone Web Components

Use the Oracle JET command-line interface (CLI) to create an Oracle JET Web Component template implemented as JavaScript or TypeScript that you can populate with content. If you're not using the tooling, you can add the Web Component files and folders manually to your Oracle JET app.

The following image shows a simple Web Component named `demo-card` that displays contact cards with the contact's name and image if available. When the user selects the card, the content flips to show additional detail about the contact.



The procedure below lists the high level steps to create a Web Component using this `demo-card` component as an example. Portions of the code are omitted for the sake of brevity, but you can find the complete example at Web Component - Basic. You can also download the demo files by clicking the download button ( ) in the cookbook.

Before you begin:

- Familiarize yourself with the steps to install the Oracle JET CLI, see Install Oracle JET Tooling.
- Familiarize yourself with the steps to add a Web Component to an Oracle JET app using the Oracle JET CLI, see Understand the Web App Workflow.
- Familiarize yourself with the list of reserved names for a Web Component that are not available for use, see valid custom element name.
- Familiarize yourself with the list of existing Web Component properties, events, and methods, see HTMLElement properties, event listeners, and methods.
- Familiarize yourself with the list of global attributes and events, see Global attributes.

To create a Web Component:

1. Determine a name for your Web Component.

   The Web Component specification restricts custom element names as follows:

   - Names must contain a hyphen.
   - Names must start with a lowercase ASCII letter.
   - Names must not contain any uppercase ASCII letters.
   - Names should use a unique prefix to reduce the risk of a naming collision with other components.

     A good pattern is to use your organization's name as the first segment of the component name, for example, *org-component-name*. Names must not start with the prefix `oj-` or `ns-`, which correspond to the root of the reserved `oj` and `ns` namespaces.
   - Names must not be any of the reserved names. Oracle JET also reserves the `oj` and the `ns` namespace and prefixes.

   For example, use `demo-card` to duplicate the contact card example.

2. Determine where to place your Web Component, using one of the following options.

   - Add the Web Component to an existing Oracle JET app that you created with the Oracle JET CLI.

     If you use this method, you'll use the CLI to create a Web Component template that contains the folders and files you'll need to store the Web Component's content.
   - Manually add the Web Component to an existing Oracle JET app that doesn't use the Oracle JET CLI.

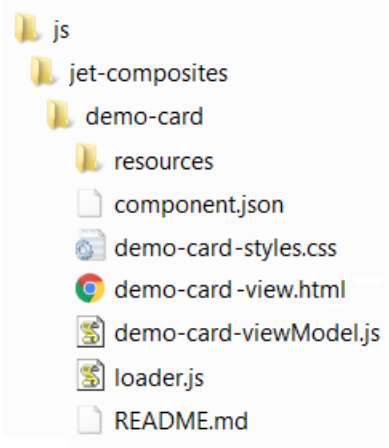
     If you use this method, you'll create the folders and files manually to store the Web Component's content.

3. Depending upon the choice you made in the previous step, perform one of the following tasks to create the Web Component.

   - If you used the Oracle JET CLI to create an app, then in the app's top level directory, enter the following command at a terminal prompt to generate the Web Component template:

     ```
     ojet create component component-name
     ```

     For example, enter `ojet create component demo-card` to create a Web Component named `demo-card` in a base app created with JavaScript. The command will add `jet-composites/demo-card` to the app's `js` folder and files containing stub content for the Web Component.



The base app's implementation of JavaScript or TypeScript determines the folder and implementation of the Web Component. If your base app has a TypeScript implementation, then the `ts` folder, not the `js` folder, contains the stub contents of the Web Component.

- If you're not using the Oracle JET CLI, create a `jet-composites` folder in your app's `js` folder or `ts` folder, and add folders containing the name of each Web Component you will create.

  For the `demo-card` example, create the `jet-composites` folder and add a `demo-card` folder to it. You'll create the individual Web Component files in the remaining steps.

4. Determine the properties, methods, and events that your Web Component will support and add them to the `component.json` file in the Web Component's root folder, creating the file if needed.

   The name of the Web Component properties, event listeners, and methods should avoid collision with the existing HTML element properties, event listeners, and methods. Additionally, the property name `slot` should not be used. Also, you must not re-define the global attributes and events.

   The demo-card example defines properties for the Web Component and the contact's full name, employee image, title, work number, and email address. The required properties are highlighted in bold.

```
 {
  "name": "demo-card",
  "description": "A card element that can display an avatar or initials on
one side and employee information on the other.",
  "version": "1.0.2",
  "displayName": "Demo Card",
  "jetVersion": ">=6.0.0 <18.1.0",
   "properties": {
    "name": {
      "description": "The employee's full name.",
      "type": "string"
    },
    "avatar": {
      "description": "The url of the employee's image.",
      "type": "string"
    },
    "workTitle": {
```

```
      "description": "The employee's job title.",
      "type": "string"
    },
    "workNumber": {
      "description": "The employee's work number.",
      "type": "number"
    },
    "email": {
      "description": "The employee's email.",
      "type": "string"
    }
  }
}
```

This basic demo-card example only defines properties for the Web Component. You can also add metadata that defines methods and events as shown below. The metadata lists the name of the method or event and supported parameters.

```
{
  "properties": {
    ... contents omitted
  },
  "methods": {
     "flipCard": {
       "description": "Method to toggle flipping a card"
     },
     "enableFlip": {
       "description": "Enables or disables the ability to flip a card.",
       "params": [
         {
           "name": "bEnable",
           "description": "True to enable card flipping and false
otherwise.",
           "type": "boolean"
         }
       ]
     },
   },
   "events": {
     "cardClick": {
       "description": "Triggered when a card is clicked and contains the
value of the clicked card..",
       "bubbles": true,
       "detail": {
         "value": {
           "description": "The value of the card.",
           "type": "string"
                }
          }
        }
    }
}
```

5. If your Web Component contains a ViewModel, add its definition to *web-component-name*`-viewModel.js` in the Web Component's root folder, creating the file if needed.

The code sample below shows the ViewModel for the demo-card Web Component. Comments describe the purpose, parameters, and return value of each function.

```
define(['knockout', 'ojs/ojknockout'],
    function(ko) {
        function model(context) {
            var self = this;
            self.initials = null;
            self.workFormatted = null;
            var element = context.element;

            /**
             * Formats a 10 digit number as a phone number.
             * @param  {number} number The number to format
             * @return {string}        The formatted phone number
             */
            var formatPhoneNumber = function(number) {
                return Number(number).toString().replace(/(\d{3})(\d{3})
(\d{4})/, '$1-$2-$3');
            }

            if (context.properties.name) {
                var initials = context.properties.name.match(/\b\w/g);
                self.initials = (initials.shift() +
initials.pop()).toUpperCase();
            }
            if (context.properties.workNumber)
                self.workFormatted =
formatPhoneNumber(context.properties.workNumber);

            /**
             * Flips a card
             * @param  {MouseEvent} event The click event
             */
            self.flipCard = function(event) {
                if (event.type === 'click' || (event.type === 'keypress'
&& event.keyCode === 13)) {
                    // It's better to look for View elements using a
selector
                    // instead of by DOM node order which isn't guaranteed.
                    $(element).children('.demo-card-flip-
container').toggleClass('demo-card-flipped');
                }
            };
        }

        return model;
    }
)
```

6. In the Web Component's root folder, add the View definition to *web-component-name*-view.html, creating the file if needed.

The View for the demo-card Web Component is shown below. Any property defined in the component's metadata is accessed using the `$properties` property of the View binding context.

```
<div tabindex="0" role="group" class="demo-card-flip-container"
  on-click="[[flipCard]]" on-keypress="[[flipCard]]" :aria-
label="[[$properties.name + ' Press Enter for
more info.']]">
  <div class="demo-card-front-side">
    <oj-avatar class="demo-card-avatar" role="img" size="lg"
initials="[[initials]]"
src="[[$properties.avatar]]" :aria-label="[['Avatar of '
+ $properties.name]]">
    </oj-avatar>
    <h2>
      <oj-bind-text value="[[$properties.name]]"></oj-bind-text>
    </h2>
  </div>

  <div class="demo-card-back-side">
    <div class="demo-card-inner-back-side">
      <h2>
              <oj-bind-text value="[[$properties.name]]"></oj-bind-text>
      </h2>
      <h5>
        <oj-bind-text value="[[$properties.workTitle]]"></oj-bind-text>
      </h5>
      <oj-bind-if test="[[$properties.workNumber != null]]">
        <h5>Work</h5>
        <span class="demo-card-text"><oj-bind-text
value="[[workFormatted]]"></oj-bind-text></span>
      </oj-bind-if>
      <oj-bind-if test="[[$properties.email != null]]">
        <h5>Email</h5>
        <span class="demo-card-text"><oj-bind-text
value="[[$properties.email]]"></oj-bind-text></span>
      </oj-bind-if>
    </div>
  </div>
</div>
```

For accessibility, the View's role is defined as `group`, with `aria-label` specified for the contact's name. In general, follow the same accessibility guidelines for the Web Component View markup that you would anywhere else within the app.

7. If you're not using the Oracle JET CLI, create the `loader.js` RequireJS module and place it in the Web Component's root folder.

   The `loader.js` module defines the Web Component dependencies and registers the component's `tagName`, `demo-card` in this example.

```
define(['ojs/ojcomposite', 'text!./demo-card-view.html', './demo-card-
viewModel',
        'text!./component.json', 'css!./demo-card-styles'],
  function(Composite, view, viewModel, metadata) {
```

```
    Composite.register('demo-card', {
      view: view,
      viewModel: viewModel,
      metadata: JSON.parse(metadata)
    });
  }
);
```

In this example, the CSS is loaded through a RequireJS plugin (`css!./demo-card-styles`), and you do not need to pass it explicitly in `Composite.register()`.

8. Configure any custom styling that your Web Component will use.

   - If you only have a few styles, add them to *web–component-name*`-styles.css` file in the Web Component's root folder, creating the file if needed.

     For example, the demo-card Web Component defines styles for the demo card's display, width, height, margin, padding, and more. It also defines the classes that will be used when the user clicks a contact card. A portion of the CSS is shown below.

     ```
     /* This is to prevent the flash of unstyled content before the Web
     Component properties have been setup. */
     demo-card:not(.oj-complete) {
       visibility: hidden;
     }

     demo-card {
       display: block;
       width: 200px;
       height: 200px;
       perspective: 800px;
       margin: 10px;
       box-sizing: border-box;
       cursor: pointer;
     }

     demo-card h2,
     demo-card h5,
     demo-card a,
     demo-card .demo-card-avatar  {
       color: #fff;
       padding: 0;
     }
      ... remaining contents omitted
     ```

   - If you used the Oracle JET tooling to create your app and want to use Sass to generate your CSS:

     a. If needed, at a terminal prompt in your app's top level directory, type the following command to add Dart Sass (`sass`) to your app: `ojet add sass`.

     b. Create *web–component-name*`-styles.scss` and place it in the Web Component's top level folder.

     c. Edit *web–component-name*`-styles.scss` with any valid SCSS syntax and save the file.

In this example, a variable defines the demo card size:

```
$demo-card-size: 200px;

/* This is to prevent the flash of unstyled content before the Web
Component properties have been setup. */
demo-card:not(.oj-complete) {
  visibility: hidden;
}

demo-card {
  display: block;
  width: $demo-card-size;
  height: $demo-card-size;
  perspective: 800px;
  margin: 10px;
  box-sizing: border-box;
  cursor: pointer;
}

demo-card h2,
demo-card h5,
demo-card a,
demo-card .demo-card-avatar  {
  color: #fff;
  padding: 0;
}
... remaining contents omitted
```

d. To compile Sass, at a terminal prompt type `ojet build` or `ojet serve` with the `--sass` flag and app-specific options.

```
ojet build|serve [options] --sass
```

`ojet build --sass` will compile your app and generate *web–component-name*`-styles.css` and *web–component-name*`-styles.css.map` files in the default platform's folder. For a web app, the command will place the CSS in **web**`/js/js-composites/`*web–component-name*.

`ojet serve --sass` will also compile your app but will display the web app in a running browser with `livereload` enabled. If you save a change to *web–component-name*`-styles.scss` in the app's **src**`/js/jet-composites/`*web–component-name* folder, Oracle JET will compile Sass again and refresh the display.

> **Tip:**
>
> For help with `ojet` command syntax, type `ojet help` at a terminal prompt.

9. If you want to add documentation for your Web Component, add content to `README.md` in your Web Component's root folder, creating the file if needed.

Your `README.md` file should include an overview of your component with well-formatted examples. Include any additional information that you want to provide to your component's consumers. The recommended standard for `README` file format is markdown.

For help with markdown, refer to GitHub documentation.

For the complete code of the demo-card Web Component CSS styles, see `demo-card-styles.css` in the Web Component - Basic cookbook sample.

For information on adding Web Component metadata that defines methods and events, see the Web Component - Events cookbook sample.

Create JET Packs

Create JET Packs to simplify project management for consumers who might pick up a component that is related to one or more components. You may require specific versions of the referenced components for individual JET Packs.

Fundamentally, the JET Pack is a library of related Web Components that does not directly include those assets, but is as an index to a particular versioned stripe of components.

> **Note:**
>
> Note there is one exception to the pack as a reference mechanism for related components. A pack might include one or more RequireJS bundle files which package up optimized forms of the component set into a small number of physical downloads. This, however, is always in addition to the actual components being available as independent entities in Oracle Component Exchange.

The components referenced by the JET Pack are intended to be used together and their usage is restricted by individual component version. Thus, the JET Pack that you create will tie very specific versions of each component into a relationship with very specific, fixed versions of the other components in the same set. Thus, a JET Pack itself has a "version stripe" which determines the specific components that users import into their apps. Since the version number of individual components may vary, the JET Pack guarantees the consumer associates their app with the version of the pack as a whole, and not with the individual components contained by the pack.

For details about versioning JET Packs, see Version Numbering Standards.

1. Create the JET Pack using the JET tooling from the root folder of your app.

   ```
   ojet create pack my-pack
   ```

   Consider the pack name carefully, as the name will determine the prefix to any components within that pack.

   The tooling adds the folder structure with the template files that you will need to modify:

   ```
   /(working folder)
     /src
       /js
         /jet-composites
   ```

```
        /my-pack
          component.json
```

2. Create the components that you want to bundle with the JET Pack by using the JET tooling from the root folder of your app. The component name that you specify must be unique within the pack.

   ```
   ojet create component my-widget-1 --pack=my-pack
   ```

   The tooling nests the component folder `/my-widget-1` under the `my-pack` root folder and the new component files resemble those created for a standalone Web Component.

   ```
   /(working folder)
     /src
       /js
         /jet-composites
           /my-pack
             component.json
             /my-widget-1
               /resources
                 /nls
                   /root
                     my-widget-1-strings.js
               component.json
               loader.js
               README.md
               my-widget-1-viewModel.js
               my-widget-1-styles.css
               my-widget-1-view.html
   ```

   Note the following about the created component files.

   - `component.json` specifies the **name** of the component as **my-widget-1** and the **pack** is set to **my-pack**, providing the complete definition of the component's identity.
   - `loader.js` registers the HTML tag for the new component as **my-pack-my-widget-1**. This is the full name of the component, which is a concatenation of the pack name and the component name. When you need to refer to this component in a dependency from another component's metadata or in HTML, you will use the full name.

3. Optionally, for any Resource components that you created, as described in Create Resource Components for JET Packs, add the component's working folder with its own `component.json` file to the pack file structure.

   The tooling nests the component folder `/my-widget-1` under the `my-pack` root folder and the new component files resemble those created for a standalone Web Component.

   ```
   /(working folder)
     /src
       /js
         /jet-composites
           /my-pack
             component.json
             /my-widget-1
               /resources
                 /nls
   ```

```
        /root
          my-widget-1-strings.js
    component.json
    loader.js
    README.md
    my-widget-1-viewModel.js
    my-widget-1-styles.css
    my-widget-1-view.html
  /my-resource-component-1
    component.json
    /converters
      file1.js
      ...
    /resources
      /nls
        /root
          strings-file.js
    /validators
      file1.js
      ...
```

4. Optionally, generate any required bundled for desired components of the pack. Refer to RequireJS documentation for details at the https://requirejs.org web site.

> **Tip:**
>
> You can use RequireJS to create optimized bundles of the pack components, so that rather than each component being downloaded separately by the consuming app at runtime, instead a single JavaScript file can be downloaded that contains multiple components. It's a good idea to use this facility if you have sets of components that are almost always used together. A pack can have any number of bundles (or none at all) in order to group the available components as required. Be aware that not every component in the pack has to be included in one of the bundles and that each component can only be part of one bundle.

5. Use a text editor to modify the `component.json` file in the pack folder root similar to the following sample, to identify pack dependencies and optional bundles. Added components must be associated by their full name and a specific version.

```
{
    "name": "my-pack",
    "version": "1.0.0",
    "type": "pack",
    "displayName": "My JET Pack",
    "description": "An example JET Pack",
    "dependencies": {
      "my-pack-my-widget-1":"1.0.0",
      ...
    },
    "bundles":{
        "my-pack/my-bundle":[
          "my-pack/my-bundle-file1/loader",
          ...
        ]
```

```
    },
   "extension": {
       "catalog": {
           "coverImage": "coverimage.png"
       }
    }
  }
}
```

Your pack component's `component.json` file must contain the following unique definitions:

- **name** is the name of the JET Pack has to be unique, and should be defined with the namespace relevant to your group. This name will be prepended to create the full name of individual components of the pack.
- **version** defines the exact version number of the pack, not a SemVer range.

> **Note:**
>
> Changes in version number with a given release of a pack should reflect the most significant change in the pack contents. For example, if the pack contained two components and as part of a release one of these had a Patch level change and the other a Major version change then the pack version number change should also be a Major version change. There is no requirement for the actual version number of the pack to match the version number(s) of any of it's referenced components. For more information see, Version Numbering Standards.

- **type** must be set to `pack`.
- **displayName** is the name of the pack component that you want displayed in in Oracle Component Exchange. Set this to something readable but not too long.
- **description** is the description that you want displayed in Oracle Component Exchange. For example, use this to explain how the pack is intended to be used.
- **dependencies** defines the set of components that make up the pack, specified by the component full name (a concatenation of pack name and component name). Note that exact version numbers are used here, not SemVer ranges. It's important that you manage revisions of dependency version numbers to reflect changes to the referenced component's version and also to specify part of the path to reach the components within the pack.

  If you want to include all components in the JET Pack directory, use a token, `"@dependencies@"`, as the value for `dependencies` rather than defining individual entries for all the components in the pack. The following snippet illustrates how you use this token in your `component.json` file:

```
{
  "name": "my-pack",
  "version": "1.0.0",
  "type": "pack",
  "displayName": "My JET Pack",
  "description": "An example JET Pack",
  "dependencies": "@dependencies@"
}
```

- **bundles** defines the available bundles (optional) and the contents of each. Note how both the bundle name and the contents of that bundle are defined with the pack name prefix as this is the RequireJS path that is needed to map those artifacts.
- **catalog** defines the working metadata for Oracle Component Exchange, including a cover image in this case.

6. Use a text editor to modify the `component.json` file in the component folder root similar to the following sample, to identify the pack relationship. Components must be identified by a unique name (without the pack prefix) and a specific version.

```
{
    "name": "my-widget-1",
    "pack": "my-pack",
    "displayName": "My Web Component",
    "description": "Fully featured component",
    "version": "1.0.0",
    "jetVersion": "^10.0.0",
    "dependencies": {
      "my-widget-file1":"^1.0.0",
      ...
    },
    ...
}
```

Your pack component's `component.json` file must contain the following unique definitions:

- **name** is the name of the component has to be unique. This name will be prepended with the pack name to create the component full name. For example,
- **pack** is the name of the JET Pack that the component is a part of.
- **displayName** is the name of the component that you want displayed in Oracle Component Exchange.
- **description** is the description that you want displayed in Oracle Component Exchange. For example, use this to explain the role of the components in the pack as they are intended to be used.
- **version** defines the exact version number of the component, not a SemVer range.
- **jetVersion** defines the compatible version(s) of Oracle JET, specified by a semantic version (SemVer) range. It's important that you manage revisions of this version number to inform consumers of the compatibility of a given change and also to specify part of the path to reach the components within the pack. For more information see, Version Numbering Standards.
- **dependencies** defines the set of libraries and other components that make up the component within the pack. In the case where a dependent component is also listed as a member of the JET Pack, specify components here by their full name (a concatenation of pack name and component name). For example, `my-pack-my-component`. Note that SemVer ranges are allowed. It's important that the range selected for a component within a particular JET Pack version overlaps with the members of that stripe.

7. Optionally, create a readme file in the root of your working folder. This should be defined as a plain text file called `README.txt` (or `README.md` when using markdown format).
8. Optionally, create a cover image in the root of your working folder to display the component on Oracle Exchange. The file name can be the same as the name attribute in the `component.json` file.

9. Use the JET tooling to create a zip archive of the JET Pack working folder when you want to upload the component to Oracle Component Exchange, as described in Package Web Components.
10. Support consuming the JET Pack in Oracle Visual Builder projects by uploading the component to Oracle Component Exchange, as described in Publish Web Components to Oracle Component Exchange.

Create Resource Components for JET Packs

Create a resource component when you want to reuse assets across web components that you assemble into JET Packs. The resource component can be reused by multiple JET Packs.

When dealing with complex sets of components you may find that it makes sense to share certain assets between multiple components. In such cases, the components can all be included into a single JET Pack and then a resource component can be added to the pack in order to hold the shared assets. There is no constraint on what can be stored in a pack, typically it may expose shared JavaScript, CSS, and JSON files and images. Note that third party libraries should generally be referenced from a reference component and should not be included into a resource component.

You don't need any tools to create the resource component. You will need to create a folder in a convenient location. This folder will ultimately be zipped to create the distributable resource component. Internally this folder can then hold any content in any structure that you desire.

To create a resource component:

1. If you have not already done so, create a JET Pack using the following command from the root folder of your app to contain the resource component(s):

   ```
   ojet create pack my-resource-pack
   ```

2. Still in the root folder of your app, create the resource component in the JET Pack:

   ```
   ojet create component my-resource-comp --type=resource --pack=my-resource-pack
   ```

   The tooling adds the folder structure with a single template `component.json` file and an index file.

   ```
   /root folder
     /src
       /js or /ts
         /jet-composites
           /my-resource-pack
           /my-resource-comp
              component.json
   ```

3. Populate the created folder (`my-resource-comp`, in our example) with the desired content. You can add content in any structure desired, with the exception of NLS content for translation bundles. In the case of NLS content, preserve the typical JET folder structure; this is important if your resource component is going to include such bundles.

   ```
   /(my-resource-folder)
     /converters
       phoneConverter.js
       phoneConverterFactory.js
     /resources
       /nls
         /root
   ```

```
      oj-ext-strings.js
    /phone
      countryCodes.json
  /validators
    emailValidator.js
    emailValidatorFactory.js
    phoneValidator.js
    phoneValidatorFactory.js
    urlValidator.js
    urlValidatorFactory.js
```

In this sample notice how the `/resources/nls` folder structure for translation bundles is preserved according to the folder structured of the app generated by JET tooling.

4. Use a text editor to update the `component.json` file in the folder root similar to the following sample, which defines the resource `my-resource-comp` for the JET Pack `my-resource-pack`.

```
{
  "name": "my-resource-comp",
  "pack": "my-resource-pack",
  "displayName": "Oracle Jet Extended Utilities",
  "description": "A set of reusable utility classes used by the Oracle JET
extended
                  component set and available for general use.  Includes
various
                  reusable validators",
  "license": "https://opensource.org/licenses/UPL",
  "type": "resource",
  "version": "2.0.2",
  "jetVersion": ">=8.0.0 <10.1.0",
  "publicModules": [
    "validators/emailValidatorFactory",
    "validators/urlValidatorFactory"
  ],
  "extension":{
    "catalog": {
      "category": "Resources",
        "coverImage": "cca-resource-folder.svg"
    }
  }
}
```

Your resource component's `component.json` file must contain the following unique definitions:

- **name** is the name of the resource component has to be unique, and should be defined with the namespace relevant to your group.
- **pack** is the name of the JET Pack containing the resource component.
- **displayName** is the name of the resource component as displayed in Oracle Component Exchange. Set this to something readable but not too long.
- **description** is the description that you want displayed in Oracle Component Exchange. For example, use this to explain the available assets provided by the component.

- **type** must be set to `resource`.
- **version** defines the semantic version (SemVer) of the resource component as a whole. It's important that you manage revisions of this version number to inform consumers of the compatibility of a given change.

> **Note:**
>
> Changes to the resource component version should roll up all of the changes within the resource component, which might not be restricted to changes only in `.js` files. A change to a CSS selector defined in a shared `.css` file can trigger a major version change when it forces consumers to make changes to their downstream uses of that selector. For more information see, Version Numbering Standards.

- **jetVersion** defines the supported Oracle JET version range using SemVer notation. This is *optional* and depends on the nature of what you include into the resource component. If the component contains JavaScript code and any of that code makes reference to Oracle JET APIs, then you really should include a JET version range in that case. For more information about specifying semantic versions see Version Numbering Standards.
- **publicModules** lists entry points within the resource component that you consider as being public and intend to be consumed by any component that depends on this component. Any API not listed in the array is considered to be pack-private and therefore can only be used by components within the same pack namespace, but may not be used externally.
- **catalog** defines the working metadata for Oracle Component Exchange, including a cover image in this case.

5. Optionally, create a readme file in the root of your working folder. A readme can be used to document the assets of the resource. This should be defined as a plain text file called `README.txt` (or `README.md` when using markdown format).

> **Tip:**
>
> Take care to explain the state of the assets. For example, you might choose to include utility classes in the resource component that are deemed public and can safely be used by external consumers (for example, code outside of the JET Pack that the component belongs to). However, you may want to document other assets as private to the pack itself.

6. Optionally, create a change log file in the root of your working folder. The change log can detail significant changes to the pack over time and is strongly recommended. This should be defined as a text file called `CHANGELOG.txt` (or `CHANGELOG.md` when using markdown format).
7. Optionally, include a License file in the root of your working folder.
8. Optionally, create a cover image in the root of your working folder to display the component on Oracle Exchange. Using the third party logo can be helpful here to identify the usage. The file name can be the same as the name attribute in the `component.json` file.
9. Create a zip archive of the working folder when you want to upload the component to Oracle Component Exchange. Oracle recommends using the format *<fullName>*-

`<version>.zip` for the archive file name. For example, `my-resource-pack-my-resource-comp-2.0.2.zip`.

For information about using the resource component in a JET Pack, see Create JET Packs.

Create Reference Components for Web Components

Create a reference component when you need to obtain a pointer to third-party libraries for use by Web Components.

Sometimes your JET Web Components need to use third party libraries to function and although it is possible to embed such libraries within the component itself, or within a resource component, it generally better to reference a shared copy of the library by defining a reference component.

Create the Reference Component

You don't need any tools to create the reference component. You will need to create a folder in a convenient location where you will define metadata for the reference component in the `component.json` file. This folder will ultimately be zipped to create the distributable reference component.

Reference components are generally standalone, so the `component.json` file you create must not be contained within a JET Pack.

To create a reference component:

1. Create the working folder and use a text editor to create a `component.json` file in the folder root similar to the following sample, which references the moment.js library.

```
{
  "name": "oj-ref-moment",
  "displayName": "Moment library",
  "description": "Supplies reference information for moment.js used to
parse,
                  validate, manipulate, and display dates and times in
JavaScript",
  "license": "https://opensource.org/licenses/MIT",
  "type": "reference",
  "package":"moment",
  "version": "2.24.0",
  "paths": {
    "npm": {
      "debug": "moment",
      "min": "min/moment.min"
    },
    "cdn": {
      "debug": "https://static.oracle.com/cdn/jet/packs/3rdparty/moment/
2.24.0/moment.min",
      "min": "https://static.oracle.com/cdn/jet/packs/3rdparty/moment/
2.24.0/moment.min"
    }
  },
  "extension": {
    "catalog": {
      "category": "Third Party",
      "tags": [
        "momentjs"
```

```
      ],
      "coverImage": "coverImage.png"
    }
  }
}
```

Your reference component's `component.json` file must contain the following unique definitions:

- **name** is the name of the reference component has to be unique, and should be defined with the namespace relevant to your group.
- **displayName** is the name of the resource component as displayed in Oracle Component Exchange. Set this to something readable but not too long.
- **description** is the description that you want displayed in Oracle Component Exchange. For example, use this to explain the function of the third party library.
- **license** comes from the third party library itself and must be specified.
- **type** must be set to `reference`.
- **package** defines the npm package name for the library. This will also be used as the name of the associated RequireJS path that will point to the library and so will be used by components that depend on this reference.
- **version** should reflect the version of the third party library that this reference component defines. If you need to be able to reference multiple versions of a given library then you will need multiple versions of the reference component in order to map each one.
- **paths** defines the CDN locations for this library. See below for more information about getting access to the Oracle CDN.
- **min** points to the optimal version of the library to consume. The debug path can point to a debug version or just the min version as here.
- **catalog** defines the working metadata for Oracle Component Exchange including a cover image in this case.

2. Optionally, create readme file in the root of your working folder. A readme can be used to point at the third party component web site for reference. This should be defined as a plain text file called `README.txt` (or `README.md` when using markdown format).
3. Optionally, create a cover image in the root of your working folder to display the component on Oracle Exchange. Using the third party logo can be helpful here to identify the usage. The file name can be the same as the name attribute in the `component.json` file.
4. Create a zip archive of the working folder when you want to upload the component to Oracle Component Exchange. Oracle recommends using the format *<fullName>*-*<version>*`.zip` for the archive file name. For example, `oj-ref-moment-2.24.0.zip`.
5. Support consuming the reference component in Oracle Visual Builder projects by uploading the component to a CDN. See below for more details.

Consume the Reference Component

When your Web Components need access to the third party library defined in one of these reference components, you use the dependency attribute metadata in the `component.json` to point to either an explicit version of the reference component or you can specify a semantic

range. Here's a simple example of a component that consumes two such reference components at specific versions:

```
{
    "name":"calendar",
    "pack":"oj-sample",
    "displayName": "JET Calendar",
    "description": "FullCalendar wrapper with Accessibility added.",
    "version": "1.0.2",
    "jetVersion": "^9.0.0",
    "dependencies": {
        "oj-ref-moment":"2.24.0",
        "oj-ref-fullcalendar":"3.9.0"
    },

    ...
```

When the above component is added to an Oracle JET or Oracle Visual Builder project this dependency information will be used to create the correct RequireJS paths for the third party libraries pointed to be the reference component.

For more information about semantic version usage, see Version Numbering Standards.

Alternatively, when you install a Web Component that depends on a reference component and you use Oracle JET CLI, the tooling will automatically do an npm install for you so that the libraries are local. However, with the same component used in Oracle Visual Builder, a CDN location must be used and therefore the reference component must exist on the CDN in order to be used in Visual Builder.

Theme Web Components

Oracle JET Web Components may need to inherit styling from consuming app, such as the background color, where the color style is themeable. Web Components may also enable theming so the consuming app can customize the provided styles. You can add theming support to your Web Component, and you can define CSS variables to support customization of your component's look and feel.

About Web Component Theming

When you theme Web Components you work with SASS partial files to define the style classes and expose the styles through CSS variables, by using the JET Tooling to derive optimized CSS.

It is not always necessary to theme custom components and component packs that you create. In many cases you may not need to enable custom theming. For example, when you only need to wrap some core JET components with no additional user interface, no theming is needed. As these two use cases suggest, it depends how you expect your components to be used in the consuming app.

- Theme-enable the custom component, or component pack, when the component needs to inherit styling from the consuming app, such as the background color that color style must be themeable in your component.
- Theme-enable the custom component, or component pack, when the component defines its own style class and that style needs to be themeable in the consuming app for customization.

Oracle JET relies on CSS variables to theme apps. The use of CSS variables to theme your Web Component supports easy integration into a consuming app. To streamline the theming process and support the generation of optimised stylesheets, your Web Component relies on SASS partial files and SCSS processing.

After you create the Web Component, you run the `ojet add theming` command to install support for scss processing into the base app where you will add the component. Then you can use the `ojet create` component command to add your component to the `jet-composites` folder. The new component project will contain a themes folder and three subfolders: `base`,`redwood`, and `stable` containing SASS partials files, as follows:

- **base** folder - contains the base SASS partial file, where you define the style classes for the component that are common across themes. The style classes reference CSS variables, the values of which are generally provided by the theme specific SASS partial file.
- **redwood** folder - contains the theme-specific SASS partial file for the Redwood theme, where you define style settings for the component, specifically you use it to set any CSS variable values that the theme variant needs to set on the base `_myComponent.scss` partial file. The Redwood theme implements the look and feel for Oracle apps, future changes will be made to address Oracle's requirements.
- **stable** folder - contains the theme-specific SASS partial file for the Stable theme, where you define style settings for the component, specifically you use it to set any CSS variable values that the theme variant needs to set on the base `_myComponent.scss` partial file. The Stable theme is recommended as the base theme for custom themes if you want to reduce the likelihood that future theme updates affect your custom theme.

The SASS partials split between these folders supports the separation of the core style definitions from the variable driven theme-specific definitions. This structure allows you to define multiple themes but in practice is not a requirement for JET web apps, where the Redwood theme runs across platforms and environments, such as iOS on a mobile device or Windows on a desktop machine.

In addition to the generated theme folders, each component also has an SCSS file (ie, `myComponent-styles.scss`) at the root of the component folder that the JET Tooling processes to generate the final CSS from the theme SASS partials. The root `.scss` file contains a single line:

```
@import "themes/redwood/_myComponent.scss";
```

Guidelines for Web Component Theming

JET relies on the use of CSS variables as the primary vehicle for theming, which allows even a single supported theme configuration to adapt to the requirements of the consuming app.

When theming the Web Component observe the following considerations to guide the process.

- The goal should be to make your themed Web Component work out of the box for most cases and still allow the component to be themeable by a consuming app. Maintain a single default theme across your entire set of components.
- While a Web Component must provide a single default theme that works out of the box, you may choose to define the settings for multiple themes within the component. Note that only one theme can be active and surfaced for the component for direct runtime use.
- Document the supported themes in the readme file for the component or pack. This information, combined with component metadata that you supply in `component.json`, specify the contract that the consuming app must fulfill.

- Use CSS variables to externalize anything that you want to make configurable within a theme. This supports component-instance style overrides where needed and also simplifies the creation of custom themes for the component.
- Inherit as much information as possible from the core JET Redwood theme, for example color ramps and sizing. This ensures that the Web Component's theming harmonizes with all of the core JET components present in the app and that your component will be able to adapt well to a custom theme. Where possible, use the `oj-flex` classes supplied by JET or other public layout styles to manage layout. This will ensure that the finer attributes of theming, such as padding, are consistent.
- CSS usage should be optimized to allow consolidation at the JET Pack level or the app level. The use of CSS variables supports CSS consolidation.

If your themed component is configured through CSS variables so that the consuming app may override at the app or instance level, then add those variables to a section called Theming in the component `README.md` and document what they do and what they should be set to. Once documented in this way, the variables become a formal part of the component API and you should follow the normal semver rules when it comes to making changes. A change to the default theme should be classed as a MAJOR version number change in semantic version number.

Theme a Web Component

You can use Oracle JET Tooling to theme-enable a custom Web Components project and work with CSS variables to theme custom components that you add to the project.

You enable theming of a new Web Components project, or JET Pack project, by using the JET Tooling to add theming support to the containing project. The tooling adds SASS partials files used to streamline the CSS generation process and is a *prerequisite* step to creating a custom component or pack to support modifications to the base theme.

After you theme-enable your containing project by running the `ojet add theming` command at the project root, new custom components that you create in the project will contain a `themes` folder with subfolders that each contain a single SASS partials file that you will modify:

- `/<componentName>/themes/base/_<componentName>.sccs` defines your custom styling for the component that you want to remain common across themes. The styling portions that need to change with theme variations will be injected via CSS variable values and come from either the base JET theme or from the component theme-specific settings.
- `/<componentName>/themes/redwood/_<componentName>.sccs` contains the theme-specific settings for the component. Specifically, in this file you can specify any CSS variable values that a custom theme based on the Redwood theme needs to set on the base partials.
- `/<componentName>/themes/stable/_<componentName>.sccs` contains the theme-specific settings for the component. Specifically, in this file you can specify any CSS variable values that a custom theme based on the Stable theme needs to set on the base partials.

Additionally, any custom component in a theme-enabled project will also contain a `<componentName>.scss` SASS file in the component's root folder. This file serves to import the CSS file that the JET build process generates from the SASS partials files.

> **Note:**
>
> A CSS build error may result if a pack or component is created before running `ojet add theming`. The `add theming` command enables SCSS compilation to generate the CSS. Be sure to run `add theming` before creating the pack or component.

Before you begin:

- Refer to the JET CSS Variables section of the JET API reference doc for an overview of the CSS variables and values to use when directly defining custom selectors of the web component.
- View the CSS Variables section in the Oracle JET Cookbook for examples of CSS variable usages.
- Optionally, download and install the CSS Variable Theme Builder - Instruction Tab app when you want to learn about the available CSS variables in this interactive demo app. Follow the instructions online to modify the theme definition files to learn how CSS variable overrides change the demo tool UI.

To add theming support and theme a component:

1. In your Web Component project's top-level directory, enter the following command at a terminal prompt to install the theming toolchain.

   ```
   ojet add theming
   ```

2. Create a Web Component in the containing project. Or, create a JET Pack and add a Web Component to the pack, as this sample shows.

   ```
   ojet create pack my-pack
   ojet create component my-widget1 --pack=my-pack
   ```

   For example, the following commands create a pack `oj-sample` and add a custom component `metric` to the pack.

   ```
   ojet create pack oj-sample
   ojet create component metric --pack=oj-sample
   ```

   These commands create a `jet-composites` folder that contains the `oj-sample` pack folder and the `metric` component.

   ```
   src
   |   index.html
   |
   +---css
   +---js
   \---ts
       +---jet-composites
       |   \---oj-sample
       |       |   component.json
       |       |
       |       \---metric
       |           |   component.json
   ```

```
|           |   loader.ts
|           |   metric-styles.scss
|           |   metric-view.html
|           |   metric-viewModel.ts
|           |   README.md
|           |
|           +---resources
|           \---themes
|               +---base
|               |       _metric.scss
|               |
|               +---redwood
|               |       _metric.scss
|               |
|               \---stable
|                       _metric.scss
|
+---viewModels
+---views
```

In the directory above, the `metric` folder shows the base SASS file `metric-styles.scss`. The `themes` subfolder contains the SASS partials files `_metric.scss` that you will modify.

> **Note:**
>
> If you see a `.css` file instead of a `.scss` file in the root folder of the component, it indicates you did not run the `add theming` command before creating the component.

3. In the `/<componentName>/themes/base` folder, edit the `_componentName.scss` SAAS partials file to define the style selectors for your component. You can define selector properties with hardcoded values or you can define properties controlled by CSS variables when you want to allow overriding of the property by a consuming app.

   Note that style and variable names must be named-spaced to the owning component's HTML element tag to avoid redefinition. For example, the style `.oj-sample-metric-value-color` is name-spaced by `oj-sample-metric`.

```
@import "oj/utilities/_oj.utilities.modules.scss";
// Even if this file is imported many times 'module-include-once' ensures
the content is included just once.
@include module-include-once("oj-sample-metric.base") {
  oj-sample-metric:not(.oj-complete){
    visibility: hidden;
  }
  // Selector definitions section
  oj-sample-metric .oj-sample-metric-value-text {
    display:inline-block;
    align-self:center;
  }
  oj-sample-metric .oj-sample-metric-label {
    font-size: var(--oj-sample-metric-label-font-size);
  }
  oj-sample-metric .oj-sample-metric-value-color {
```

```
    color: var(--oj-sample-metric-value-color);
  }
  oj-sample-metric .oj-sample-metric-label-color {
    color: var(--oj-sample-metric-label-color);
  }
  ...
}
```

In this sample, only the selector `.oj-sample-metric-value-text` selector is completely defined in place and references no CSS variables. Properties of this style cannot be overridden. The `font-size` property for the `.oj-sample-metric-label` selector references a CSS variable that you will define in the theme-specific SASS partials file. The `color` property for the `.oj-sample-metric-value-color` and `.oj-sample-metric-label-color` selectors also references a CSS variable to be defined. The usage of CSS variables allows these properties to be overridden in the consuming app.

When you do not need to allow overriding of a selector property value in the consuming app, you can reference variables defined by JET itself without definition in the theme-specific SASS partials file. In this next sample, the selector is defined directly by `--oj-core-text-color-secondary`, a JET CSS variable that defines text color.

```
  ...
  oj-sample-metric .oj-sample-metric-label-color {
    color: var(--oj-core-text-color-secondary);
  }
  ...
}
```

For a list of JET CSS variables, refer to the resources listed in the Before You Begin section of this topic.

> **Note:**
>
> If you define mappings such as:
>
> ```
> color: rgb(var(--oj-palette-neutral-rgb-60));
> ```
>
> Then, when you build your project, you will encounter an error: "Error: Function rgb is missing argument $green" in your `.scss` files. This is a known SASS compiler issue and the workaround is to use uppercase for the function name, like this:
>
> ```
> color: RGB(var(--oj-palette-neutral-rgb-60));
> ```

4. In the `/<componentName>/themes/redwood` or `/<componentName>/themes/stable` folder, edit the `_componentName.scss` SAAS partials file to provide the theme-specific settings required by the base partials SCCS file. Within root you can define all component-specific CSS variables as hardcoded values or JET CSS variables.

```
@include module-include-once("_metric.redwood") {
  :root {
    --oj-sample-metric-label-font-size:0.875rem;
    --oj-sample-metric-value-color:var(--oj-core-text-color-primary);
    --oj-sample-metric-label-color:var(--oj-core-text-color-secondary);
```

```
    ...
  }
}
```

In this sample, only the CSS variable `--oj-sample-metric-label-font-size` is hardcoded. The other two variables inherit values from underlying JET variables. However, all settings defined in the theme-partials file may be overridden in the app that consumes the web component.

For a list of JET CSS variables, refer to the resources listed in the Before You Begin section of this topic.

To prepare the themed component or pack for a consuming app, see these topics:

- Consolidate CSS for JET Packs
- Optimize CSS to Allow Consuming Apps to Provide Styles

Consolidate CSS for JET Packs

You can consolidate the stylesheets of custom components in a JET Pack to a single CSS file.

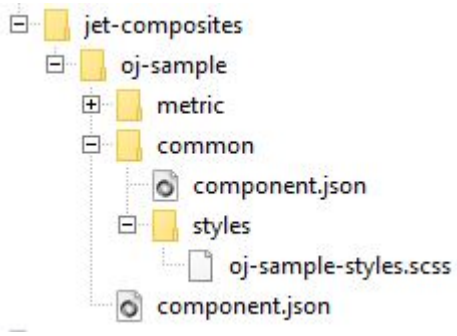
When you add custom Web Components to a JET Pack, initially, each component provides it's own CSS file. Although this approach works, it's optimal in a production app to reduce the number of physical roundtrips to the server and take full advantage of browser caching and the CDN. To reduce the number of roundtrips, you can enable loading from a single CSS file through the use of a shared Resource component that you add to the pack.

Before you begin:

- Create a Resource component with a working folder `common` and its `component.json` file, as described in Create Resource Components for JET Packs.

To create a single shared CSS file in a pack:

1. Add the Resource component that you created with a `component.json` file to the pack file structure and add a subfolder `styles` with a single SCSS file that you will use to import the CSS of the individual components in the pack.



In the figure above, the folder `common` is the working folder for the Resource component and the `styles` subfolder shows a SCSS file `oj-sample-style.scss` named for the pack.

2. Edit the SCCS file in the `styles` folder to replicate the import commands for the SCSS partials files of the components that you want to combine into the consolidated stylesheet.

```
@import "../../metric/themes/redwood/_metric.scss";
@import "../../input-email/themes/redwood/_input-email.scss";
@import "../../input-email/themes/redwood/_input-url.scss";
```

In this sample, we show import statements for three components from the `oj-sample` pack example. In addition to the `metric` component's import, we include the import statements for the `input-email` component and the `input-url` component (omitted from the figure).

3. For each component that you want to consolidate into a shared stylesheet, edit the component's `loader.js` file (`.ts` in the case of a TypeScript project) to reference the shared stylesheet named in the Resource component. Do not edit the loader file for any component that you want to continue to use the component-specific stylesheet.

```
define (['ojs/ojcomposite',
         'text!./metric-view.html',
         './metric-viewModel',
         'text!./component.json',
         'css!oj-sample/common/styles/oj-sample-styles'
         ],

function(Composite, view, viewModel, metadata){
  Composite.register("oj-sample-metric", {
    view: view,
    viewModel: viewModel,
    metadata: JSON.parse(metadata)
  });
 }
);
```

In this JavaScript sample, the loader script for the `metric` component shows the modified path statement references `oj-sample-style` from the SCSS file that was added to the Resource component's `/common/styles` folder.

In the case of a TypeScript app, modify the import statement like this sample:

```
import "css!oj-sample/common/styles/oj-sample-styles";
```

> **Note:**
>
> Do not use relative path mapping to reference a pack component. Identify the path to the consolidated file in the pack component as shown in the previous examples for a JavaScript and TypeScript app.

To perform a further optimization that allows the consuming app to manage the style completely in its own CSS, you can perform this task:

- Optimize CSS to Allow Consuming Apps to Provide Styles

Optimize CSS to Allow Consuming Apps to Provide Styles

You can optimize the loading of component styles in the theme CSS for the consuming app.

To support downstream usages of the themed component, the Web Component needs to establish a contract for what the component needs. The developer who consumes your themed custom component into their app should be able to incorporate the styles unchanged and, optionally, perform a CSS optimization to enable loading from a single CSS file.

To enable these usages you must establish a contract that the downstream developer can observe when adding your component to their app.

To define and support the contract, you decorate the custom component with metadata that specifies the theme SCSS partials that the component exposes, and you modify the default loader script for the component CSS to specify the CSS requireJS plugin with the configurable `ojcss` plugin.

Before you begin:

- Set up the consolidation of component stylesheets with the aid of a Resource component you create at the pack level, as described in Consolidate CSS for JET Packs.

To define a theme-consuming contract for the component:

1. For each component that you want to consolidate into a shared stylesheet, edit the component's `loader.js` file (`.ts` in the case of a TypeScript project) and modify the import statement for the consolidated stylesheet by replacing the normal CSS requireJS plugin with the configurable `ojcss` plugin.

```
define (['ojs/ojcomposite',
         'text!./metric-view.html',
         './metric-viewModel',
         'text!./component.json',
         'ojcss!oj-sample/common/styles/oj-sample-styles'
         ],

function(Composite, view, viewModel, metadata){
  Composite.register("oj-sample-metric", {
    view: view,
    viewModel: viewModel,
    metadata: JSON.parse(metadata)
  });
 }
);
```

   In the case of a TypeScript app, modify the import statement like this sample:

```
import "ojcss!oj-sample/common/styles/oj-sample-styles";
```

   Note that the `ojcss` plugin import gives the downstream developer the ability to set up their app to configure the processing of the plugin directive and optionally suppress loading of custom component stylesheets when the consuming app defines the styles.

2. For these same components, edit the component's `component.json` file to decorate the component with the extension metadata to specify the SCSS partials that define its theme.

```
{
    "name": "metric",
    "pack": "oj-sample",
    "type":"composite",
    "displayName": "Metric Component",
    "license": "https://opensource.org/licenses/UPL",
    "description": "A simple tile that displays a label and value.",
    "version": "4.0.0",
    "jetVersion": ">=8.0.0 <10.0.0",
    "icon": {
        "iconPath": "extension/images/cca-metric.svg",
        "selectedIconPath": "extension/images/cca-metric.svg",
        "hoverIconPath": "extension/images/cca-metric-ovr.svg"
    },
    "properties": {
      . . .
    },
    "events": {
        . . .
    },
    "extension": {
        "themes":{
            "default":"Redwood",
            "unsupportedThemes":["Alta"]
            "partials":{
              "Redwood":"themes/redwood/_oj-sample-metric.scss"}
      }
        "catalog": {
            . . .
        },
        "vbdt": {
            . . .
        }
    }

  }
```

In this sample, the `extension : themes` attribute defines an array of themes supported by the component. Each theme points to the relative path of the root partial as the source for that component's theme implementation. The definition also indicates whether or not it is the expected default theme. Components may support multiple themes, but only one of can be marked as the default.

Note that this metadata is not required by JET but is recommended for any themeable custom component. This information will be used by the downstream developer to set up their app to pull in these SCSS partials into its own SCSS file. They can then build the app to assemble individual stylesheets into a combined CSS file.

In normal usage there is no difference in operation between the `ojcss` and `css` plugins. However, the consuming app now has the ability to control the processing of the plugin directive.

To understand how the consuming app controls the processing of the plugin directive to suppress the loading of the component CSS and define style classes within their app's custom theme, see Incorporate Themed Components into a Consuming App.

Incorporate Themed Components into a Consuming App

You can override CSS variables in a properly themed Web Component by making changes to the style classes in your consuming app's theme.

When consuming a custom component and its theme is the same as your app's theme, then you can add the custom component to your app and use it without changes. However, when the consuming app theme changes the styles of the custom component, you need to reference the custom component theme partials SCSS files in your app.
For example, if your app's custom theme alters the default Redwood theme color ramp, you want to ensure that the same colors are inherited by the themeable custom components that you add to your app. This process of overriding the CSS variables of the custom component by the consuming app requires that you incorporate the partials supplied by each component that you want to consume. This process is statically defined and therefore you will need to re-perform this whenever you add a new component that requires theming into the consuming app.

Note that you also can use this process when you do not need to make changes to the underlying Redwood theme and only want to optimize the number of CSS files by consolidating both the core Redwood styles, plus the extra styles defined for the custom components you will use.

Before you begin:

1. Create a working project that you will use to carry out the theme creation process.

   ```
   ojet create redwood-plus-theme-source --template=basic
   ```

   In this example, the working project is named `redwood-plus-theme-source`.

2. Add theming support to the project and create the custom theme.

   ```
   ojet add theming
   ojet create theme redwood-plus --basetheme=redwood
   ```

   In this example, the theme name `redwood-plus` distinguishes the custom theme from the out-of-the-box `redwood` theme name. Note that JET, as of release 11.0.0, supports another out-of-the-box theme (`stable`) that is less likely to be affected by future changes to the out-of-the-box `redwood` theme. Choose `stable` as the `--basetheme` argument value if the consuming app uses a theme based on the `stable` theme.

3. When working with Oracle Component Exchange to add Web Components, set up access to Component Exchange.

   ```
   ojet configure --exchange-url=https://xxxxx-
   cloud01.developer.ocp.oraclecloud.com/xxxxx-cloud01/s/xxxxx-
   cloud01_sharedcomponentcatalog_8325/compcatalog/0.2.0
   ```

4. Add the components that you want to include into your theme. This process will download the components into the `/jet_components` cache folder in the root of your theme source project.

You can add the components one by one:

```
ojet add component oj-sample-metric@^4.0.0
```

Or, you can add an entire pack at once.

```
ojet add pack oj-sample@^4.0.0
```

In this example, the component name and pack name are reused from the examples shown in the previous topics in this section.

Note when adding the components or packs, you should specify the versions that you need. Similar to the way you would use the core JET theme, you can expect to rebuild and regenerate the theme if there is a change in the major version number of the component or pack.

To incorporate themeable components into the custom theme:

1. In the working project, where you added the custom components, open the `/src/themes/<themeName>/web` folder and examine the folder structure.

```
/src
  /themes
    /redwood-plus
      /web
        _redwood-plus.components.scss
        _redwood-plus.cssvars.settings.scss
        _redwood-plus.sass.settings.scss
        redwood-plus.scss
```

In this sample, we see three SCSS partials files and the CSS aggregating file `redwood-plus.scss`. In the context of customizing the theme in the working project with themeable custom components, the four files have these purposes.

- `_redwood-plus.components.sccs` allows you to tune the overall size of your theme by specifying that it should only style specific components from JET core. Usually you will not touch this file as you'll be styling the whole component set. See also Optimize the CSS in a Custom Theme.
- `_redwood-plus.cssvars.settings.scss` is the file that you use to make most of the style changes in your custom theme, assuming you need to make any. You will use it to set the CSS variable values that you want to be used by the various JET Core components (and possibly by the custom components, as well). For example, to change the primary text color for JET, as a whole, you can do it in this file by uncommenting and setting the value for `--oj-core-text-color-primary`. Assuming this same variable happens to be used by one of the custom components, then it would share this common change. See also Modify the Custom Theme with the JET CLI.
- `_redwood-plus.sass.settings` defines other aspects of the theme that you may need to change but that can't be expressed as simple variable values, for example animation effects. For basic theming scenarios, you probably don't need to touch this file.
- `redwood-plus.scss` this is the main CSS aggregating file for the theme as a whole and is the one that will be transformed into the final output, as the `redwood-plus.css` file.

2. In the `/src/themes/<themeName>/web` folder, edit the CSS aggregating file `<themeName>.scss` and add the import statements for the theme-able custom components that you added to the working project.

```
...
// import SASS custom variable overrides
@import "_redwood-plus.sass.settings.scss";

// Imports all jet components styles
//@import "oj/all-components/themes/redwood/_oj-all-components.scss";

// To optimize performance, consider commenting out the above oj-all-components
// import and uncomment _redwood-plus.components.scss below.
// Then in _redwood-plus.components.scss uncomment only the component
// imports that your app needs.
//

// @import "_redwood-plus.components.scss";

// Import components from oj-sample JET Pack
@import "oj-sample/metric/themes/redwood/_oj-sample-metric.scss";
@import "oj-sample/input-email/themes/redwood/_input-email.scss";
@import "oj-sample/input-url/themes/redwood/_input-url.scss";

// import CSS Custom properties
@import "_redwood-plus.cssvars.settings.scss";
```

   In this sample, the import statements for the three theme-able custom components in the `oj-sample` pack are added. With these import statements referencing each component partials' includePath added to the CSS aggregating file, the indicated component partials will be incorporated into the custom theme.

3. Optionally, if you are making changes to the theme (and not using the process only to consolidate CSS), you can test your theme by embedding sample components, including your custom components, into the working project's `index.html` page and run it. You can then verify the theme changes are what you expect. However, before you run the project, you must ensure that the custom components can pick up their styles from the theme, and not from their own stylesheets.

   a. To suppress the loading of component CSS, edit the `main.js` file and add an `ojcss` configuration section in the `requirejs` configuration that names the CSS loading to suppress.

```
requirejs.config(
    {
      baseUrl: 'js',
      ojcss: {
        'exclude': ['oj-sample']
       },
      paths:
      /* DO NOT MODIFY
      ** All paths are dynamically generated from the
path_mappings.json file.
      ** Add any new library dependencies in path_mappings json file
      */
      // injector:mainReleasePaths
      {
        ...
      }
```

```
        // endinjector
        ,

      }
    );
}());
```

Make sure that you add the `ojcss` configuration section outside of the `//injector` block, as shown. Otherwise, the build process will remove and ignore the section.

In this sample, the `ojcss` section of the `requirejs` configuration excludes the consolidated stylesheet of the JET Pack `oj-sample`. The `ojcss` section defines an array of requireJS paths that will be excluded from loading. Note this configuration depends upon imports within the component using consistent Reliable Referencing based paths that match.

b. Run the project to verify the changes.

```
ojet serve --theme=redwood-plus
```

4. Update the version number in the `/src/themes/<myTheme>/theme.json` file to one that is suitable for the target CSS and build the consolidated theme. By default the version number is initially `0.0.1`.

```
ojet build --theme=redwood-plus
```

Test Web Components

Test Oracle JET Web Components using your favorite testing tools for client-side JavaScript apps.

Regardless of the test method you choose, be sure that your tests fully exercise the Web Component's:

- ViewModel (if it exists)

  Ideally, your test results should be verifiable via code coverage numbers.
- HTML view

  Be sure to include any DOM branches that might be conditionally rendered, and test all slots with and without default content.
- Properties and property values
- Events
- Methods
- Accessibility
- Security

For additional information about testing Oracle JET apps, see Test Oracle JET Apps.

Add Web Components to Your Page

To use an Oracle JET Web Component, you must register the Web Component's loader file in your app and you must also include the Web Component element in the app's HTML. You can add any supporting CSS or files as needed.

1. In the Web Components's root folder, open `component.json` and verify that your version of Oracle JET is compatible with the version specified in `jetVersion`.

   For example, the demo-card example specifies the following `jetVersion`:

   ```
   "jetVersion": ">=3.0.0 <18.1.0"
   ```

   This indicates that the component is compatible with JET versions greater than or equal to 3.0.0 and less than 18.1.0.

   If your version of Oracle JET is lower than the `jetVersion`, you must update your version of Oracle JET before using the component. If your version of Oracle JET is greater than the `jetVersion`, contact the developer to get an updated version of the component.

2. In your app's `index.html` or main app HTML, add the component and any associated property declarations.

   For example, to use the `demo-card` standalone Web Component, add it to your `index.html` file and add declarations for `name`, `avatar`, `work-title`, `work-number`, `email`, and `background-image`.

   ```
   <div id="composite-container" class="oj-flex oj-sm-flex-items-initial">
      <oj-bind-for-each data="[[employees]]">
        <template>
          <demo-card class="oj-flex-item"
                     name="[[$current.data.name]]"
                     avatar="[[$current.data.avatar]]"
                     work-title="[[$current.data.title]]"
                     work-number="[[$current.data.work]]"
                     email="[[$current.data.email]]">
          </demo-card>
        </template>
      </oj-bind-for-each>
   </div>
   ```

   In the case of components within a JET Pack, the HTML tag name is the component full name. The full name of a pack's member component is always a concatenation of the pack name and the component name, as specified by the **dependencies** attribute of the pack-level `component.json` file (located in the pack root folder under `jet-composites`). For example, a component `widget-1` that is a member of the JET Pack `my-pack`, has the following full name that you can reference as the HTML tag name.

   ```
   my-pack-widget-1
   ```

   Note that the framework maps the attribute names in the markup to the component's properties.

- Attribute names are converted to lowercase. For example, a `workTitle` attribute will map to a `worktitle` property.
- Attribute names with dashes are converted to camelCase by capitalizing the first character after a dash and then removing the dashes. For example, the `work-title` attribute will map to a `workTitle` property.

You can access the mapped properties programmatically as shown in the following markup:

```
<h5><oj-bind-text value="[[properties.workTitle]]"></oj-bind-text></h5>
```

3. In your app's ViewModel, set values for the properties you declared in the previous step and add the component's loader file to the list of app dependencies.

   For example, the following code adds the ViewModel to the app's RequireJS bootstrap file. The code also defines the `jet-composites/demo-card/loader` dependency.

```
require(['ojs/ojbootstrap', 'knockout', 'ojs/ojknockout', 'demo-card/
loader'],
function(Bootstrap, ko) {
  function model() {
    var self = this;
    self.employees = [
      {
        name: 'Deb Raphaely',
        avatar: 'images/composites/debraphaely.png',
        title: 'Purchasing Director',
        work: 5171278899,
        email: 'deb.raphaely@oracle.com'
      },
      {
        name: 'Adam Fripp',
        avatar: null,
        title: 'IT Manager',
        work: 6501232234,
        email: 'adam.fripp@oracle.com'
      }
    ];
  }

  Bootstrap.whenDocumentReady().then(function()
    {
      ko.applyBindings(new model(), document.getElementById('composite-
container'));
    }
  );
});
```

   In the case of a JET Pack, you add the loader file for the JET Pack by specifying the path based on the pack root and folder name of the component contained within the pack.

```
'my-pack/widget-1/loader'
```

4. Add any supporting CSS, folders, and files as needed.

For example, the demo card example defines a background image for the contact card in the app's `demo.css`:

```
#composite-container demo-card .demo-card-front-side {
    background-image: url('images/composites/card-background_1.png');
}
```

Build Web Components

You can build your Oracle JET Web Component to optimize the files and to generate a minified folder of the component that can be shared with the consumers.

When your Web Component is configured and is ready to be used in different apps, you can build the Web Components of the type: standalone Web Component, JET Pack, and Resource component. Building these components using JET tooling generates a minified content with the optimized component files. This minified version of the component can be easily shared with the consumers for use. For example, you would build the component before publishing it to Oracle Component Exchange. To build the Web Component, use the following command from the root folder of the JET app containing the component:

```
ojet build component my-web-component-name
```

For example, if your Web Component name is `demo-card-example`, use the following command:

```
ojet build component demo-card-example
```

For a JET Pack, specify the pack name.

```
ojet build component my-pack-name
```

Note that the building individual components within the pack is not supported, and the whole pack must be built at once.

This command creates a `/min` folder in the `web/js/jet-composites/demo-card-example/x.x.x` directory of your Oracle JET web app, where `x.x.x` is the version number of the component. The `/min` folder contains the minified (release) version of your Web Component files.

Reference component do not require minification or bundling and therefore do not need to be built.

When you build Web Components:

- If your JET app contains more than one component, you can build the containing JET app to build and optimize all components together. The `build component` command with the component name provides the capability to build a single component.
- You can optionally use the `--release` flag with the build command, but it is not necessary since the `build` command generates both the debug and minified version of the component.
- You can optionally use the `--optimize=none` flags with the `build` command when you want to generate compiled output that is more readable and suitable for debugging. The

component's `loader.js` file will contain the minified app source, but content readability is improved, as line breaks and white space will be preserved from the original source.

Generate API Documentation for VComponent-based Web Components

The Oracle JET CLI includes a command (`ojet add docgen`) that you can use to assist with the generation of API documentation for the VComponent-based web components (VComponent) that you develop.

When you run the command from the root of your project, the JSDoc NPM package is installed and an `apidoc_template` directory is added to the `src` directory of your project. The `apidoc_template` directory contains HTML files (footer, header, and main) that you can customize with appropriate titles, subtitles, and footer information, such as copyright information, for the API reference documentation that you'll subsequently generate for your VComponent(s). The command also adds an `"enableDocGen": true` entry to the `oraclejetconfig.json` file of the Oracle JET app. When `true`, API doc is generated. When `false`, no API doc is generated for the VComponents.

You write comments in the source file of your VComponent, as in the following example:

```
import { ExtendGlobalProps, registerCustomElement } from "ojs/ojvcomponent";
. . .

type Props = Readonly<{
  message?: string;
  address?: string;
}>;

/**
 *
 * @ojmetadata version "1.0.0"
 * @ojmetadata displayName "A user friendly, translatable name of the
component"
 * @ojmetadata description "<p>Write a description here.</p>
                           <p>Use HTML tags to put in new paragraphs</p>
                           <ul>
                               <li>Bullet list item 1</li>
                               <li>Bullet list item 2</li></ul>
        * <p>Everything before the closing quote is rendered</p>
 * "
 *
 */


function StandaloneVcompFuncImpl({ address = "Redwood shores",
                         message = "Hello from  standalone-vcomp-func" }:
Props) {
  return (
    <div>
    . . .
    </div>
  );
}
```

Once you have completed documenting your VComponent's API in the source file, you run the `build` command for your component or the JET Pack, if the component is part of a JET pack (`ojet build component` *`component-name`* or `ojet build component` *`jet-pack-name`*) to generate API reference doc in the `appRootDir/web/js/jet-composites/`*`component-or-pack-name`*`/`*`vcomponent-version`*`/docs` directory.

The following `/docs` directory listing shows the files that the Oracle JET CLI generates for a standalone VComponent. You can't generate the API documentation by building the Oracle JET app that contains the component. You have to build the individual VComponent or the JET Pack that contains VComponents. Note too that you can't generate API doc for CCA-based web components using the Oracle JET CLI `ojet add docgen` command.

```
appRootDir/web/js/jet-composites/standalone-vcomp-func/1.0.0/docs
|   index.html
|   jsDocMd.json
|   standalone-vcomp-func.html
|   standalone.StandaloneVcompFunc.html
|
+---scripts
|   |   deprecated.js
|   |
|   \---prettify
|           Apache-License-2.0.txt
|           lang-css.js
|           prettify.js
|
\---styles
    |   jsdoc-default.css
    |   prettify-jsdoc.css
    |   prettify-tomorrow.css
    |
    \---images
            bookmark.png
            linesarrowup.png
            linesarrowup_blue.png
            linesarrowup_hov.png
            linesarrowup_white.png
            oracle_logo_sm.png
```

One final thing to note is that if you want to include an alternative logo and/or CSS styles to change the appearance of the generated API doc, you update the content in the following directory `appRootDir/node_modules/@oracle/oraclejet/dist/jsdoc/static/styles/`.

Package Web Components

You can create a sharable zip file archive of the minified Oracle JET Web Component from the Command-Line Interface.

When you want to share Web Components with other developers, you can create an archive file of the generated output contained in the `jet-composites` subfolder of the app's `/web`. After you build a standalone Web Component or a Resource component, you use the JET tooling to

run the `package` command and create a zip file that contains the Web Component compiled and minified source.

```
ojet package component my-web-component-name
```

Similarly, in the case of JET packs, you cannot create a zip file directly from the file system. It is necessary to use the JET tooling to package JET packs because the output under the `/jet-composites/<packName>` subfolder contains nested component folders and the tooling ensures that each component has its own zip file.

```
ojet package pack my-JET-Pack-name
```

The `package` command packages the component's minified source from the `/web/js/jet-composites` directory and makes it available as a zip file in a `/dist` folder at the root of the containing app. This zip file will contain both the specified component and a minified version of that component in a `/min` subfolder.

Reference components do not require minification or bundling and therefore do not need to be built. You can archive the Reference component by creating a simple zip archive of the component's folder.

The zip archive of the packaged component is suitable to share, for example, on Oracle Component Exchange, as described in Publish Web Components to Oracle Component Exchange. To help organize components that you want to publish, the JET tooling appends the value of the `version` property from the `component.json` file for the JET pack and the individual components to the generated zip in the `dist` folder. Assume, for example, that you have a component pack, `my-component-pack`, that has a `version` value of `1.0.0` and the indiviudal components (`my-widget-1`, and so on) within the pack also have version values of `1.0.0`, then the zip file names for the generated files will be as follows:

```
appRootDir/dist/
my-web-component-name_1-0-0.zip
my-component-pack_1-0-0.zip
my-component-pack-my-widget-1_1-0-0.zip
my-component-pack-my-widget-2_1-0-0.zip
my-component-pack-my-widget-3_1-0-0.zip
```

You can also generate an archive file when you want to upload the component to a CDN. In the CDN case, additional steps are required before you can share the component, as described in Upload and Consume Web Components on a CDN.

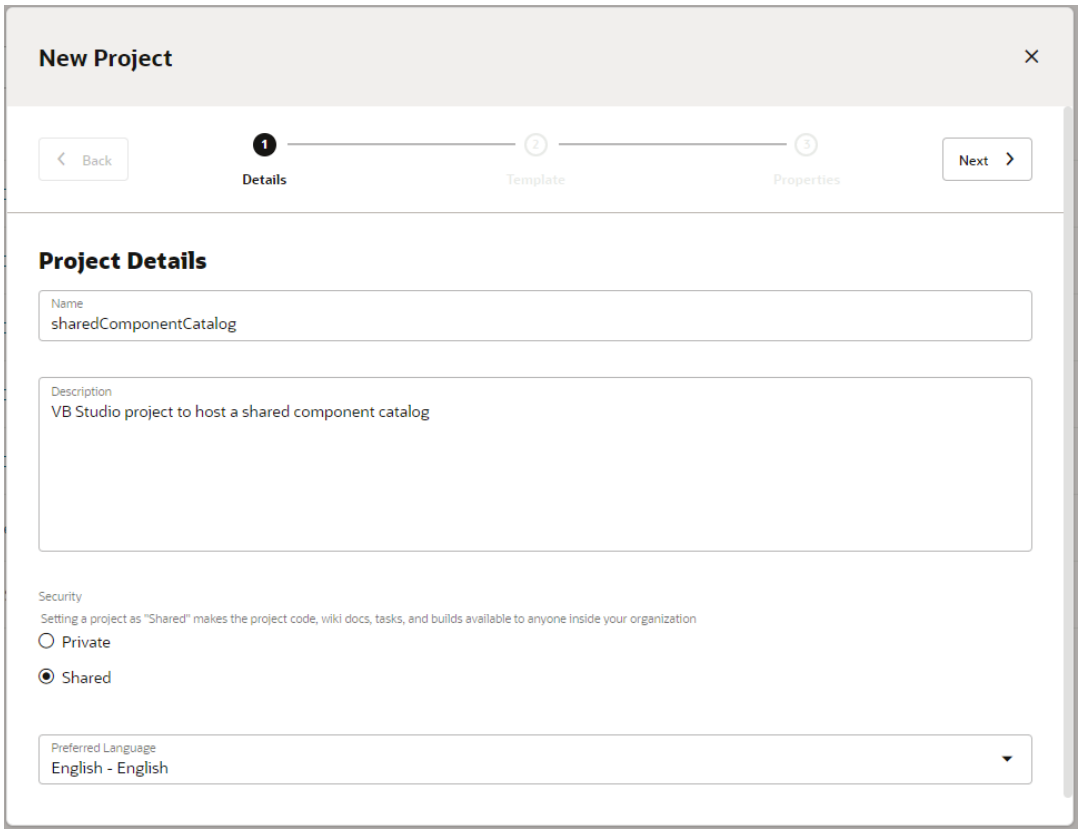
Create a Project to Host a Shared Oracle Component Exchange

When you want to store and share Web Components across machines for re-use by other developers, you can use a Component Exchange that you create in Oracle Visual Builder Studio.

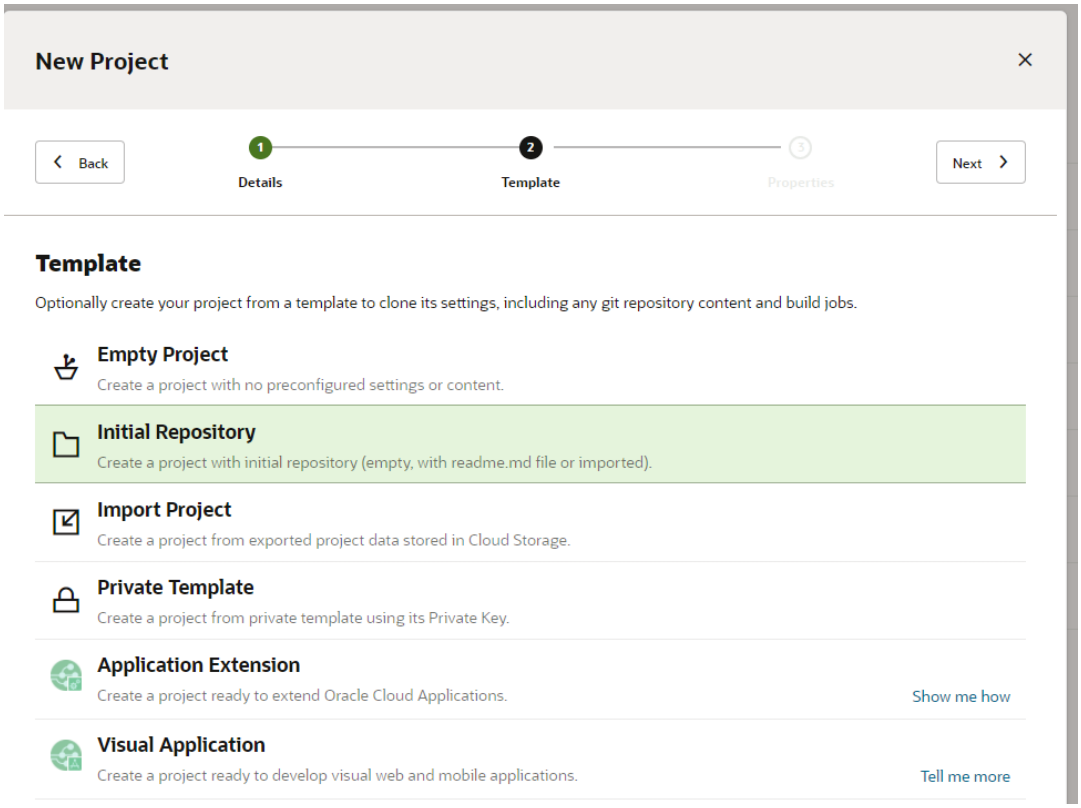
When you want to share Web Components, you generally need to set up a dedicated project to specifically host a Shared Exchange. Although every project within Oracle Visual Builder Studio has a private Component Exchange instance, the Shared Exchange makes uploaded components accessible to other developers. You can use the same project in your Shared Exchange to host the source code repository of your components.

To create the Shared Exchange:

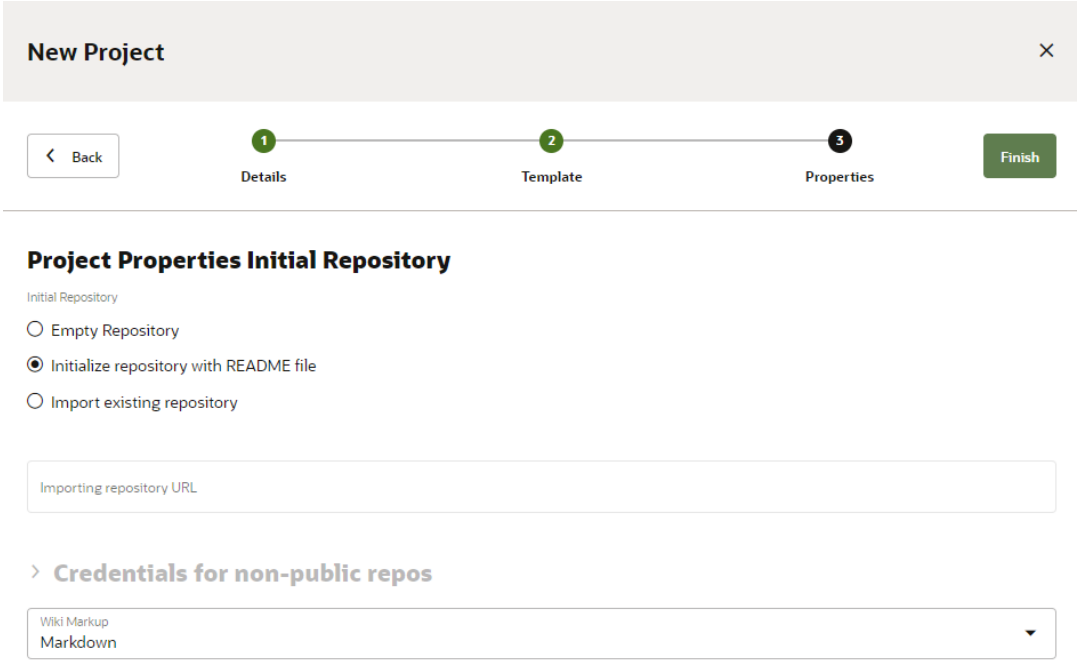
1. Create a new project in Oracle Visual Builder Studio.



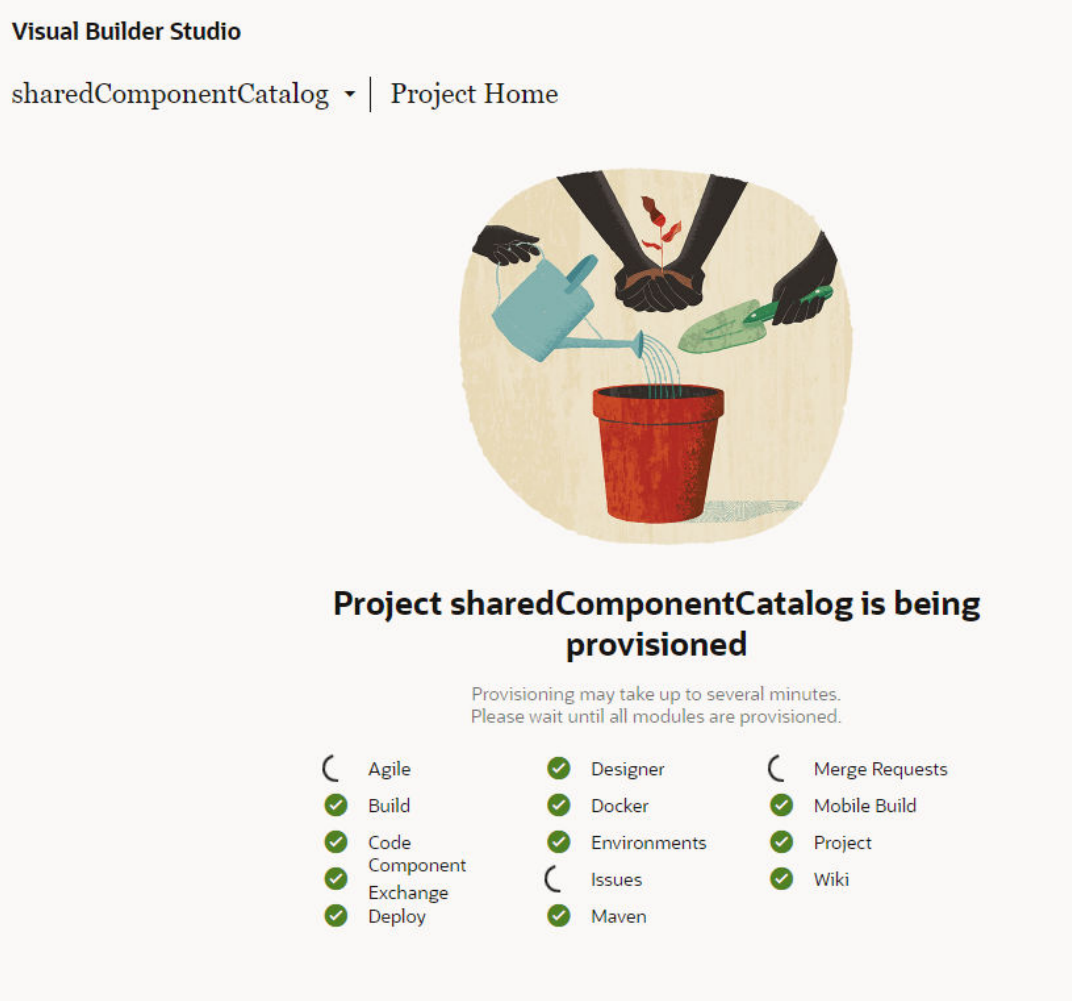
2. Select the **Initial Repository** template for the new project.



3. Leave the defaults settings unchanged or, optionally, select the settings for importing your component source code and click **Finish**.



4. In the project provisioning screen, verify that **Component Exchange** is one of the services being created.



5. Share the project with all users who will need to access published components by adding them as members to the project. They will then use their own user name and password to access the Shared Component Exchange.

Once the project is created you can obtain the correct URL for use with the Oracle JET tooling to access the Shared Component Exchange, as described in Publish Web Components to Oracle Component Exchange.

Publish Web Components to Oracle Component Exchange

When you want to store and share Web Components across machines for re-use by other developers, you can use the Oracle JET CLI (Command-Line Interface) to configure access to a Shared Component Exchange defined in Oracle Visual Builder Studio and then publish components to a public project.

When you want to share Web Components, you can use Oracle JET CLI to configure access to a Shared Component Exchange by supplying the URL to the target Component Exchange. Once you have configured the JET tooling for a specific Component Exchange, you can run the `publish component` command in the JET CLI to upload specified components. Users with access rights can use the CLI to search the Component Exchange for components by keyword and add components to their web app project.

Before you begin:

- Create a project in a Shared Component Exchange, as described in Create a Project to Host a Shared Oracle Component Exchange.
- Share the project with all users who will need to access components by adding them as members to the project, as described in Create a Project to Host a Shared Oracle Component Exchange. Users will need to create their own user name and password to access the Shared Component Exchange.

To publish components to a Shared Component Exchange:

1. Obtain the URL to the Shared Component Exchange that you can use to configure Oracle JET tooling.
   a. From the Shared Component Exchange that you created, copy the URL that you would use to clone the GIT repository. The URL will look similar to this.

      ```
      https://john.doe@example.org@xxxxx-
      cloud01.developer.ocp.oraclecloud.com/xxxxx-cloud01/s/xxxxx-
      cloud01_sharedcomponentcatalog_8325/scm/sharedcomponentcatalog.git
      ```

   b. Using the copied URL, remove the user name prefix (for example, `john.doe@example.org@`) and remove elements from `/scm` onwards to obtain a root of just the project, similar to this.

      ```
      https://xxxxx-cloud01.developer.ocp.oraclecloud.com/xxxxx-cloud01/s/
      xxxxx-cloud01_sharedcomponentcatalog_8325
      ```

   c. Next, append `/compcatalog/0.2.0`.

      In this example, the Exchange URL that you need for the Oracle JET tooling looks like this.

      ```
      https://xxxxx-cloud01.developer.ocp.oraclecloud.com/xxxxx-cloud01/s/
      xxxxx-cloud01_sharedcomponentcatalog_8325/compcatalog/0.2.0
      ```

2. Configure Oracle JET tooling to access the Component Exchange project by running the `ojet configure` command in the Oracle JET CLI. Set the `--exchange-url` flag on the command to pass the Component Exchange URL you obtained.

   ```
   ojet configure --exchange-url=https://xxxxx-
   cloud01.developer.ocp.oraclecloud.com/xxxxx-cloud01/s/xxxxx-
   cloud01_sharedcomponentcatalog_8325/compcatalog/0.2.0
   ```

> **Tip:**
>
> You can use the JET CLI to define an Exchange URL that is global to all projects for your user:
>
> ```
> ojet configure --global --exchange-url=myExchange.org
> ```
>
> The URL that you pass to the `ojet configure` command at the project level, overrides the global definition.

3. In the Oracle JET CLI, publish a component to the configured Component Exchange by running the `publish component` command.

   ```
   ojet publish component my-demo-card
   ```

   Optionally, you can supply Component Exchange login user name and password with the `publish` command. For more information, enter `ojet help publish` in the CLI.

Oracle JET tooling supports searching the configured Component Exchange by running the `search exchange` command with a keyword, such as the component name. Additionally, you can add components to your web app by running the `add component` command for the configured Component Exchange. For more information, use `ojet help` in the JET CLI.

Upload and Consume Web Components on a CDN

You can package a Web Component to make it available on a Content Delivery Network (CDN) and you can reuse the component in your app.

You can include the CDN location of the component in the component metadata as defined in the component's `component.json` file. By doing this, tools such as Oracle JET tooling and Oracle Visual Builder will be able to point to the CDN location when you build your apps in release mode.

CDN information is encoded by using the `paths` attribute in the `component.json` file. A typical example looks similar to this.

```
{
  "name": "demo-samplecomponent",
  "displayName": "Sample component",
  "version": "1.0.0",
  "paths": {
    "cdn": {
      "min": "https://static.example.com/cdn/jet/components/demo-
samplecomponent/1.0.0/min",
      "debug": "https://static.example.com/cdn/jet/components/demo-
samplecomponent/1.0.0"
    }
  },
...
```

The following notes apply to the `paths` attribute that you specify:

- The exact CDN root location will depend on your CDN provider, and it is the final part of the location that is needed.
- The location has a folder with the same name as the component (in this example `demo-samplecomponent`) followed by the version number of the component (`1.0.0`). As you release new versions of the component, you will create a new version-number folder under the component root.
- You can provide both a `min` and `debug` path for the component, where the `debug` path is optional.

To prepare for CDN distribution, package the component by running the `package component` command in the Oracle JET Command-Line Interface. This will produce a zip file with the same name as the component (for example, `demo-samplecomponent.zip`) in the `/dist` folder of

the containing Oracle JET app. This zip file will contain both the specified component and a minified version of that component in a `/min` subfolder.

Unpack the zip file that you created under the `/<component-name>/<version>` folder for your CDN as identified in the `component.json` file.

After a component is available on a CDN, you can then use it in your JET app by pointing the requireJS path for the component to the CDN location that you identified for the component. If you use Oracle JET tooling, this will be done for you; however, if you need to define the requireJS paths manually, such a mapping will look similar to this in `main.js` file:

```
requirejs.config(
    {
      baseUrl: 'js',
      paths:
      {
        'demo-samplecomponent': '"https://static.example.com/cdn/jet/
components/demo-samplecomponent/1.0.0/min',
      ...
```

References to the component can be made using the path `<component-name>/loader`.

9

Use Oracle JET REST Data Provider APIs

Use Oracle JET's REST Data Provider APIs (`RESTDataProvider` and `RESTTreeDataProvider`) to send queries to JSON-based REST services and fetch the resulting data to your app.

The `RESTDataProvider` implements Oracle JET's `DataProvider` interface while the `RESTTreeDataProvider` implements the `TreeDataProvider` interface. We'll first describe `RESTDataProvider` as it is the foundation API to fetch data from JSON-based REST services using the Fetch API. The `RESTTreeDataProvider` builds on the `RESTDataProvider` to retrieve hierarchical data.

About the Oracle JET REST Data Provider

Use `RESTDataProvider` when you want to send query parameters to the JSON-based REST service so that the subset of data that matches your query is returned to the JET app rather than all data.

Use it, if, for example, you want to display a list of employees in your app who match a certain criterion, such as members of a department. You could fetch the full list of employees to your JET app where you query the returned data before you display the data rows that match your criterion. This is inefficient because you send a request for data that you don't need. Instead, using an instance of `RESTDataProvider`, you construct a request to the REST service that returns the appropriate subset of data.

`RESTDataProvider` accomplishes this through the use of `transforms`, which is a property option that you specify when you initialize a `RESTDataProvider` instance. `RESTDataProvider` has three `transforms` property options. These are `fetchFirst`, `fetchByOffset` and `fetchByKeys`. These correspond to the three available fetch methods on data providers. Each property option defines the following functions:

- `request`: A function that creates a Request object to use for the Fetch API call to the the REST API. This is where any required query parameters for paging, filtering, and sorting can be applied to the URL which can then be used to create a request. Other request options, such as `headers`, `body` and `method` can also be added to the request if needed.
- `response`: A function that extracts the data and other relevant values from the response body. The function must, at a minimum, return an object with a `data` property that is an array of items of type D (generic passed into `RESTDataProvider` class). The generic type `D` corresponds to the type of the entries in the loaded data. For example, for personal information data, you may have entries of type string and number, as in the following entries: `{ name: string, age: number }`.

The Oracle JET Cookbook includes a series of demos that show usage of the `RESTDataProvider`. The Overview demo shows you how to initialize an instance of `RESTDataProvider` using the `fetchFirst` method that includes request and response functions. See REST Data Provider in the Oracle JET Cookbook and REST Data Provider in the *Oracle® JavaScript Extension Toolkit (JET) API Reference for Oracle JET*.

About the Oracle JET REST Tree Data Provider

Use `RESTTreeDataProvider` when you want to retrieve hierarchical data from JSON-based REST services.

The main difference between `RESTDataProvider` and `RESTTreeDataProvider` is that `RESTTreeDataProvider` exposes an additional instance method, `getChildDataProvider`, and a constructor option that is also named `getChildDataProvider`. They have different signatures.

- Instance method's signature is `getChildDataProvider(parentKey: K): TreeDataProvider<K, D> | null`
- Constructor option's signature is `getChildDataProvider(item: Item<K, D>): DataProvider<K, D> | null`

The `getChildDataProvider` method takes as an argument the key of the parent node to create a child data provider for. It then returns a `RESTTreeDataProvider` instance that loads the children of the parent node or `null` if the node is a leaf node. A leaf node is a node that cannot have child nodes.

If the key of the parent node that you pass to `getChildDataProvider` does not correspond to a previously fetched item or is an item that metadata identifies as a leaf node, `getChildDataProvider` returns `null`. Note that calls to `getChildDataProvider(parentKey: K)` internally call `getChildDataProvider(item: Item<K, D>)` after retrieving the item corresponding to `parentKey`. Therefore, it is the constructor option that accesses the metadata of the item corresponding to a node's key. Since the constructor option is defined by the JET app, it is the JET app, and not `RESTTreeDataProvider`, that decides whether to return `null` or a `DataProvider` instance. This whole flow, which starts from the JET app returning metadata through the fetch response transform, is a mechanism to give apps enough information to determine whether a node has children or not.

Note too that in the Oracle JET Cookbook the leaf field name is based on the metadata provided by the cookbook's mock server through the response transform. Your REST service is likely to use a different field name.

The metadata used by the Oracle JET Cookbook's mock server is of type `{ key: K, leaf: boolean}[]`. This indicates whether the corresponding node is a leaf node. This is useful when invoking `getChildDataProvider`. It returns `null` when the parent node is a leaf node. Otherwise, the node renders with an expansion arrow in the `oj-tree-view` component even though it cannot have child nodes.

One other thing to note is that create operations for the `RESTTreeDataProvider` need to use the `parentKeys` option. This can be seen in the `addChildNode` method of the Events demo in the Oracle JET Cookbook entry for REST Tree Data Provider. See more detail about the `parentKeys` option in Data Provider Add Operation Event Detail of the *Oracle® JavaScript Extension Toolkit (JET) API Reference for Oracle JET*.

The Oracle JET Cookbook includes a series of demos that show usage of the `RESTTreeDataProvider` API. The Overview demo shows you how to create an instance of `RESTTreeDataProvider` using the `getChildDataProvider` method. See REST Tree Data Provider. In addition to the demos in the Oracle JET Cookbook, see also the entry for the REST Tree Data Provider in the *Oracle® JavaScript Extension Toolkit (JET) API Reference for Oracle JET*.

Create a CRUD App Using Oracle JET REST Data Providers

Use the Oracle JET REST Data Provider APIs to create apps that perform CRUD (Create, Read, Update, Delete) operations on data returned from a REST Service API.

Unlike the Common Model and Collection APIs that we recommended you to use prior to release 11, `RESTDataProvider` and `RESTTreeDataProvider` do not provide methods such as `Collection.remove` or `Model.destroy`. Instead you use the Fetch API and the appropriate HTTP request method to send a request to the REST service to perform the appropriate operation. In conjunction with this step, you use the data provider's `mutate` method to update the data provider instance.

> **Note:**
>
> The steps that follow make specific reference to `RESTDataProvider`, but the general steps also apply to `RESTTreeDataProvider`. View the **Events** demo in the Oracle JET Cookbook entry for REST Tree Data Provider to see an implementation of CRUD-type functionality that uses `RESTTreeDataProvider`. Note that the demos in the Oracle JET Cookbook use a mock REST server, so operations and responses in an actual REST service may differ from Cookbook demonstration.

Define the Data Model for REST Data Provider

Identify the data source for your app and create the data model.

1. Identify your data source and examine the data. For data originating from a REST service, identify the service URL and navigate to it in a browser.

   The following example shows a sample of the response body for a request sent to a REST service for department data.

```
[
  {
    "DepartmentId": 20,
    "DepartmentName": "HR",
    "LocationId": 200,
    "ManagerId": 300,
    "Date": "01 Oct 2015"
  },
  {
    "DepartmentId": 100,
    "DepartmentName": "Facility",
    "LocationId": 200,
    "ManagerId": 300,
    "Date": "13 Oct 2002"
  }
]
```

   In this example, each department is identified by the `DepartmentId` attribute.

2. Add code to your app that creates an instance of the REST data provider and fetches data from the REST service.

```
import { RESTDataProvider } from "ojs/ojrestdataprovider";
import "ojs/ojtable";

type D = { DepartmentId: number; DepartmentName: string; Date: string };
type K = D["DepartmentId"];

class ViewModel {

    dataprovider: RESTDataProvider<K, D>;
    keyAttributes = "DepartmentId";
    url = "https://restServiceURL/departments";

    constructor() {
        this.dataprovider = new RESTDataProvider({
            keyAttributes: this.keyAttributes,
            url: this.url,

            transforms: {
                fetchFirst: {
                    request: async (options) => {
                        const url = new URL(options.url);
                        const { size, offset } = options.fetchParameters;
                        url.searchParams.set("limit", String(size));
                        url.searchParams.set("offset", String(offset));
                        return new Request(url.href);
                    },
                    response: async ({ body }) => {
                        const { items } = body;
                        // If the response body returns, for example,
"items".
                        // We need to assign "items" to "data"
                        return { data: items };
                    },
                },
            },
        });
    }
}

export = ViewModel;
```

Read Records

To read the records, define the Oracle JET elements that will read the records in your app's page.

The following sample code shows a portion of the `html` file that displays a table of records using the `oj-table` element and the `data` attribute that references the `dataprovider`. In this example, the table element creates columns for `Department Id`, `Department Name`, and so on.

```
<oj-table id="table" data="[[dataprovider]]"
      columns='[{"headerText": "Department Id",
```

```
          "field": "DepartmentId"},
         {"headerText": "Department Name",
          "field": "DepartmentName"},
         {"headerText": "Location Id",
          "field": "LocationId"},
         {"headerText": "Manager Id",
          "field": "ManagerId"}]'>
</oj-table>
```

Create Records

To add the ability to create new records, add elements to your HTML page that accept input from the user and create a function that sends the new record to the REST service.

1. Add elements to the app's HTML page that accept input from the user.

   The code in the following example adds `oj-input-*` elements to an HTML page to allow a user to create a new entry for a department.

   ```
   <oj-form-layout readonly="false" colspan-wrap="wrap" max-columns="1">
     <div>
       <oj-input-number id="departmentIdInput" label-hint="Department Id"
   value="{{inputDepartmentId}}"></oj-input-number>
       <oj-input-text id="departmentNameInput" label-hint="Department Name"
   value="{{inputDepartmentName}}"></oj-input-text>
       <oj-input-number id="locationIdInput" label-hint="Location Id"
   value="{{inputLocationId}}const addedRowKey =
   addedRow[this.keyAttributes]"></oj-input-number>
       <oj-input-number id="managerIdInput" label-hint="Manager Id"
   value="{{inputManagerId}}"></oj-input-number>
     </div>
     <div>
       <oj-button id="addButton" on-oj-action="[[addRow]]"
   disabled="[[disabledAdd]]">Create</oj-button>
       . . .
     </div>
   </oj-form-layout>
   ```

   The `oj-button`'s `on-oj-action` attribute is bound to the `addRow` function which is defined in the next step.

2. Add code to the ViewModel to send the user's input as a new request to the REST service and update the `RESTDataProvider` instance using the `mutate` method.

   ```
   // add to the observableArray
     addRow = async () => {
       // Create row object based on form inputs
       const row = {
         DepartmentId: this.inputDepartmentId(),
         DepartmentName: this.inputDepartmentName(),
         LocationId: this.inputLocationId(),
         ManagerId: this.inputManagerId(),
       };
       // Create and send request to REST service to add row
       const request = new Request(this.restServerUrl, {
   ```

```
      headers: new Headers({
        "Content-type": "application/json; charset=UTF-8",
      }),
      body: JSON.stringify(row),
      method: "POST",
    });
    const response = await fetch(request);
    const addedRow = await response.json();
    // Create add mutate event and call mutate method
    // to notify dataprovider consumers that a row has been
    // added
    const addedRowIndex = addedRow.index;
    delete addedRow.index;
    const addedRowKey = addedRow[this.keyAttributes];
    const addedRowMetaData = { key: addedRowKey };
    this.dataprovider.mutate({
      add: {
        data: [addedRow],
        indexes: [addedRowIndex],
        keys: new Set([addedRowKey]),
        metadata: [addedRowMetaData],
      },
    });
  };
```

Update Records

To add the ability to update records, add elements to your HTML page that accept input from the user and create a function that sends the updated record to the REST service.

1. Add elements to the app page that identify updatable elements and enables the user to perform an action to update them.

   The code in the following example adds `oj-input-*` elements to an HTML page to allow a user to update a selected entry for a department.

```
<oj-form-layout readonly="false" colspan-wrap="wrap" max-columns="1">
  <div>
    <oj-input-number id="departmentIdInput" label-hint="Department Id"
value="{{inputDepartmentId}}"></oj-input-number>
    <oj-input-text id="departmentNameInput" label-hint="Department Name"
value="{{inputDepartmentName}}"></oj-input-text>
    <oj-input-number id="locationIdInput" label-hint="Location Id"
value="{{inputLocationId}}"></oj-input-number>
    <oj-input-number id="managerIdInput" label-hint="Manager Id"
value="{{inputManagerId}}"></oj-input-number>
  </div>
  <div>
    <oj-button id="addButton" on-oj-action="[[updateRow]]"
disabled="[[ disabledUpdate]]">Update</oj-button>
    . . .
  </div>
</oj-form-layout>
```

The `oj-button`'s `on-oj-action` attribute is bound to the `updateRow` function which is defined in the next step.

2. Add code to the ViewModel to send the user's update as a new request to the REST service and update the `RESTDataProvider` instance using the `mutate` method.

```
// used to update the fields based on the selected row
  updateRow = async () => {
    const currentRow = this.selectedRow;
    if (currentRow != null) {
      // Create row object to update based on form inputs
      const row = {
        DepartmentId: this.inputDepartmentId(),
        DepartmentName: this.inputDepartmentName(),
        LocationId: this.inputLocationId(),
        ManagerId: this.inputManagerId(),
      };
      // Create and send request to update row on the REST service
      const request = new Request(
        `${this.restServerUrl}/${this.selectedKey}`,
        {
          headers: new Headers({
            "Content-type": "application/json; charset=UTF-8",
          }),
          body: JSON.stringify(row),
          method: "PUT",
        }
      );
      const response = await fetch(request);
      const updatedRow = await response.json();
      const updatedRowIndex = updatedRow.index;
      delete updatedRow.index;
      // Create update mutate event and call mutate method
      // to notify dataprovider consumers that a row has been
      // updated
      const updatedRowKey = this.selectedKey;
      const updatedRowMetaData = { key: updatedRowKey };
      this.dataprovider.mutate({
        update: {
          data: [updatedRow],
          indexes: [updatedRowIndex],
          keys: new Set([updatedRowKey]),
          metadata: [updatedRowMetaData],
        },
      });
    }
  };
```

Delete Records

To add the ability to delete records, add elements to your HTML that accept input from the user and create a function that sends the record for deletion to the REST service.

1. Add elements to the app page that identifies records marked for deletion and enables the user to perform an action to delete them.

The `oj-button`'s `on-oj-action` attribute in the following example is bound to the `removeRow` function which is defined in the next step.

```
<oj-button id="removeButton" on-oj-action="[[removeRow]]"
disabled="[[disabledRemove]]">Remove</oj-button>
```

2. Add code to the ViewModel to delete the record or records submitted by the user.

```
// used to remove the selected row
removeRow = async () => {
  const currentRow = this.selectedRow;
  if (currentRow != null) {
    // Create and send request to delete row on REST server
    const request = new Request(
      `${this.restServerUrl}/${this.selectedKey}`,
      { method: "DELETE" }
    );
    const response = await fetch(request);
    const removedRow = await response.json();
    const removedRowIndex = removedRow.index;
    delete removedRow.index;
    // Create remove mutate event and call mutate method
    // to notify dataprovider consumers that a row has been
    // removed
    const removedRowKey = removedRow[this.keyAttributes];
    const removedRowMetaData = { key: removedRowKey };
    this.dataprovider.mutate({
      remove: {
        data: [removedRow],
        indexes: [removedRowIndex],
        keys: new Set([removedRowKey]),
        metadata: [removedRowMetaData],
      },
    });
  }
  this.disabledUpdate(true);
  this.disabledRemove(true);
};
```

10

Validate and Convert Input

Oracle JET includes validators and converters on a number of Oracle JET editable elements, including `oj-combobox`, `oj-input*`, and `oj-text-area`. You can use them as is or customize them for validating and converting input in your Oracle JET app.
Some editable elements such as `oj-checkboxset`, `oj-radioset`, and `oj-select` have a simple attribute for `required` values that implicitly creates a built-in validator.

> **Note:**
>
> The `oj-input*` mentioned above refers to the family of input components such as `oj-input-date-time`, `oj-input-text`, and `oj-input-password`, among others.

About Oracle JET Validators and Converters

Oracle JET provides converter classes that convert user input strings into the data type expected by the app and validator classes that enforce a validation rule on those input strings.

For example, you can use Oracle JET's `IntlDateTimeConverter` to convert a user-entered date to a date-only ISO string and then use `DateTimeRangeValidator` to validate that input against a specified date range. If the converters or validators included in Oracle JET are not sufficient for your app, you can create custom converters or validators.

About Validators

All Oracle JET editable elements support a `value` attribute and provide UI elements that allow the user to enter or choose a value. These elements also support other attributes that you can set to instruct the element how to validate its value.

An editable element may implicitly create a built-in converter and/or built-in validators for its normal functioning when certain attributes are set. For example, editable elements that support a `required` property create the required validator implicitly when the property is set to `true`. Other elements like `oj-input-date`, `oj-input-date-time`, and `oj-input-time` create a datetime converter to implement its basic functionality.

About the Oracle JET Validators

The following table describes the Oracle JET validators and provides links to the API documentation:

Validator	Description	Link to API	Module
`DateTimeRangeValidator`	Validates that the input date is between two dates, between two times, or within two date and time ranges	DateTimeRangeValidator	`ojvalidation-datetimerange`

Validator	Description	Link to API	Module
`DateRestrictionValidator`	Validates that the input date is not a restricted date	DateRestrictionValidator	`ojvalidation-daterestriction`
`LengthValidator`	Validates that an input string is within a specified length	LengthValidator	`ojvalidation-length`
`NumberRangeValidator`	Validates that an input number is within a specified range	NumberRangeValidator	`ojvalidation-numberrange`
`RegExpValidator`	Validates that the regular expression matches a specified pattern	RegExpValidator	`ojvalidation-regexp`
`RequiredValidator`	Validates that a required entry exists	RequiredValidator	`ojvalidation-required`

About Oracle JET Component Validation Attributes

The attributes that a component supports are part of its API, and the following validation specific attributes apply to most editable elements.

Element Attribute	Description
`converter`	When specified, the `converter` instance is used over any internal converter the element might create. On elements such as `oj-input-text`, you may need to specify this attribute if the value must be processed to and from a number or a date value.
`countBy`	When specified on `LengthValidator`, `countBy` enables you to change the validator's default counting behavior. By default, this property is set to `codeUnit`, which uses JavaScript's String length property to count a UTF-16 surrogate pair as length === 2. Set this to `codePoint` to count surrogate pairs as length ===1.
`max`	When specified on an Oracle JET element like `oj-input-date` or `oj-input-number`, the element creates an implicit range validator.
`min`	When specified on an Oracle JET element like `oj-input-date` or `oj-input-number`, the component creates an implicit range validator.
`required`	When specified on an Oracle JET element, the element creates an implicit required validator.
`validators`	When specified, the element uses these validators along with the implicit validators to validate the UI value. Can be implemented with `Validators` or `AsyncValidators` to validate the user input on the server asynchronously.

Some editable elements do not support specific validation attributes as they might be irrelevant to its intrinsic functioning. For example, `oj-radioset` and `oj-checkboxset` do not support a converter attribute since there is nothing for the converter to convert. For an exact list of attributes and how to use them, refer to the `Attributes` section in the element's API documentation. For Oracle JET API documentation, see API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET). Select the component you're interested in viewing from the API list.

About Oracle JET Component Validation Methods

Oracle JET editable elements support the following methods for validation purposes. For details on how to call this method, its parameters and return values, refer to the component's API documentation.

Element Method	Description
`reset()`	Use this method to reset the element by clearing all messages and messages attributes - `messagesCustom` - and update the element's display value using the attribute value. User entered values will be erased when this method is called.
`validate()`	Use this method to validate the component using the current display value.

For details on calling an element's method, parameters, and return values, see the `Methods` section of the element's API documentation in API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET). You can also find detail on how to register a callback for or bind to the event and for information about what triggers the events. Select the component you're interested in viewing from the API list.

About Oracle JET Converters

Oracle JET provides converters to convert date, date-time, number, color, and string.

These converters extend the `Converter` object which defines a basic contract for converter implementations. The converter API is based on the ECMAScript Internationalization API specification (ECMA-402 Edition 1.0) and uses the Unicode Common Locale Data Repository (CLDR) for its locale data. Both converters are initialized through their constructors, which accept options defined by the API specification. For additional information about the ECMA-402 API specification, see https://www.ecma-international.org/publications-and-standards/standards/ecma-402/. For information about the Unicode CLDR, see http://cldr.unicode.org.

The Oracle JET implementation extends the ECMA-402 specification by introducing additional options. You can use the converters with an Oracle JET component or instantiate and use them directly on the page. Each converter has a `ConverterOptions` type definition that specifies the conversion options it supports. The following table describes the available converters and provides a link to the API documentation for each converter, including detailed descriptions of the properties supported by each converter's `ConverterOptions` type definition.

Converter	Description	Link to API
`ColorConverter`	Converts Color object formats	ColorConverter
`IntlDateTimeConverter`	Parses a string into an ISO string format (`yyyy-mm-dd`) or converts an ISO string into a standard string format. For example: • `10/12/2020` ---> `2020-10-12` • `2020-10-12` ---> `10/12/2020`	IntlDateTimeConverter
`NumberConverter`	Converts a string into a number and formats a number into a locale-specific string	NumberConverter
`BigDecimalStringConverter`	Converts a big-decimal string into a locale-specific string. Use for very large numbers (greater than `Number.MAX_SAFE_INTEGER`) and numbers with large scale (more than 17 fractional digits). Can only be used with components that expect a string value, such as `oj-c-input-text`.	BigDecimalStringConverter

Converter	Description	Link to API
`LocalDateConverter`	Converts a date-only ISO string to a formatted string or a string to a date-only ISO string	LocalDateConverter

In addition to the API documentation, see the following demos in the Oracle JET Cookbook that show converters in use:

- Converters
- Color Palette
- Color Spectrum

The Oracle JET converters support lenient number and date parsing when the user input does not exactly match the expected pattern. The parser does the lenient parsing based on the leniency rules for the specific converter. For example, both `NumberConverter` and `BigDecimalStringConverter` remove unexpected characters. The `ConverterOptions` type definition typically includes a `lenientParse` property that you can set to `none` so that user input matches the expected input or an exception is thrown. For specific details, see the API documentation for the converter that you are using.

The resource bundles that hold the locale symbols and data used by the Oracle JET converters are downloaded automatically based on the locale set on the page when using RequireJS and the `ojs/ojvalidation-datetime` or `ojs/ojvalidation-number` module. If your app does not use RequireJS, the locale data will not be downloaded automatically.

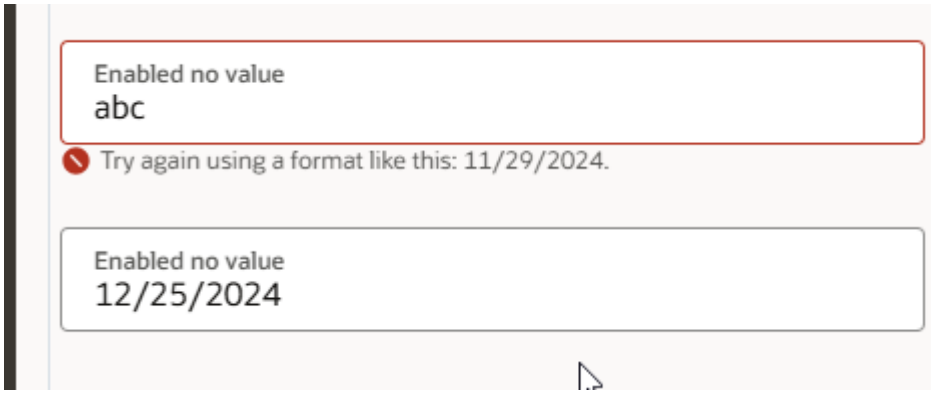
You can use the converters with an Oracle JET component or instantiate and use them directly on the page.

Use Oracle JET Converters with Oracle JET Components

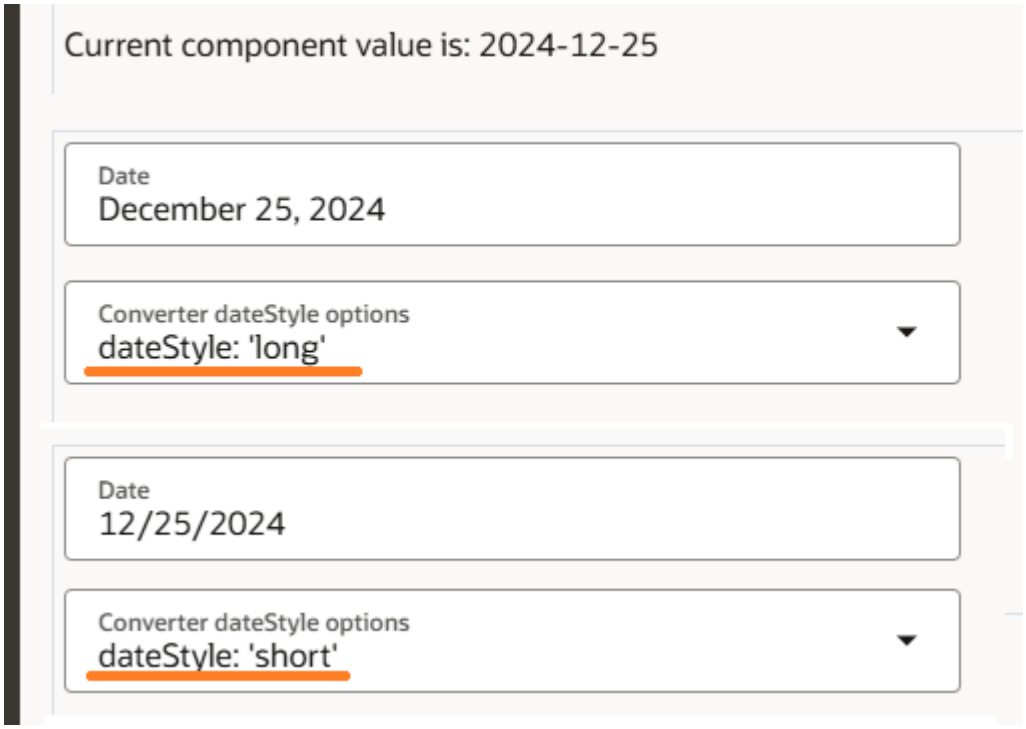
Oracle JET components that accept user input, such as `oj-c-input-date-text`, already include an implicit converter that parses user input. However, you can also specify an explicit converter on the component that will be used instead when converting data.

In the following example, taken from the Overview demo for the `oj-c-input-date-text` component in the Oracle JET Cookbook, an error message displays if you enter "`abc`" while the component accepts an input of "`12/25/24`" and renders it as "`12/25/2024`" using its implicit converter.

The error that the converter throws when there are errors during parsing or formatting operations is represented by the `ConverterError` object, and the error message is represented by an object of type `Message`. The messages that Oracle JET converters use are resources that are defined in the translation bundle included with Oracle JET.



You can also specify the converter directly on the component's converter `attribute`, if it exists. For example, the `oj-c-input-date-text` component uses the `converter` attribute to change the date style that the component renders to one of the converter options supported by `LocalDateConverter`.



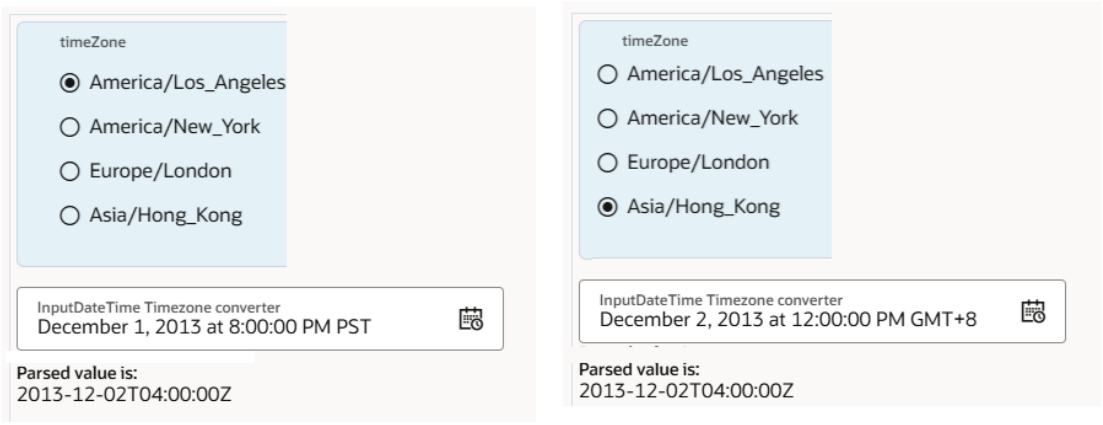
The Oracle JET Cookbook includes implementations of the options discussed here. See:

- Converters
- Input Date Text - Core Pack

Understand Time Zone Support in Oracle JET

Oracle JET input components, such as `oj-input-date-time`, support local time zone input. You can enable time zone support by using Oracle JET's `IntlDateTimeConverter`, which relies on JavaScript's `Intl.DateTimeFormat` API.

In the following image, the Input Date Time component's `converter` attribute references code that renders the input time of `2013-12-02T04:00:00Z` appropriately depending on whether the time zone is `America/Los_Angeles` or `Asia/Hong_Kong`.



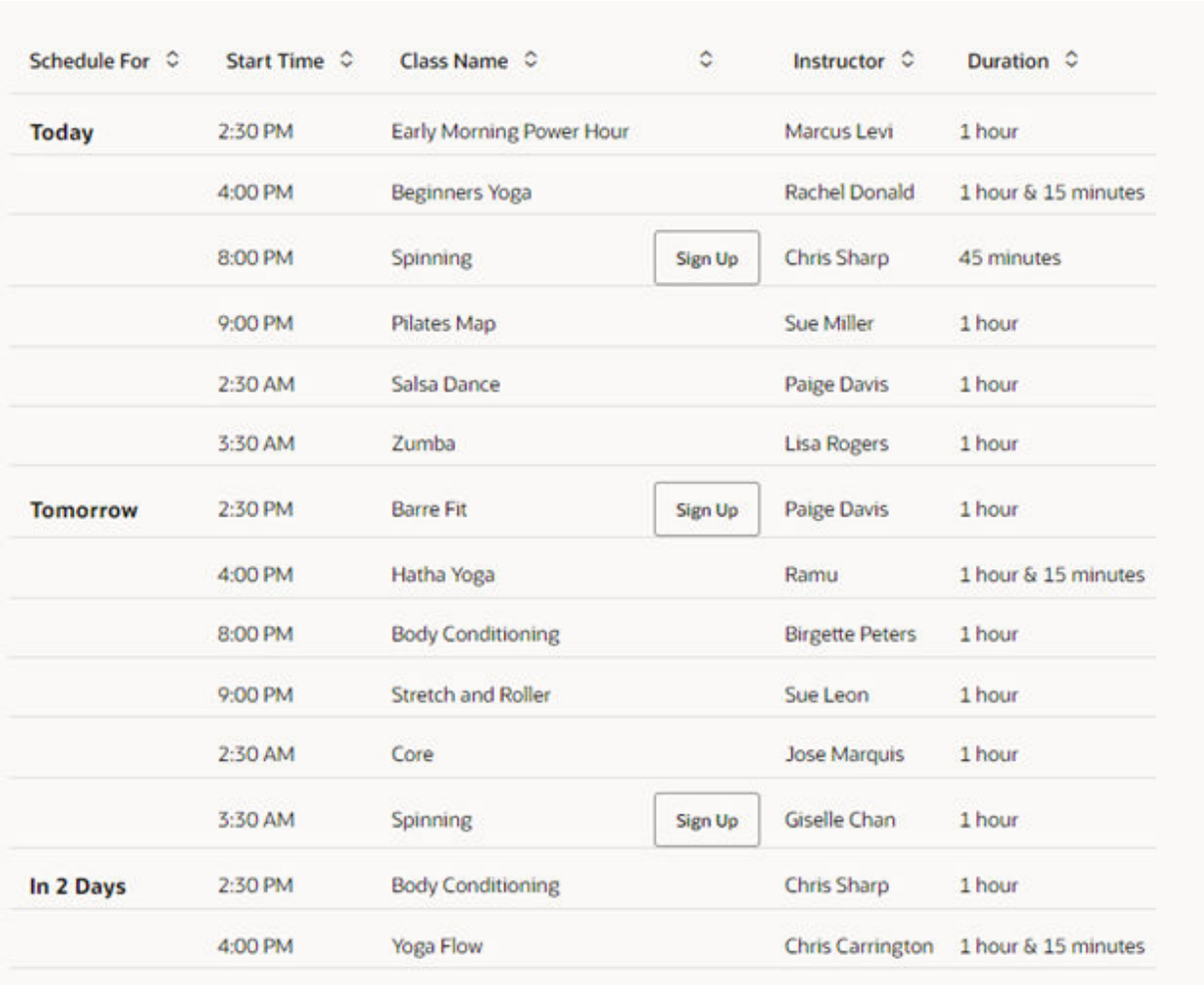
The Oracle JET Cookbook includes implementations of the options discussed here. See Input Date and Time - Time Zone.

For more information about Oracle JET's `IntlDateTimeConverter`, see IntlDateTimeConverter.

Use Custom Converters in Oracle JET

You can create custom converters in Oracle JET by extending `Converter`. Custom converters can be used with Oracle JET components, provided they don't violate the integrity of the component. As with the built-in Oracle JET converters, you can also use them directly on the page.

The following image shows an implementation of a custom converter that converts the current date to a relative term. The `Schedule For` column uses a `RelativeDateTimeConverter` to convert the date that the page is run in the `en-US` locale to display `Today`, `Tomorrow`, and the date in two days.



Schedule For	Start Time	Class Name		Instructor	Duration
Today	2:30 PM	Early Morning Power Hour		Marcus Levi	1 hour
	4:00 PM	Beginners Yoga		Rachel Donald	1 hour & 15 minutes
	8:00 PM	Spinning	Sign Up	Chris Sharp	45 minutes
	9:00 PM	Pilates Map		Sue Miller	1 hour
	2:30 AM	Salsa Dance		Paige Davis	1 hour
	3:30 AM	Zumba		Lisa Rogers	1 hour
Tomorrow	2:30 PM	Barre Fit	Sign Up	Paige Davis	1 hour
	4:00 PM	Hatha Yoga		Ramu	1 hour & 15 minutes
	8:00 PM	Body Conditioning		Birgette Peters	1 hour
	9:00 PM	Stretch and Roller		Sue Leon	1 hour
	2:30 AM	Core		Jose Marquis	1 hour
	3:30 AM	Spinning	Sign Up	Giselle Chan	1 hour
In 2 Days	2:30 PM	Body Conditioning		Chris Sharp	1 hour
	4:00 PM	Yoga Flow		Chris Carrington	1 hour & 15 minutes

To create and use a custom converter in Oracle JET:

1. Define the custom converter.

   The Oracle JET Cookbook's Custom Converter sample uses `RelativeDateTimeConverter` to wrap the Oracle JET `IntlDateTimeConverter` by providing a specialized `format()` method that turns dates close to the current date into relative terms for display. For example, in the `en-US` locale, the relative terms display `Today`, `Yesterday`, and `Tomorrow`. If a relative notation for the date value does not exist, then the date is formatted using the regular Oracle JET `format()` method supported by the Oracle JET `IntlDateTimeConverter`.

   For additional details about the `IntlDateTimeConverter` converter's supported methods, see the IntlDateTimeConverter API documentation.

2. Add code to your app that uses the custom converter, as in the following example.

   The code sample below shows how you could add code to your script to use the custom converter.

   ```
   let rdConverter = new RelativeDateTimeConverter(relativeDayOptions);
   ```

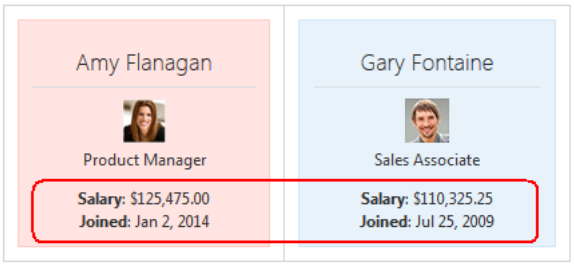
3. Add the Oracle JET element or elements that will use the custom converter to your page.

The Oracle JET Cookbook includes implementations of the options discussed here. See Converters (Custom).

Use Oracle JET Converters Without Oracle JET Components

If you want to use a converter without binding it to an Oracle JET component, create the converter using the constructor for the converter of your choice.

The Oracle JET Cookbook includes a demo that shows how to use the number and date time converters directly in your pages without binding them to an Oracle JET component. In the demo image, the salary is a number formatted as currency, and the start date is an ISO string formatted as a date.



The viewModel defines a `salaryConverter` method to format a number as currency and a `dateConverter` that formats the start date using the date format style and medium date format while the view HTML binds to the returned values, as shown in the following snippet for Amy Flanagan's salary and start date.

```
. . .
<p>Product Manager</p>
 <span style="white-space:nowrap;"><b>Salary</b>:
<span>
  <oj-bind-text value="[[amySalary]]"></oj-bind-text>
</span>
 </span>
 <br />
 <span style="white-space:nowrap;"><b>Joined</b>:
<span>
  <oj-bind-text value="[[amyStartDate]]"></oj-bind-text>
</span>
. . .
```

The Oracle JET Cookbook includes implementations of the options discussed here. See Create Converter.

About Oracle JET Validators

Oracle JET validators provide properties that allow callers to customize the validator instance. The properties are documented as part of the validators' API. Unlike converters where only one instance of a converter can be set on an element, you can associate one or more validators with an element.

When a user interacts with the element to change its value, the validators associated with the element run in order. When the value violates a validation rule, the `value` attribute is not populated, and the validator highlights the element with an error.

You can use the validators with an Oracle JET element or instantiate and use them directly on the page.

Use Oracle JET Validators with Oracle JET Components

Oracle JET editable elements, such as `oj-input-text` and `oj-input-date`, set up validators both implicitly, based on certain attributes they support such as `required`, `min`, `max`, and so on, and explicitly by providing a means to set up one or more validators using the component's `validators` attribute.

For example, the following code shows an `oj-input-date` element that uses the default validator supplied by the component implicitly. When the `oj-input-date` component reads the `min` and `max` attributes, it creates the implicit `DateTimeRangeValidator`.

```
<oj-form-layout id="validator-example">
  <oj-input-date
    id="dateTimeRange1"
    value="{{dateValue1}}"
    min="2000-01-01T08:00:00.000"
    help.instruction="enter a date that falls in the current
                  millenium and not greater than today's date."
    max="[[todayIsoDate]]"
    label-hint="'min' attribute and 'max' option">
  </oj-input-date>
</oj-form-layout>
```

The script to create the view model for this example is shown below.

- TypeScript
- JavaScript

TypeScript

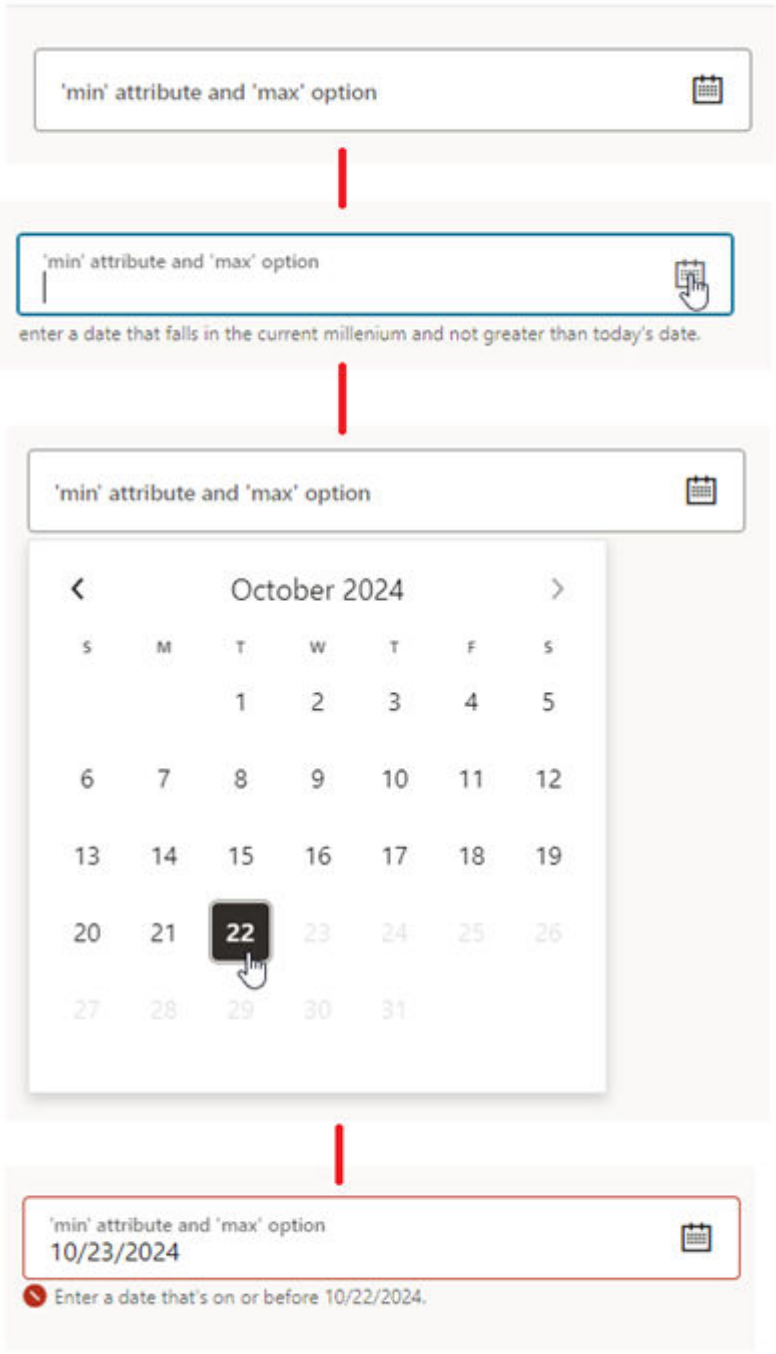
```
import * as ko from "knockout";
import * as ConverterUtilsI18n from "ojs/ojconverterutils-i18n";
import "ojs/ojknockout";
import "ojs/ojformlayout";
import "ojs/ojdatetimepicker";

class DemoViewModel {

  dateValue1: ko.Observable<string>;
  dateValue2: ko.Observable<string>;
  todayIsoDate: ko.Observable<string>;
  milleniumStartIsoDate: ko.Observable<string>;

constructor() {
```

```
  this.dateValue1 = ko.observable("");
  this.dateValue2 = ko.observable("");
  this.todayIsoDate = ko.observable(
    ConverterUtilsI18n.IntlConverterUtils.dateToLocalIso(new Date())
  );
  this.milleniumStartIsoDate = ko.observable(
    ConverterUtilsI18n.IntlConverterUtils.dateToLocalIso(new Date(2000, 0, 1))
    );
  }
}

export = DemoViewModel;
```

JavaScript

```
define([
  "knockout",
  "ojs/ojconverterutils-i18n",
  "ojs/ojknockout",
  "ojs/ojdatetimepicker",
  "ojs/ojformlayout",
], function (
  ko,
  ConverterUtilsI18n
) {
  function DemoViewModel() {

  this.dateValue1 = ko.observable();
  this.dateValue2 = ko.observable();
  this.todayIsoDate = ko.observable(
   ConverterUtilsI18n.IntlConverterUtils.dateToLocalIso(new Date())
  );
  this.milleniumStartIsoDate = ko.observable(
   ConverterUtilsI18n.IntlConverterUtils.dateToLocalIso(new Date(2000, 0, 1))
  );
 }
  return DemoViewModel;
});
```

When the user runs the page, the `oj-input-date` element displays an input field with a calendar icon. The `help.instruction` attribute set on the element displays below the input field when you click on the input field. If you input data that is not within the expected range, the built-in validator displays an error message with the expected range.



The error thrown by the Oracle JET validator when validation fails is represented by the `ValidatorError` object, and the error message is represented by an object of type `Message`. The messages and hints that Oracle JET validators use when they throw an error are

resources that are defined in the translation bundle included with Oracle JET. For more information about messaging in Oracle JET, see Work with User Assistance.

You can also specify the validator on the element's `validators` attribute, if it exists. The code sample below shows another `oj-input-date` element that calls a function which specifies the `DateTimeRangeValidator` validator (`dateTimeRange`) in the `validators` attribute.

```
<oj-form-layout id="validator-example">
 <oj-input-date
  id="dateTimeRange2"
  value="{{dateValue2}}"
  validators="[[validators]]"
  help.instruction="enter a date that falls in the current millenium and
                    not greater than today's date."
  label-hint="'dateTimeRange' type in 'validators' option">
 </oj-input-date>
</oj-form-layout>
```

The code below shows the additions to the viewModel with options that set the valid minimum and maximum dates and a hint that displays when the user sets the focus in the field.

- TypeScript
- JavaScript

TypeScript

```
import * as ko from "knockout";
import * as ConverterUtilsI18n from "ojs/ojconverterutils-i18n";
import AsyncDateTimeRangeValidator
                = require("ojs/ojasyncvalidator-datetimerange");
import * as DateTimeConverter from "ojs/ojconverter-datetime";
import "ojs/ojknockout";
import "ojs/ojformlayout";
import "ojs/ojdatetimepicker";

class DemoViewModel {
  dateValue1: ko.Observable<string>;
  dateValue2: ko.Observable<string>;
  todayIsoDate: ko.Observable<string>;
  milleniumStartIsoDate: ko.Observable<string>;
  validators: ko.Computed<AsyncDateTimeRangeValidator<string>[]>;

  constructor() {
    this.dateValue1 = ko.observable("");
    this.dateValue2 = ko.observable("");
    this.todayIsoDate = ko.observable(
      ConverterUtilsI18n.IntlConverterUtils.dateToLocalIso(new Date())
    );
    this.milleniumStartIsoDate = ko.observable(
    ConverterUtilsI18n.IntlConverterUtils.
                    dateToLocalIso(new Date(2000, 0, 1))
    );
```

```
    this.validators = ko.computed(() => {
      return [
        new AsyncDateTimeRangeValidator({
          max: this.todayIsoDate(),
          min: this.milleniumStartIsoDate(),
          hint: {
            inRange: "Enter a date that falls in the
                      current millennium.",
          },
          converter: new DateTimeConverter.IntlDateTimeConverter({
            day: "2-digit",
            month: "2-digit",
            year: "2-digit",
          }),
        }),
      ];
    });
  }
}

export = DemoViewModel;
```

JavaScript

```
define([
  "knockout",
  "ojs/ojconverterutils-i18n",
  "ojs/ojasyncvalidator-datetimerange",
  "ojs/ojconverter-datetime",
  "ojs/ojknockout",
  "ojs/ojdatetimepicker",
  "ojs/ojformlayout",
], function (
  ko,
  ConverterUtilsI18n,
  AsyncDateTimeRangeValidator,
  DateTimeConverter
) {
  function DemoViewModel() {

    this.dateValue1 = ko.observable();
    this.dateValue2 = ko.observable();
    this.todayIsoDate = ko.observable(
      ConverterUtilsI18n.IntlConverterUtils.
                      dateToLocalIso(new Date())
    );
    this.milleniumStartIsoDate = ko.observable(
      ConverterUtilsI18n.IntlConverterUtils.
             dateToLocalIso(new Date(2000, 0, 1))
    );

    this.validators = ko.computed(
      function () {
        return [
```

```
            new AsyncDateTimeRangeValidator({
              max: this.todayIsoDate(),
              min: this.milleniumStartIsoDate(),
              hint: {
                inRange: "Enter a date that falls
                          in the current millennium.",
              },
              converter: new DateTimeConverter.
                                IntlDateTimeConverter({
                day: "2-digit",
                month: "2-digit",
                year: "2-digit",
              }),
            }),
          ];
        }.bind(this)
      );
    }

    return DemoViewModel;
});
```

When the user runs the page for the `en-US` locale, the `oj-input-date` element displays an input field that expects the user's input date to be between `01/01/2000` and the current date. When entering a date value into the field, the date converter accepts alternate input as long as it can parse it unambiguously. This offers end users a great deal of leniency when entering date values. For example, typing `1-2-3` converts to a `Date` that falls on the 2nd day of January, 2003. If the `Date` value also happens to fall in the expected `Date` range set in the validator, then the value is accepted. If validation fails, the component displays an error.

Oracle JET elements can also use a `regExp` validator. If the regular expression pattern requires a backslash, while specifying the expression within an Oracle JET element, you need to use double backslashes. The options that each validator accepts are specified in API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET).

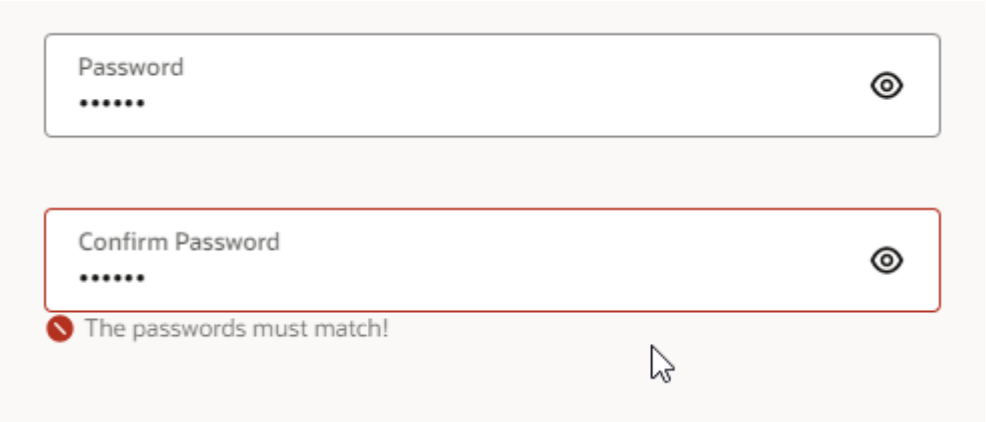
The Oracle JET Cookbook contains the complete example discussed in this section as well as examples that show the built-in validators for date restrictions, length, number range, regular expression, and required fields. For details, see Validators.

For more information about Oracle JET component validation, see Understand How Validation and Messaging Works in Oracle JET Editable Components.

Use Custom Validators in Oracle JET

You can create custom validators in Oracle JET that you can reference like built-in validators from the `validators` attribute.

The following image shows a custom validator that displays an error message if the user's password doesn't match.



To create and use a custom validator:

1. Define the custom validator.

   The Oracle JET Cookbook sample defines an `equalToPassword` custom validator of type `Validator`. Because the custom validator provides the methods expected of a Validator instance, Oracle JET accepts it.

2. Add code to your app that uses the custom validator.

   The code sample below shows how you could add code to your view page to use the custom validator. In this example, both input fields are defined as `oj-c-input-password` elements. The first instance validates that the user entered a password that meets the app's password requirements. The second instance uses the `equalToPassword` validator to verify that the password in the second field is equal to the password entered in the first field.

```
<oj-form-layout id="custom-validator-example">
  <oj-c-input-password
    id="password"
    required
    value="{{password}}"
    validators="[[validators]]"
    label-hint="Password"
    mask-icon="visible"></oj-c-input-password>
  <oj-c-input-password
    id="cpassword"
    value="{{passwordRepeat}}"
    validators="[[[equalToPassword]]]"
    label-hint="Confirm Password"
    mask-icon="visible"></oj-c-input-password>
</oj-form-layout>
```

   The Oracle JET Cookbook contains the complete code sample discussed in this section. See Validators (Custom).

About Asynchronous Validators

Oracle JET input components support asynchronous server-side validation via the `validators` attribute. That means you can check input values against server data without the need to submit a form or refresh a page.

Two example scenarios illustrate where you can use asynchronous server-side validation:

- In a form that collects new user data, you validate input in an email field to check if the input value has been registered previously.
- Set number range validators that check against volatile data. For example, on an e-commerce website, you can check the user's cart against the available inventory and inform the user if the goods are unavailable without them submitting the cart for checkout.

The Oracle JET Cookbook has a sample that uses the `validators` attribute and dummy data to simulate server-side validation.

The following code shows an `oj-input-text` element with the `validators` attribute set to `validators` and `asyncValidator` observables in the viewModel code. The `validators` attribute must be of type `AsyncValidator` to fulfill the API contract required to create the asynchronous validator.

```
<oj-form-layout id="fl1">
  <oj-c-input-text
    id="input-text"
    required
    label-hint="Quantity Limit"
    on-valid-changed="[[validChangedListener]]"
    validators="[[[validators, asyncValidator]]]"
    value="{{quantityLimit}}"
    converter="[[currencyConverter]]">
  </oj-c-input-text>
</oj-form-layout>
```

The viewModel code includes a number range validator created in the asyncValidator object that returns a Promise. A Promise object represents a value that may not be available yet, but will be resolved at some point in the future. In asynchronous validation, the `AsyncValidator.validate()` function returns a Promise that evaluates to Boolean `true` if validation passes and if validation fails, it returns an Error. For more information, see the `validators` attribute section of Input Text or see Promise (MDN).

The Oracle JET Cookbook includes implementations of the options discussed here. See Async Validators.

11

Work with User Assistance

The Oracle JET user assistance framework includes support for user assistance on the editable components in the form of help, hints, and messaging that you can customize as needed for your app. Editable components include `oj-checkboxset`, `oj-color*`, `oj-combobox*`, `oj-input*`, `oj-radioset`, `oj-select*`, `oj-slider`, `oj-switch`, and `oj-text-area`.

> **Note:**
>
> The `oj-input*` mentioned above refers to the family of input components such as `oj-input-date-time`, `oj-input-text`, and `oj-input-password`, among others. `oj-color*`, `oj-combobox*`, and `oj-select*` each represent two components.

> **Tip:**
>
> To add tooltips to plain text or other non-editable components, use `oj-popup`. See the Tooltip example in the Oracle JET Cookbook at Popups.

Understand Oracle JET's Messaging APIs on Editable Components

Oracle JET provides a messaging API to support messaging on Oracle JET editable components.

Editable components include the following:

- `oj-checkboxset`
- `oj-color-palette`
- `oj-color-spectrum`
- `oj-combobox-many`
- `oj-combobox-one`
- `oj-input-date`
- `oj-input-date-time`
- `oj-input-number`
- `oj-input-password`
- `oj-input-text`
- `oj-input-time`
- `oj-radioset`

- `oj-select-many`
- `oj-select-one`
- `oj-slider`
- `oj-switch`
- `oj-text-area`

The Oracle JET Cookbook also includes descriptions and examples for working with each component at: Form Controls.

About Oracle JET Editable Component Messaging Attributes

The following attributes impact messaging on editable elements.

Element Attribute	Description
`converter`	Default `converter` hint displayed as placeholder text when a placeholder attribute is not already set.
`display-options`	JSON object literal that specifies the location where the element should display auxiliary content such as `messages`, `converterHint`, `validatorHint`, and `helpInstruction` in relation to itself. Refer to the element's API documentation for details.
`help`	Help message displayed on an `oj-label` element when the user hovers over the Help icon. No formatted text is available for the message. The `oj-label` has two exclusive attributes, `help.definition` and `help.source`. The `help.definition` attribute's value appears and the attribute's value is read by a screen reader when you hover with a mouse, when you tab into the Help icon, or when you press and hold on a device. The default value is `null`. The `help.source` attribute's value is a link, which is opened when you click with a mouse or tap on a device. The default value is `null`.
`help.instruction`	Displays text in a note window that displays when the user sets focus on the input field. You can format the text string using standard HTML formatting tags.
`messages-custom`	List of messages that the app provides when it encounters business validation errors or messages of other severity type.
`placeholder`	The placeholder text to set on the element.
`translations`	Object containing all translated resources relevant to the component and all its superclasses. Use sub-properties to modify the component's translated resources.
`validators`	List of validators used by element when performing validation. Validator hints are displayed in a note window by default.

See the `Attributes` section of the element's API documentation in API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) for additional details about its messaging properties. Select the component you're interested in viewing from the API list.

About Oracle JET Component Messaging Methods

Editable value components support the following method for messaging purposes.

Component Event	Description
`showMessages`	Takes all deferred messages and shows them. If there were no deferred messages this method simply returns. When the user sets focus on the component, the deferred messages will display inline.

See the `Methods` section of the component's API documentation in the API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) for details on how to call the method, its parameters, and return value. Select the component you're interested in viewing from the list.

Understand How Validation and Messaging Works in Oracle JET Editable Components

The actions performed on an Oracle JET component, the properties set on it, and the methods called on it, all instruct the component on how it should validate the value and what content it should show as part of its messaging.

Editable components always perform either normal or deferred validation in some situations. In other situations, the editable component decides to perform either normal or deferred validation based on the component's state. Understanding the normal and deferred validation process may be helpful for determining what message properties to set on your components.

- Normal Validation: During normal validation, the component clears all `messages` properties (`messages-custom`), parses the UI value, and performs validation. Validation errors are reported to the user immediately. If there are no validation errors, the `value` attribute is updated, and the value is formatted and pushed to the display.

  The editable component always runs normal validation when:

  – The user interacts with an editable component and changes its value in the UI.

  – The app calls `validate()` on the component.

  > **Note:**
  >
  > When the app changes certain properties, the component might decide to run normal validation depending on its current state. See Mixed Validation below for additional details.

- Deferred Validation: Uses the `required` validator to validate the component's `value`. The `required` validator is the only validator that participates in deferred validation. During deferred validation all `messages` properties are cleared unless specified otherwise. If the `value` fails deferred validation, validation errors are not shown to the user immediately.

  The editable component always runs deferred validation when:

  – A component is created. None of the `messages` properties are cleared.

  – The app calls the `reset()` method on the component.

  – The app changes the `value` property on the component programmatically.

  > **Note:**
  >
  > When the app changes certain properties programmatically, the component might decide to run deferred validation depending on its current state. See Mixed Validation below for additional details.

- Mixed Validation: Runs when the following properties are changed or methods are called by the app. Either deferred or normal validation is run based on the component's current

state, and any validation errors are either hidden or shown to the user. Mixed validation runs when:

- `converter` property changes
- `disabled` property changes
- `readOnly` properties change
- `required` property changes
- `validators` property changes
- `refresh()` method is called

The Oracle JET Cookbook includes additional examples that show normal and deferred validation at Validators (Component). For additional information about the validators and converters included with Oracle JET, see Validate and Convert Input.

Understand How an Oracle JET Editable Component Performs Normal Validation

An Oracle JET editable component runs normal validation when the user changes the value in the UI or when the app calls the component's `validate()` method. In both cases, error messages are displayed immediately.

About the Normal Validation Process When User Changes Value of an Editable Component

When a user changes an editable value:

1. All `messages-custom` messages are cleared. An `onMessagesCustomChanged` event is triggered if applicable and if the change in value is obvious.
2. If a converter is set on the component, the UI value is parsed. If there is a parse error, then processing jumps to step **5**.
3. If one or more validators are set on the component:
   a. The parsed (converted) value is validated in sequence using the specified validators, with the implicit required validator being the first to run if present. The value that is passed to the implicit required validator is trimmed of white space.
   b. If the validator throws an error, the error is remembered, and the next validator runs if it exists.

      After all validators complete, if there are one or more errors, processing jumps to step **5**.
4. If all validations pass:
   a. The parsed value is written to the component's `value` attribute, and an `onValueChanged` event is triggered for the `value` attribute.
   b. The new value is formatted for display using the converter again and displayed on the component.

> **Note:**
>
> If the component's value property happens to be bound to a Knockout observable, then the value is written to the observable as well.

5. If one or more errors occurred in an earlier step:
   a. The component's `value` property remains unchanged.
   b. Errors are displayed by default inline with the component. The user can also view the details of the error by setting focus on the component.
6. When the user fixes the error, the validation process begins again.

About the Normal Validation Process When Validate() is Called on Editable Component

The `validate()` method validates the component's current display value using the converter and all validators registered on the component and updates the `value` attribute if validation passes.

The method is only available on editable components where it makes sense, for example, `oj-input-number`. For details about the `validate()` method, see validate().

Understand How an Oracle JET Editable Component Performs Deferred Validation

An Oracle JET editable component runs deferred validation when the component is created, when its `value` or `required` property is changed programmatically, or when the component's `reset()` method is called. This section provides additional detail about the deferred validation process when an Oracle JET editable component is created and when the `value` property is changed programmatically.

You can also find additional detail in the API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET). Select the component you're interested in from the navigation list.

About the Deferred Validation Process When an Oracle JET Editable Component is Created

When an editable element is created, as one of the last steps, it runs deferred validation on the component's initialized value.

1. The required validator is run, and a validation error raised if the value is empty or null.
2. If a validation error is raised, the component updates the messaging framework. No messaging themes are applied on the component nor does it show the error message in the note window because the validation error message is deferred.

> **Note:**
>
> Page authors can call `showMessages()` at any time to reveal deferred messages.

About the Deferred Validation Process When value Property is Changed Programmatically

An Oracle JET editable element's `value` property can change programmatically if:

- The page has code that changes the element's value attribute, or
- The page author refreshes the ViewModel observable with a new server value.

In both cases, the element will update itself to show the new value as follows:

1. All messages properties are cleared on the editable element and `onMessagesCustomChanged` events triggered if applicable.
2. An `onValueChanged` event is triggered for the `value` attribute if applicable.
3. If a converter is set on the element, the `value` attribute is retrieved and formatted before it's displayed. If there is a format error, then processing jumps to step 5. Otherwise the formatted value is displayed on the element.
4. Deferred validators are run on the new value. Any validation errors raised are communicated to the messaging framework, but the errors themselves are not shown to the user.
5. If there was a formatting error, the validation error message is processed and the component's `messages-custom` attribute updated. Formatting errors are shown right away.

> **Note:**
>
> Page authors should ensure that the value you set is the expected type as defined by the component's API and that the value can be formatted without any errors for display.

Use Oracle JET Messaging

Use the Oracle JET messaging framework to notify an Oracle JET app of a component's messages and validity as well as notify an Oracle JET component of a business validation failure.

Notify an Oracle JET Editable Component of Business Validation Errors

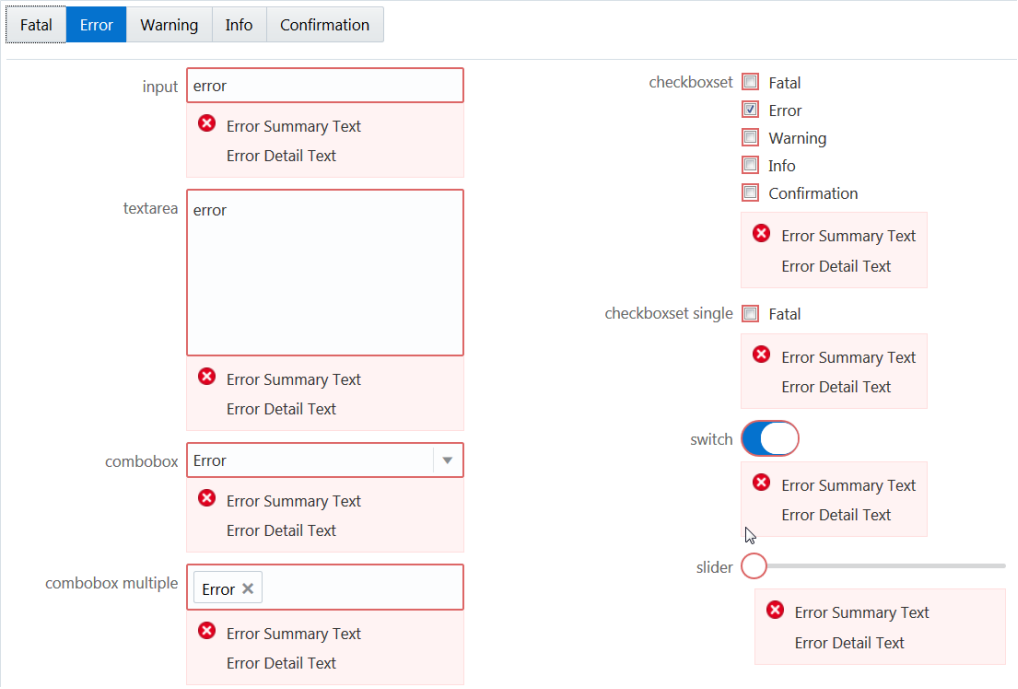
You can notify Oracle JET editable elements of business validation errors using the `messages-custom` attribute and the `showMessages()` method.

Use the messages-custom Attribute

Your app can use this attribute to notify Oracle JET components to show new or updated custom messages. These could be a result of business validation that originates in the viewModel layer or on the server. When this property is set, the message shows to the user immediately. The `messages-custom` attribute takes an Object that duck-types `Message` with detail, summary, and severity fields.

In this example, the severity type button is toggled and a message of the selected severity type is pushed onto the `messages-custom` array. The `messages-custom` attribute is set on every form

control in this example. When the `messages-custom` attribute is changed, it is shown immediately. In this example, the user selected the `Error` severity type, and the associated messages are shown for the various text input and selection components.



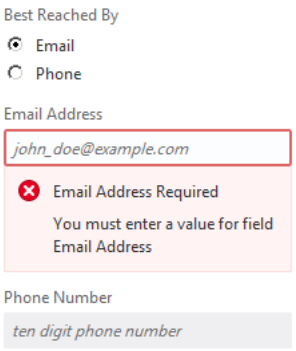
In this example an observable, `appMessages`, is declared and is bound to the `messages-custom` attribute for various elements. The following code describes how you can associate the observable with the `messages-custom` attribute for the `oj-switch` element .

```
<oj-switch id="switch" value="{{switchValue}}"
           messages-custom="{{appMessages}}">
</oj-switch>
```

In the corresponding JavaScript file, set the severity type and pass it to the observable, `appMessages`, to display associated messages.

```
if (summary && detail)
{
 msgs.push({summary: summary, detail: detail, severity: type});
}
self.appMessages(msgs);
```

In the example below, an instance of a cross-field custom validator is associated with the `emailAddress` observable, ensuring that its value is non-empty when the user chooses **Email** as their contact preference.



In the corresponding JavaScript file you must set the `messages-custom` attribute as `emailAddressMessages`.

```
<oj-input-text id="emailId" type="email"  name="emailId"
               placeholder="john_doe@example.com" value="{{emailAddress}}"
               messages-custom="{{emailAddressMessages}}"
               disabled="[[contactPref() !== 'email']]">
</oj-input-text>
```

For the complete example and code used to create the custom validator, see Cross-Field Validation. The demo uses a custom validator to validate an observable value against another. If validation fails the custom validator updates the `messages-custom` attribute.

Use the showMessages() Method on Editable Components

Use this method to instruct the component to show its deferred messages. When this method is called, the Oracle JET editable component automatically takes all deferred messages and shows them. This causes the component to display the deferred messages to the user.

For an example, see Show Deferred Messages.

Understand the oj-validation-group Component

The `oj-validation-group` component is an Oracle JET element that tracks the validity of a group of components and allows a page author to enforce validation best practices in Oracle JET apps.

Apps can use this component to:

- Determine whether there are invalid components in the group that are currently showing messages using the `invalidShown` value of the `valid` property.
- Determine whether there are invalid components that have deferred messages, such as messages that are currently hidden in the group, using the `invalidHidden` value of the `valid` property.
- Set focus on the first enabled component in the group using the `focusOn()` method. They can also focus on the first enabled component showing invalid messages using `focusOn("@firstInvalidShown")`.
- Show deferred messages on all tracked components using the `showMessages()` method.

For details about the `oj-validation-group` component's attributes and methods, see oj-validation-group.

Track the Validity of a Group of Editable Components Using oj-validation-group

You can track the validity of a group of editable components by wrapping them in the `oj-validation-group` component.

The `oj-validation-group` searches all its descendants for a `valid` property, and adds them to the list of components it is tracking. When it adds a component, it does not check the tracked component's children since the component's valid state should already be based on the valid state of its children, if applicable.

When it finds all such components, it determines its own `valid` property value based on all the enabled (including hidden) components it tracks. Any disabled or readonly components are ignored in calculating the `valid` state.

The most invalid component's valid property value will be the `oj-validation-group` element's valid property value. When any of the tracked component's valid value changes, `oj-validation-group` will be notified and will update its own valid value if it has changed.

The following code sample shows how an `oj-validation-group` can be used to track the overall validity of a typical form.

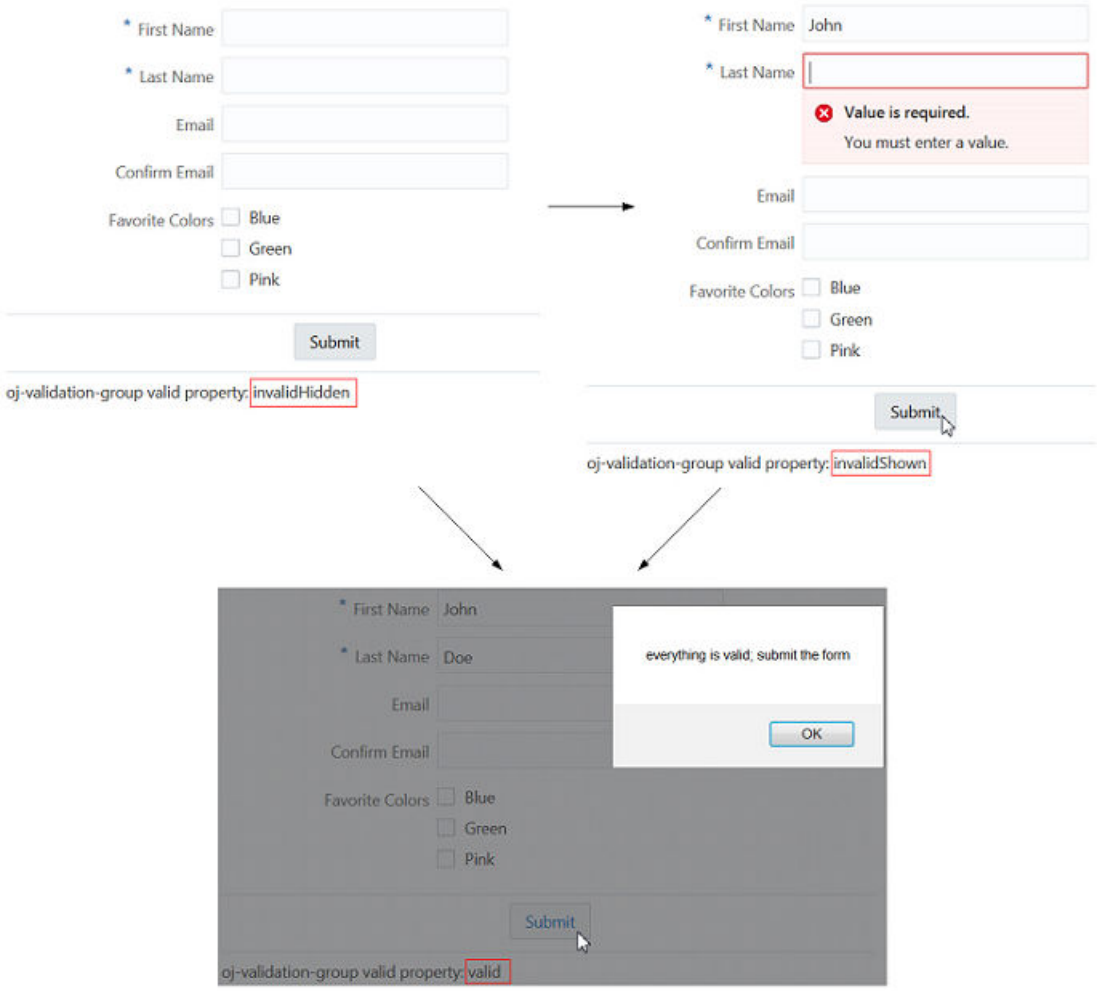
```
<div id="validation-usecase">
  <oj-validation-group id="tracker" valid="{{groupValid}}">
    <oj-form-layout label-edge="inside" id="fl1">

      <oj-input-text id="firstname"  required
                     autocomplete="off"
                     label-hint="First Name"
                     name="firstname" >
      </oj-input-text>
      <oj-input-text id="lastname" required
                     value="{{lastName}}"
                     autocomplete="off"
                     label-hint="Last Name">
      </oj-input-text>
      <oj-input-text id="email"
                     on-value-changed="[[firstEmailValueChanged]]"
                     autocomplete="off"
                     label-hint="Email"
                     value="{{email}}" >
      </oj-input-text>
      <oj-input-text id="email2"
                     autocomplete="off"
                     label-hint="Confirm Email"
                     validators="[[emailMatchValidator]]"
                     value="{{email2}}">
      </oj-input-text>
      <oj-checkboxset id="colors" label-hint="Favorite Colors">
        <oj-option id="blueopt" value="blue">Blue</oj-option>
        <oj-option id="greenopt" value="green">Green</oj-option>
        <oj-option id="pinkopt" value="pink">Pink</oj-option>
      </oj-checkboxset>
    </oj-form-layout>
  </oj-validation-group>
  <hr/>
  <div class="oj-flex">
    <div class="oj-flex-item"> </div>
    <div class="oj-flex-item">
```

```
      <oj-button id="submitBtn"
                 on-oj-action="[[submit]]">Submit</oj-button>
    </div>
  </div>
  <hr/>
  <span>oj-validation-group valid property: </span>
  <span id="namevalid">
    <oj-bind-text value="[[groupValid]]"></oj-bind-text>
  </span>
</div>
```

A portion of the script to create the view model for this example is shown below. This portion pertains to the `oj-validation-group` used above. The full script is contained in the Cookbook sample linked below.

```
require([ 'knockout', 'ojs/ojbootstrap', 'ojs/ojknockout', 'ojs/ojbutton', 'ojs/
ojcheckboxset', 'ojs/ojformlayout', 'ojs/ojinputtext', 'ojs/ojvalidationgroup'],
  function (ko, Bootstrap)
  // this callback gets executed when all required modules
  // for validation are loaded
  {
    function DemoViewModel() {
      var self = this;
      self.tracker = ko.observable();

      ...

      // to show the oj-validation-group's valid property value
      self.groupValid = ko.observable();

      // User presses the submit button
      self.submit = function () {

        var tracker = document.getElementById("tracker");

        if (tracker.valid === "valid") {
          // submit the form would go here
          alert("everything is valid; submit the form");
        }
        else {
           // show messages on all the components that have messages hidden.
           tracker.showMessages();
           tracker.focusOn("@firstInvalidShown");
        }
      };
    };

    Bootstrap.whenDocumentReady().then(
       function ()
       {
         ko.applyBindings(new DemoViewModel(), document.getElementById('validation-
usecase'));
       }
    );
  });
```

The figure below shows the output for the code sample. The status text at the bottom of each instance shows the valid state of the `oj-validation-group`, and by extension, the form.



The Oracle JET Cookbook contains the complete example used in this section. See Form Fields.

For an example showing the `oj-validation-group` used for cross-field validation, see Cross-Field Validation.

Create Page and Section Level Messaging

Oracle JET includes a Message Banner (`oj-message-banner`) element. Use it to display brief, medium disruption, semi-permanent messages on your app page or on sections within a page that communicate relevant and useful information to your app users.

The Cookbook contains various examples that showcase usage of the `oj-message-banner` element and the API doc provides details about its attributes and methods. See Message Banner demo in the Oracle JET Cookbook and the Message Banner entry in the *Oracle® JavaScript Extension Toolkit (JET) API Reference for Oracle JET*.

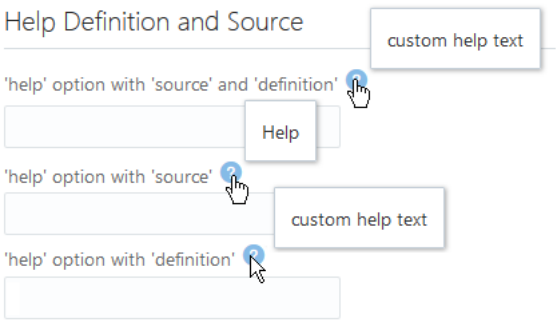
Configure an Editable Component's oj-label Help Attribute

Configure an `oj-label` element's `help` attribute to add a **Help** icon that displays descriptive text, includes a clickable link to a URL for additional information, or displays both the help text and clickable link when the user hovers over it.

The `help` attribute includes two sub-properties that control the help definition text and help icon:

- `definition`: Contains the help definition text that displays when the user does one of the following:
  - hovers over the help icon
  - tabs into the help icon with the keyboard
  - presses and holds the help icon on a mobile device
- `source`: Contains the URL to be used in the help icon's link

The following image shows three `oj-label` components configured to use the help attribute. The top component is configured with both a `definition` and `source` help sub-property, and the image shows the text and clickable pointer that displays when the user hovers over the help icon. In the middle image, the `oj-label` component includes a help icon that links to a URL when the user clicks it. In the bottom image, the `oj-label` displays custom help text when the user hovers over the label or help icon.



Before you begin:

- Familiarize yourself with the process of adding an Oracle JET element to your page. See Add an Oracle JET Component to Your Page.

To configure an `oj-label` element's `help` property:

1. Add the `oj-label` element to your page.
2. In the markup, add the `help` attribute and the `definition` or `source` sub-property.

   The markup for the `ojInputText` components is shown below. Each `ojInputText` component uses the `ojComponent` binding to define the component and set the help sub-properties. In this example, the user will be directed to `http://www.oracle.com` after clicking **Help**.

```
<div id="form-container" class="oj-form">
  <h3 class="oj-header-border">Help Definition and Source</h3>
  <oj-label id="ltext10" for="text10" help.definition="custom help text"
```

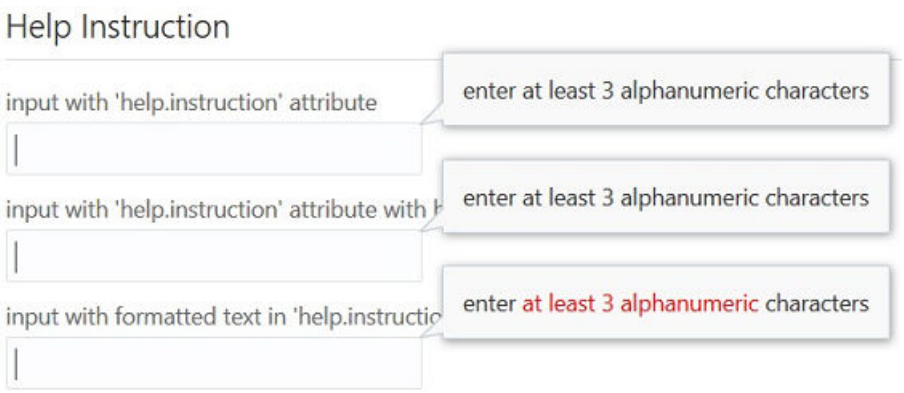
```
    help.source="http://www.oracle.com">'help' property with 'source' and
'definition'</oj-label>
  <oj-input-text id="text10" name="text10" value="{{text}}"></oj-input-
text>

  <oj-label id="ltext11" for="text11"
    help.source="http://www.oracle.com">'help' property with 'source'</oj-label>
  <oj-input-text id="text11" name="text11" value="{{text}}"></oj-input-text>

  <oj-label id="ltext12" for="text12"
    help.definition="custom help text">'help' property with 'definition'</oj-
label>
  <oj-input-text id="text12" name="text12" value="{{text}}"></oj-input-text>
</div>
```

See the Oracle JET Cookbook at Help and Title for the complete example to configure the `help` property on the ojInputText component.

Configure an Editable Component's help.instruction Attribute

Configure an editable component's help text using the `help.instruction` attribute. This will display a note window with advisory text (often called a tooltip) when the input component has focus.

The following image shows two `oj-input-text` elements configured to display text when the user sets focus in the input field. In the first example, the `help.instruction` attribute is defined for the `oj-input-text` element without formatting. In the second example, the attribute value has HTML formatting added to the advisory text.



Before you begin:

- Familiarize yourself with the process of adding an Oracle JET element to your page. See Add an Oracle JET Component to Your Page.

To configure an editable element's `help.instruction` attribute:

1. Add the editable element to your page.
2. In the markup, add the `help.instruction` attribute in the element tag.

   The following code sample shows the markup for defining the three `oj-input-text` components.

```
<div id="form-container" class="oj-form">
  <h3 class="oj-header-border">Help Instruction</h3>
```

```
    <oj-label for="text20">input with 'help.instruction' attribute</oj-label>
    <oj-input-text id="text20" name="text20"  autocomplete="off"
validators="[[validators]]"
        help.instruction="enter at least 3 alphanumeric characters"
value="{{text}}"></oj-input-text>

    <oj-label for="text21">input with 'help.instruction' attribute with binding</oj-
label>
    <oj-input-text id="text21" name="text21"  autocomplete="off"
validators="[[validators]]"
        help.instruction="{{helpInstruction}}" value="{{text}}"></oj-input-text>

    <oj-label for="text22">input with formatted text in 'help.instruction'
attribute</oj-label>
    <oj-input-text id="text22" name="text22"  autocomplete="off"
validators="[[validators]]"
        help.instruction="<html>enter <span style='color:red'>at least 3
alphanumeric</span> characters</html>"
        value="{{text}}"></oj-input-text>
</div>
```

3. In your app script, bind the component's `value` to a Knockout observable.

   In this example, each `oj-input-text` element's `value` attribute is defined as `text` which is set to a Knockout observable in the following script. The script also defines regular expression validators to ensure that the user enters at least three letters or numbers.

```
require(['knockout', 'ojs/ojbootstrap, 'ojs/ojknockout', 'ojs/ojinputtext', 'ojs/
ojlabel'],
  function(ko, Bootstrap)
  {
    function MemberViewModel()
    {
      var self = this;
      self.text = ko.observable();

      self.validators = ko.computed(function()
        {
          return [{
             type: 'regExp',
             options: {
               pattern: '[a-zA-Z0-9]{3,}',
               messageDetail: 'You must enter at least 3 letters or
numbers'}}];
        });
      self.helpInstruction = "enter at least 3 alphanumeric characters";
    };

    Bootstrap.whenDocumentReady().then(
      function ()
      {
        ko.applyBindings(new MemberViewModel(), document.getElementById('form-
container'));
      }
    );
  });
```

For the complete example, see Help and Title in the Oracle JET Cookbook. For additional detail about the `oj-input-text` component, see the ojInputText API documentation.

For additional information about the regular expression validator, see About Oracle JET Validators and Converters.

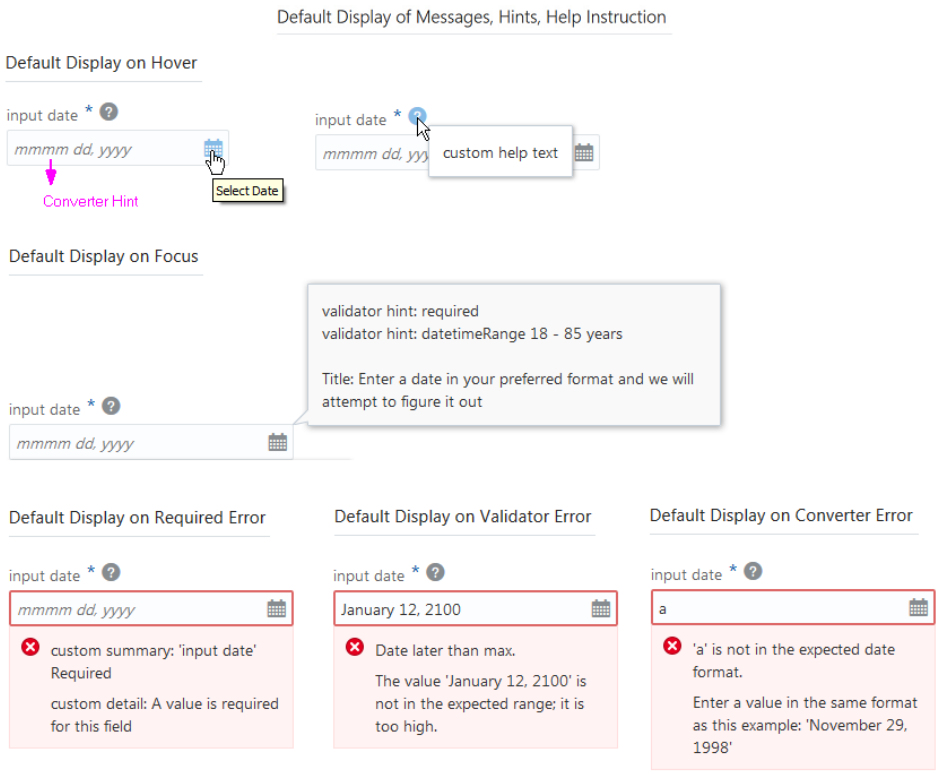
Control the Display of Hints, Help, and Messages

Use the `display-options` attribute to control the placement and visibility of converter and validator hints, messages, and help instructions.

The following image shows the default placement and visibility of help, converter and validator hints, and messages. This example uses the `oj-input-date` component, but the behavior is the same on all editable components where it makes sense:

- validator `hint`: Displays in a note window on focus
- converter `hint`: Used as the input field's placeholder, or displays in a note window if the `placeholder` attribute is defined.
- `messages`: Displays inline on error
- `help.instruction`: Displays in a note window on focus

The `oj-label` exclusive attribute `help.definition` displays in a note window on hover.



The code sample below shows the markup for the `oj-input-date` component used in this example. The example includes definitions for `help.instruction`, validator hints, and a data value for custom messages on validation failure. The sample also shows the markup for a `oj-label` element with the `help` attribute.

```
<div id="form-container" class="oj-form">
 <h3 class="oj-header-border">Default Display of Messages, Hints, Help Instruction</h3>
 <oj-label for="date10" help.definition="custom help text"> Input Date</oj-label>
 <oj-input-date id="date10" size="30" name="date10" required placeholder="month day,
year"
```

```
                help.instruction='enter a date in your preferred format and we
will attempt to figure it out'
                converter="[[longDateConverter]]"
                value="{{birthdate}}" validators="[[validators]]"
                translations='{
                  "required": {
                  "hint": "validator hint: required",
                  "messageSummary": "<html>custom summary: {label}
<b>Required</b></html>",
                  "messageDetail": "<html>custom detail: A value is required for
this field</html>"}}'>
 </oj-input-date>
```

The code sample below shows the custom messages on validation failure set in the app's script.

```
function MemberViewModel()
{
   var self = this;
   self.validators = ko.computed(function()
   {
      return [{
          type: 'datetimeRange',
          options: {
            min: ValidationBase.IntlConverterUtils.dateToLocalIso(new Date(1930, 00,
01)),
            max: ValidationBase.IntlConverterUtils.dateToLocalIso(new Date(1995, 11,31)),
            hint: {
              inRange: 'Validator hint: datetimeRange: January 1, 1930 - November
30, 1995 years'},
            messageSummary:{
              rangeOverflow: 'Date later than max.',
              rangeUnderflow: 'Date earlier than min.'},
            messageDetail: {
              rangeOverflow: 'The value \'{value}\' is not in the expected range;
it is too high.',
              rangeUnderflow: 'The value \'{value}\' is not in the expected
range; it is too low.'}
              }}];
   });
   //...Contents Omitted

}

Bootstrap.whenDocumentReady().then(
    function ()
    {
      ko.applyBindings(new MemberViewModel(), document.getElementById('form-container'));
    }
  );
});
```

Using the `display-options` element attribute in your markup, you can change the default behavior of the hints, help, and messaging properties of a single editable component on your page. To control the behavior of all editable components on the page, you can use the `Component.setDefaultOptions()` method in your app script to set `displayOptions` values.

`display-options` allows you to change the default behavior as follows:

- `helpInstruction`: Set to `none` to turn off the help instruction display.

- `converterHint`: Set to `none` to turn off the display or set to `notewindow` to change the default placement from placeholder text to a note window.
- `validatorHint`: Set to `none` to turn off the display.
- `messages`: Set to `none` to turn off the display or set to `notewindow` to change the default placement from inline to a note window.

Before you begin:

- Familiarize yourself with the process of adding an Oracle JET element to your page. See Add an Oracle JET Component to Your Page.

To change the default display type (inline or note window) and display options for hints, help, and messages:

1. Add the editable element to your page.
2. To change the default display type (inline or note window) for an individual editable component, add the `display-options` attribute to your component definition and set it as needed.

   For example, to turn off the display of hints and `help.instruction` and to display messages in a note window, add the highlighted markup to your component definition:

   ```
   <oj-input-date id="date12" required value="{{birthdate}}"
     converter="[[longDateConverter]]" validators="[[validators]]"
     help.instruction="enter a date in your preferred format and we will attempt to
   figure it out"
     display-options='{"converterHint": "none", "validatorHint": "none",
   "helpInstruction": "none", "messages": "notewindow"}'
       ... contents omitted
   }"
   ```

3. To change the default display and location for all editable components in your app, add the `Component.setDefaultOptions()` method to your app's script and specify the desired `displayOptions`.

   For example, to turn off the display of hints and help and to display messages in a note window, add the `ojComponent.setDefaultOptions()` method with the arguments shown below.

   ```
   Components.setDefaultOptions({
     'editableValue':
     {
       'displayOptions':
       {
         'converterHint': ['none'],
         'validatorHint': ['none'],
         'messages': ['notewindow'],
         'helpInstruction': ['none']
       }
     }});
   ```

The Oracle JET cookbook contains the complete code for this example at User Assistance. You can also find additional examples that illustrate hints, help, and messaging configuration.

12

Develop Accessible Oracle JET Apps

Oracle JET and Oracle JET components have built-in accessibility features for people with disabilities. Use these features to create accessible Oracle JET web app pages.

About Oracle JET and Accessibility

Oracle JET components have built-in accessibility support that conforms with the Web Content Accessibility Guidelines version 2.2 at the AA level (WCAG 2.2 AA), developed by the World Wide Web Consortium (W3C).

Accessibility involves making your app usable for people with disabilities such as low vision or blindness, deafness, or other physical limitations. This means, for example, creating apps that can be:

- Used without a mouse (keyboard only)
- Used with assistive technologies such as screen readers and screen magnifiers
- Used without reliance on sound, color, animation, or timing

Oracle JET components provide support for:

- Keyboard and touch navigation

  Oracle JET components follow the Web Accessibility Initiative - Accessible Rich Internet Application (WAI-ARIA) Developing a Keyboard Interface guidelines. Follow these guidelines when using Oracle JET components in your app. For example, avoid setting `tabindex` to values greater than `0`. The API documentation for each Oracle JET component lists its keyboard and touch end user information when applicable, including a few deviations from the WAI-ARIA guidelines.

- Zoom

  Oracle JET supports browser zooming up to 400%. For example, on the Firefox browser, you can choose **View -> Zoom -> Zoom In**.

- Screen reader

  Oracle JET supports screen readers such as JAWS, Apple VoiceOver, and Google TalkBack by generating content that complies with WAI-ARIA standards, and no special mode is needed.

- Oracle JET component roles and names

  Each Oracle JET component has an appropriate role, such as `button`, `link`, and so on, and each component supports an associated name (label), if applicable.

- Sufficient color contrast

  Oracle JET provides the Redwood theme, starting in Oracle JET release 9.0.0, which is designed to provide a luminosity contrast ratio of at least 4.5:1.

Oracle JET does not support the use of access keys due to their negative impact on assistive tooling and accessibility.

Oracle documents the degree of conformance of each product with the applicable accessibility standards using the Voluntary Product Accessibility Template (VPAT). You should review the appropriate VPAT for the version of Oracle JET that you are using for important information including known exceptions and defects, if any.

While Oracle JET is capable of rendering an app that conforms to WCAG 2.2 AA to the degree indicated by the VPAT, it is the responsibility of the app designer and developer to understand the applicable accessibility standards fully, use Oracle JET appropriately, and perform accessibility testing with disabled users and assistive technology.

About the Accessibility Features of Oracle JET Components

Oracle JET components are designed to generate content that conforms to the WCAG 2.2 AA standards. In most cases, you don't need to do anything to add accessibility to the Oracle JET component. However, there are some components where you may need to supply a label or other property.

For those components, the component's API documentation contains an Accessibility section that provides the information you need to ensure the component's accessibility.

> **Note:**
>
> Some Oracle products have runtime accessibility modes that render content optimized for certain types of users, such as users of screen readers. For the most part, Oracle JET renders all accessibility-related content all of the time. There is only a mode for users that rely on the operating system's high contrast mode, which is described in Create Accessible Oracle JET Pages.

Oracle JET components that provide keyboard and touch navigation list the keystroke and gesture end user information in their API documentation. Since the navigation is built into the component, you do not need to do anything to configure it.

You can access an individual Oracle JET component's accessibility features and requirements in the API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET). Select the component you're interested in from the list on the left. You can also find the list of supported keystrokes and gestures for each Oracle JET component that supports keystrokes and gestures in the Oracle® JavaScript Extension Toolkit (JET) Keyboard and Touch Reference.

Create Accessible Oracle JET Pages

Content generated by Oracle JET is designed to conform to the WCAG 2.2 AA standards. However, many standards are not under the complete control of Oracle JET, such as overall UI consistency, the use of color, the quality of on-screen text and instructions, and so on.

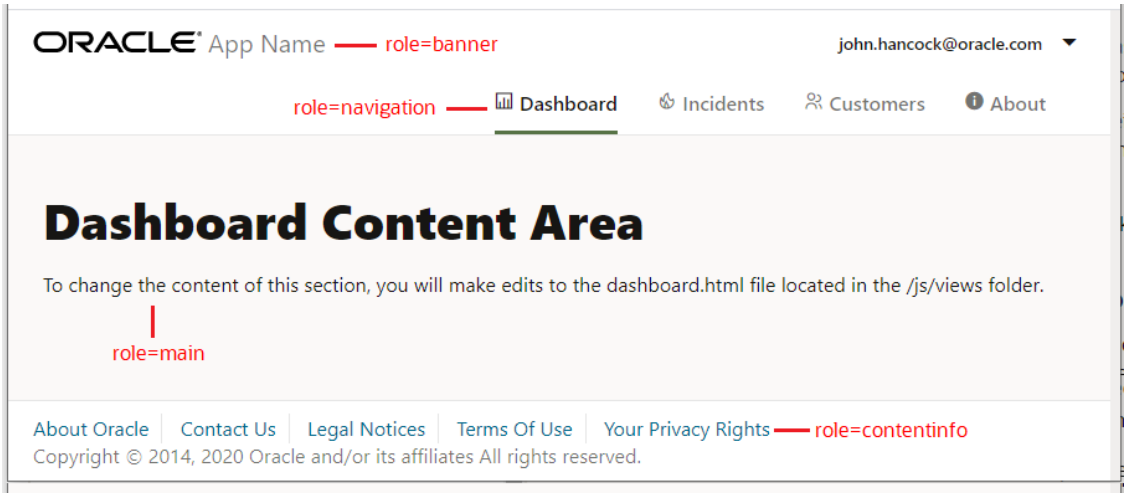
A complete product development plan that addresses accessibility should include proper UI design, coding, and testing with an array of tools, assistive technology, and disabled users.

> **Note:**
>
> In most cases, end-user documentation for your app must describe information about accessibility, such as example keystrokes needed to operate certain components.

Configure WAI-ARIA Landmarks

WAI-ARIA landmarks provide navigational information to assistive technology users. Landmark roles identify the purpose of a page region and allow the user to navigate directly to a desired region. Without landmarks, assistive technology users must use the TAB key to navigate through a page.

The Oracle JET team recommends the use of WAI-ARIA landmarks to ensure page accessibility and provides examples you can use in the Oracle JET Starter Template collection. The following figure shows the runtime view of the Oracle JET Web Nav Drawer Starter Template (`navdrawer`). In this example, the page is organized into regions compatible with WAI-ARIA landmark regions, including regions for the `banner`, `navigation`, `main`, and `contentinfo` landmarks.



The code in the following example shows the landmarks for the `navdrawer` template. Each landmark is placed on the HTML element that defines the landmark region: `div` for the `navigation` regions, `header` for the `banner` region, `oj-module` for the `main` region, and `footer` for the `contentinfo` region.

```
<!DOCTYPE html>
<html lang="en-us">
  <head>
    <title>Oracle JET Starter Template - Web Nav Drawer</title>
    ...contents omitted
  </head>

  <body class="oj-web-applayout-body">
    ...contents omitted

    <div id="globalBody" class="oj-offcanvas-outer-wrapper oj-offcanvas-page">
      <div
        id="navDrawer"
        role="navigation"
        class="oj-contrast-marker oj-web-applayout-offcanvas oj-offcanvas-
start">
        <oj-navigation-list
          id="navDrawerList"
```

```
          data="[[navDataProvider]]"
          item.renderer="[[KnockoutTemplateUtils.getRenderer('navTemplate',
true)]]"
          on-click="[[toggleDrawer]]"
          selection="{{selection.path}}"></oj-navigation-list>
      </div>
      <div id="pageContent" class="oj-web-applayout-page">
        <header role="banner" class="oj-web-applayout-header">
          <div
            class="oj-web-applayout-max-width oj-flex-bar oj-sm-align-items-
center">
            ...contents omitted
          </div>
          <div
            role="navigation"
            class="oj-web-applayout-max-width oj-web-applayout-navbar">
            <oj-tab-bar
              id="navTabBar"
              class="oj-sm-only-hide oj-md-condense oj-md-justify-content-
flex-end"
              data="[[navDataProvider]]"
              edge="top"

item.renderer="[[oj.KnockoutTemplateUtils.getRenderer('navTemplate', true)]]"
              selection="{{selection.path}}"></oj-tab-bar>
          </div>
        </header>
        <oj-module
          role="main"
          class="oj-web-applayout-max-width oj-web-applayout-content"
          config="[[moduleAdapter.koObservableConfig]]"></oj-module>

        <footer class="oj-web-applayout-footer" role="contentinfo">
          ...contents omitted
        </footer>
      </div>
    </div>
   ...contents omitted
  </body>
</html>
```

If your app includes a complementary region, add `role="complementary"` to the HTML `div` element:

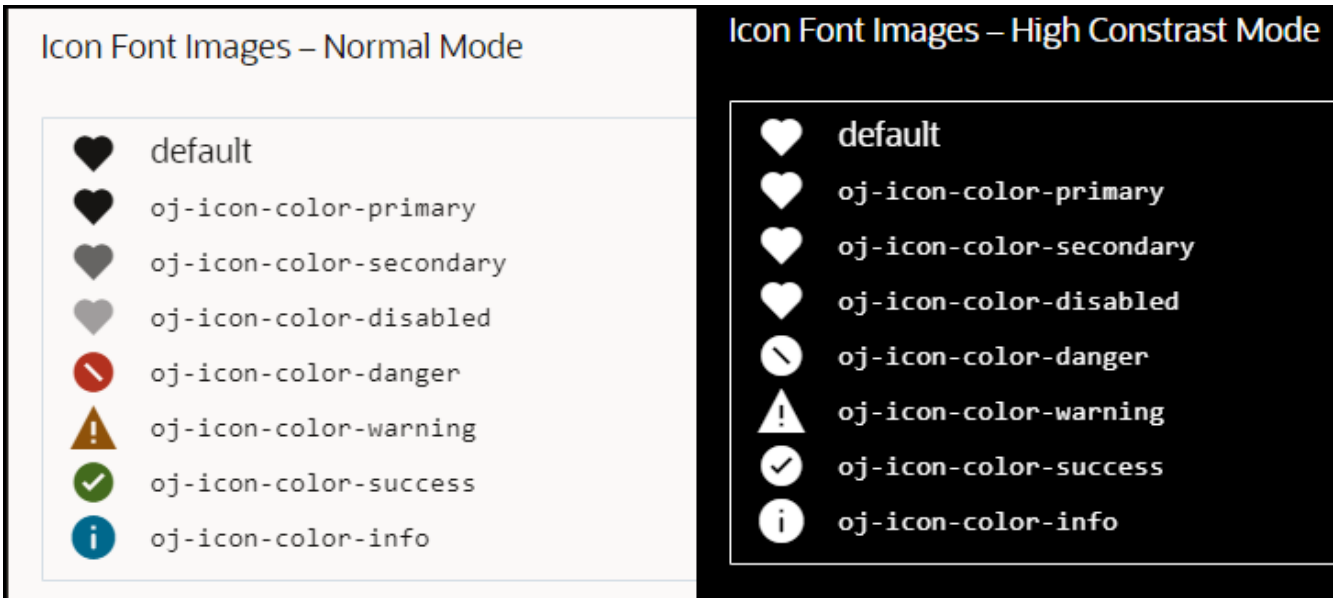
```
<div role="complementary"></div>
```

For additional information about WAI-ARIA landmark roles, see landmark_roles.

Configure High Contrast Mode

High contrast mode is for people who require a very high level of contrast in order to distinguish components on the page. Operating systems such as Microsoft Windows and macOS provide methods for users to run in high contrast mode.

The graphic below shows the effect of changing to high contrast mode on Oracle JET icon font images.



Understand Color and Background Image Limitations in High Contrast Mode

There are color and background image limitations in high contrast mode that your app may need to work around.

In high contrast mode the colors in the CSS may be ignored or overridden, including background, border, and text colors. Therefore, in high contrast mode you may need to find an alternative way to show state. For example, you might need to add or change the border to show that something is selected. To change CSS in high contrast mode, see forced-colors.

Also, your app may need to show alternate high contrast images that work on either a dark or light background color. Consider providing an image that includes a background, so either black on a white background or white on a black background. That way it won't matter what the background color is on the page since the contrast is in the image itself.

Test High Contrast Mode

The recommended way to test high contrast mode in Oracle JET app is to set high contrast mode at the operating system level and test your app in browsers that support high contrast mode, such as Google Chrome, Microsoft Edge, and Mozilla Firefox.

For information about enabling high contrast mode, see the documentation for your computer's operating system. For example, to turn high contrast mode on and off in Microsoft Windows, use the following key combination: Left Alt+Left Shift+PrtScn. You may need to refresh your browser to see the new mode.

Hide Screen Reader Content

Sometimes you want to have some text on the page that is read to the screen reader user, but the sighted user doesn't see. Oracle JET provides the `oj-helper-hidden-accessible` class that you can use to hide content.

You can find the `.oj-helper-hidden-accessible` style defaults in the Redwood theme CSS file (`web/css/redwood/x.x.x/web/redwood.css`). For additional information about theming and Oracle JET, see Use CSS and Themes in Oracle JET Apps.

Use ARIA Live Region

An ARIA live region is a mechanism to notify assistive technologies when a web page content is updated.

When using a single page app, if there are any changes to the page or the page region, the user of assistive technologies, such as screen readers, is not notified about the changes in the page content since the URL of the app has not changed. To help provide a notification to the screen readers when a page or segments of a page change, an ARIA live region announces the dynamic changes within a web page. When the update takes place within an ARIA live region, a screen reader is automatically notified, wherever its focus is at the time, and it announces the updated content to the user. This can be achieved by using the `aria-live` attribute.

The `aria-live` attribute identifies an element as a live region. It takes three possible values:

- `off`: No notification
- `polite`: Screen reader notifies user once the current task is complete
- `assertive`: Screen reader interrupts the current task to notify user

If the value of the `aria-live` attribute defined for an element is set to `polite`, your screen reader will not be interrupted and will announce the changes in the ARIA live region when the user has no activity on the screen. If the value is set to `assertive`, the new information has high priority and should be notified or announced to the user immediately.

You can also use some of the advanced ARIA live region attributes to communicate information about the entire live region or a portion of the live region to assistive technologies. Some of the advanced live region attributes to use are:

- `aria-atomic`: The `aria-atomic` attribute is used along with `aria-live` attribute when the page contains live regions. This attribute is used to set whether the assistive technologies should present the entire live region as a single unit or to only announce the regions that have been changed. The possible values are `true` or `false`. The default value is `false`.
- `aria-relevant`: This attribute is used in conjunction with the `aria-live` attribute to describe the types of changes that have occurred within a given live region that should be announced to the user. The possible settings are `additions`, `removals`, `text`, and `all`. The default setting is `additions text`.

The following example an ARIA live region defined in the `index.html` file of an Oracle JET app created with the `navdrawer` template. In this example, the `aria-live` attribute is set to `assertive` and the `aria-atomic` attribute is set to `true`:

```
. . .
<div id="globalBody">
  <div id="announce" tabindex="-1"
       class="send-off-screen"
       :aria-live="[[manner]]" aria-atomic="true">
    <p id="ariaLiveMessage"><oj-bind-text value="[[message]]"></oj-bind-
text></p>
  </div>
. . .
```

The following example shows how to set up the observables and event listeners in the `appController` file of the app.

- TypeScript
- JavaScript

TypeScript

```
. . .
class RootViewModel {
  manner: ko.Observable<string>;
  message: ko.Observable<string | undefined>;
  . . .

  constructor() {
    // handle announcements sent when pages change, for Accessibility.
    this.manner = ko.observable('polite');
    this.message = ko.observable();

    let globalBodyElement: HTMLElement = document.getElementById(
      'globalBody'
    ) as HTMLElement;
    globalBodyElement.addEventListener(
      'announce',
      this.announcementHandler,
      false
    );
```

JavaScript

```
. . .
function ControllerViewModel() {
. . .

  // Handle announcements sent when pages change, for Accessibility.
  this.manner = ko.observable('polite');
  this.message = ko.observable();
  announcementHandler = (event) => {
    this.message(event.detail.message);
    this.manner(event.detail.manner);
  };

 document
  .getElementById('globalBody')
  .addEventListener('announce', announcementHandler, false);
```

By default, each of the Oracle JET apps that you scaffold with a starter template (for example, `navdrawer`) include an accessibility utility module that defines a function (`announce`) to send an announcement to an Aria Live region. This module is in the `ts` or `js` sub-directory of your

`appRootDir/src` directory, depending on whether your app is TypeScript or JavaScript-based. An example usage is also implemented in each of the generated modules (`dashboard`, for example) of the Oracle JET app.

- TypeScript
- JavaScript

TypeScript

```
. . .
import * as AccUtils from '../accUtils';

class DashboardViewModel {
  constructor() {}
  connected(): void {
    AccUtils.announce('Dashboard page loaded.');
    document.title = 'Dashboard';
    // implement further logic if needed
  }
. . .
```

JavaScript

```
define(['../accUtils'], function (accUtils) {

  function DashboardViewModel() {

    this.connected = () => {
      accUtils.announce('Dashboard page loaded.', 'assertive');
      document.title = 'Dashboard';
      // Implement further logic if needed
    };
. . .
```

13

Internationalize and Localize Oracle JET Apps

Oracle JET includes support for internationalization (I18N), localization (L10N), and use of Oracle National Language Support (NLS) translation bundles in Oracle JET apps. Configure your Oracle JET app so that the app can be used in a variety of locales and international user environments.

About Internationalizing and Localizing Oracle JET Apps

Internationalization (I18N) is the process of designing software so that it can be adapted to various languages and regions easily, cost effectively, and, in particular, without engineering changes to the software. Localization (L10N) is the use of locale-specific language and constructs at runtime.

Oracle has adopted the industry standards for I18N and L10N, such as World Wide Web Consortium (W3C) recommendations, Unicode technologies, and Internet Engineering Task Force (IETF) specifications to enable support for the various languages, writing systems, and regional conventions of the world. Languages and locales are identified with a standard language tag and processed as defined in BCP 47. Oracle JET includes Oracle National Language Support (NLS) translation support for the languages listed in the following table.

Language	Language Tag
Arabic	`ar`
Brazilian Portuguese	`pt`
Bulgarian	`bg-BG`
Canadian French	`fr-CA`
Chinese (Simplified)	`zh-Hans` (or `zh-CN`)
Chinese (Traditional)	`zh-Hant` (or `zh-TW`)
Croatian	`hr`
Czech	`cs`
Danish	`da`
Dutch	`nl`
Estonian	`et`
Finnish	`fi`
French	`fr`
German	`de`
Greek	`el`
Hebrew	`he`
Hungarian	`hu`
Icelandic	`is`
Italian	`it`
Japanese	`ja`

Language	Language Tag
Korean	`ko`
Latin_Serbian	`sr-Latn`
Latvian	`lv`
Lithuanian	`lt`
Malay	`ms-MY`
Norwegian	`no`
Polish	`pl`
Portuguese	`pt-PT`
Romania	`ro`
Russian	`ru`
Serbian	`sr`
Slovak	`sk`
Slovenian	`sl`
Spanish	`es`
Swedish	`sv`
Thai	`th`
Turkish	`tr`
Ukrainian	`uk-UA`
Vietnamese	`vi`

Oracle JET translations are stored in a resource bundle. You can add your own translations to the bundles. For additional information, see Add Translation Bundles to Oracle JET.

Oracle JET also includes formatting support for over 196 locales. Oracle JET locale elements are based upon the Unicode Common Locale Data Repository (CLDR) and are stored in locale bundles. For additional information about Unicode CLDR, see http://cldr.unicode.org. You can find the supported locale bundles in the Oracle JET distribution:

```
js/libs/oj/18.1.0/resources/nls
```

It is the app's responsibility to determine the locale used on the page. Typically, the app determines the locale by calculating it on the server side from the browser locale setting or by using the user locale preference stored in an identity store and the supported translation languages of the app.

Once the locale is determined, your app must communicate this locale to Oracle JET for its locale-sensitive operations, such as loading resource bundles and formatting date-time data. Oracle JET determines the locale for locale-sensitive operations in the following order:

1. Locale specification in the ojL10n plugin of the RequireJS configuration.
2. `lang` attribute of the `html` tag.
3. `navigator.language` browser property.

If your app will not provide an option for users to change the locale dynamically, then setting the `lang` attribute on the `html` tag is the recommended practice because, in addition to setting the locale for Oracle JET, it sets the locale for all HTML elements as well. Oracle JET automatically loads the translations bundle for the current language and the locale bundle for the locale that was set. If you don't set a locale, Oracle JET will default to the browser property.

If, however, your app provides an option to change the locale dynamically, we recommend that you set the locale specification in the ojL10n plugin if your app uses RequireJS. Oracle JET loads the locale and resource bundles automatically when your app initializes.

If you use Webpack rather than RequireJS as the module bundler for your Oracle JET app, we recommend that you generate one code bundle for each locale that you want to support and deploy each bundle to a different URL for the locale that you want to support. If, for example, your app URL is https://www.oracle.com/index.html and you want to support the French and Spanish locales, deploy the bundles for these locales to https://www.oracle.com/fr/index.html and https://www.oracle.com/es/index.html respectively.

Finally, Oracle JET includes validators and converters that use the locale bundles. When you change the locale on the page, an Oracle JET component has built-in support for displaying content in the new locale. For additional information about Oracle JET's validators and converters, see Validate and Convert Input.

Internationalize and Localize Oracle JET Apps

Configure your app to use Oracle JET's built-in support for internationalization and localization.

Use Oracle JET's Internationalization and Localization Support

To use Oracle JET's built-in internationalization and localization support, you can specify one of the supported languages or locales on the `lang` attribute of the `html` element on your page. For example, the following setting will set the language to the French (France) locale.

```
<html lang="fr-FR">
```

If you want to specify the French (Canada) locale, you would specify the following instead:

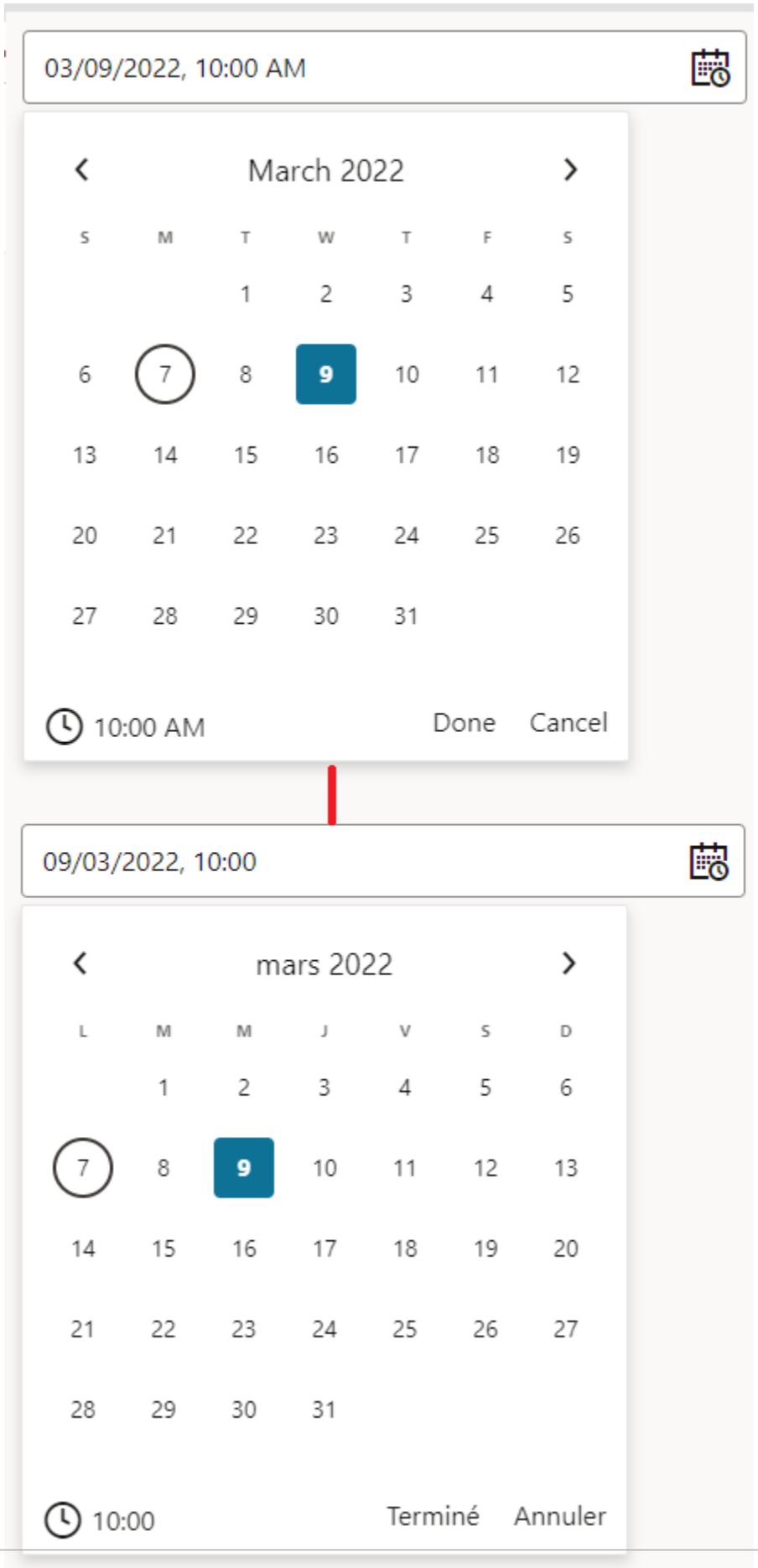
```
<html lang="fr-CA">
```

> **Tip:**
>
> The `locale` specification isn't case sensitive. Oracle JET will accept `FR-FR`, `fr-fr`, and so on, and map it to the correct resource bundle directory.

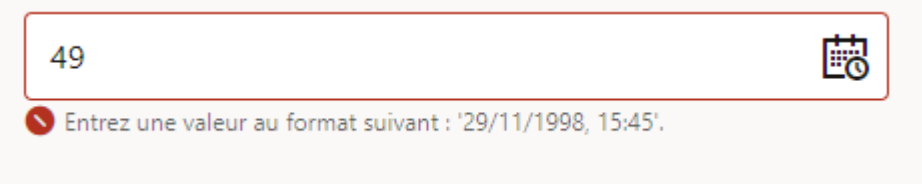
When you specify the locale in this manner, any Oracle JET component on your page will display in the specified language and use locale constructs appropriate for the locale.

If the locale doesn't have an associated resource bundle, Oracle JET will load the lesser significant language bundle. If Oracle JET doesn't have a bundle for the lesser significant language, it will use the default root bundle. For example, if Oracle JET doesn't have a translation bundle for `fr-CA`, it will look for the `fr` resource bundle. If the `fr` bundle doesn't exist, Oracle JET will use the default root bundle and display the strings in English.

In the image below, the page is configured with the `oj-input-date-time` component. The image shows the effect of changing the `lang` attribute to `fr-FR`.


03/09/2022, 10:00 AM
March 2022
S M T W T F S
1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
10:00 AM
Done
Cancel
09/03/2022, 10:00
mars 2022
L M M J V S D
1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
10:00
Terminé
Annuler

If you type an incorrect value in the `oj-input-date-time` field, the error text displays in the specified language. In this example, the error displays in French.

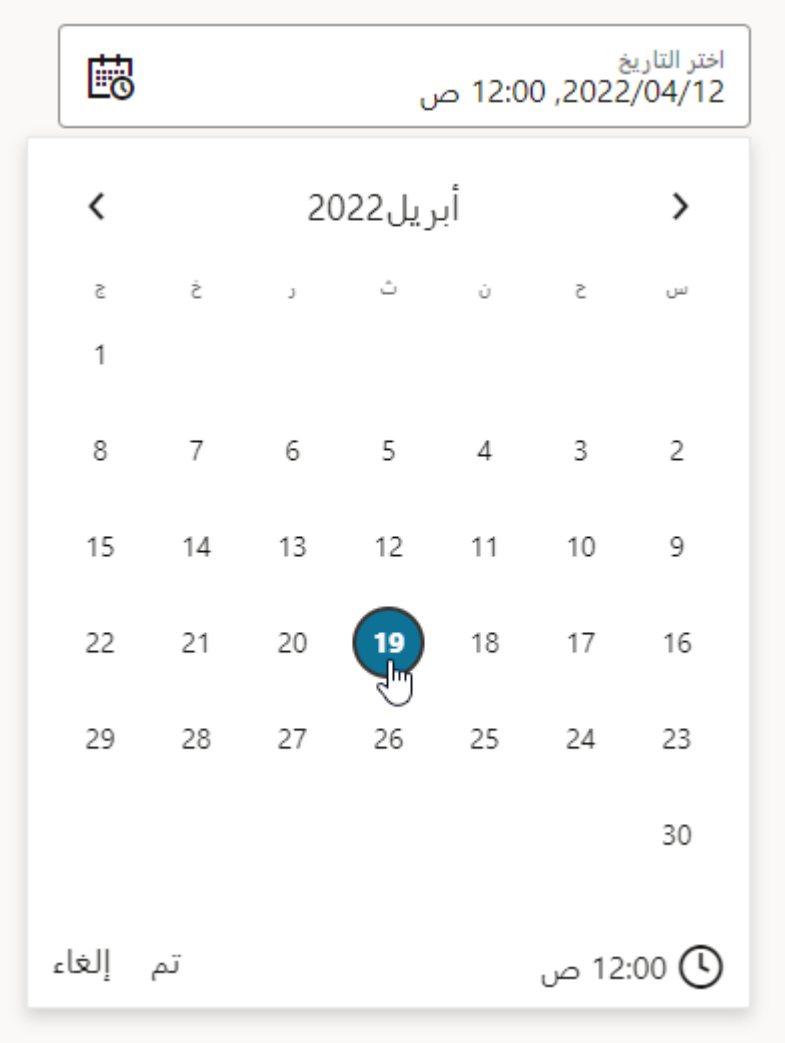


Enable Bidirectional (BiDi) Support in Oracle JET

If the language you specify uses a right-to-left (RTL) direction instead of the default left-to-right (LTR) direction, such as Arabic and Hebrew, you must specify the `dir` attribute on the `html` tag.

```
<html dir="rtl">
```

The image below shows the `oj-input-date-time` field that displays if you specify the Arabic (Egypt) language code and change the `dir` attribute to `rtl`.



Once you have enabled BiDi support in your Oracle JET app, you must still ensure that your app displays properly in the desired layout and renders strings as expected.

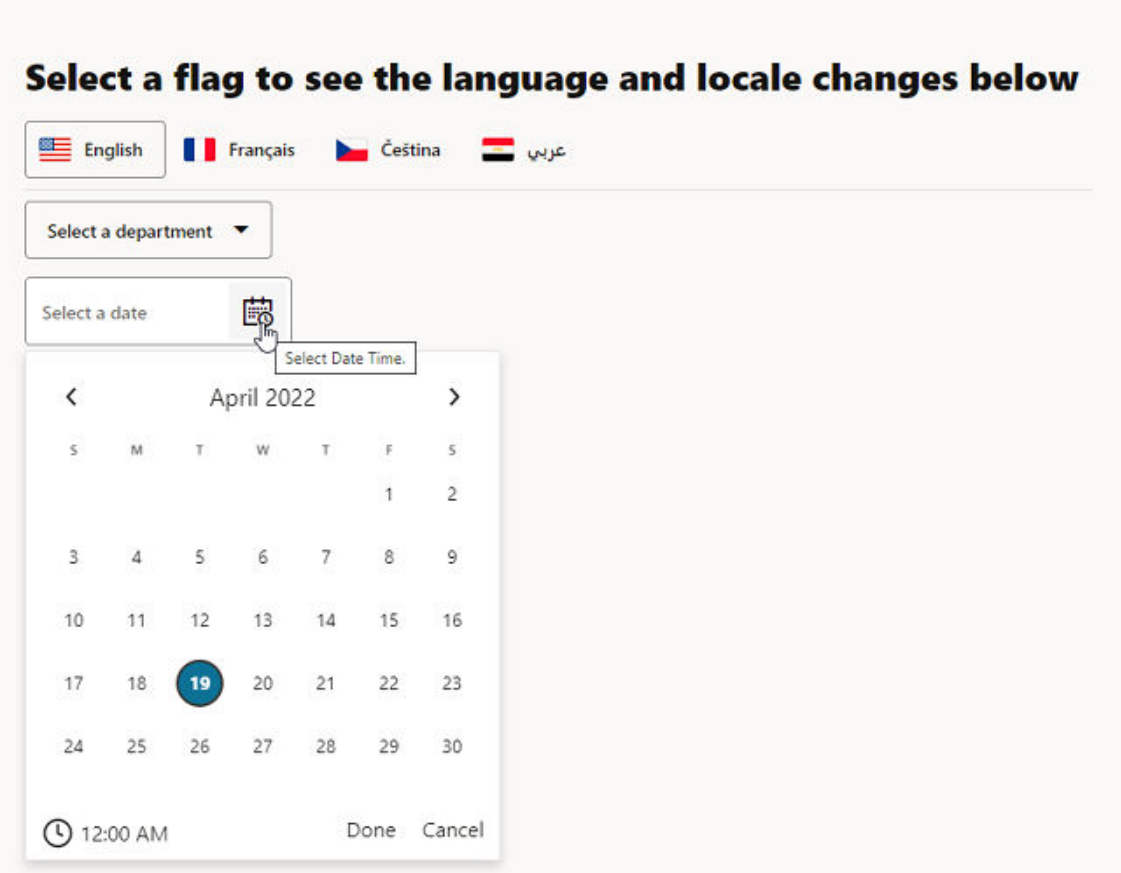
> **Note:**
>
> Oracle JET does not support the setting of the `dir` attribute on individual HTML elements which would cause a page to show mixed directions. Also, if you programmatically change the `dir` attribute after the page has been initialized, you must reload the page or refresh each JET component.

Set the Locale and Direction Dynamically

You can configure your app to change its locale and direction dynamically by setting a key-value pair in the app's local storage that your app's RequireJS ojL10n plugin reads when you reload the app URL.

The image below shows an Oracle JET app configured to display a menu that displays a department list when clicked and a date picker. By default, the app is set to the `en-US` locale. Both the menu and date picker display in English.

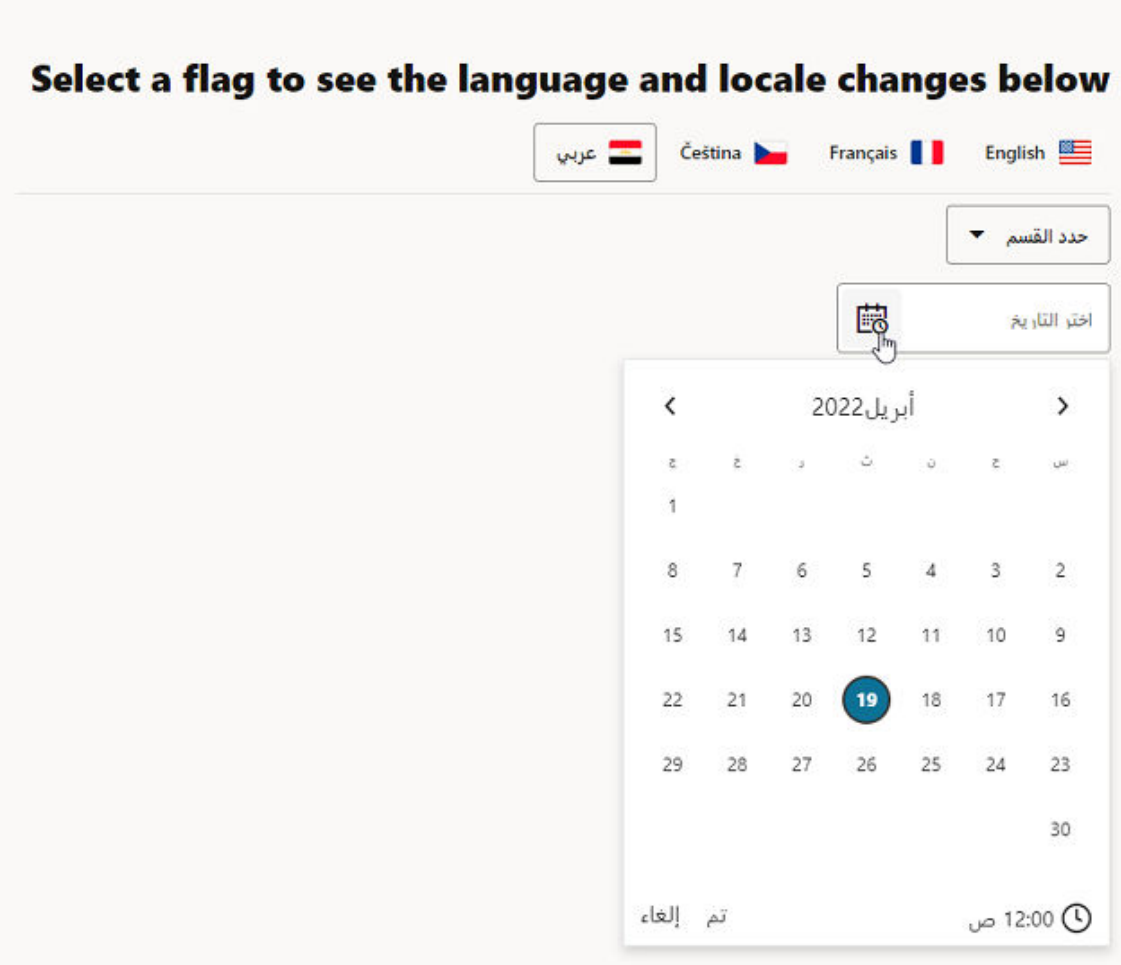
You can run the Oracle JET app shown in the image if you download the JET-Localization.zip and run the Oracle JET CLI `ojet restore` and `ojet serve` commands in the directory where you extract the ZIP file.



The app also includes a button set that shows the United States of America, France, Czech Republic, and Egypt flags. When the user clicks one of the flags, the app locale is set to the locale represented by the flag: `en-US`, `fr-FR`, `cs-CZ`, or `ar-EG`.

> **Note:**
>
> The flags used in this example are for illustrative use only. Using national flags to select a UI language is strongly discouraged because multiple languages are spoken in one country, and a language may be spoken in multiple countries as well. In a real app, you can use clickable text instead that indicates the preferred language to replace the flag icons.

The image below shows the updated page after the user clicks the Egyptian flag.



Implementing this behavior requires you to make changes in the view, the viewModel, and the `appRootDir/src/main.js` file of your app. In the view code of the app, the `on-value-changed` property change listener attribute specifies a `setLang` function that is called when a user changes the selected button.

```
<oj-buttonset-one . . . on-value-changed="[[setLang]]">
```

In the viewModel code of the app, this `setLang` function determines what locale the user selected and sets entries in window.localStorage so that a user's selection persists across browser sessions. The final step in the function is to reload the current URL using the location.reload() method.

```
setLang = (evt) => {
    let newLocale = "";
    let lang = evt.detail.value;
    switch (lang) {
      case "Čeština":
        newLocale = "cs-CZ";
        break;
      case "Français":
        newLocale = "fr-FR";
        break;
      case "عربي":
          newLocale = "ar-EG";
          break;
      default:
        newLocale = "en-US";
    }
    window.localStorage.setItem('mylocale',newLocale);
    window.localStorage.setItem('mylang',lang);
    location.reload();
  };
```

To set the newly-selected locale in the ojL10n plugin of our app's `appRootDir/src/main.js` file, we write the following entries that read the updated locale value from local storage, and set it on the `locale` specification in the ojL10n plugin. We also include a check that sets the direction to `rtl` if the specified locale is Egyptian Arabic (`ar-EG`).

```
(function () {
  ...
const localeOverride = window.localStorage.getItem("mylocale");
  if (localeOverride) {
    // Set dir attribute on <html> element.
    // Note that other Arabic locales and Hebrew also use the rtl direction.
    // Include a check here for other locales that your app must support.
    if(localeOverride === "ar-EG"){
      document.getElementsByTagName('html')[0].setAttribute('dir','rtl');
    } else {
      document.getElementsByTagName('html')[0].setAttribute('dir','ltr');
    }
    requirejs.config({
      config: {
        ojL10n: {
          locale: localeOverride,
        },
      },
    });
  }
})();
...
```

For information about defining your own translation strings and adding them to the Oracle JET resource bundle, see Add Translation Bundles to Oracle JET.

When you use this approach to internationalize and localize your app, you must consider every component and element on your page and provide translation strings where needed. If your page includes a large number of translation strings, the page can take a performance hit.

Also, if SEO (Search Engine Optimization) is important for your app, be aware that search engines normally do not run JavaScript and access static text only.

> **Tip:**
>
> To work around issues with performance or SEO, you can add pages to your app that are already translated in the desired language. When you use pages that are already translated, the Knockout bindings are executed only for truly dynamic pieces.

Work with Currency, Dates, Time, and Numbers

When you use the converters included with Oracle JET, dates, times, numbers, and currency are automatically converted based on the locale settings. You can also provide custom converters if the Oracle JET converters are not sufficient for your app. For additional information about Oracle JET converters, see About Oracle JET Converters. For information about adding custom converters to your app, see Use Custom Converters in Oracle JET.

Work with Oracle JET Translation Bundles

Oracle JET includes a translation bundle that translates strings generated by Oracle JET components into all supported languages. Add your own translation bundle by merging it with the Oracle JET bundle.

About Oracle JET Translation Bundles

Oracle JET includes a translation bundle that translates strings generated by Oracle JET components into all supported languages. You can add your own translation bundle following the same format used in Oracle JET.

The Oracle JET translation bundle follows a specified format for the content and directory layout but also allows some leniency regarding case and certain characters.

Translation Bundle Location

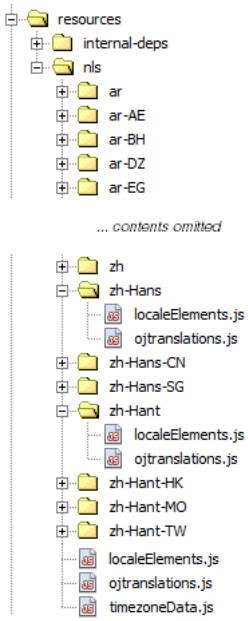
The location of the Oracle JET translation bundle, which is named `ojtranslations.js`, is in the following directory:

```
libs/oj/v18.1.0/resources/nls/ojtranslations
```

Each supported language is contained in a directory under the `nls` directory. The directory names use the following convention:

- lowercase for the language sub-tag (`zh`, `sr`, and so on.)
- title case for the script sub-tag (`Hant`, `Latn`, and so on.)
- uppercase for the region sub-tag (`HK`, `BA`, and so on.)

The language, script, and region sub-tags are separated by hyphens (-). The following image shows a portion of the directory structure.



Top-Level Module

The `ojtranslations.js` file contains the strings that Oracle JET translates and lists the languages that have translations. This is the top-level module or root bundle. In the root bundle, the strings are in English and are the runtime default values when a translation isn't available for the user's preferred language.

Translation Bundle Format

Oracle JET expects the top-level root bundle and translations to follow a specified format. The root bundle contains the Oracle JET strings with default translations and the list of locales that have translations.

```
define({
// root bundle
  root: {
    "oj-message":{
        fatal:"Fatal",
        error:"Error",
        warning:"Warning",
        info:"Info",
        confirmation:"Confirmation",
        "compact-type-summary":"{0}: {1}"
     },
     // ... contents omitted
  },

// supported locales.
  "fr-CA":1,
   ar:1,
   ro:1,
```

```
   "zh-Hant":1,
   nl:1,
   it:1,
   fr:1,
   //  ... contents omitted
   tr:1,fi:1
});
```

The strings are defined in nested JSON objects so that each string is referenced by a name with a prefix: `oj-message.fatal`, `oj-message.error`, and so on.

The language translation resource bundles contain the Oracle JET string definitions with the translated strings. For example, the following code sample shows a portion of the French (Canada) translation resource bundle, contained in `nls/fr-CA/ojtranslations.js`.

```
define({
  "oj-message":{
     fatal:"Fatale",
     error:"Erreur",
     warning:"Avertissement",
     info:"Infos",
     confirmation:"Confirmation",
     "compact-type-summary":"{0}: {1}"
     },
    //  ... contents omitted
});
```

When there is no translation available for the user's dialect, the strings in the base language bundle will be displayed. If there are no translations for the user's preferred language, the root language bundle, English, will be displayed.

Named Message Tokens

Some messages may contain values that aren't known until runtime. For example, in the message `"User foo was not found in group bar"`, the `foo` user and `bar` group is determined at runtime. In this case, you can define `{username}` and `{groupname}` as named message tokens as shown in the following code.

```
"MyUserKey":"User {username} was not found in group {groupname}."
```

At runtime, the actual values are replaced into the message at the position of the tokens by calling the `Translations.applyParameters()` method with the key of the message as the first argument and the parameters to be inserted into the translated pattern as the second argument.

```
let parMyUserKey = { 'username': 'Foo', 'groupname': 'Test' };
let tmpString = Translations.applyParameters(MenuBundle.MyUserKey,
parMyUserKey);
this.MyUserKey = Translations.getTranslatedString(tmpString);
```

Numeric Message Tokens

Alternatively, you can define numeric tokens instead of named tokens. For example, in the message `"This item will be available in 5 days"`, the number `5` is determined at

runtime. In this case, you can define the message with a message token of `{0}` as shown in the following code.

```
"MyKey": "This item will be available in {0} days."
```

A message can have up to 10 numeric tokens. For example, the message `"Sales order {0} has {1} items"` contains two numeric tokens. When translated, the tokens can be reordered so that message token `{1}` appears before message token `{0}` in the translated string, if required by the target language grammar. The code that calls the `applyParameters()` and `getTranslatedString()` methods remains the same no matter how the tokens are reordered in the translated string.

> **Tip:**
>
> Use named tokens instead of numeric tokens to improve readability and reuse.

Escape Characters in Resource Bundle Strings

The dollar sign, curly braces and square brackets require escaping if you want them to show up in the output. Add a dollar sign ($) before the characters as shown in the following table.

Escaped Form	Output
$$	$
${	{
$}	}
$[	[
$]	]

For example, if you want your output to display `[Date: {01/02/2020}, Time: {01:02 PM}, Cost: $38.99, Book Name: JET developer's guide]`, enter the following in your resource bundle string:

```
"productDetail": "$[Date: ${01/02/2020$}, Time: ${01:02 PM$}, Cost: $$38.99,
Book Name: {bookName}$]"
```

You then use the `Translations.applyParameters()` method to return the string with the escaped characters and substituted tokens, if any, to display in the UI, as shown in the following example:

```
let parProductDetail = { bookName: "JET developer's guide"};
this.productDetail = Translations.applyParameters(MenuBundle.productDetail,
parProductDetail);
```

Format Translated Strings

In some situations, you may want to apply formatting to strings in the resource bundle to appear in the UI. Take, for example, a book title to which we may want to apply the `<i>` tag so

that the book title renders using italics in the HTML output. In this scenario, you might define the following entry in the resource bundle(s):

```
// root bundle
"FormatTranslatedString": "The <i>{booktitle}</i> describes how to develop
Oracle JET apps"
```

And then use Oracle JET's oj-bind-dom element to render the string in the UI, as in the following example:

```
<p><span>
  <oj-bind-dom config="{{ formatTranslatedString() }}"></oj-bind-dom>
</span></p>
```

> **Caution:**
>
> The oj-bind-dom element does not validate HTML input provided by an app for integrity or security violations. It is the app's responsibility to sanitize the input to prevent unsafe content from being added to the page.

In our viewModel, we use Oracle JET's HtmlUtils utility class to parse the string from the resource bundle.

```
. . .
import * as HtmlUtils from "ojs/ojhtmlutils";
import "ojs/ojbinddom";
. . .
class DashboardViewModel {
 . . .
  FormatTranslatedString: String;
 . . .
  formatTranslatedString = () => {

    var parBookTitle = { 'booktitle': 'Oracle JET Developer Guide' };
    let strTitle =
Translations.applyParameters(MenuBundle.FormatTranslatedString, parBookTitle);
    this.FormatTranslatedString = Translations.getTranslatedString(strTitle);

    return {
      view: HtmlUtils.stringToNodeArray(
        `<span>${this.FormatTranslatedString}</span>`,
      ),
    };
   };
. . .
```

Add Translation Bundles to Oracle JET

You can add a translation bundle to your Oracle JET app with the custom strings that your app UI needs and the translations that you want your app to support.

To add translation bundles to Oracle JET:

1. Define the translations.

   For example, the following code defines a translation set for a menu containing a button label and three menu items. The default language is set to English, and the default label and menu items will be displayed in English. The root object in the file is the default resource bundle. The other properties list the supported locales, `fr`, `cs`, and `ar`.

```
define({
  "root": {
    "label": "Select a department",
    "menu1": "Sales",
    "menu2": "Human Resources",
    "menu3": "Transportation"
},
  "fr": true,
  "cs": true,
  "ar": true
});
```

   To add a prefix to the resource names (for example, `department.label`, `department.menu1`, and so on), add it to your bundles as shown below.

```
define({
  "root": {
    "department": {
      "label": "Select a department",
      "menu1": "Sales",
      "menu2": "Human Resources",
      "menu3": "Transportation"
    }
  }
},
  "fr": true,
  "cs": true,
  "ar": true
});
```

   When the locale is set to a French locale, the French translation bundle is loaded. The code below shows the definition for the label and menu items in French.

```
define({
  "label": "Sélectionnez un département",
  "menu1": "Ventes",
  "menu2": "Ressources humaines",
  "menu3": "Transports"
})
```

   You can also provide regional dialects for your base language bundle by just defining what you need for that dialect.

```
define({
    "label": "Canadian French message here"
});
```

When there is no translation available for the user's dialect, the strings in the base language bundle will be displayed. In this example, the menu items will be displayed using the French translations. If there are no translations for the user's preferred language, the root language bundle, whatever language it is, will be displayed.

2. Include each definition in a file located in a directory named `nls`.

   For example, the default translation in the previous step is placed in a file named `menu.js` in the `appRootDir/src/ts/resources/nls` directory. The supported translations are located in a file named `menu.js` in child sub-directories that use the name of the locale:

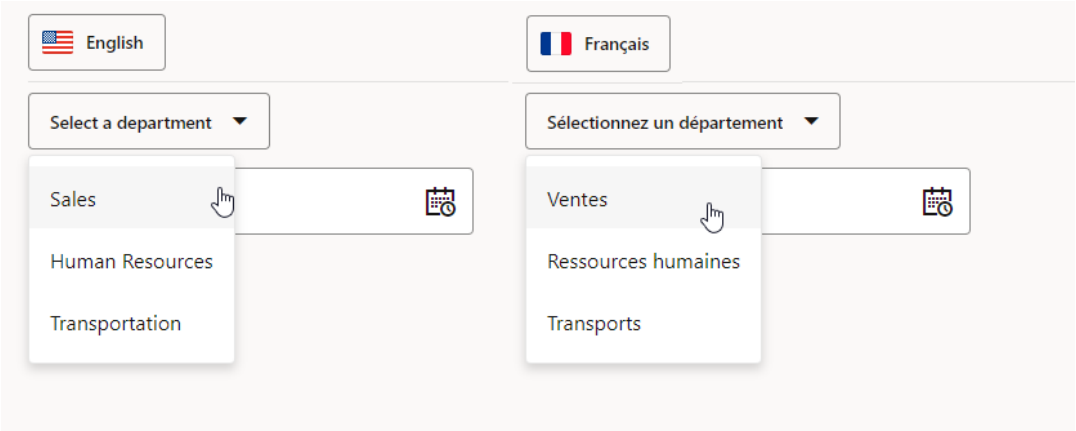
```
appRootDir/src/ts/resources/nls
|   menu.js
|
+---ar
|     menu.js
|
+---cs
|     menu.js
|
+---fr
      menu.js
```

   The directory name examples use `ts` as the examples assume that the Oracle JET app is a TypeScript-based app. If the app is JavaScript-based, then the directory name is `js`, as in `appRootDir/src/js/resources/nls`.

3. You need to configure changes in the view and viewModel code of your app to make sure that it retrieves the appropriate translation.

   If, for example, you want to implement the following UI where the menu labels change to English or French depending on the locale, you need to reference `btnLocaleLabel` and `menuNames` Knockout variables in the view code, as follows:

```
<oj-menu-button id="menuButton1">
  <span>
    <oj-bind-text value="[[btnLocaleLabel]]"></oj-bind-text>
  </span>
<oj-menu id="myMenu1" slot="menu" on-oj-action="[[changeLabel]]">
  <oj-bind-for-each data="[[menuNames]]">
    <template>
      <oj-option
value="[[$current.data.itemName]]" :id="[[$current.data.id]]">
    <span>
      <oj-bind-text value="[[$current.data.itemName]]"></oj-bind-text>
    </span>
      </oj-option>
  . . .
```



4. In the viewModel code, you first import the translation bundles where you defined the default and translated strings:

```
import * as MenuBundle from "ojL10n!../resources/nls/menu";
```

5. You then declare the Knockout observables that reference the imported translation bundle (`MenuBundle`) for the appropriate value to use, as demonstrated by the following excerpts from the viewModel file:

```
import * as MenuBundle from "ojL10n!../resources/nls/menu";

class DashboardViewModel {
  btnLocaleLabel: ko.Observable<string>;
  menuNames: ko.ObservableArray<object>;

  constructor() {
    // setting up knockout observables for the
    // button label and the menu items
    this.btnLocaleLabel = ko.observable();
    this.menuNames = ko.observableArray([{}]);
      . . .
  }

  loadMenu = () => {
    // These lines pull the translated values for the menu items
    // from the appropriate resource file in the /resources/nls directory
    this.menuNames([
      { itemName: MenuBundle.menu1, id:'menu1' },
      { itemName: MenuBundle.menu2, id: 'menu2' },
      { itemName: MenuBundle.menu3, id: 'menu3' },
    ]);
    this.btnLocaleLabel(MenuBundle.label);
  };

  /**
   * Optional ViewModel method invoked after transition to the new View is
complete.
   * That includes any possible animation between the old and the new View.
   */
  transitionCompleted(): void {
    // Call the function that pulls the translated values.
    this.loadMenu();
  }
```

```
}

export = DashboardViewModel;
```

6. If the strings from your resource bundle include message tokens or reserved characters (`$`, `{`, `}`, `[`, `]`) that need to be escaped, you must use Oracle JET's `Translation.applyParameters` API.

   The following code demonstrates how to render characters that must be escaped and a string that requires a parameter value (also referred to as a token value).

```
// // App UI is going to render the following strings:
$ { } [ ]
[The Oracle JET Developer's Guide costs $38.99]

// To accomplish this, we enter the following entries in the app's
resource bundle(s):
(appRootDir/src/ts/resources/nls/menu.js)
. . .
        "EscapeChar": "$$ ${ $} $[ $]",
        "EscapeCharToken": "$[The {bookName} costs $$38.99$]"
    },
. . .

// appRoot/src/ts/viewModels/dashboard.ts includes the following entries:
// Import the Translations module that includes the applyParameters() API
import * as Translations from "ojs/ojtranslation";

// Define types:
. . .
  EscapeChar: String;
  EscapeCharToken: String;

constructor() {

. . .

// Escape characters in resource bundle strings
let parEscapeChar = { };
this.EscapeChar = Translations.applyParameters(MenuBundle.EscapeChar,
parEscapeChar)

// Substitute a token and escape a character
let parEscapeCharToken = { bookName: "Oracle JET Developer's Guide"};
this.EscapeCharToken =
Translations.applyParameters(MenuBundle.EscapeCharToken,
parEscapeCharToken)

// appRoot/src/ts/views/dashboard.html includes the following entries to
render the final string
. . .
<oj-bind-text value="[[EscapeChar]]"></oj-bind-text>
. . .
<oj-bind-text value="[[EscapeCharToken]]"></oj-bind-text>
```

You can run an Oracle JET app that includes these code snippets if you download the JET-Localization.zip and run the Oracle JET CLI `ojet restore` and `ojet serve` commands in the directory where you extract the ZIP file.

14

Use CSS and Themes in Oracle JET Apps

Oracle JET includes the Redwood theme that provides styling across web apps and implements Oracle Redwood Design System. The Redwood theme provides hundreds of custom properties (also called CSS variables) to achieve its look and feel. You can use the Redwood theme as provided, or you can customize the custom properties manually.

> **Note:**
>
> Starting in Oracle JET release 9.0.0, the Redwood theme is the default theme for all new JET web apps.

About the Redwood Theme Included with Oracle JET

Oracle Redwood Design System is the Oracle standard for app look and feel. It is being implemented company-wide to unify the user interface of all Oracle product offerings and is implemented in Oracle JET as the Redwood theme.

Oracle JET takes this opportunity to refresh the toolkit look and feel with this dynamic, forward thinking design system and also to introduce all new components that rely on the user experience of Oracle Redwood Design System, such as JET waterfall layout and JET stream list component.

All starter templates use the Redwood theme. Because all apps will be created with the Redwood theme by default, you no longer need to specify the theme type when using the JET Tooling `create` and `serve` commands. There are no theme variations specific to the mobile platforms. For details, see Create an App with the Redwood Theme.

If you have an existing app that you want to migrate from the Alta theme, you can migrate to the current JET release and configure the app to run with the out-of-the-box CSS for the Redwood theme. For details, see Migrate to the Redwood Theme CSS.

Customizing Redwood theme is supported by working with CSS variables. For details, see About CSS Variables and Custom Themes in Oracle JET.

CSS Files Included with the Redwood Theme

Oracle JET includes CSS files designed for display on web browsers that implement Oracle Redwood Design System. In JET, the Redwood theme includes minified and readable versions of the CSS.

Starting in Oracle JET release 9.0.0, app themes are based on Redwood theme, a theme for both mobile and browser apps that replaces the multiple themes needed in prior releases.

The Redwood CSS is included with the Oracle JET distribution and is located in the `/<app_root>/node_modules/@oracle/oraclejet/dist/css/redwood` folder. The Redwood CSS distribution contains the following files:

- `oj-redwood.css`: Readable version of the CSS for the out-of-the-box Redwood theme

- `oj-redwood-min.css`: Minified version of the CSS for the out-of-the-box Redwood theme

In addition, the Redwood theme includes the following CSS:

- `oj-redwood-notag.css`: Readable version of the CSS without tag selectors
- `oj-redwood-notag-min.css`: Minified version of the CSS without tag selectors.

For additional details about Oracle JET theming and tag selectors, see Disable JET Styling of Base HTML Tags.

Always use the recommended standards to work with your CSS and themes. For more information, see Best Practices for Using CSS and Themes.

> **Important:**
>
> Do not override the style classes in the Oracle JET CSS distribution. The CSS files shipped with Oracle JET are considered private and must not be modified. Such modifications may prevent you from migrating your theme to a future release.

Create an App with the Redwood Theme

The Redwood theme is the Oracle JET implementation of Oracle Redwood Design System and is the default theme for apps that you create in JET release 9.0.0 and later.

Oracle Redwood Design System is the new Oracle user experience design language, and Redwood theme is the recommended theme if you are creating a new JET web app.

You use the `ojet create` command to scaffold an app that by default uses the Redwood CSS files provided with the Redwood theme distribution. When you build the app, the JET Tooling loads `redwood.css`.

When you create your app, JET Tooling will use the out-of-the-box Redwood theme file as configured by the property `defaultTheme=redwood` in the `oraclejetconfig.json` file. There are no other configuration settings needed for the tooling to use the `redwood.css` when you build and run the app.

```
{
    "paths": {
        ...
    },
    "defaultBrowser": "chrome",
    "sassVer": "1.80.5",
    "defaultTheme": "redwood",
    "architecture": "mvvm",
    "generatorVersion": "18.1.0"
}
```

Customization of the theme and styling of individual JET components is supported by overriding JET CSS variables. For details about overriding CSS variables, see Work with Custom Themes and Component Styles.

To create an app with the Redwood theme:

1. Create the app.

   ```
   ojet create my-web-app
   ```

   The tooling creates the `index.html` file in the `src` folder with an injector token that will load the CSS for the out-of-the-box Redwood theme.

   ```
   <!-- This is the main css file for the default theme -->
   <!-- injector:theme -->
   <!-- endinjector -->
   ```

2. Build a development version of your app.

   ```
   ojet build
   ```

   The tooling outputs the built app source to the `web` build folder in the app's root and populates the `/web/css/redwood/18.1.0/web` folder with the set of available Redwood theme files, including the default theme `redwood.css`, as well as fonts and images.

   ```
   AppRootDir/web/css/redwood/18.1.0/web
     fonts
     images
     redwood-notag-min.css
     redwood-notag.css
     redwood.css
     redwood.min.css
   ```

   Theme files with the `oj-` prefix and `-notag` suffix provide alternatives to the out-of-the-box Redwood theme that allow you to work with custom themes, as described in Work with Custom Themes and Component Styles.

   Note that `index.html` in the build output folder by default loads the expanded CSS from `redwood.css`. To load minified CSS from `redwood-min.css`, build the app in release mode.

   ```
   ojet build --release
   ```

3. Run the app to view the out-of-the-box Redwood theme in the browser.

   ```
   ojet serve
   ```

Adjust the Scale of the Redwood Theme

Apps and their users have different scale needs and requirements. The Redwood theme supports three different scale sizes.

The Redwood theme by default uses a large scale size. The large scale is recommended for lower-density pages and is optimized for readability and human scale.

The medium scale is more compact than Redwood's default scale and is designed to increase efficiency by allowing more data to be shown above the page's fold.

The small scale has highly compact proportions and is designed for canvas-style tools with a high level of information and UI density. This scale is not touch-friendly and should therefore only be used on desktop apps.

To change the Redwood theme to use a small or medium scale, use the Oracle JET scale classes `oj-scale-sm` or `oj-scale-md`, respectively. Add these classes to the `html` element in your app's `index.html` file.

```
<html class="oj-scale-md">
   <!-- content -->
</html>
```

The entire page must be one of these scales—specifying different scales on a single page is not supported.

Best Practices for Using CSS and Themes

Use the recommended styling standards for creating CSS and themes in your web app. These practices apply to all themes in Oracle JET.

Standard	Details	Example

Never override Oracle JET classes	While there are ways to override Oracle JET CSS style classes containing the `oj-` prefix, none of them is allowed, whether public or private.	Here is one example of an overridden Oracle JET class that is not allowed:
		
		`.acme-branding-header .oj-button{`
		`  color: white;`
		`  background: blue;`
		`}`

Use mobile-first design	Apps should be mobile first, meaning that they should work on a phone and tablet. This means they must be touch friendly, including sizing tap targets appropriately. There is no hover in mobile, so the design should not rely on the use of hover.	

Standard	Details	Example

Follow naming conventions	Use the `.namespace-block-element` naming convention for your CSS file.
	For example, if you are writing a branding bar that contains a header element for your company acme, then choose `acme` as the namespace, `branding` as the block, and `header` as the element. So, the selector name will be `.acme-branding-header`. Oracle JET uses dashes in their selector names, but you may use other naming conventions such as Block, Element, Modifier (BEM).
	Including a namespace is important in order to minimize the chances of your base CSS (for example, one provided by a client) on the page affecting your app's CSS and vice-versa. For example, if your CSS and the client-provided CSS both have a class named `branding-header` as seen below, your branding header text color will be red.
	Client CSS:
	`.branding-header{color: red}`
	Your CSS:
	`.branding-header{background-color: blue}`
	Using `.acme-branding-header` rather than just `.branding-header` will greatly reduce the chance that your client will use a class with the same name.
		`.acme-branding-header`

Use only Oracle JET public classes	Use only the public classes documented in the Oracle® JavaScript Extension Toolkit (JET) Styling Reference. All other Oracle JET classes are considered private and subject to change without notice.	

Standard	Details	Example
Minimize custom CSS	Oracle JET has CSS Utility classes that help you minimize custom CSS. The following list identifies the JET's CSS Utility classes that can assist you: • Font-size and font-weight: use JET's Typography classes • Color: use JET's Text Colors or icon font color classes • Background-color: use JET's Background Colors classes • Border: use JET's Divider or Panel classes • Padding and margins: use JET's Spacing classes • Display none: use If Binding (`oj-bind-if`). JET also has Hide classes (to always hide use `oj-sm-hide`, that hides on all screen sizes), but these can cause issues in components that need to be notified when made visible, so use of `oj-bind-if` is safer. • Display inline-block: use oj-helper-inline-block • Text-align: use JET's Text Align classes • Text-overflow: ellipsis use the Line Clamp (`oj-line-clamp-1`) class instead • Breaking long words • Flex: use JET's Flex Layouts classes • Widths: use JET's Responsive Width or Responsive Grids classes • Float: use JET's Float Start and Float End classes • White-space: use JET's White-Space classes When you use Oracle JET CSS utility classes in your app, you also ensure compatibility with the Oracle JET-provided theme. For more details see the CSS Utility classes demos in the Oracle JET Cookbook and the API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) .	

Don't set the font family in the CSS	The app should set the font family for text once on the root of the page which allows the app to change the font family as needed. In order to blend in with the font family chosen by the app, do not set the font family in the CSS.	Do not set the font family like this:
		`.acme-branding-header {`
		`  font-family: arial;`
		`}`

Use REM for font sizes and CSS styles	Consider using REM (root em) for font sizes and other CSS style properties where relative sizing allows your app to scale based on the root `html` element font size. Oracle JET components rely on REM, which allows your app to adjust to any changes to the underlying CSS font sizes for the themes included with Oracle JET. Likewise, your app can benefit by working with rem units instead of pixels or some other absolute unit. You can use rem units where ever an HTML style can benefit from scalable length units, such as the CSS properties for line-height, width, height, padding, margin, and so on. If you do not want to set a style directly on the `html` tag, you can reference the `oj-html` class.	`.acme-branding-header {`
		`  font-size: 1.2rem;`
		`}`

Standard	Details	Example

Add bi-directional (BIDI) styling support	Oracle JET apps are expected to set `dir="rtl"` for right-to-left (RTL) languages as described in Set the Text Direction. You can use this setting to support both left-to-right (LTR) and RTL languages in your CSS. To minimize the need to use `dir`, consider using CSS logical properties in your CSS.	`html:not([dir="rtl"]) .acme-branding-header { right: 0; }`
		
		`html[dir="rtl"] .acme-branding-header { left: 0; }`

Use `oj-hicontrast` for high contrast styling	When Oracle JET detects high contrast mode, it places the `oj-hicontrast` selector on the body element which you can use to change the CSS as needed. See Configure High Contrast Mode.	`.acme-branding-header { border: 1px; }`
		
		`.oj-hicontrast .acme-branding-header { border: 2px; }`

Avoid `!important`	Avoid the use of `!important` in your CSS as it makes it problematic to override the value. Where possible, use higher specificity instead. See the Mozilla specificity page for more information.	⚠ Avoid using `!important`. `.acme-branding-header { font-size: 1.2rem !important; }`
Optimize image use	All image systems have advantages and disadvantages. See Work with Images to decide if icon fonts are right for you. Always consider performance when using images. For tips, see Add Performance Optimization to an Oracle JET App	

DOCTYPE Requirement

In order for Oracle JET's theming to work properly, you must include the following line at the top of all HTML5 pages:

```
<!DOCTYPE html>
```

If you don't include this line, the CSS in your app may not function as expected. For example, you may notice that some elements aren't properly aligned.

> **Tip:**
>
> If you create an Oracle JET app using the tooling or one of the sample apps, this line is already added for you, and you do not need to add it yourself.

Set the Text Direction

If the language you specify uses a right-to-left (RTL) direction instead of the default left-to-right (LTR) direction, such as Arabic and Hebrew, you must specify the `dir` attribute on the `html` tag: `<html lang=name dir="rtl">`.

For example, the following code specifies the Hebrew Israel (he-IL) locale with right-to-left direction enabled:

```
<html lang="he-IL" dir="rtl">
```

Oracle JET does not support multiple directions on a page. The reason this is not supported is that the proximity of elements in the document tree has no effect on the CSS specificity, so switching directions multiple times in the page may not work the way you might expect. The code sample below shows an example.

```
<style>
  [dir=ltr] .foo {color: blue}
  [dir=rtl] .foo {color: red}
</style>
<span dir="rtl">
  <span dir="ltr">
    <span class="foo">You might think I will be blue because dir=ltr is on,
                      a closer ancestor than dir=rtl. But css doesn't care
                      about proximity, so instead I am red
                      (because [dir=rtl] .foo {color: red} was defined last).
                      Isn't that surprising?
    </span>
  </span>
</span>
```

For more information about localizing your app and adding bidirectional support, see Enable Bidirectional (BiDi) Support in Oracle JET. For more information about CSS specificity, see https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity.

Work with Images

Oracle JET uses icon fonts whenever possible to render images provided by the Redwood theme. When icon fonts are not possible, Oracle JET uses SVG images.

You may also find the following topics helpful when working with images.

Image Considerations

There are a variety of ways to load icons, such as sprites, data URIs, icon fonts, and so on. Factors to consider when choosing an image strategy include:

- Themable: Can you use CSS to change the image? Can you replace a single image easily?
- High contrast mode: Does the image render properly in high contrast mode for accessibility?
- High resolution support: Does the image look acceptable on high resolution (retina) displays?

- Image limitations: Are there limitations that impact your use case? For example, icon fonts are a single color, and small SVG images often do not render well.
- Performance: Is image size a factor? Do you need alternate versions of an image for different resolutions or states such as disabled, enabled, hover, and active?

Icon Font Considerations

Oracle JET uses icon fonts whenever possible because icon fonts have certain advantages over other formats.

- Themable: You can use style classes to change their color instead of having to replace the image, making them very easy to theme.
- High contrast mode: Icon fonts are optimal for high contrast mode as they are considered text. However, keep in mind that you can't rely on color in high contrast mode, and you may need to indicate state (active, hover, and so on) using another visual indicator. For example, you can add a border or change the icon font's size. For additional information about Oracle JET and high contrast mode, see Configure High Contrast Mode.
- High resolution: Icon fonts look good on a high resolution (retina) display without providing alternate icons.
- Performance: You can change icon font colors using CSS so alternate icons are not required to indicate state changes. Alternate images are also not required for high resolution displays.

Icon fonts also have disadvantages. It can be difficult to replace a single image, and they only show one color. You can use text shadows to provide some depth to the icon font.

Work with Custom Themes and Component Styles

When you add a custom theme to your scaffolded app, Oracle JET adds CSS variable definition files that you can modify to customize your app’s look and feel.

About CSS Variables and Custom Themes in Oracle JET

JET introduced the Redwood theme in release 9.0.0. In release 10.0.0, support for theming based on the out-of-the-box Redwood theme with CSS variables instead of Sass variables was introduced. Starting with JET release 11.0.0, you can also create a custom theme based on another out-of-the-box theme, Stable, that JET provides. We refer to these two themes (Redwood and Stable) as *base themes*.

Choose the Stable theme as the base theme for your custom theme if you want to reduce the likelihood that future updates to the Redwood theme affect the custom theme that you develop. Use Redwood as the base theme for your custom theme if you want to inherit future updates to the Redwood theme. Redwood theme implements the look and feel for Oracle apps, and future changes will be made to address Oracle’s requirements. The Stable theme, as the name suggests, intends to be stable and unlikely to change frequently. That said, Oracle cannot guarantee that the Stable theme will remain unchanged. We may, for example, need to change the Stable theme to incorporate component enhancements. As of its initial release (JET release 11.0.0), Stable theme implements the same component behavior as the Redwood theme.

Options to theme your app using CSS variables include:

- Add variable overrides in your app's `/appRootDir/src/css/app.css` file while your app continues to use the Redwood or Stable CSS file. Choose this option if you want to add a limited number of app-specific overrides to the base theme.
- Use the JET CLI to create a new custom theme that your app will use. Choose this option if you want to make extensive changes, or you want to make changes that will be reused by a variety of apps.

Use the Theme Builder app (links below) to assist you with these tasks as it contains a variety of JET components that allow you to preview the affect of changes to CSS variables.

Note that Sass is not completely eliminated in JET; JET continues to use Sass when you work with the Redwood theme for bundling and there are a few cases, for example in media queries, where CSS variables aren't supported. For the few exceptions, Sass variables are needed (such as `$screenSmallMinWidth` and `$screenSmallMaxWidth`). Therefore Sass is still part of the theming toolchain when you use `ojet add theming` in the JET CLI. If you use Sass to generate your own styles and don't rely on any JET Sass variables, then you can continue to use Sass as before. However, if you do rely on JET Sass variables, then most of those Sass variables will no longer be available, see the Sass-based Theme Builder - Migration tab for information about how to migrate JET Sass variables to CSS variables.

Use these links to view Theme Builder pages:

- CSS Variable Theme Builder
- CSS Variable Theme Builder - Instruction Tab
- Sass-based Theme Builder - Migration Tab

Here are some recent significant updates in JET's support of theming.

Release	Event
JET 9.0.0	You can use the Redwood CSS out of the box, without any changes.
JET 10.0.0	• You can customize the Redwood theme using CSS variable overrides. • Alta themes have been deprecated in JET release 10.0.0. If you have an existing theme that extends Alta and you want to migrate it to extend the Redwood theme (and use CSS variables), it is a manual process. There is information on variable migration on the Theme Builder - Migration tab. More information about migrating your app to a Redwood theme can also be found in Migrate to the Redwood Theme CSS. Note that Alta themes will continue to use Sass, and they will not switch to use CSS variables.
JET 11.0.0	You can choose between adding a custom theme to your JET app based on the Stable theme or the Redwood theme. Choose the Stable theme as the base theme for your custom theme if you want to reduce the likelihood that future updates to the Redwood theme affect the custom theme that you develop. Use Redwood as the base theme for your custom theme if you want your custom theme to inherit future updates to the Redwood theme.

Add Custom Theme Support with the JET CLI

You can use Oracle JET Tooling to add a custom theme to your app.

When you use the JET Tooling to add theming support to your app, it adds the theme definition files. Adding theming support is a prerequisite step to performing any modifications to the out-of-the-box Redwood or Stable themes.

Creation of the theme generates these theme definition files in the app `/src/themes` folder.
To add theming support:

1. In your app's top-level directory, enter the following command at a terminal prompt to install the theming toolchain.

   ```
   ojet add theming
   ```

2. Create the custom theme. This adds the custom theme settings files to your app in a folder named for your custom theme. You must choose between the `stable` or the `redwood` theme as the base theme for your custom theme. Choose `stable` as the base theme if you want to reduce the likelihood that future updates to the Redwood theme affect the custome theme that you develop. Use `redwood` if you want your custom theme to inherit future updates to the Redwood theme.

   ```
   ojet create theme themeName --basetheme=stable|redwood
   ```

   For example, the following command creates a custom theme named `myTheme` that uses the `stable` theme as the base theme.

   ```
   ojet create theme myTheme --basetheme=stable
   ```

   The command creates a folder with the custom theme name in the app `/src/themes` directory.

   ```
   src
   |   index.html
   |
   +---css
   |
   +---themes
   |
   |   \---myTheme
   |       |   theme.json
   |       |
   |       \---web
   |               myTheme.scss
   |               _myTheme.cssvars.settings.scss
   |               _myTheme.optimize-components.scss
   |               _myTheme.sass.settings.scss
   ```

   In the directory listing above, the `web` folder shows the `.scss` custom theme settings files that you can modify. You'll find out how to work with these files at the end of this section.

   - `_<themeName>.cssvars.settings.scss` is the primary file that you will use to create your custom theme. You edit this file to set CSS variable values that are then used by JET components, app-wide. So, for example, to change the primary text color for JET as a whole, you can edit this file by uncommenting and setting the value for `--oj-core-text-color-primary`.
   - `_<themeName>.optimize-components.sccs` allows you to tune the overall CSS size of your theme by specifying styling of specific JET components. You do not need to edit this file if you are styling the entire JET component set.
   - `_<themeName>.sass.settings.scss` defines other customizable aspects of the theme that are not yet supported by CSS variables. This includes media queries and responsive screen sizes. For most theming needs, you will not need to edit this file.

- `<themeName>.scss` is the main aggregating file for the theme and is the one the JET Tooling uses to generate the custom CSS at build time. By default it is configured to apply your CSS overrides to all JET components. You may need to edit this file when you minimize the JET component CSS that you want to load or you can specify not to load the CSS for base HTML tags.

In the `myTheme` folder, the `theme.json` file contains the version number of the theme, starting with `0.0.1`.

3. Optionally, in the app root, edit the `oraclejetconfig.json` file and set the `defaultTheme` property to the custom theme name that you specified with the `ojet create theme` command.

```
{
    "paths": {
         ...
    },
    "defaultBrowser": "chrome",
       "sassVer": "1.80.5",
    "defaultTheme": "myTheme",
       "architecture": "mvvm",
    "generatorVersion": "18.1.0"
}
```

In this sample, `myTheme` is the name of the custom theme shown in the previous step. When you edit the `oraclejetconfig.json` file, each time you build or serve the app, you can omit the `--theme=<themeName>` build command option, and the tooling loads your custom theme.

After you add support for custom theming to your app, when you build and run the app, the CSS will load the out-of-the-box Redwood or Stable theme that you specified as the value for the `--basetheme` argument.

To work with the settings files to customize the theme, you can perform these additional tasks:

- Modify the Custom Theme with the JET CLI
- Optimize the CSS in a Custom Theme
- Disable JET Styling of Base HTML Tags

Additionally, the CSS Variables section in the Oracle JET Cookbook includes examples of the type of look and feel customizations described in the above list of topics.

Modify the Custom Theme with the JET CLI

You can override CSS variables defined by the base theme in the custom theme that you add to your JET app.

You modify the generated file `_<themeName>.cssvars.settings.scss` located in the `/src/themes/<themeName>/web` folder. This file contains the list of available JET CSS variables in `:root` where you set values for the variables to apply application-wide to the JET components.

Before you begin:

- Install the theming toolchain and configure a custom theme, as described in Add Custom Theme Support with the JET CLI. The `themes` folder with custom theme settings files are added to your application source.
- Examine the App Wide Theming section in the Oracle JET Cookbook for examples.
- Refer to the JET CSS Variables section of the JET API reference doc for an overview of the CSS variables and their values.
- Optionally, download and install the Theme Builder app. The Theme Builder app uses the CLI and shows various JET components to easily see the effect of a creating a theme. See CSS Variable Theme Builder - Instruction Tab.

To override base theme CSS variables in your application CSS:

1. In the `/src/themes/<themeName>/web` folder, edit the generated `_<themeName>.cssvars.settings.scss` theme file and modify the CSS variables under `:root` that you want to override.

   For example, to change the height of all JET buttons in your application, you can override the JET CSS variable `--oj-button-height`, like this:

   ```
   :root {
     ...
     --oj-button-height: 4rem;
     ...
     }
   ```

2. In the `/src/themes/<themeName>` folder, edit the generated `theme.json` configuration file and set the `generatedFileType` property to the appropriate value:

   - `"generatedFileType": "combined"`

     Use when you need to change the value of CSS breakpoints, or you want to optimize performance by only downloading the CSS you need. With this option the JET CSS and the customizations are combined into one custom theme file. This custom theme file needs to be regenerated every time that you upgrade the JET version of your app. The following entry appears in the `index.html` page when you serve your app with this option:

     ```
     <!-- injector:theme -->
         <link rel="stylesheet" href="css/myStableTheme/0.0.1/web/
     myStableTheme.css" id="css" />
     <!-- endinjector -->
     ```

     If you use `combined` because you need to change the value of CSS breakpoints, you also need to uncomment the following entry in the aggregating file for the theme (`appRootDir/src/themes/<themeName>/web/myStableTheme.scss` in our example):

     ```
     //@import "_myStableTheme.sass.settings.scss";
     ```

     You must then uncomment one of the following files:

     ```
     // @import "oj/all-components/themes/stable/_oj-all-components.scss";
     or:
     // @import "_myStableTheme.optimize-components.scss";
     ```

If you use `combined` to optimize performance by only downloading the CSS you need, you must uncomment the following line in the aggregating file for the theme:

```
// @import "_myStableTheme.optimize-components.scss";
```

- `"generatedFileType": "add-on"`

  This is the default value. Leave unchanged if you don't need to change the value of CSS breakpoints or optimize performance by only downloading the CSS you need. With this option the theme file just contains the CSS variable overrides, so it can be loaded separately after the base Stable or Redwood CSS file. You do not need to regenerate this theme file every time that you upgrade to a new JET release. The following entries appear in the `index.html` page when you serve your app if your app uses a custom theme (`myStableTheme`) that uses the Stable theme as the base theme.

```
<!-- injector:theme -->
 <link rel="stylesheet" href="css/stable/11.0.0/web/stable.css"
id="css" />
 <link rel="stylesheet" href="css/myStableTheme/0.0.1/web/
myStableTheme.css" id="css" />
<!-- endinjector -->
```

3. Build a development version of your application.

```
ojet build --theme=<themeName>
```

You can omit the `--theme` option if you set the `defaultTheme` property in the `oraclejetconfig.json` file.

The tooling outputs the built application source to the `web` build folder in the application's root and populates the `/web/css` folder with the CSS from the theme that you specified as the base theme when creating a custom theme (Redwood or Stable) and the CSS with your overrides, such as the generated `myTheme.css` file. The following directory listing shows the generated entries for a theme, `myStableTheme`, that uses the Stable theme as its base theme. The directory listing includes `stable.css` and `stable.min.css` files because the app uses `"generatedFileType": "add-on"` in the `appRootDir/src/themes/myStableTheme/theme.json` file.

```
web
|
+---css
|   +---fonts
|   +---images
|   +---myThemeStable
|   |   \---0.0.1
|   |       +---common
|   |       \---web
|   |               myThemeStable.css
|   |
|   +---redwood
|   |   \---11.0.0
|   |       \---web
|   |           +---fonts
|   |           |       internal_iconfont.woff2
|   |           |
```

```
|   |           \---images
|   |                   avatar-pattern1.png
|   |                   ...
|   |                   spinner_full.gif
|   |
|   \---stable
|       \---11.0.0
|           \---web
|               |   stable.css
|               |   stable.min.css
|               |
|               +---fonts
|               \---images
|
+---js
\---themes
    \---myThemeStable
        |   theme.json
        |
        \---web
                myThemeStable.scss
                _myThemeStable.cssvars.settings.scss
                _myThemeStable.optimize-components.scss
                _myThemeStable.sass.settings.scss
```

Theme files with the `oj-` prefix and `-notag` suffix provide alternatives to the out-of-the-box Redwood theme that allow you to work with a reduced CSS, as described in Work with Custom Themes and Component Styles.

To load minified CSS, build the application in release mode.

```
ojet build --release
```

4. Run your application.

   ```
   ojet serve --theme=<themeName>
   ```

   Again, you can omit the `--theme` option if you set the `defaultTheme` property in the `oraclejetconfig.json` file.

5. Optionally, to refine the theme, modify CSS variables in your application's `_<themeName>.cssvars.settings.scss` theme file.

   The Oracle JET serve process will immediately reload the running application in the browser where you can observe the effect of any CSS variable overrides.

6. To exit the application, press Ctrl+C at the terminal prompt.

Modify the Custom Theme with Theme Builder

You can use the JET Theme Builder application to override CSS variables defined by the Redwood or Stable theme and reuse the generated theme definition files to theme your own application.

Theme Builder is a JET application that contains a variety of JET components that allow you to visualize look and feel changes that you make by overriding CSS variable values. Because

Theme Builder is a JET application, the process of theming uses the same theme definition files that your application relies on when you create a custom theme. This allows you to use the Theme Builder application to create a custom theme and then reuse those generated theme definition files in your own application. Alternatively, you can use Theme Builder strictly as a tool to learn the names of the CSS variables, and then work with JET Tooling to create a custom theme in your own application, as described in Modify the Custom Theme with the JET CLI.

To work with Theme Builder, you download the application as described in the Theme Builder - Instruction tab. You then use the `ojet add theming` and `ojet create theme` commands in the JET CLI to generate the needed theme definition files that allow you to override CSS variables and create a custom theme CSS. When you are satisfied with the custom CSS in Theme Builder, you will then prepare your own application to generate CSS from theme definition files before copying over the theme definition files you modified in Theme Builder.

Before you begin:

- Download and install the CSS Variable Theme Builder - Instruction Tab application. You will modify the generated theme definition files to override Redwood CSS variables and reuse the modified files in your application.
- In the application that you want to reuse the Theme Builder generated CSS, install the theming toolchain and configure a custom theme, as described in Add Custom Theme Support with the JET CLI. The `themes` folder with custom theme definition files is added to your application source. This is the folder that you will copy Theme Builder theme definition files into.

To reuse a Theme Builder generated CSS in your application:

1. After you have downloaded Theme Builder and run the commands to support theming, in Theme Builder edit the generated `/src/themes/<themeName>/web/_<themeName>.cssvars.settings.scss` theme file and begin modifying the Redwood CSS variables under `:root` that you want to override.
2. While the Theme Builder application is running, modify CSS variables and visualize the changes in the running application until you are satisfied with the look and feel that you have created.
3. When you are satisfied with the look and feel in Theme Builder and have no further customizations, press Ctrl+C at the terminal prompt to exit the application.
4. In the Theme Builder application copy all theme definition files in the `/src/themes/<themeName>/web` folder and paste them into the `/src/themes/<themeName>/web` folder of the target application that you have already prepared to work with theming.
5. Build a development version of the application where you added the theme definition files.

   ```
   ojet build --theme=<themeName>
   ```

   You can omit the `--theme` option if you set the `defaultTheme` property in the `oraclejetconfig.json` file.
6. Run your application.

   ```
   ojet serve --theme=<themeName>
   ```

   Again, you can omit the `--theme` option if you set the `defaultTheme` property in the `oraclejetconfig.json` file.

7. Optionally, to refine the theme, modify CSS variables in your target application's `_<themeName>.cssvars.settings.scss` theme file.

   The Oracle JET serve process will immediately reload the running application in the browser where you can observe the effect of any CSS variable overrides.

8. To exit the application, press Ctrl+C at the terminal prompt.

Optimize the CSS in a Custom Theme

By default when you build the application, JET Tooling loads CSS with all Redwood style classes enabled. You can configure the CSS to use just the styles required by the components of your application.

With the custom theme settings files added to the `/src/themes` folder of your project by the JET Tooling, you can reduce the size of the Redwood CSS by commenting and uncommenting import statements in the settings files for specific JET component style classes. With the appropriate import settings completed, JET Tooling will process the CSS settings files and load CSS for only the JET components you specified.

Before you begin:

- Install the theming toolchain and configure a custom theme, as described in Add Custom Theme Support with the JET CLI. The `themes` folder with custom theme settings files are added to your application source.

To reduce CSS by limiting component CSS:

1. In the `/src/themes/themeName/web` folder of your application, edit the aggregating `themeName.scss` file, and comment out the import statement that by default imports all JET component styles.

   ```
   // The import statement contains redwood or stable, depending on the value
   that you
   // supplied to the --basetheme argument when you created the custom theme.
   //@import "oj/all-components/themes/redwood/_oj-all-components.scss";
   or:
   //@import "oj/all-components/themes/stable/_oj-all-components.scss";
   ```

   And, then uncomment the import statement to enable compiling with the file that controls the CSS to include.

   ```
   @import "_themeName.optimize-components.scss";
   ```

2. Edit the `_themeName.optimize-components.scss` file and uncomment the import statements for the JET components in your application. Note that the following example demonstrates a theme that uses `redwood` as the value for the `--basetheme` argument.

   ```
   ...
   //@import "oj/avatar/themes/redwood/oj-avatar.scss";
   //@import "oj/badge/themes/redwood/oj-badge.scss";
   @import "oj/button/themes/redwood/oj-button.scss";
   @import "oj/buttonset-one/themes/redwood/oj-buttonset-one.scss";
   @import "oj/buttonset-many/themes/redwood/oj-buttonset-many.scss";
   //@import "oj/card/themes/redwood/oj-card.scss";
   //@import "oj/chart/themes/redwood/oj-chart.scss";
   ...
   ```

In this example, only the import statement for buttons and button sets are uncommented to include the CSS for those components. All import statements in the `_themeName.omptimize-components.scss` file remain commented out, by default.

3. If your application is already running, you can immediately observe the effect of commenting and uncommenting import statements, after you save the changes in the `_themeName.optimize-components.scss` file. Otherwise, serve your application to enable live reload.

   ```
   ojet serve
   ```

4. To exit the application, press Ctrl+C at the terminal prompt.

Style Component Instances with CSS Variables

When you want to customize the appearance of a component instance, you create style classes that override the CSS variables at the selector level and then apply the style classes to the component instance in your app's HTML.

You can define style classes in your `app.css` file in the `/src/css` folder or, when you theme your application, in the generated `_<themeName>.cssvars.settings.scss` theme file located in the `/src/themes/<themeName>/web` folder. The `_<themeName>.cssvars.settings.scss` file contains the list of available CSS variables in `:root`, where you can set values for the variables to apply application-wide to the JET components. When you want to define a style class to customize individual component instances, you must add your style class definitions with CSS overrides so they appear at the end of the file, after all CSS variables and they must not be contained within `:root`.

Note that while most customization is done with CSS variables, but in a few cases, such as for media queries, Sass variables are needed.

Before you begin:

- Examine the Component Styling section in the Oracle JET Cookbook for examples.
- Refer to the JET CSS Variables section of the JET API reference doc for an overview of the CSS variables and their values.
- Optionally, download and install the CSS Variable Theme Builder - Instruction Tab application when you want to learn about the available CSS variables in this interactive demo application. Follow the instructions online to modify the theme definition files to learn how CSS variable overrides change the demo tool UI.

To override JET CSS variables to style component instances:

1. If you have not added theming support to the application, open the `/src/css` folder of your application and edit the application CSS that you maintain. You can add a style class that defines the CSS variable overrides that you intend to apply to certain components.

   For example, to change the height of certain buttons in your application, you can create a style class `.demo-jumbo-button` and override the JET CSS variable `--oj-button-height` in the class, like this:

   ```
   .demo-jumbo-button {
     --oj-button-height: 4rem;
     }
   ```

   And then, in your application HTML, you can style buttons with the class as needed.

   ```
   <oj-button class="demo-jumbo-button"></button>
   ```

2. Alternatively, if you are theming your application, you can add the style class to the generated `_<themeName>.cssvars.settings.scss` theme file located in the `/src/themes/<themeName>/web` folder.

   When you edit the generated file, you must add your style class definitions with JET CSS overrides so they appear at the end of the file, after all JET CSS variables and they must not be contained within `:root`.

3. To run your application with live reload enabled, enter `ojet serve` with the `--theme` option to specify the CSS that you maintain for your application.

   ```
   ojet serve --theme=themeName
   ```

   You can omit the `--theme` option if you set the `defaultTheme` property in the `oraclejetconfig.json` file.

4. Use Oracle JET live reload to immediately observe the effect of any CSS variable overrides.

5. To exit the application, press Ctrl+C at the terminal prompt.

Disable JET Styling of Base HTML Tags

By default, Oracle JET applies styles to HTML tag elements such as `a`, `h1`, `h2`, and so on. This feature makes it easier to code a page since you do not have to apply selectors to each occurrence of the element.

If you do not want to apply styles to all base HTML tags by default in your custom theme, you can specify that Oracle JET generate style classes that can be placed on tags. The base theme CSS styles loaded at build time are controlled by import statements in custom theme settings files that you can optionally add to your project. To work with the CSS settings files in your project, you need to add theming support to the app and then create a custom theme.

With the custom theme settings files added to the `/src/themes` folder of your project, you can comment out an import statement that by default enables importing all JET component styles and add in its place the no-tag version of the import statement. With the appropriate import setting completed, when you build your app, JET Tooling will process the CSS settings files and load CSS for all JET components, but without the JET style classes for HTML base tags. To apply JET style to the HTML tags with the no-tag import enabled, you must explicitly set the JET style class on the desired HTML tag. For example, you can apply the JET style for links on the HTML anchor tag, like this `<a class=oj-link>`.

The following table lists the HTML tags with default Oracle JET tag styles and the corresponding Oracle JET style class that you can optionally apply when you enable the no-tag import statement.

HTML Tag	Oracle JET Style Class
`html`	`oj-html`
`body`	`oj-body`
`a`	`oj-link`
`h1, h2, h3, h4`	`oj-header`
`hr`	`oj-hr`
`p`	`oj-p`
`ul, ol`	`oj-ul, oj-ol`

Before you begin:

- Install the theming toolchain and configure a custom theme, as described in Add Custom Theme Support with the JET CLI. The `themes` folder with custom theme settings files are added to your app source.

To disable JET styling of base HTML tag:

1. In the `/src/themes/`*`themeName`*`/web` folder of your app, edit the aggregating *`themeName`*`.scss` file, and comment out the import statement that enables importing all JET component styles.

   ```
   // The import statement contains redwood or stable, depending on the value
   that you
   // supplied to the --basetheme argument when you created the custom theme.
   //@import "oj/all-components/themes/redwood/_oj-all-components.scss";
   or:
   //@import "oj/all-components/themes/stable/_oj-all-components.scss";
   ```

   And, then add the import statement to enable compiling with Oracle JET style classes that you can optionally apply to HTML tags.

   ```
   @import "oj/all-components/themes/redwood/_oj-all-components-notag.scss";
   or:
   @import "oj/all-components/themes/stable/_oj-all-components-notag.scss";
   ```

   This will generate style classes that you can apply to HTML tags, for example `<a class=oj-link>`.

2. If your app is already running, you can immediately observe the styling changes. Otherwise, serve your app to enable live reload.

   ```
   ojet serve
   ```

3. To exit the app, press Ctrl+C at the terminal prompt.

15

Secure Oracle JET Apps

Oracle JET follows security best practices for Oracle JET components and provides the `OAuth` class to help you manage access to users' private data.

About Securing Oracle JET Apps

Oracle JET apps are client-side HTML apps written in JavaScript, and you should follow best practices for securing your Oracle JET apps.

There are a number of Internet resources available that can assist you, including the Open Web Application Security Project (OWASP), Web Application Security Project (WASP), Web Application Security Working Group (WASWG), and various commercial sites.

Oracle JET Components and Security

Oracle JET components follow best practices for security. In particular:

- All JavaScript code is executed in `strict` mode using the `use strict` directive.

  Strict mode changes warnings about poor syntax, such as using undeclared variables, into actual errors that you must correct. For more information, see http://www.w3schools.com/js/js_strict.asp.

- Oracle JET code does not use inline `script` elements.

  Because browsers can't tell where the inline script originated, the World Wide Web Consortium (W3C) Content Security Policy prohibits the use of inline scripts. For additional information, see https://w3c.github.io/webappsec/specs/content-security-policy.

- Oracle JET code does not generate random numbers.
- Any HTML generated by an Oracle JET component is either escaped or sanitized.

Oracle JET Security and Developer Responsibilities

Oracle JET components follow established security guidelines and ensure that strings provided as options and user input will never be executed as JavaScript to prevent XSS attacks. However, Oracle JET does not include a mechanism for sanitizing strings, and you should consult established guidelines for dealing with XSS attacks in your own code and content.

You can find more information about securing JavaScript apps in the DOM based XSS Prevention Cheat Sheet.

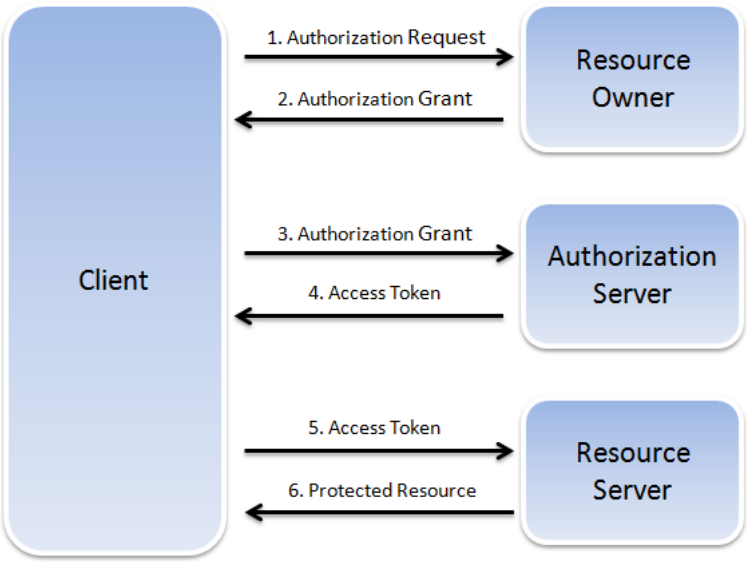
Oracle JET Security Features

The Oracle JET API provides the `OAuth` authorization plugin which supports the OAuth 2.0 open protocol. OAuth standardizes the way desktop and web apps access a user's private data. It provides a mechanism for users to grant access to private data without sharing their private username and password credentials.

OAuth 2.0 defines the following roles:

- Resource owner: An entity that can grant access to a protected resource, such as the end user.
- Client: app making protected and authorized resource requests on behalf of the resource owner.
- Resource server: Server hosting the protected resources that can accept and respond to protected resource requests using access tokens.
- Authorization server: Server that issues access tokens to the client after it successfully authenticates the resource owner and obtains authorization. The authorization server can be the same server as the resource server. In addition, an authorization server can issue access tokens accepted by multiple resource servers.

OAuth 2.0 Request for Comments (RFC) 6749 describes the interaction between the four roles as an abstract flow.



1. The client requests authorization from the resource owner, either directly or through the authorization server. Note that the RFC specifies that the authorization server is preferred.
2. The client receives an authorization grant, which is defined as the credential representing the resource owner's authorization.
3. The client requests an access token from the authorization server by authenticating with the server and presenting the authorization grant.
4. The authorization server issues the access token after authenticating the client and validating the authorization grant.
5. The client presents the access token to the resource server and requests the protected resource.
6. The resource server validates the access token and serves the request if validated.

The access token is a unique identifier issued by the server and used by the client to associate authenticated requests with the resource owner whose authorization is requested or has been obtained by the client.

The Oracle JET `OAuth` plugin provides functions for the following tasks:

- Getting access token credentials if initialized by client credentials.
- Caching access token credentials.
- Creating the header array with bearer token.

For details about using the `OAuth` plugin, see Use OAuth in Your Oracle JET App. For additional information about OAuth 2.0, see https://tools.ietf.org/html/rfc6749.

Oracle JET Secure Response Headers

Oracle JET recommends the usage of HTTP response headers to securely host your JET web application. These response headers protect your web applications from cross scripting attacks (XSS) attacks, packet sniffing, and clickjacking.

You must configure these response headers on the server where the JET application is hosted. As the configuration of these response headers is dependent on the type of server, you must refer to the documentation of your server for configuration steps.

You must use the secure response headers to notify the user agent to only connect to a given site over HTTPS and load all resources over secure channels to control XSS attacks and packet sniffing. You can configure the response header in the server to specify the protocols that are allowed to be used; for example, a server can specify that all content must be loaded using HTTPS protocol.

It is highly recommended to host your JET app using the HTTPS protocol to reduce the cross scripting attacks or packet sniffing. Also some of the new browser features only work under HTTPS protocol. The below table lists some of the secure response headers along with the HTTPS column that indicates which of these headers are specific for HTTPS based configuration.

Table 15-1 Secure Response Header Options

Option	Value	HTTPS Related	Description
`Content-Security-Policy`	See Table 15-2, Content-Security-Policy Header Options.	No	Specifies fine-grained resource access.
`X-XSS-Protection`	`1; mode=block`	No	Blocks a page when cross site scripting attempt is detected. **NOTE**: • The `X-XSS-Protection` directive is a defense-in-depth mechanism to mitigate the effect of reflected XSS vulnerabilities and does not detect or block persistent or DOM based XSS attacks. Apps must still perform proper input validation on the server and output encoding as the primary defense against XSS. • Mozilla Firefox does not implement cross site scripting protection.
`X-Permitted-Cross-Domain-Policies`	`none`	No	Cross-domain policy file is an XML document that grants a web client permission to handle data across domains.

Table 15-1 (Cont.) Secure Response Header Options

Option	Value	HTTPS Related	Description
`X-Frame-Options`	`deny`	No	Prevents clickjacking for browsers. This directive can only be set using an HTTP response header. To frame your content from the same origin, use `sameorigin`. If hosted by known host(s), specify `allow-frame hostname`. **NOTE**: • If a request contains both a CSP `frame-ancestors` and `X-Frame-Options` directive, browsers that support both will ignore the `X-Frame-Options` directive in favor of the standardized CSP `frame-ancestors` directive. • The `allow-frame` directive does not support wildcards.
`X-Content-Type-Options`	`nosniff`	No	Ensures browser uses MIME type to determine the content type. Use of this directive with images requires the image format to match its specified MIME type. Use of this directive on JavasSript files requires the MIME type to be set to `text/javascript`.
`Strict-Transport-Security`	`max-age=<secs>; includeSubDomains`	Yes	Tells the browser to communicate only with the specified site (and any subdomains) over HTTPS and prevents the user from overriding an invalid or self-signed certificate.
`Referrer-Policy`	`no-referrer`	No	Tells the browser to include referrer information on outbound link requests.
`Public-Key-Pins`	`pin-sha256="<sha256>"; max-age=<secs>`	Yes	Prevents use of incorrect or fraudulent certificates.
`Expect-CT`	`max-age=86400, enforce`	Yes	Signals to the browser that compliance to the Certificate Transparency Policy should be enforced.

Content Security Policy Headers

Content Security Policy (CSP) is delivered through an HTTP response header and controls the resources that an Oracle JET web app can use.

The CSP header provides a mechanism to restrict the locations from which JavaScript running in a browser can load the required resources, restrict the execution of JavaScript, and control situations in which a page can be framed. This can mitigate Cross Site Scripting (XSS) vulnerabilities as well as provide protection against clickjacking attacks.

You should enable CSP for browsers to help the server administrators reduce or eliminate the attacks by specifying the domains that the browser should consider to be valid sources for loading executable scripts, stylesheets, images, fonts, and so on. See the Browser Compatibility Matrix for the browser versions that support CSP.

Configuring CSP involves adding the `Content-Security-Policy` HTTP header to a web page and giving it values to control resources the user agent is allowed to load for that page. To add

`Content-Security-Policy` HTTP header to a web page, configure your web server to return the `Content-Security-Policy` HTTP response header. For example, here is a basic CSP response header, where `script-src` directive specifies an executable script as the resource type and `'self'` is a constant that specifies the current domain as the approved source that the browser may load script from:

```
Content-Security-Policy: script-src 'self'
```

CSP has some of the following commonly used constants:

- `'none'`: Blocks the use of certain resource type.
- `'self'`: Matches the current origin (but not subdomains).
- `'unsafe-eval'`: Allows the use of mechanisms like `eval()`.

  Note as an alternative to the `unsafe-eval` CSP domain, JET provides an expression evaluator that allows JET expression syntax to be evaluated in a way that is compliant with Content Security Policies that prohibit unsafe evaluations. The default CSP use of the `unsafe-eval` domain remains unchanged, but apps can opt into the JET behavior with some restrictions on the types of expressions that are supported. See details on creation, usage, supported expressions and limitations in the CspExpressionEvaluator API documentation.

Alternatively, you can also use the HTML meta tags to configure CSP. For example:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; img-
src https://*; frame-src 'none';">
```

Note that some of the CSP directives do not work with the HTML meta tags, for example `frame-ancestors`.

The below table describes the out-of-box settings required by a JET web app to run in its most secure mode without changing the JET functionality. The JET web app may need to modify these settings for additional resource origins. The table lists the different response header directives that can be used while enabling CSP based on the two following scenarios:

- Co-Hosted: When the JET and the app source codes are hosted on the same server
- Content Delivery Network (CDN): When the JET code is from the JET CDN and the app code is from a different server

Table 15-2 Content-Security-Policy Header Options

CSP Version	Header Options	Co-Hosted	CDN	Description
CSP 1.0	`default-src`	`'none'`	`'none'`	Serves as a default setting that ensures resource loading is blocked if the resource type is not specified. All other settings need to be explicitly enabled for specific resource origins.

Table 15-2 (Cont.) Content-Security-Policy Header Options

CSP Version	Header Options	Co-Hosted	CDN	Description
CSP 1.0	`connect-src`	`'self'`	`'self'`	Manages the REST and Web Sockets to be accessed.
CSP 1.0	`font-src`	`'self'`	`'self' static.oracle.com`	Specifies valid sources for fonts.
CSP 1.0	`img-src`	`data: 'self'`	`'self' data: static.oracle.com`	Specifies valid sources for images. Allows JET inline images.
CSP 1.0	`media-src`	`'none'`	`'none'`	Specifies valid sources for loading media using the `<audio>`, `<video>`, and `<track>` elements.
CSP 1.0	`object-src`	`'none'`	`'none'`	Specifies valid sources for the `<object>`, `<embed>`, and `<applet>` elements.
CSP 1.0	`script-src`	`'self' 'unsafe-eval'`	`'self' static.oracle.com 'unsafe-eval';`	Specifies valid sources for JavaScript. This directive is used for knockout expressions and JET function creation. **Note:** The use of `unsafe-eval` is currently required by Knockout to resolve expressions, but JET provides an alternative expression evaluator when you require your app to run in a strict Content Security Policy environment. For usage information, see the CspExpressionEvaluator API documentation.
CSP 1.0	`sandbox`	-	-	Runs the page as in a sandboxed iframe.

Table 15-2 (Cont.) Content-Security-Policy Header Options

CSP Version	Header Options	Co-Hosted	CDN	Description
CSP 2.0	`form-action`	-	-	This directive is used for form submits. Not applicable for JET.
CSP 2.0	`frame-ancestors`	`'none'`	`'none'`	Specifies valid sources for nested browsing contexts loading using elements such as `<frame>` and `<iframe>` and prevents clickjacking for browsers. This directive can only be set using an HTTP response header. To frame your content from the same origin, use `'self'`. If hosted by known host(s), specify the hosts.

When `default-src` is set to `none`, you must explicitly enable all the other needed settings for specific resource origins.

The following example shows how to set up CSP if a website administrator wants to allow content from a trusted domain and all its subdomains:

```
Content-Security-Policy: default-src 'self' *.trusted.com
```

The following example shows how to set up CSP if a website administrator wants to allow users of a web app to include images from any origin in their own content, but to restrict audio or video media to trusted providers, and all scripts only to a specific server that hosts trusted code.

```
Content-Security-Policy: default-src 'self'; img-src *; media-src media1.com
media2.com; script-src userscripts.example.com
```

Use OAuth in Your Oracle JET App

You can use the `OAuth` plugin to manage access to client (end user) private data. The Oracle JET API includes the `OAuth` class which provides the methods you can use to initialize the `OAuth` object, verify initialization, and calculate the authorization header based on client credentials or access token.

Initialize OAuth

You can create an instance of a specific `OAuth` object using the `OAuth` constructor:

```
new OAuth(header, attributes)
```

The `attributes` and `header` parameters are optional.

Parameter	Type	Description
`header`	String	MIME Header name. Defaults to `Authorization`
`attributes`	Object	Contains client credentials or access/bearer token. Client credentials contain: • `client_id` (required): public client Credentials • `client_secret` (required): secret client credentials • `bearer_url` (required): URL for token bearer and refresh credentials • Additional attributes as needed (optional) Access/bearer tokens contain: • `access_token` (required): Bearer token • Additional attributes as needed (optional)

The code sample below shows three examples for initializing `OAuth`.

```
// Initialize OAuth with client credentials
var myOAuth = new OAuth('X-Header', {...Client credentials...});

// Initialize OAuth with token credentials
var myOAuth = new OAuth('X-Header', {...Access/Bearer token...});

// Initialize OAuth manually
var myOAuth = new OAuth();
```

If you choose to initialize OAuth manually, you can add the client credentials or access/bearer token using methods shown in the following code sample.

```
// Initializing client credentials manually
myOAuth.setAccessTokenRequest({...Client Credentials ...});
myOAuth.clientCredentialGrant();

// Initializing access bearer token manually
myOAuth.setAccessTokenResponse({...Access Token...});
```

The OAuth API also includes methods for getting and cleaning the client credentials or access tokens. For additional information, see the OAuth API documentation.

Verify OAuth Initialization

Use the `isInitialized()` method to verify that the initialization succeeded.

```
var initFlag = myOAuth.isInitialized();
```

Obtain the OAuth Header

Use the `getHeader()` method to get the OAuth header. The method calculates the authorization header based on the client credentials or access token.

```
// Client credentials
var myOAuth = new OAuth('New-Header', {...Client credentials...});
var myHeaders =  myOAuth.getHeader();

// Access token
var myOAuth = new OAuth('New-Header', {...Access/Bearer token...});
var myHeaders =  myOAuth.getHeader();

// Manual initialization, client credentials
var myOAuth = new OAuth();
myOAuth.setAccessTokenRequest({...Client credentials...});
var myHeaders =  myOAuth.getHeader();

// Manual initialization, access token
var myOAuth = new OAuth('New-Header', {...Access/Bearer token...});
var myHeaders =  myOAuth.getHeader();
```

About Cross-Origin Resource Sharing (CORS)

CORS is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served. The same-origin security policy of JavaScript forbids certain cross-domain requests, notably Ajax requests, by default.

Rejected resource requests due to CORS can affect web apps. Apps that encounter a rejection receive messages such as the following example in response to resource requests:

```
No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

Server-side administrators can specify the origins allowed to access their resources by modifying the policy used by their remote server to allow cross-site requests from trusted clients. For example, to access a remote service managed by Oracle's Mobile Cloud Service (MCS), an MCS administrator configures MCS's `Security_AllowOrigin` environment policy with a comma-separated list of URL patterns that identify the remote services that serve resources from different domains.

If you serve your web app to the local browser for testing, you may encounter CORS rejections. Some browsers provide options to disable CORS, such as Chrome's `--disable-web-security` and Firefox's `security.fileuri.strict_origin_policy` and some browsers support plugins that work around CORS.

Only use these options when testing your app and ensure that you complete further testing in a production-like environment without these options to be sure that your app will not encounter CORS issues in production.

16

Configure Data Cache and Offline Support

Use the Oracle Offline Persistence Toolkit to enable data caching and offline support within your Oracle JET app.

About the Oracle Offline Persistence Toolkit

The toolkit is a client-side JavaScript library that Oracle maintains as an open-source project. The toolkit provides caching and offline support at the HTTP request layer.

This support is transparent to the user and is done through the Fetch API and an XHR adapter. HTTP requests made while the client or client device is offline are captured for replay when connection to the server is restored. Additional capabilities include a persistent storage layer, synchronization manager, binary data support and various configuration APIs for customizing the default behavior. This toolkit can be used in both ServiceWorker and non-ServiceWorker contexts within web apps.

Using the toolkit, you can configure your app to:

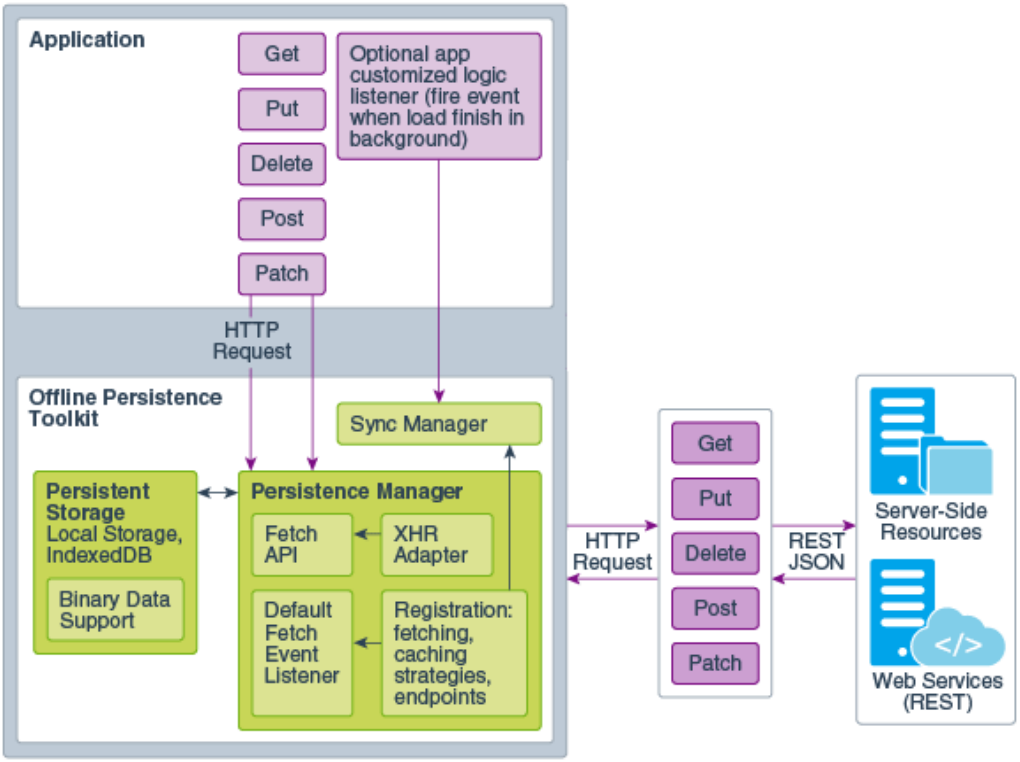
- Download content for offline reading where connectivity isn't available.

  For example, an app could include product inventory data that a salesperson could download and read at customer sites where connectivity isn't available.
- Cache content for improved performance.
- Perform transactions on the downloaded content where connectivity isn't available and upload the transactions when connectivity returns.

  For example, the salesperson could visit a site with no Internet access and enter an order for some number of product items. When connectivity returns, the app can automatically send the transaction to the server.
- Provide conflict resolution when the offline data can't merge with the server.

  If the salesperson's request exceeds the amount of available inventory, the app can configure a message asking the salesperson to cancel the order or place the item on back order.

The architecture diagram illustrates the major components of the toolkit and how an app interacts with it.



Install the Offline Persistence Toolkit

Use npm to install the offline persistence toolkit. After installation, you must update your application's `src/js/path_mapping.json` file to recognize the new package.

1. Change to your application's top level directory and open a terminal window.
2. At the terminal prompt, enter the following command to install the toolkit: `npm install @oracle/offline-persistence-toolkit --save`.
3. Change to your app's `src/js` directory and open `path_mapping.json` for editing.
4. Add the persistence toolkit to the `"libs"` entry.

   The easiest way to add the toolkit is to copy an existing entry that's similar to your library and modify as needed. A sample entry for `offline-persistence-toolkit`, which started with a copy of the `"ojs"` entry, is shown below.

```
"libs": {

    "offline-persistence-toolkit": {
      "cdn": "",
      "cwd": "node_modules/@oracle/offline-persistence-toolkit/dist",
      "debug": {
        "cwd": "debug",
        "src": ["**"],
        "path": "libs/offline-persistence-toolkit/debug",
        "cdn": ""
      },
      "release": {
        "cwd": "min",
```

```
            "src": ["**"],
            "path": "libs/offline-persistence-toolkit/min",
            "cdn": ""
          }
    },
```

For information about using the toolkit once you have it installed in your Oracle JET app, see the `README.md` and Wiki for the persistence toolkit on Github at https://github.com/oracle/offline-persistence-toolkit.

17

Optimize Performance of Oracle JET Apps

Oracle JET applications are client-side HTML5 applications. Most performance optimization recommendations relating to client-side HTML applications also apply to applications developed using Oracle JET or to Oracle JET components. In addition, some Oracle JET components have performance recommendations that are specific to the component.

About Performance and Oracle JET Apps

In general, you can optimize an Oracle JET app the same way that you would optimize performance for any client-side HTML5 app.

There are many online resources that provide tips for performance optimization. For example, the Google Developers website describes their tools for improving the performance of the app.

Most of the recommendations made by the Google tools are up to you to implement, but Oracle JET includes features that can reduce the payload size and the number of trips to retrieve the Oracle JET app's CSS. In general, strive to follow these guidelines.

1. Always minify and bundle app resources to reduce the number of requests from the server.
2. Configure the app to load resources from Oracle CDN to minimize network usage.
3. Configure the app to use the Oracle JET library on CDN so that the `bundles-config.js` script will load minified bundles and modules by default.
4. Compress the app with gzip to reduce the size (and enable compression on the web server.)
5. Enable HTTP caching on web server so that some requests can be served from the cache instead of from the server. Use ETags on files that should always be served from the server.
6. Take advantage of HTTP/2 to serve page resources faster than is possible with HTTP/1.1.
7. Use a single page app, so that the browser isn’t forced to tear down and rebuild the whole app.
8. Avoid putting too many data-centric components into a single page.
9. Optimize graphic images: prefer vector format; choose the appropriate image format based on the best overall compression available.

For more information about these optimization tips and others, see Add Performance Optimization to an Oracle JET App.

Add Performance Optimization to an Oracle JET App

Most tips for optimizing performance of web apps also apply to Oracle JET apps. However, there are some steps you can take that apply specifically to Oracle JET apps to optimize JavaScript, CSS, Oracle JET components, REST calls, and images.

JavaScript Performance Tips

Performance Tip	Details
Maintain the expected JavaScript folder structure	Use folder organization generated by the Oracle JET tooling and maintain all JavaScript files inside the `js` folder of the app root directory.
Send only the JavaScript code that your app needs.	Oracle JET includes modules that you can load with RequireJS. For additional information, see Use RequireJS for Modular Development. One approach is to preload the JavaScript modules that your app will use. The following sample shows how to modify the require function in the app `main.js`.

```
require(['ojs/ojbootstrap', 'knockout', 'ojs/ojknockout'],
  function (Bootstrap) {
    Bootstrap.whenDocumentReady().then(
      function () {
        function init() {
        }

        // If running in a hybrid (e.g. Cordova) environment,
we need to wait for the deviceready
        // event before executing any code that might interact
with Cordova APIs or plugins.
        if (document.body.classList.contains('oj-hybrid')) {
          document.addEventListener('deviceready', init);
        } else {
          init();
        }

        //after main doc is done preload some js
        require(['ojs/ojbutton', 'ojs/ojdvt-base', 'ojs/
ojtree', 'ojs/ojaccordion', 'ojs/ojtreemap'], //example of what
can be preloaded
          function()
          {
          }
        );
      }
    );
  }
);
```

Performance Tip	Details

Send minified/obfuscated JavaScript.	Oracle JET provides minified versions of the Oracle JET library as well as third-party libraries when available. By default, the path mappings for the minified versions of these libraries in `path_mapping.json` will be injected into the Oracle JET RequireJS bootstrap file included with all Oracle JET distributions when you build a release version of the app. The following sample shows a single library from `path_mappings.json` where the minified library is available for release mode.
	
	`"jquery": {`
	`"cdn": "3rdparty",`
	`"cwd": "node_modules/jquery/dist",`
	`"debug": {`
	`"src": "jquery.js",`
	`"path": "libs/jquery/jquery-#{version}.js",`
	`"cdnPath": "jquery/jquery-3.x.x"`
	`},`
	`"release": {`
	`"src": "jquery.min.js",`
	`"path": "libs/jquery/jquery-#{version}.min.js",`
	`"cdnPath": "jquery/jquery-3.x.x.min"`
	`}`
	`},`
	
	For additional information about using the RequireJS bootstrap file in your Oracle JET app, see About RequireJS in an Oracle JET App.

Minimize the number of trips to retrieve the JavaScript.	Oracle JET doesn't provide support for minimizing the number of trips, but RequireJS has an optimization tool that you can use to combine modules. For additional detail, see the documentation for the RequireJS optimizer. Alternatively, use a JavaScript code minifier, such as Terser to bundle and minify the JavaScript source.
Use lazy loading for JavaScript not needed on first render.	You can lazy load content that is not needed on first render. For example, you can configure the `oj-film-strip` component to retrieve child node data only when requested. For an example, see the Lazy Loading (oj-film-strip) Oracle JET Cookbook example.
Compress or zip the payload.	Oracle JET has no control over the server, and this recommendation is up to you to implement. For some additional information and tips, see https://developers.google.com/speed/docs/best-practices/payload#GzipCompression.
Set cache headers.	JET has no control over the server, and this recommendation is up to you to implement. For additional information about cache optimization, see https://developers.google.com/speed/docs/best-practices/caching.

CSS Performance Tips

Performance Tip	Details
Maintain the expected CSS folder structure	Use folder organization generated by the Oracle JET tooling and maintain all CSS files inside the `css` folder of the app root directory.
Render pages for all the needed CSS once	Link to style sheets inside the HEAD section of the HTML page and do not use CSS links in the page body to avoid rerendering an already loaded page.
Send only the CSS that your app needs.	To control the CSS content that goes into your app, create a custom theme and limit what it includes to the CSS that your app needs. See Optimize the CSS in a Custom Theme. Also, if you're using the Oracle JET grid system, you can also control which responsive classes get included in the CSS. For details, see Control the Size and Generation of the CSS.

Performance Tip	Details
Send minified/obfuscated CSS.	By default, Oracle JET includes minified CSS. However, if you want to modify the CSS to send only what your app needs, you can use Sass to minimize your output. For additional information, see the `:compressed` option at: http://sass-lang.com/documentation/file.SASS_REFERENCE.html#output_style.

Oracle JET App and Component Performance Tips

Performance Tip	Details
Configure your app to load bundled JET modules and libraries using Oracle CDN and the bundle configuration support it provides for JET.	Leveraging the Oracle Content Delivery Network (CDN) and bundle configuration support optimizes the app startup performance of enterprise apps and also ensures that your app builds with the module and library versions required for a particular Oracle JET release. Referring directly to the bundles within the app is not recommended and that includes adding or modifying links that make direct reference to the configuration of the bundles. For additional information, see About Configuring the App for Oracle CDN Optimization.
Consider using plain HTML components were possible.	For stamping components (like Table or ListView) or declarative components (like a composite component), if you embed a simple Oracle JET component like a read-only input component or button, consider using a plain HTML component instead. The plain HTML component can be lighter weight (less DOM), and if the component is stamped that can add up. However, note that with plain HTML, you also won't have access to any built-in accessibility support, such as converters and validators, which the Oracle JET component provides.
Follow Oracle JET component best practices.	Consult the API documentation for the Oracle JET component. The API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) includes a performance section for a component when applicable. For example, see the ojDataGrid—Performance section.
Limit number of Oracle JET components per page.	The number of components on the page will impact the page load time. If you want to reduce the load time, place fewer data-centric components in the page.
Use the `oj-defer` element to delay binding execution	When a component contains hidden content to display, the `oj-defer` element delays the process of applying bindings to the child components until the hidden content becomes visible. For additional information on `oj-defer`, see the oj-defer API documentation.
Limit the fetch size for collection components	Set the collection component `scroll-policy-options.fetch-size` attribute equal to the number the number of items to display in the component viewport. Also set `scroll-policy="loadMoreOnScroll"` to ensure that the fetch for additional items occurs only when the user scrolls toward the end of the fetched list.

REST Request Performance Tips

Performance Tip	Details
Reduce the number of roundtrips between client and server.	There are a number of techniques that you can use to reduce the number of roundtrips, and here are some examples: • The number of REST requests on the page will impact the page load time. If you need to reduce the load time, simplify your page and make fewer REST requests. • A REST call that needs data from a previous REST call creates a dependency that triggers serialization and increases the data fetch response time. To reduce response time, minimize the number of dependent REST calls by redefining your REST calls to fetch only the data your UI requires. Ideally, the REST endpoint supports fetching of the exact data or can be redesigned as needed. • All REST operations should be executed asynchronously. To manage async state, invoke REST endpoints with a method to return a Promise from the start. • If the REST endpoint is slow to respond, and the data is essential to the app, consider prefetching the data in a non-blocking REST endpoint invocation. • The REST endpoint should be designed to support pagination.

Image Optimization

Performance Tip	Details
Maintain the expected images folder structure	Use folder organization generated by the Oracle JET tooling and maintain all image files inside the `images` folder of the app root.
Reduce image size.	Reducing the size of the images will result in faster downloads and reduce the time it takes to render the content on the screen. For example, Scalable Vector Graphics (SVG) images are usually smaller than Portable Network Graphics (PNG) images and scale on high resolution devices. There are also a number of third-party tools that you can use to reduce the size of your images. The tool that you select will depend on the image type, for example: • imagemin: Utility to compress PNG images • svgomg: Utility to compress SVG images. You can use this tool online or download svgo to work with the images on your own system.
Reduce the number of roundtrips between client and server.	There are a number of techniques that you can use to reduce the number of roundtrips, and here are some examples: • Icon fonts Icon fonts are useful when your icon uses a single color. Oracle JET uses icon fonts, and you can see examples of them at: Icon Fonts. You can find utilities on the Internet such as IcoMoon that you can use to generate icon fonts. • Image Sprites An image sprite is a collection of images combined into a single image, reducing the number of server requests. You can find examples of them at http://www.w3schools.com/css/css_image_sprites.asp. • Lazy loading You can use lazy loading to defer the loading of images not in the user's viewport. You can find many examples and utilities on the Internet that use this technique. • Base64 Encoding You can use Base64 Encoding to inline image data. They are commonly used in data Uniform Resource Indicators (URIs), and you can find additional information about them at https://developer.mozilla.org/docs/Web/HTTP/data_URIs.

For additional performance tips, see the Google Developers documentation for improving the quality of web pages, including how to audit for performance.

About Configuring the App for Oracle CDN Optimization

You can configure an Oracle JET app to minimize the network load at app startup through the use of the Oracle Content Delivery Network (CDN) and the Oracle JET distributions that the CDN supports.

The local loading of the required Oracle JET libraries and modules, which is configured by default when you create the app, is not recommended for a production app because it does not use the Oracle CDN and requires each app running in the browser to load libraries and modules in a standalone manner. This default configuration can be used when you build and serve the app locally until you need to stage the app in a test environment to simulate network access.

To enable Oracle CDN optimization, configure the `path_mapping.json` file in your app. The options configured in the path mapping file determine the library and module settings for the entire app. With the `path_mapping.json` file, you do not need to edit the path URL of the required libraries and modules in any other app file. When you configure path mapping for CDN optimization, you will determine how the tooling updates the `require` block of the `main.js` file and how the app will load modules and libraries as follows:

- If you configure the path mappings to use the CDN without accessing the bundles configuration, such that modules and libraries will be loaded individually from the CDN, the tooling injects the URLs from the path mapping file into the `require` block of the `main.js` file.
- If you configure the path mappings to use CDN bundle loading, the tooling updates the `index.html` file to execute the bundles configuration script file (`bundles-config.js`) from the following script reference:

```
<body>
  <script type="text/javascript" src="https://static.oracle.com/cdn/jet/
18.1.0/default/js/bundles-config.js"></script>
..
</body>
```

> **Note:**
>
> Starting in JET release 9.0.0, the convention of using the leading character "`v`" to identify the release number has changed. As the above sample shows, the release identifier is now a semver value specified as 18.1.0 with no leading character.

The bundles configuration file specifies its own `require` block that the app executes to load as a set of bundled modules and libraries from the CDN. When you configure the app this way, the `main.js` file is updated by the tooling to display only a URL list comprising third-party libraries. In the bundles configuration scenario, the injected `require` block in the `main.js` file becomes a placeholder for any app-specific libraries that you want to add to the list. Where URL duplications may occur between the `require` block of the bundles configuration file and the app's `main.js` file, the bundles configuration takes precedence to ensure bundle loading from the CDN has priority.

> **Tip:**
>
> Configuring your app to reference the bundles configuration script file on the Oracle CDN is recommended because Oracle maintains the configuration for each release. By pointing your app to the current bundles configuration, you will ensure that your app runs with the latest supported library and module versions.

Configure Bundled Loading of Libraries and Modules

For Oracle CDN optimization, you may use the bundles configuration script that loads a set of bundled libraries and modules from the CDN.

To configure bundle loading of the libraries and modules using the bundles configuration script, perform the following steps.

1. Open the `path_mapping.json` file in the `appRootDir/src/js` directory of your app and change the `use` element's value to `cdn`:

   ```
   "use": "cdn"
   ```

2. Leave the `cdns` element unchanged. It should show the following as the default path definitions for Oracle JET and third-party libraries:

   ```
   "cdns": {
       "jet": {
          "prefix": "https://static.oracle.com/cdn/jet/18.1.0/default/js",
          "css": "https://static.oracle.com/cdn/jet/18.1.0/default/css",
          "config": "bundles-config.js"
       },
       "3rdparty": "https://static.oracle.com/cdn/jet/18.1.0/3rdparty"
   },
   ```

   > **Note:**
   >
   > Starting in JET release 9.0.0, the convention of using the leading character "`v`" to identify the release number has changed. As the above sample shows, the release identifier is now a semver value specified as 18.1.0 with no leading character.

3. Optionally, in the case of bundle loading, for each third-party library, update the value of the `cdn` element from `3rdparty` to `jet`. For example, this Knockout library path definition shows `"cdn": "jet"` to prevent the Knockout library's URL from being injected into the `main.js` file:

   ```
   "libs": {

     "knockout": {
       "cdn": "jet",
       "cwd": "node_modules/knockout/build/output",
      ...
    },
   ```

Setting `"cdn": "jet"` for each third-party library prevents the libraries' URLs from being injected into the `require` block of the `main.js` file. This update is not necessary to ensure bundle loading of third-party libraries because the bundles configuration script overrides duplicate URL paths that appear in the `main.js` file.

4. Save the file and either build or serve the app to complete the bundle loading configuration.

Configure Individual Loading of Libraries and Modules

You can configure path mappings to use the CDN without accessing the bundles configuration script, so that modules and libraries are loaded individually from the CDN.

To configure individual loading of the libraries and modules based on the `require` block of the `main.js` file (without the use of the bundles configuration script), perform the following steps.

1. Open the `path_mapping.json` file in the `appRootDir/src/js` directory of your app and change the `use` element's value to `cdn`.

   ```
   "use": "cdn"
   ```

2. Edit the `cdns` element to remove the `config` element so that the `cdns` path definitions for Oracle JET and third-party libraries are formatted as follows:

   ```
   "cdns": {
       "jet": "https://static.oracle.com/cdn/jet/18.1.0/default/js",
       "css": "https://static.oracle.com/cdn/jet/18.1.0/default/css",
       "3rdparty": "https://static.oracle.com/cdn/jet/18.1.0/3rdparty"
   }
   ```

3. Save the file and either build or serve the app to complete the `main.js` file's `require` block configuration.

When you need to make a change to the list of required libraries (for example, to specify a different release version), do not edit the `main.js` file; instead, edit the `path_mapping.json` file and, in the case of bundle loading, also edit the `bundles-config.js` URL in your app's `index.html` file. You will need to rebuild the app to apply the changes. For details about the `path_mapping.json` file and the configuration updates performed by the tooling, see Understand the Path Mapping Script File and Configuration Options.

Understand the Path Mapping Script File and Configuration Options

You may need to integrate additional functionality into your apps using libraries and modules that are not provided by Oracle JET. Apps built using Oracle JET's command-line interface (CLI) have a mechanism that helps with this process: the `path_mapping.json` file.

When you build or serve your app, Oracle JET tooling invokes the `appRootDir/src/js/path_mapping.json` configuration file and determines the URI path for the Oracle JET modules and libraries based on the settings you configured for the path mapping `use` attribute.

The `use` attribute, set to `local` by default, specifies the location of required libraries including core Oracle JET libraries (such as `ojs`, `ojL10n`, and `ojtranslations`) and third-party dependency libraries (such as `knockout`, `jquery`, and `hammerjs`).

When you build your app, Oracle JET defines load paths for each library specified in the `path_mapping.json` file using the `requirejs.config()` function of the app's `main.js` file. Each library path URI is determined based on the `path` or `cdnPath` attribute of the library listed in the `libs` map.

For example, the path mapping entry for the Knockout library shows the following details.

```
"libs": {
  "knockout": {
    "cdn": "3rdparty",
    "cwd": "node_modules/knockout/build/output",
    "debug": {
      "src": "knockout-latest.debug.js",
      "path": "libs/knockout/knockout-#{version}.debug.js",
      "cdnPath": "knockout/knockout-3.x.x.debug"
    },
    "release": {
      "src": "knockout-latest.js",
      "path": "libs/knockout/knockout-#{version}.js",
      "cdnPath": "knockout/knockout-3.x.x"
    }
},
```

After building or serving the app, the `main.js` file's `requirejs.config()` paths map contains the following path mapping:

```
"knockout":"libs/knockout/knockout-x.x.x.debug"
```

For more information on the `path_mapping.json` configuration file and how to use it, see Add Third-Party Tools or Libraries to Your Oracle JET App.

Work with Libraries and Modules on Content Delivery Networks

The following list includes common scenarios encountered in working with libraries and modules hosted on CDNs and your Oracle JET apps.

When configured for the Oracle CDN, the `main.js` file's `require` block is determined either entirely by the path mapping file local to the app or, in the case of the bundle loading optimization, partially from the path mapping file and partially from the `require` block of the `bundles-config.js` file maintained by Oracle on the Oracle CDN. Path injector markers in the `main.js` file indicate where the release-specific URLs appear.

- **CDN Scenario 1:** To load libraries and modules as bundles from the Oracle CDN, by default only the path mappings for third-party libraries will appear in the URL library list in the `require` block of the `main.js` file.

  For example, the path mapping definition for the Knockout library shows the following details. Note that the `config` attribute specifies the name of the bundles configuration script file as `bundles-config.js`.

  ```
  "baseUrl": "js" <==ignored
  "use": "cdn"

  "cdns": {
      "jet": {
         "prefix": "https://static.oracle.com/cdn/jet/18.1.0/default/js",
  ```

```
        "css": "https://static.oracle.com/cdn/jet/18.1.0/default/css",
        "config": "bundles-config.js"
      },
     "3rdparty": "https://static.oracle.com/cdn/jet/18.1.0/3rdparty"
},

"libs": {
   ...
     "cdnPath": "knockout/knockout-3.x.x"
}
```

After building or serving your app, the `main.js` file's `require` block contains a list of third-party library URLs as a placeholder.

Note that loading libraries and module as specified in the `bundles-config.js` file's `require` block takes precedence over any duplicate libraries that may appear in the `main.js` file's `require` block. However, if you prefer, you can configure the third-party library path mapping so that their URLs do not appear in the `main.js` file's `require` block. To accomplish this, edit `"cdn": "3rdparty"` in the `path_mapping.json` file to `"cdn": "jet"` for each third-party library path definition.

- **CDN Scenario 2:** To load libraries individually from the Oracle CDN using the path mapping URLs to specify the location, the list of library URLs will appear entirely in the `main.js` file's `require` block.

  For example, the path mapping definition for the Knockout library shows the following details after you edit the `cdns` element to remove the bundles configuration script reference.

  ```
  "baseUrl": "js" <==ignored
  "use": "cdn"

  "cdns": {
      "jet": "https://static.oracle.com/cdn/jet/18.1.0/default/js",
      "css": "https://static.oracle.com/cdn/jet/18.1.0/default/css",
      "3rdparty": "https://static.oracle.com/cdn/jet/18.1.0/3rdparty"
  }

  "libs": {
     ...
       "cdnPath": "knockout/knockout-3.x.x"
  }
  ```

  After a build or serve, the `main.js` file's `require` block contains the following URL (along with the URLs for all other base libraries and modules):

  ```
  "knockout": "https://static.oracle.com/cdn/jet/18.1.0/3rdparty/knockout/
  knockout-3.x.x"
  ```

- **CDN Scenario 3:** If your app needs to access libraries that reside on a non-Oracle CDN, you can update the `path_mapping.json` file to specify your own CDN endpoint and library definition.

  Depending on whether you use the bundles configuration script, add your CDN name and endpoint URI to the `cdns` definition as follows.

When using the bundles configuration script to load libraries and modules:

```
"cdns": {
    "jet": {
      "prefix": "https://static.oracle.com/cdn/jet/18.1.0/default/js",
      "css": "https://static.oracle.com/cdn/jet/18.1.0/default/css",
      "config": "bundles-config.js"
    },
    "3rdparty": "https://static.oracle.com/cdn/jet/18.1.0/3rdparty"
    "yourCDN": "endPoint to your own CDN"
},
...
```

Or, when loading libraries and modules individually (not using the bundles configuration script):

```
"cdns": {
    "jet": "https://static.oracle.com/cdn/jet/18.1.0/default/js",
    "css": "https://static.oracle.com/cdn/jet/18.1.0/default/css",
    "3rdparty": "https://static.oracle.com/cdn/jet/18.1.0/3rdparty",
    "yourCDN": "endPoint to your own CDN"
},
...
```

Then, in the list of libraries, define your library entry similar to the following sample.

```
"yourLib": {
  "cdn": "yourCDN",
  "cwd": "node_modules/yourLib",
  "debug": {
    "src": "yourLib.js",
    "path": "libs/yourLib/yourLib.js",
    "cdnPath": "yourLib/yourLib.js"
  },
  "release": {
    "src": "yourLib.min.js",
    "path": "libs/yourLib/yourLib.min.js",
    "cdnPath": "yourLib/yourLib.min.js"
  }
},
```

18

Audit Oracle JET App Files

An Oracle JET audit runs against the app files of your JET project and performs a static analysis of the source code from an Oracle JET perspective. Audit diagnostic messages result from an invocation of the Oracle JET Audit Framework (JAF) command-line utility and are governed by rules that are specific to a JET release version.
Oracle JET Audit Framework (JAF) relies on the configuration file created by the JET tooling when you invoke the JAF initialization command `ojaf --init` in a Command Prompt window on the JET app.

The `oraclejafconfig.json` file that you create when you initialize Oracle JAF the first time defines the properties that you can use to control many aspects of your JET app audit. For example, by configuring the JAF audit, you can perform the following.

- Specify the JET version when you want to use audit rules that are specific to a JET version. This is configured by default as the JET version of the app to be audited.
- Specify the file set when you want to exclude app directories and file types. This is configured by default to include all files of the app to be audited.
- Invoke custom audit rules that are user-defined and assembled as a JAF rule pack for distribution.
- Prevent specific audit rules from running in the audit or limiting the audit to only rules of a certain severity level.
- Include the metadata of Oracle JET Web Components to audit the HTML files of your app's custom components.
- Control the JavaScript source code to audit based on JAF comments that you embed in your source files.
- Work with the output of the audit to customize the presentation of audit messages or to suppress audit messages.

The properties in the `oraclejafconfig.json` file configuration settings are up to you to specify. By doing so, you can fine-tune the audit to focus audit results on only the source that you intend. Multiple configuration files can created for specific runtime criteria or projects. The configuration files are JSON format, but JavaScript style comments are permitted for documentation purposes. The configuration file to be used can be specified on the command-line.

Each time you run the audit from a Command Prompt window, Oracle JAF searches the directory in which you initiated the audit for the JAF configuration file `oraclejafconfig.json`. If no configuration file is found there, then JAF processes only HTML files found in the current directory. In that case, the default JAF configuration settings are used for the audit.

If the built-in audit rules provided with the JAF installation do not meet all the diagnostic requirements of your app, you can write custom audit rules to extend JAF. You implement user-defined audit rules as JavaScript files. The JAF API allows you to register event listeners and handle the audit context created by JAF on the file set of your JET projects. Custom audit rules can be assembled into distributable rule packs and invoked by developers on any Oracle JET app.

For more information about JAF, see *Using and Extending the Oracle JET Audit Framework*.

19

Test and Debug Oracle JET Apps

Test and debug Oracle JET web apps using a recommended set of testing and debugging tools for client-side apps.

Test Oracle JET Apps

Tests help you build complex Oracle JET apps quickly and reliably by preventing regressions and encouraging you to create apps that are composed of testable functions, modules, classes, and components.

We recommend that you write tests as early as possible in your app's development cycle. The longer that you delay testing, the more dependencies the app is likely to have, and the more difficult it will be to begin testing.

Testing Types

There are three main testing types that you should consider when testing Oracle JET apps.

1. Unit Testing
   - Unit testing checks that all inputs to a given function, class, or component are producing the expected output or response.
   - These tests are typically applied to self-contained business logic, components, classes, modules, or functions that do not involve UI rendering, network requests, or other environmental concerns.

     Note that REST service APIs should be tested independently.
   - Unit tests are aware of the implementation details and dependencies of a component and focus on isolating the tested component.
2. Component Testing
   - Component testing checks that individual components can be interacted with and behave as expected. These tests import more code than unit tests, are more complex, and require more time to execute.
   - Component tests should catch issues related to your component's properties, events, the slots that it provides, styles, classes, lifecycle hooks, and more.
   - These tests are unaware of the implementation details of a component; they mock up as little as possible in order to test the integration of your component and the entire system.

     You should not mock up child components in component tests but instead check the interactions between your component and its children with a test that interacts with the components as a user would (for example, by clicking on an element).
3. End-to-End Testing
   - End-to-end testing, which often involves setting up a database or other backend service, checks features that span multiple pages and make real network requests against a production-built JET app.

End-to-end testing is meant to test the functionality of an entire app, not just its individual components. Therefore, use unit tests and component tests when testing specific components of your Oracle JET apps.

Unit Testing

Unit testing should be the first and most comprehensive form of testing that you perform.

The purpose of unit testing is to ensure that each unit of software code is coded correctly, works as expected, and returns the expected outputs for all relevant inputs. A unit can be a function, method, module, object, or other entity in an app's source code.

Unit tests are small, efficient tests created to execute and verify the lowest-level of code and to test those individual entities in isolation. By isolating functionality, we remove external dependencies that aren't relevant to the unit being tested and increase the visibility into the source of failures.

Unit tests should interact with the component's public application programming interface (API) and pass the API as many different combinations of test data as necessary to exercise as much of the component's code paths as possible. This includes testing the component's properties, events, methods, and slots.

Unit tests that you create should adhere to the following principles:

- **Easy to write:** Unit testing should be your main testing focus; therefore, tests should typically be easy to write because many will be written. The standard testing technology stack combined with recommended development environments ensures that the tests are easily and quickly written.
- **Readable:** The intent of each test should be clearly documented, not just in comments, but the code should also allow for easy interpretation of what its purpose is. Keeping tests readable is important should someone need to debug when a failure occurs.
- **Reliable:** Tests should consistently pass when no bugs are introduced into the component code and only fail when there are true bugs or new, unimplemented behaviors. The tests should also execute reliably regardless of the order in which they're run.
- **Fast:** Tests should be able to execute quickly and report issues immediately to the developer. If a test runs slowly, it could be a sign that it is dependent upon an external system or interacting with an external system.
- **Discrete:** Tests should exercise the smallest unit of work possible, not only to ensure that all units are properly verified but also to aid in the detection of bugs when failures occur. In each unit test, individual test cases should independently target a single attribute of the code to be verified.
- **Independent:** Above all else, unit tests should be independent of one another, free of external dependencies, and be able to run consistently irrespective of the environment in which they're executed.

To shield unit tests from external changes that may affect their outcomes, unit tests focus solely on verifying code that is wholly owned by the component and avoid verifying the behaviors of anything external to that component. When external dependencies are needed, consider using mocks to stand in their place.

Component Testing

The purpose of component testing is to establish that an individual component behaves and can be interacted with according to its specifications. In addition to verifying that your component accepts the correct inputs and produces the right outputs, component tests also

include checking for issues related to your component's properties, events, slots, styles, classes, lifecycle hooks, and so on.

A component is made up of many units of code, therefore component testing is more complex and takes longer to conduct than unit testing. However, it is still very necessary; the individual units within your component may work on their own, but issues can occur when you use them together.

Component testing is a form of closed-box testing, meaning that the test evaluates the behavior of the program without considering the details of the underlying code. You should begin testing a component in its entirety immediately after development, though the tested component may in part depend on other components that have not yet been developed. Depending on the development lifecycle model, component testing can be done in isolation from other components in the system, in order to prevent external influences.

If the components that your component depends on have not yet been developed, then use dummy objects instead of the real components. These dummy objects are the stub (called function) and the controller (called function).

Depending on the depth of the test level, there are two types of component tests: small component tests and large component tests.

When component testing is done in isolation from other components, it is called "small component testing." Small component tests do not consider the component's integration with other components.

When component testing is performed without isolating the component from other components, it is called "large component testing", or "component testing" in general. These tests are done when there is a dependency on the flow of functionality of the components, and therefore we cannot isolate them.

End-to-End Testing

End-to-end testing is a method of evaluating a software product by examining its behavior from start to finish. This approach verifies that the app operates as intended and confirms that all integrated components function correctly in relation to one another. Additionally, end-to-end testing defines the system dependencies of the product to ensure optimal performance.

The primary goal of end-to-end testing is to replicate the end-user experience by simulating real-world scenarios and evaluating the system and its components for proper integration and data consistency. This approach allows for the validation of the system's performance from the perspective of the user.

End-to-end testing is a widely adopted and reliable technique that provides the following advantages.

- Comprehensive test coverage
- Assurance of app's accuracy
- Faster time to market
- Reduced costs
- Identification of bugs

Modern software systems are increasingly interconnected, with various subsystems that can cause adverse effects throughout the entire system if they fail. End-to-end testing can help prevent these risks by:

- Verifying the system's flow
- Increasing the coverage of testing areas

- Identifying issues related to subsystems

End-to-end testing is beneficial for a variety of stakeholders:

- Developers appreciate end-to-end testing as it allows them to offload testing responsibilities.
- Testers find it useful as it enables them to write tests that simulate real-world scenarios and avoid potential problems.
- Managers benefit from end-to-end testing as it allows them to understand the impact of a failing test on the end-user.

The end-to-end testing process comprises four stages:

1. **Test Planning:** Outlining key tasks, schedules, and resources required
2. **Test Design:** Creating test specifications, identifying test cases, assessing risks, analyzing usage, and scheduling tests
3. **Test Execution:** Carrying out the test cases and documenting the results
4. **Results Analysis:** Reviewing the test results, evaluating the testing process, and conducting further testing as required

There are two approaches to end-to-end testing:

- **Horizontal Testing:** This method involves testing across multiple apps and is often used in a single ERP (Enterprise Resource Planning) system.
- **Vertical Testing:** This approach involves testing in layers, where tests are conducted in a sequential, hierarchical order. This method is used to test critical components of a complex computing system and does not typically involve users or interfaces.

End-to-end testing is typically performed on finished products and systems, with each review serving as a test of the completed system. If the system does not produce the expected output or if a problem is detected, a second test will be conducted. In this case, the team will need to record and analyze the data to determine the source of the issue, fix it, and retest.

While testing your app end-to-end, consider the following metrics:

- **Test Case Preparation Status:** This metric is used to track the progress of test cases that are currently being prepared in comparison to the planned test cases.
- **Test Progress Tracking:** Regular monitoring of test progress on a weekly basis to provide updates on test completion percentage and the status of passed/failed, executed/unexecuted, and valid/invalid test cases.
- **Defects Status and Details:** Provides a weekly percentage of open and closed defects and a breakdown of defects by severity and priority.
- **Environment Availability:** Information on the number of operational hours and hours scheduled for testing each day.

Composite Component Unit Testing

Composite components comprise a few different pieces, including the view, the viewModel, and the bindings that connect them. The view is the visible part of the component and the means by which the user interacts with it, whereas the viewModel controls the behavior of the component in response to some stimulus. The view-to-viewModel bindings tie together the view and its user interactions and the viewModel behaviors that are run in response.

There are generally two approaches to unit testing components:

1. **View testing**, or DOM (Document Object Model) testing, interacts with the component's UI in much the same way that a user would: by clicking, typing, and generally interacting with the component through its visible elements. While it may invoke actions in the controller due to bindings written into the view, the purpose is to verify not the behavior but rather that the bindings themselves call the correct actions.
2. **ViewModel testing** focuses solely on the controller layer of the component. This type of testing instantiates the controller class and calls its public functions to assert behavior. Any dependencies that the controller may have on external modules are mocked out so that the test can be concerned with only the viewModel code. This testing is done independently of the view. The test is not interacting with the view to call the controller functions; rather, it calls the functions directly and performs assertions on the results from the call.

> **Note:**
>
> The view-to-viewModel bindings are the key to interactions; they define what is called for each stimulus. However, they are usually only tested during integration, not during unit testing.

We recommend unit testing composite components using direct viewModel testing, which focuses on the smallest pieces of the code that are public, namely, the API. For composite components, the public API is defined by the viewModel and surfaced on the custom web element. This makes it possible to test the viewModel methods without running the entire component through the browser and rendering a UI. ViewModel testing allows inputs to methods to be supplied by the test, and the output can be examined for correctness. This form of testing discourages direct UI interactions and instead prefers to simulate the environment in which the component would run.

Unit tests should verify the public aspects of your component, and those exposed in the API include:

- Properties
- Events
- Methods
- Slots

In addition to the API, other aspects of composites include:

- Accessibility
- Security
- Localization
- User Interaction

These aren't typically defined as part of the public API, but they should also be verified through testing.

There are circumstances where viewModel testing alone isn't sufficient to cover all aspects of the component, such as simulating user events to ensure that bindings are correctly applied. For these scenarios, view testing via the DOM or WebDriver can be used.

About the Oracle JET Testing Technology Stack

The recommended technology stack for testing Oracle JET apps includes Karma, Mocha, Chai, and Sinon.

To aid developers in testing their apps and leveraging these technologies, the `ojet add testing` command was added to the Oracle JET CLI with the release of JET 15. The command configures an app's testing environment and sets up a framework for testing JET components. Once an app is configured for testing, you can use the recommended testing technology stack to write and run your tests.

1. Karma is a test runner for JavaScript that runs on NodeJS. It runs an HTTP server to make project files available to browser instances that it launches and manages. Karma loads the necessary files into the browsers, executing source code against test code.

   > **Note:**
   >
   > Since different browsers can have different DOM implementations, testing against most of the major browsers is essential if you want to ensure that your app will behave properly for the majority of its users.

   Karma monitors files specified in its main configuration file, `karma.conf.js`, for changes, triggering test runs by sending a signal to the testing server to inform all connected browsers to run the test code again. The server collects the test results against each browser and presents them to the developer in the CLI.

   Essentially, Karma starts both the browsers and Mocha. Mocha in turn executes the tests.

2. Mocha is a JavaScript testing framework, the library against which tests are written; it allows for the declaration of test suites and cases. It can be installed globally and set as a development dependency for your project, or you can set it up to run test cases directly on the web browser.

   Mocha tests run serially, allowing for flexible and accurate reporting while mapping uncaught exceptions to the correct test cases. Mocha simplifies asynchronous testing with features that invoke the callback once the test is finished; it enables synchronous testing by omitting the callback.

3. Chai is an assertion library for NodeJS and the browser that can be paired with any JavaScript testing framework. It has several interfaces that a developer can choose from, and tests that you write in Chai resemble English sentence construction.

4. Where mocks are required, we recommend using Sinon, a library that enables the mocking of objects and functions to allow the tests to run without requiring external dependencies.

For UI automation testing, we recommend using Selenium WebDriver in conjunction with the Oracle® JavaScript Extension Toolkit (Oracle JET) WebDriver.

Configure Oracle JET Apps for Testing

The `ojet add testing` Command

Use the `ojet add testing` CLI command to add testing capability to your Oracle JET app by setting up the framework and libraries required for testing for JET components.

Run the command from a terminal window in your app's root directory. After it configures your app's testing environment, you can proceed with testing your app by using Karma as a test runner and Mocha and Chai to write your tests. You can also use Sinon to include spies, stubs, and mocks for your tests.

The configuration performed by the `ojet add testing` command also creates some essential testing directories and files within your app.

The `test-config` folder is added to your app's root directory. It contains three configuration files that are required for testing: `karma.conf.js`, `test-main.js`, and `tsconfig.json`. The Karma configuration file `karma.conf.js` is the main configuration file for your tests, `test-main.js` is the RequireJS configuration file that is loaded by Karma, and `tsconfig.json` is the TypeScript configuration file for the test files.

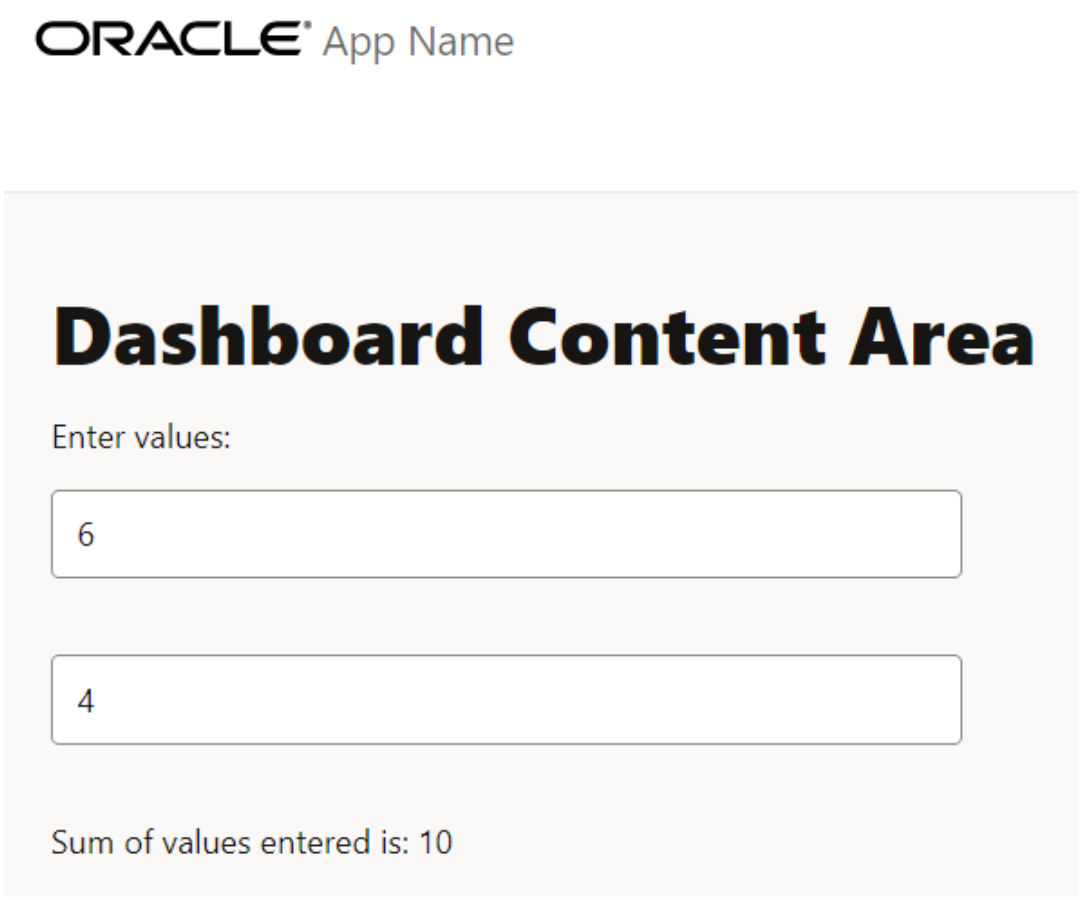
Additionally, existing components are checked for test files. The extension for files containing tests, known as "spec files," should be `.spec.ts` so that tooling recognizes them. If spec files are missing from a component, then they are injected. Spec files are located within a `__tests__` folder inside the component folder, such as `/src/ts/jet-composites/oj-calculate-value/__tests__` . This folder holds the test files you write for your component and, by default, is created with three test files containing dummy tests: `oj-calculate-value-knockout.spec.ts`, `oj-calculate-value-ui.spec.ts`, and `oj-calculate-value-viewmodel.spec.ts`.

> **Note:**
>
> If you create a new component or pack from the command line after running the `ojet add testing` command on your project, then the `__tests__` folder and files are injected by default.

Testing Demo

Here we will use the `ojet add testing` command to configure a testing environment for an Oracle JET app. The app used in this testing demo contains a calculator component that takes two user-submitted numbers, calculates their sum, and displays the result in the app's dashboard. Once the app's testing environment is set up, we will run tests on the calculator component.

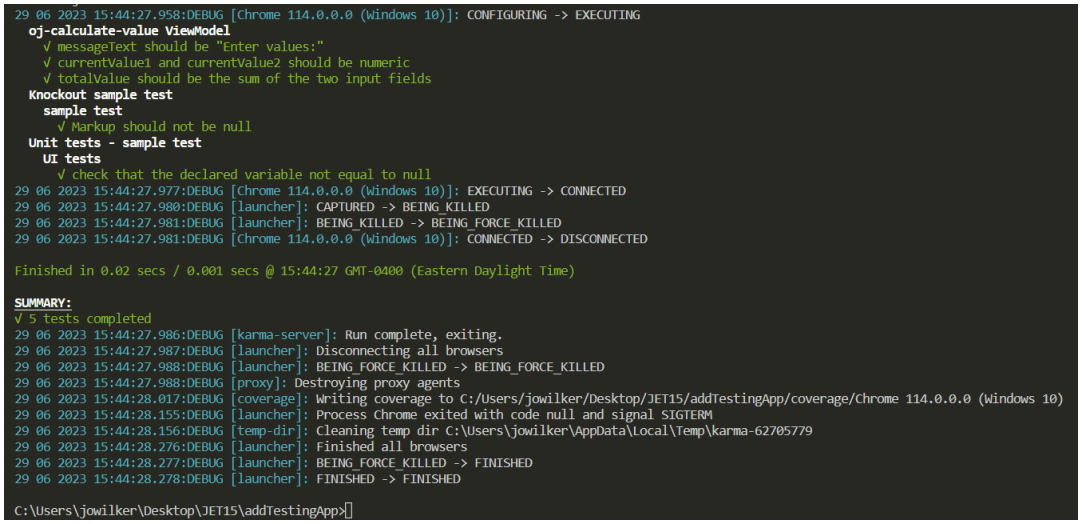
1. First, download the JET-Test-Example.zip and unzip it to your working directory. Open a terminal window in your app's root directory and run the `ojet restore` command.
2. Observe the app's directory structure. In the `/src/ts/jet-composites` directory is the `oj-calculate-value` component. The tests you will run on the calculator component, after you have configured your app for testing, are included in the app's root directory as the text file `oj-calculate-value-viewmodel-spec-ts.txt`.
   Use the `ojet serve` command to run the app and manually test that the calculator component works as expected.



> **Note:**
>
> Before running tests in your project, you must first run `ojet build` or `ojet serve`, or the tests will fail.

3. Run the `ojet add testing` command from a terminal window in your app's root directory. In your app's directory structure, you can see that the `test-config` folder was added to your app's root directory and the `__tests__` folder was added to the `/src/ts/jet-composites/oj-calculate-value` directory.
4. Rename the file `oj-calculate-value-viewmodel-spec-ts.txt` in your app's root directory to `oj-calculate-value-viewmodel.spec.ts` and replace the file with the same name in the `\src\ts\jet-composites\oj-calculate-value\__tests__` directory. The file contains three sample unit tests written in Chai for the `oj-calculate-value` component's viewModel.

5. Run the tests. Enter the script `npm run test` in the command line and observe the results in the terminal window.



6. Note the `coverage` directory that was created in the root directory of your app, after running the tests. It contains the `oj-calculate-value-viewModel.js.html` file, which you can open in your browser to view more information about the test run.

Use BusyContext API in Automated Testing

Use `BusyContext` to wait for a component or other condition to complete some action before interacting with it.

The purpose of the BusyContext API is to accommodate sequential dependencies of asynchronous operations. Typically, you use `BusyContext` in test automation when you want to wait for an animation to complete, a JET page to load, or a data fetch to complete.

> **Note:**
>
> Animations should not be disabled during testing. Tests should run on what the user sees and encounters in your app. For example, by turning off animations, you remove the possibility of finding race conditions that would only appear with the animations enabled.

Wait Scenarios

The Busy Context API will block until all the busy states resolve or a timeout period lapses. There are four primary wait scenarios:

- Components that implement animation effects
- Components that fetch data from a REST endpoint
- Pages that load bootstrap files, such as the Oracle JET libraries loaded with RequireJS
- Customer-defined scenarios that are not limited to Oracle JET, such as blocking conditions associated with app domain logic

Determining the Busy Context's Scope

The first step for waiting on a busy context is to determine the wait condition. You can scope the granularity of a busy context for the entirety of the page or limit the scope to a specific DOM element. Busy contexts have hierarchical dependencies mirroring the document's DOM structure, with the root being the page context. Depending on your particular scenario, target one of the following busy context scopes:

- Scoped for the page

  Choose the page busy context to represent the page as a whole. Automation developers commonly need to wait until the page is fully loaded before starting automation. Also, automation developers are usually interested in testing the functionality of an app that has multiple Oracle JET components rather than a single component.

  ```
  var busyContext = Context.getPageContext().getBusyContext();
  ```

- Scoped for the nearest DOM element

  Choose a busy context scoped for a DOM node when your app must wait until a specific component's operation completes. For example, you may want to wait until an `ojPopup` completes an open or close animation before initiating the next task in the app flow. Use the `data-oj-context` marker attribute to define a busy context for a DOM subtree.

  ```
  <div id="mycontext" data-oj-context>
    ...
    <!-- JET content -->
    ...
  </div>

  var node = document.querySelector("#mycontext");
  var busyContext = Context.getContext(node).getBusyContext();
  ```

Determining the Ready State

After obtaining a busy context, the next step is to inquire the busy state. `BusyContext` has two operations for inquiring the ready state: isReady() and whenReady(). The `isReady()` method immediately returns the state of the busy context. The `whenReady()` method returns a Promise that resolves when the busy states resolve or a timeout period lapses.

The following example shows how you can use `isReady()` with WebDriver.

```
public static void waitForJetPageReady(WebDriver webDriver, long
timeoutInMillis)
{
  try
  {
    final WebDriverWait wait = new WebDriverWait(webDriver, timeoutInMillis /
_THOUSAND_MILLIS);
    // Eat any WebDriverException
    // "ExpectedConditions.jsReturnsValue" will continue to be called if it
doesn't return a value.
    // /ExpectedConditions.java#L1519
    wait.ignoring(WebDriverException.class);
    wait.until(ExpectedConditions.jsReturnsValue(_PAGE_WHEN_READY_SCRIPT));
  }
```

```
  catch (TimeoutException toe)
  {
    String evalString = "return
Context.getPageContext().getBusyContext().getBusyStates().join('\\n');";
    Object busyStatesLog =
((JavascriptExecutor)webDriver).executeScript(evalString);
    String retValue = "";
    if (busyStatesLog != null){
      retValue = busyStatesLog.toString();
      Assert.fail("waitForJetPageReady failed - !
Context.getPageContext().getBusyContext().isReady() - busyStates: " +
        retValue);  }
}

// The assumption with the page when ready script is that it will continue to
execute until a value is returned or
// reached the timeout period.
//
// There are three areas of concern:
// 1) Has the app opt'd in on the whenReady wait for bootstrap?
// 2) If the app has opt'd in on the jet whenReady strategy for bootstrap
"('oj_whenReady' in window)",
// wait until jet core is loaded and have a ready state.
// 3) If not opt-ing in on jet whenReady bootstrap, make the is ready check
if jet core has loaded. If jet core is
// not loaded, we assume it is not a jet page.

// Check to determine if the page is participating in the jet whenReady
bootstrap wait period.

static private final String _BOOTSTRAP_WHEN_READY_EXP = "(('oj_whenReady' in
window) && window['oj_whenReady'])";

// Assumption is we must wait until jet core is loaded and the busy state is
ready.
static private final String _WHEN_READY_WITH_BOOTSTRAP_EXP =
"(window['oj'] && window['oj']['Context'] &&
Context.getPageContext().getBusyContext().isReady() ?" +
" 'ready' : '')";

// Assumption is the jet libraries have already been loaded. If they have
not, it's not a Jet page.
// Return jet missing in action "JetMIA" if jet core is not loaded.
static private final String _WHEN_READY_NO_BOOTSTRAP_EXP =
"(window['oj'] && window['oj']['Context'] ? " +
"(Context.getPageContext().getBusyContext().isReady() ? 'ready' : '') :
'JetMIA')";

// Complete when ready script
static private final String _PAGE_WHEN_READY_SCRIPT =
"return (" + _BOOTSTRAP_WHEN_READY_EXP + " ? " +
_WHEN_READY_WITH_BOOTSTRAP_EXP + " : " +
_WHEN_READY_NO_BOOTSTRAP_EXP + ");";
```

The following example shows how you can use `whenReady()` with QUnit.

```
// Utility function for creating a promise error handler
function getExceptionHandler(assert, done, busyContext)
{
  return function (reason)
    {
      if (reason && reason['busyStates'])
      {
        // whenReady timeout
        assert.ok(false, reason.toString());
      }
      else
      {
        // Unhandled JS Exception
        var msg = reason ? reason.toString() : "Unknown Reason";
        if (busyContext)
          msg += "\n" + busyContext;
        assert.ok(false, msg);
      }

      // invoke done callback
      if (done)
        done();
    };
};

QUnit.test("popup open", function (assert)

{
  // default whenReady timeout used when argument is not provided
  Context.setBusyContextDefaultTimeout(18000);

  var done = assert.async();
  assert.expect(1);

  var popup = document.getElementById("popup1");


  // busy context scoped for the popup
  var busyContext = Context.getContext(popup).getBusyContext();
  var errorHandler = getExceptionHandler(assert, done, busyContext);

  popup.open("#showPopup1");

  busyContext.whenReady().then(function ()
  {
    assert.ok(popup.isOpen(), "popup is open");
    popup.close();
    busyContext.whenReady().then(function ()
    {
      done();
    }).catch(errorHandler);
  }).catch(errorHandler);
});
```

Creating Wait Conditions

JET components use the busy context to communicate blocking operations. You can add busy states to any scope of the busy context to block operations such as asynchronous data fetch.

The following high-level steps describe how to add a busy context:

1. Create a Scoped Busy Context.
2. Add a busy state to the busy context. You must add a description that describes the purpose of the busy state. The busy state returns a resolve function that is called when it's time to remove the busy state.

   Busy context dependency relationships are determined at the point the first busy state is added. If the DOM node is re-parented after a busy context was added, the context will maintain dependencies with any parent DOM contexts.
3. Perform the operation that needs to be guarded with a busy state. These are usually asynchronous operations that some other app flow depends on for its completion.
4. Resolve the busy state when the operation completes.

The app is responsible for releasing the busy state. The app must manage a reference to the resolve function associated with a busy state, and it must be called to release the busy state. If the DOM node that the busy context is applied to is removed in the document before the busy state is resolved, the busy state will be orphaned and will never resolve.

Debug Oracle JET Apps

Since Oracle JET web apps are client-side HTML5 apps written in JavaScript or Typescript, you can use your favorite browser's debugging facilities.

Debug Web Apps

Use your source code editor and browser's developer tools to debug your Oracle JET app.

Developer tools for widely used browsers like Chrome, Edge, and Firefox provide a range of features that assist you in inspecting and debugging your Oracle JET app as it runs in the browser. Read more about the usage of these developer tools in the documentation for your browser.

By default, the `ojet build` and `ojet serve` commands use debug versions of the Oracle JET libraries. If you build or serve your Oracle JET app in release mode (by appending the `--release` parameter to the `ojet build` or `ojet serve` command), your app uses minified versions of the Oracle JET libraries. If you choose to debug an Oracle JET app that you built in release mode, you can use the `--optimize=none` parameter to make the minified output more readable by preserving line breaks and white space:

```
ojet build --release --optimize=none
ojet serve --release --optimize=none
```

Note that browser developer tools offer the option to "pretty print" minified source files to make them more readable, if you choose not to use the `--optimize=none` parameter.

You may also be able to install browser extensions that further assist you in debugging your app.

Finally, if you use a source code editor, such as Visual Studio Code, familiarize yourself with the debugging tools that it provides to assist you as develop and debug your Oracle JET app.

20

Package and Deploy Oracle JET Apps

If you used Oracle JET tooling to create your Oracle JET app, you can package web apps for deployment to a web or app server.

Package Web Apps

If you created your app using the tooling, use the Oracle JET command-line interface (CLI) to create a release version of your app containing your app scripts and applicable Oracle JET code in minified format.

1. From a terminal prompt in your app's root directory, enter the following command: `ojet build --release`.

   The command will take some time to complete. When it's successful, you'll see the following message: `Build finished!`.

   The command replaces the development version of the libraries and scripts in `web/js/` with minified versions where available.

2. To verify that the app still works as you expect, run `ojet serve` with the `release` option.

   The `ojet serve --release` command takes the same arguments that you used to serve your web app in development mode.

   ```
   ojet serve --release [--serverPort=server-port-number --serverOnly]
   ```

   > **Tip:**
   >
   > For a complete list of options, type `ojet help serve` at the terminal prompt.

Deploy Web Apps

Oracle JET is a collection of HTML, JavaScript, and CSS files that you can deploy to any type of web or app server. There are no unique requirements for deploying Oracle JET apps.

Deployment methods are quite varied and depend upon the type of server environment your app is designed to run in. However, you should be able to use the same method for deploying Oracle JET apps that you would for any other client interface in your specific environment.

For example, if you normally deploy apps as zip files, you can zip the `web` directory and use your normal deployment process.

Remove and Restore Non-Source Files from Your JET App

The Oracle JET CLI provides commands (clean, strip, and restore) that manage the source code of your JET app by removing extraneous files, such as the build output for the platforms your JET app supports or npm modules installed into your project.

Consider using these commands when you want to package your source code for distribution to colleagues or others who may work on the source code with you. Use of these commands may not be appropriate in all circumstances. Use of the clean and strip commands will, for example, remove the content in the `web` directory that is created when you run `ojet build` or `ojet serve`.

ojet clean

Use the `ojet clean` command to clean the build output of your JET app. Specify the `web` parameter with the `ojet clean` command (`ojet clean web`) to remove the contents of your app's root directory's `web` directory.

ojet strip

Use `ojet strip` when you want to remove all non-source files from your JET app. In addition to the build output removed by the `ojet clean` command, `ojet strip` removes additional dependencies, such as npm modules installed into your project. A typical usage scenario for the `ojet strip` command is when you want to distribute the source files of your JET app to a colleague and you want to reduce the number of files to transmit to the minimum.

The `ojet strip` command relies on the presence of the `.gitignore` file in the root directory of your app to determine what to remove. The file lists the directories that are installed by the tooling and can therefore be restored by the tooling. Only those directories and files listed will be removed when you run `ojet clean` on the app folder.

If you do not use Git and you want to run `ojet strip` to make a project easier to transmit, you can create the `.gitignore` file and add it to your app's root folder with a list of the folders and files to remove, like this:

```
#List of web app folders to remove
/node_modules
/bower_components
/themes
/web
```

As an alternative to the `.gitignore` file, you can include a `stripList` property that takes an array of glob pattern values in your app's `oraclejetconfig.json` file. When you specify the `stripList` parameter in the `oraclejetconfig.json` file, Oracle JET ignores the `.gitignore` file and its entries. Specify the list of directories and file types that you want to remove when you run `ojet strip`, as demonstrated by the following example.

```
{
. . .
  "generatorVersion": "18.1.0",
  "stripList": [
    "jet_components",
    "node_modules",
    "bower_components",
    "dist",
    "web",
    "staged-themes",
    "themes",
    "myfiles/*.txt"
  ]
}
```

ojet restore

Use the `ojet restore` command to restore the dependencies, plugins, libraries, and Web Components that the `ojet strip` command removes. After the `ojet restore` command completes, use the `ojet build` and/or `ojet serve` commands to build and serve your JET app.

The `ojet restore` command supports a number of additional parameters, such as `ojet restore --ci` that invokes the `npm ci` command instead of the default `npm install` command. This option (`ojet restore --ci`) fetches the dependencies specified in the `package-lock.json` file, and can be useful in CI/CD pipelines.

For additional help with CLI commands, enter `ojet help` at a terminal prompt.

A
Troubleshooting

Follow the same procedure for troubleshooting your Oracle JET app that you would follow for any client-side JavaScript app.

If you're having issues troubleshooting a specific Oracle JET component or toolkit feature, see Oracle JET Support. Before requesting support, be sure to check the product Release Notes.

B

Oracle JET App Migration for Release 18.1.0

If you used Oracle JET tooling to scaffold your app with Oracle JET version `5.x.0` or later, you can migrate your app manually to version `18.1.0`.

Before you migrate your app, be sure to check the Oracle JET Release Notes for any component, framework, or other change that could impact your app.

> **Important:**
>
> This process is not supported for Oracle JET releases prior to version `5.0.0`.

If you use the Alta theme in your Oracle JET app, we encourage you to switch to using the Redwood theme as soon as possible. Note the following about the Alta theme:

- The Alta theme was deprecated in release 10.0.0 and supported through the 12.x releases. In release 13.0.0 and later, Oracle JET provides "best effort" support for Alta. No bug fixes or new features will be provided for Alta-only issues. For more information about Oracle JET's release schedule, see What is the release schedule for Oracle JET?
- The content that described how to use the Alta theme in an Oracle JET app has been removed from the documentation and it has been removed from the Oracle JET Cookbook.
- In release 18.0.0, Oracle JET replaced its dependency on `node-sass` with `sass` (Dart Sass). The third-party author of `node-sass` no longer supports it. If you continue to use Alta or a custom theme based on Alta, you need to install `node-sass`. For information about how to install `node-sass`, including the Node.js version support policy of `node-sass`, see the `node-sass` page on the NPM registry website.

Prepare for Oracle JET App Migration

To migrate your Oracle JET app source from version `9.x.0` or later to the latest version 18.1.0, you must upgrade NPM packages, update theme and library reference paths, and replace the `path_mapping.json` file. Additionally, you replace references to the `oj-redwood-cssvars*.css` files that introduced CSS variables as a preview feature in JET release 9.0.0. CSS variables were a production feature in JET release `10.0.0`. Finally, you include script injector tokens in `appDirRoot/src/index.html` that will automatically be replaced with the required scripts tags at build time.

You can migrate your Oracle JET app using the `ojet migrate` command that Oracle JET introduced in JET release 18.0.0, but before you begin the migration task, perform a pre-migration audit and the other preliminary tasks listed below. Once you have completed these tasks, proceed to migrate your app using the Oracle CLI `migrate` command or manually migrate the app as described in subsequent sections of this chapter.

1. Create a backup copy of the Oracle JET app that you want to migrate.
2. Uninstall any globally-installed instances of the Oracle JET CLI

As Administrator on Windows or using `sudo` as needed on Macintosh and Linux systems, enter the following commands in a terminal window to ensure that your system has no globally-installed instances of the `ojet-cli` tooling package:

a. `[sudo] npm uninstall -g ojet-cli`

   Releases 3.2.0 and earlier of the Oracle JET CLI were published without using "`@oracle`" in the namespace name of the package.

b. `[sudo] npm uninstall -g @oracle/ojet-cli`

   Post-3.2.0 releases of the the Oracle JET CLI were published using "`@oracle`" in the namespace name of the package.

3. A maintenance or active long-term support (LTS) version of Node.js is required. Open a terminal window as an administrator and check your Node.js version.

   ```
   node -v
   ```

   If your Node.js version is earlier than the versions listed as maintenance or active LTS on the Releases page of the Nodejs.org website, download a newer LTS version. Go to the Nodejs.org website. Under **LTS Version (Recommended for Most Users),** download the installer for your system. In the Download dialog box, choose a location for the file and click **Save**. Run the downloaded installer as an administrator and follow the steps in the installation wizard to install Node.js.

4. We recommend that you audit your app with Oracle JET Audit Framework (JAF) before any migration. The built-in audit rules provided with JAF will help you to identify and fix invalid functionality, including deprecated components and APIs. Implement the audit results with some attention to detail to ensure a successful migration to JET release 18.1.0.

   As Administrator on Windows or using `sudo` as needed on Macintosh and Linux systems, enter the following command in a terminal window:

   ```
   npm install -g @oracle/oraclejet-audit
   ```

   On your app root, run the following JAF command to initialize the audit of your app.

   ```
   ojaf --init
   ```

   In the generated `AppRootDir/oraclejafconfig.json` file, set the value of the `jetVer` property to the release to which you are migrating your app. For example, `"jetVer": "18.1.0"`.

   Run the following command to audit your app and review any issues that the audit identifies.

   ```
   ojaf
   ```

   For more details about JAF, see Initialize Oracle JAF and Run an Audit in *Using and Extending the Oracle JET Audit Framework*.

5. At a terminal prompt, create a temporary app to obtain the files that you will copy to your migrating app.

The type of temporary app you create depends on the kind of app you're migrating, whether it's a TypeScript-based MVVM app, a JavaScript-based MVVM app, or a virtual DOM-based app.

- Create a JavaScript-based MVVM app

  ```
  npx @oracle/ojet-cli create tempApp --template=navdrawer
  ```

- Create a TypeScript-based MVVM app

  ```
  npx @oracle/ojet-cli create tempApp --template=navdrawer --typescript
  ```

- Create a virtual DOM app

  ```
  npx @oracle/ojet-cli create tempApp --template=basic --vdom
  ```

Migrate an App Using the Oracle JET CLI

The Oracle JET CLI includes a `migrate` command that migrates Oracle JET apps created in release `13.0.0` or later and that use the Redwood theme (or a Redwood-based custom theme) to the current release.

The `migrate` command processes your app's source and configuration files and makes changes necessary to migrate the app. This includes validating that the version the Oracle JET that you are migrating from is release `13.0.0` or later. In addition, it updates and validates the:

- `oraclejetconfig.json` file
- `path_mappings.json` file
- `main.js` file
- `index.html` file
- Hook files in the `scripts` directory
- Components from the Oracle Component Exchange

For Oracle JET apps created prior to release `13.0.0` or apps that don't use the Redwood theme, see the subsequent sections for information about how to manually migrate these apps.

To migrate your app using the Oracle JET CLI:

1. Review and complete the preparatory tasks described in Prepare for Oracle JET App Migration.
2. Make a backup copy of the source of the Oracle JET app that you want to migrate.
3. In the app's top-level directory, delete the `package-lock.json` file.
4. In a terminal window, navigate to the app's top-level directory and enter the following commands to upgrade the local NPM dependencies:

   ```
   npm uninstall @oracle/oraclejet @oracle/oraclejet-core-pack @oracle/
   oraclejet-tooling @oracle/ojet-cli
   npm install @oracle/oraclejet @oracle/oraclejet-core-pack --save
   npm install @oracle/ojet-cli --save-dev
   ```

After you run these commands, open the `package.json` file of the app and ensure it contains the following package entries and version numbers for the `18.1.0` release of Oracle JET that you migrate to.

```
{
  "name": "...",
  . . .
  "dependencies": {
    "@oracle/oraclejet": "~18.1.0",
    "@oracle/oraclejet-core-pack": "~18.1.0"
  },
  "devDependencies": {
    "@oracle/ojet-cli": "~18.1.0",
    . . .
  },
  . . .
```

5. Still in the app's top-level directory, enter the appropriate command to upgrade the local NPM dependencies that you have specified in the app's `package.json` file:
   - If your Oracle JET app includes components installed from a component exchange, use the following command to upgrade the component dependencies:

     ```
     npx @oracle/ojet-cli restore
     ```

     This command invokes `npm install` as a sub-task, so you do not need to run the `npm install` command separately to upgrade local NPM dependencies.
   - `npm install`
6. Enter the appropriate command to migrate the app to the `18.1.0` release of Oracle JET.
   - Migrate the app and set the default theme to `redwood` and the version of `sass` to `1.80.5`.

     ```
     npx @oracle/ojet-cli migrate
     ```
   - If your app uses a Redwood-based custom theme, include the `--theme` option so that the migrated app continues to use the custom theme post-migration.

     ```
     npx @oracle/ojet-cli migrate --theme=myRedwoodTheme
     ```
   - For a list of available options, enter the following command at a terminal prompt in your app's top-level directory: `npx @oracle/ojet-cli help migrate`.
7. Test the migration and verify the look and feel.

   Run `npx @oracle/ojet-cli build` and `npx @oracle/ojet-cli serve` with appropriate options to build and serve the app.

Migrate Redwood-themed Apps from Releases 9.x.0 or Later to Release 18.1.0

To migrate your Oracle JET app source from version `9.x.0` or later to the latest version `18.1.0`, you must upgrade NPM packages, update theme and library reference paths, and replace the `path_mapping.json` file. Additionally, you replace references to the `oj-redwood-cssvars*.css` files that introduced CSS variables as a preview feature in JET release 9.0.0. CSS variables were a production feature in JET release `10.0.0`. Finally, you include script injector tokens in

`appDirRoot/src/index.html` that will automatically be replaced with the required scripts tags at build time.

To migrate your app:

1. Review and complete the preparatory tasks described in Prepare for Oracle JET App Migration.
2. In the app's top-level directory, delete the `package-lock.json` file.
3. In a terminal window, navigate to the app's top-level directory and enter the following commands to upgrade the local NPM dependencies:

   ```
   npm uninstall @oracle/oraclejet @oracle/oraclejet-core-pack @oracle/
   oraclejet-tooling @oracle/ojet-cli
   npm install @oracle/oraclejet @oracle/oraclejet-core-pack --save
   npm install @oracle/ojet-cli --save-dev
   ```

   After you run these commands, open the `package.json` file in your app root folder and ensure it contains the following package entries and version numbers for the `18.1.0` release of Oracle JET.

   ```
   {
     "name": "...",
     . . .
     "dependencies": {
       "@oracle/oraclejet": "~18.1.0",
       "@oracle/oraclejet-core-pack": "~18.1.0"
     },
     "devDependencies": {
       "@oracle/ojet-cli": "~18.1.0",
        . . .,
       "typescript": "5.7.2",
       . . .
     },
     . . .
   ```

   Review the entries in the temporary app that you created (`./tempApp/package.json`) to ensure that release `18.1.0` of Oracle JET does not include new or modified entries that you should make in the app that you are migrating.

   > **Note:**
   >
   > You do not require the `"typescript": "5.7.2` entry if you are migrating an Oracle JET JavaScript-based MVVM app.

4. Open the `oraclejetconfig.json` file in your app root folder and ensure it contains the following properties and settings for `sassVer` and `defaultTheme`.

   The following example specifies `redwood` so that the app uses the Redwood theme. If your app uses a custom theme, specify the name of the custom theme. For example, `"defaultTheme": "myCustomTheme",.`

   ```
   {
     "paths": {
   ```

```
    "source": {
    "common": "src",
    "web": "src-web",
    "hybrid": "src-hybrid",
    "javascript": "js",
    "typescript": "ts",
    "styles": "css",
    "themes": "themes"
    },
    "staging": {
    "web": "web",
    "hybrid": "hybrid",
    "themes": "staged-themes"
    }
  },
  "unversioned": true,
  "defaultBrowser": "chrome",
  "sassVer": "1.80.5",
  "defaultTheme": "redwood",
   "fontUrl": ". . .",
  "typescriptLibraries": "...",
  "jsdocLibraries": ". . .",
  "webpackLibraries": "...",
  "mochaTestingLibraries": "...",
  "jestTestingLibraries": "...",
  "architecture": "mvvm"
}
```

If your app uses TypeScript, Webpack, JSDoc, or testing libraries, ensure that the `*Libraries` properties for these features are up-to-date by reviewing the entries in the temporary app that you created (`./tempApp/oraclejetconfig.json`) when you prepared for this migration.

5. Update the Oracle JET library configuration paths to reference the `18.1.0` versions of Oracle libraries by copying the `path_mappings.json` file from the temporary app.

   a. Rename your migrating app's `src/js/path_mapping.json` as `migrating-path_mapping.json`.

   b. Copy `tempApp/src/js/path_mapping.json` to your migrating app's `src/js` directory.

   c. If you added any third-party libraries to your app, open `path_mapping.json` for editing and add an entry for each library that appears in `migrating-path_mapping.json`, copying an existing entry and modifying as needed. The code sample below shows the addition you might make for a third-party library named `my-library`.

```
"libs": {

    "my-library": {
      "cdn": "3rdparty",
      "cwd": "node_modules/my-library/build/output",
      "debug": {
        "src": "my-library.debug.js",
        "path": "libs/my-library/my-library.debug.js",
        "cdnPath": ""
      },
      "release": {
```

```
            "src": "my-library.js",
            "path": "libs/my-library/my-library.js",
            "cdnPath": ""
          }
        },
...
```

6. Update the app script templates by copying from the temporary app.
   a. Copy any new script template files from the `tempApp/scripts/hooks` directory to your migrating app's `scripts/hooks` directory.
   b. Copy the `hooks.json` scripting configuration file from the `tempApp/scripts/hooks` directory to your migrating app's `scripts/hooks` directory. The updated configuration file associates any new script template files with their corresponding build system hook point and allows the Oracle JET CLI to call your scripts.
7. In your app, open the `main.js` app bootstrap file and verify that it contains the following code.
   a. Verify that the `paths` property of the `requirejs.config` definition includes the opening and closing `//injector` and `//endinjector` comments. If the comments were removed, add them to your `requirejs.config()` definition, as shown.

```
requirejs.config(
  {
   baseUrl: 'js',

    paths:
    /* DO NOT MODIFY
    ** All paths are dynamicaly generated from the path_mappings.json
file.
    ** Add any new library dependencies in path_mappings json file.
    */
    //injector:mainReleasePaths
    {
       ...no need to revise...
    }
    //endinjector
  }
);
```

   When you build or serve the app, the tooling relies on the presence of these injector comments to inject release-specific library paths in `main.js`. The updated `path_mapping.json` file (from the previous migration step) ensures that the migrated app has the correct library configuration paths for this release.
   b. Verify that any modifications you made to `app.loadModule()` in your `main.js` file appear.

   Starting in JET release 9.0.0, `app.loadModule()` is deprecated and has been removed from `main.js`, but your bootstrap code may continue to use the function, for example to change a path. Because migrating apps created prior to release 9.x.0 rely on `loadModule()`, the function should remain in your migrated `main.js` file, and your migrated `appController.js` file should contain the `loadModule()` definition.

   In new apps that you create, starting in JET release 9.0.0, the `loadModule()` observable dependency is no longer used to support the deprecated `ojRouter` for use

with the `oj-module` element. Starter app templates now use `CoreRouter` and work with `oj-module` though the `ModuleRouterAdapter`.

c. Remove the private function `_ojIsIE11`, if it appears in the `main.js` app bootstrap file. This function was included previously to detect whether an app was running in the IE11 web browser in order to load the required polyfills and the transpiled to ES5 version of Oracle JET. Starting in JET release 11.0.0, JET removed support for IE11 and no longer distributes the polyfills and other resources to run JET apps in the IE11 web browser. As a result, the `_ojIsIE11` function serves no purpose, and can be removed.

```
(function () {

   function _ojIsIE11() {
      var nAgt = navigator.userAgent;
      return nAgt.indexOf('MSIE') !== -1 || !!nAgt.match(/
Trident.*rv:11./);
   };
   var _ojNeedsES5 = _ojIsIE11();

   requirejs.config(
     {
      ...
     }
   );
}());
```

8. Enter the following commands to change to the app's top-level directory and upgrade the local NPM dependencies that you have updated in the app's `package.json` file:

```
cd appDir
npm install
```

9. In the app's `src` directory, replace any hardcoded references to a previous version.

   a. If CSS references to Redwood or references to icon fonts, similar to the following, appear in the `src/index.html` file:

```
<!-- This is the main css file for the default theme -->
 <!-- injector:theme -->
  <link rel="stylesheet" href="css/libs/oj/v9.0.0/redwood/oj-redwood-
min.css"
        type="text/css"/>
 <!-- endinjector -->

<!-- This contains icon fonts used by the starter template -->
  <link rel="stylesheet" id="uxiconFont" href="https://
static.oracle.com . . .
```

   Remove the `<link>` tag so that only the `injector:theme` and `injector:font` entries remain:

```
<!-- This is the main css file for the default theme -->
 <!-- injector:theme -->
 <!-- endinjector -->
```

```
    <!-- This contains icon fonts used by the starter template -->
     <!-- injector:font -->
     <!-- endinjector:font -->
```

b. Search for hardcoded references to a previous release version that may appear in `.html` and `.js` files and replace those with the current release version.

10. If present in the `src/index.html` file, replace the following script tags:

```
. . .
    <script type="text/javascript" src="js/libs/require/require.js"></
script>
    <script type="text/javascript" src="js/main.js"></script>
  </body>
</html>
```

With these script tag injector tokens:

```
 . . .
  <!-- This injects script tags for the main javascript files -->
    <!-- injector:scripts -->
    <!-- endinjector -->
  </body>
</html>
```

At build time, these tokens will automatically be replaced with the required scripts tags. During debug builds, the tokens will be replaced with script tags that load the `require.js` and `main.js` files. During release builds, the tokens will be replaced with script tags that load the `require.js` and `bundle.js` files. Because it is no longer used during release builds, the `main.js` file will be deleted at the end of the build. This means that if your app does not use the script tag injector tokens, it will have to include a script tag in the `appRootDir/src/index.html` file that loads `bundle.js` instead of `main.js`.

11. Test the migration and verify the look and feel.

a. Run `npx @oracle/ojet-cli build` and `npx @oracle/ojet-cli serve` with appropriate options to build and serve the app.

   For a list of available options, enter the following command at a terminal prompt in your app's top level directory: `npx @oracle/ojet-cli help`.

b. If your app uses a custom theme, you may encounter the following console error message if you do not recompile the custom theme after you update it from the previous to the current release:
   `"Your CSS file is incompatible with this version of JET (18.1.0)"`

c. To make your custom theme compatible with the new JET version, run `npx @oracle/ojet-cli add sass` to enable Sass processing.

d. Run `npx @oracle/ojet-cli build` and `npx @oracle/ojet-cli serve` with appropriate options to build and serve the app.

   If your app uses a custom theme, be sure to include the `--theme` option to recompile the CSS:

   ```
   npx @oracle/ojet-cli build [options] --theme=myTheme
   ```

To specify multiple custom themes, use:

```
npx @oracle/ojet-cli build [options] --theme=myTheme --
themes=myTheme,myTheme1,myTheme2
```

12. Optional: When you are satisfied that your app has migrated successfully, remove the temporary app and delete the renamed `migrating-path_mapping.json` and `migrating-main.js` files from your migrated app. Should you find issues, you can re-run the JAF audit tool for an audit report.

Migrate to the Redwood Theme CSS

Redwood theme is the Oracle JET standard for app look and feel and is available when you want to migrate your app to the Redwood theme.

If you have an existing app that you want to migrate from the Alta theme, you can migrate to JET release `18.1.0` and configure the app to run with the Redwood CSS included with Oracle JET. You obtained the Redwood CSS distribution when you completed the app source migration process.

Migrating your app's Alta theme to Redwood theme requires making a change at the app level. You edit the `oraclejetconfig.json` file to control whether JET Tooling builds with the Redwood or Alta CSS. With the setting configured, you can rebuild your app and all the pages will use the appropriate CSS, as specified by the stylesheet injector in your app's `index.html` file.

After you set the Redwood theme as the new default and run your app, you will find the look and feel of your app changes considerably. To adjust to the Redwood theme, you will need to make manual updates to app layout for new fonts, sizes, and patterns.

CSS variables, which are supported by the Redwood theme, were introduced in the `oj-redwood-cssvars*.css` files as a preview feature in JET release `9.0.0`. In JET release `10.0.0`, CSS variables became a production feature and are now included in the `oj-redwood*.css` files. Replace references to the `oj-redwood-cssvars*.css` files with references to the `oj-redwood*.css` files in your migrated apps. Be aware of this change if you migrated your app to use the Redwood theme and started to use CSS variables in JET releases prior to release `10.0.0`.

Before You Begin:

- Complete migration of your app source files before attempting to migrate to the Redwood theme. First migrate with the Alta theme preserved and then migrate to the Redwood theme. This way you can test your app with the Redwood theme and easily revert back to the Alta theme, if desired. See Migrate Alta-themed Apps from Releases Prior to 8.3.0 to Release 18.1.0 for details.
- If you use a custom theme, review the Theme Changes section in the release notes and update your custom theme manually.

  Be aware that Sass variables that you may have overridden in an Alta theme will need to be migrated to CSS variables in the Redwood theme. For more information about migrating a custom theme, please see About CSS Variables and Custom Themes in Oracle JET.
- Review app images and consider how you will replace images that belong to the deprecated Oracle JET framework images library with public domain images, such as those found on Oracle Apex Universal Theme and on Font Awesome. The Oracle JET framework image classes are no longer shown in JET Cookbook, starting in release 9.0.0.

To migrate to the Redwood theme CSS:

1. Configure the app to load the Redwood CSS.

   Edit the `<app_root>/oraclejetconfig.json` file and change the `defaultTheme` property to `redwood`.

   ```
   {
     "paths": {
          ...
       }
    },
    "defaultBrowser": "chrome",
    "sassVer": "1.80.5",
    "defaultTheme": "redwood",
    "fontUrl": "...",
     ...,
    "architecture": "mvvm"
   }
   ```

   This configures JET Tooling to inject `oj-redwood-min.css` into the stylesheet link in your app's `index.html` file.

2. Test the migration and verify the look and feel.

   Run `npx @oracle/ojet-cli build` and `npx @oracle/ojet-cli serve` with appropriate options to build and serve the app.

   For a list of available options, enter the following command at a terminal prompt in your app's top-level directory: `npx @oracle/ojet-cli help`.

   If your app uses a custom theme, be sure to include the `--theme` option to regenerate the CSS:

   ```
   npx @oracle/ojet-cli build [options] --theme=myTheme
   ```

   To specify multiple custom themes, use:

   ```
   npx @oracle/ojet-cli build [options] --theme=myTheme --
   themes=myTheme,myTheme1,myTheme2
   ```

Migrate Alta-themed Apps from Releases Prior to 8.3.0 to Release 18.1.0

To migrate your Oracle JET app source from version `5.x.0` through version `8.3.0` to the latest version 18.1.0, you must upgrade NPM packages, update theme and library reference paths, and replace the `path_mapping.json` file. Additionally, you must update the `oraclejetconfig.json` file to enable Alta CSS usage in JET release 18.1.0. We recommend that you migrate to the Redwood CSS theme only after successful migration of the app source has been verified. Finally, you include script injector tokens in `appDirRoot/src/index.html` that will automatically be replaced with the required scripts tags at build time.

To migrate your app:

1. Review and complete the preparatory tasks described in Prepare for Oracle JET App Migration.
2. In the app's top-level directory, delete the `package-lock.json` file.
3. In a terminal window, navigate to the app's top-level directory and enter the following commands to upgrade the local NPM dependencies:

   ```
   npm uninstall @oracle/oraclejet @oracle/oraclejet-core-pack @oracle/
   oraclejet-tooling @oracle/ojet-cli
   npm install @oracle/oraclejet @oracle/oraclejet-core-pack --save
   npm install @oracle/ojet-cli --save-dev
   ```

   After you run these commands, open the `package.json` file in your app root folder and ensure it contains the following package entries and version numbers for the `18.1.0` release of Oracle JET.

   ```
   {
     "name": "...",
     . . .
     "dependencies": {
       "@oracle/oraclejet": "~18.1.0",
       "@oracle/oraclejet-core-pack": "~18.1.0"
     },
     "devDependencies": {
       "@oracle/ojet-cli": "~18.1.0",
        . . .,
       "typescript": "5.7.2",
       . . .
     },
     . . .
   ```

   Review the entries in the temporary app that you created (`./tempApp/package.json`) to ensure that release `18.1.0` of Oracle JET does not include new or modified entries that you should make in the app that you are migrating.

   > **Note:**
   >
   > You do not require the `"typescript": "5.7.2` entry if you are migrating an Oracle JET JavaScript-based MVVM app.

4. Open the `oraclejetconfig.json` file in your app root folder and ensure it contains values for the `sassVer` and `defaultTheme` properties.

   The following example specifies `alta` so that the app uses the Alta theme.

> **Note:**
>
> Oracle JET replaced its dependency on `node-sass` with `sass` (Dart Sass) in release `18.0.0`. The third-party author of `node-sass` no longer supports it. If you continue to use Alta or a custom theme based on Alta, you need to install `node-sass`. For information about how to install `node-sass`, including the Node.js version support policy of `node-sass`, see the `node-sass` page on the NPM registry website. Specify the version of `node-sass` that you install as the value for the `sassVer` property.

```
{
  "paths": {
    "source": {
    "common": "src",
    "web": "src-web",
    "hybrid": "src-hybrid",
    "javascript": "js",
    "typescript": "ts",
    "styles": "css",
    "themes": "themes"
    },
    "staging": {
    "web": "web",
    "hybrid": "hybrid",
    "themes": "staged-themes"
    }
  },
  "unversioned": true,
  "defaultBrowser": "chrome",
  "sassVer": "...",
  "defaultTheme": "alta",
  "fontUrl": "...",
  "typescriptLibraries": "...",
  "jsdocLibraries": "...",
  "webpackLibraries": "...",
  "mochaTestingLibraries": "...",
  "jestTestingLibraries": "...",
  "architecture": "mvvm"
}
```

The `"defaultTheme": "alta"` setting enables your app to migrate and remain on the Alta theme. This property supports migrating to the Redwood theme in a later migration process. If your app uses a custom theme, specify the name of the custom theme instead. For example, `"defaultTheme": "myCustomTheme",`

If your app uses TypeScript, Webpack or testing libraries, ensure that the `*Libraries` properties for these features are up-to-date by reviewing the entries in the temporary app that you created (`./tempApp/oraclejetconfig.json`).

5. Update the Oracle JET library configuration paths to reference the `18.1.0` versions of Oracle libraries by copying the `path_mappings.json` file from the temporary app.
   a. Rename your migrating app's `src/js/path_mapping.json` as `migrating-path_mapping.json`.

b. Copy `tempApp/src/js/path_mapping.json` to your migrating app's `src/js` directory.

c. If you added any third-party libraries to your app, open `path_mapping.json` for editing and add an entry for each library that appears in `migrating-path_mapping.json`, copying an existing entry and modifying as needed. The code sample below shows the addition you might make for a third-party library named `my-library`.

```
"libs": {

    "my-library": {
      "cdn": "3rdparty",
      "cwd": "node_modules/my-library/build/output",
      "debug": {
        "src": "my-library.debug.js",
        "path": "libs/my-library/my-library.debug.js",
        "cdnPath": ""
      },
      "release": {
        "src": "my-library.js",
        "path": "libs/my-library/my-library.js",
        "cdnPath": ""
      }
    },
...
```

6. Update the app script templates by copying from the temporary app.

   a. Copy any new script template files from the `tempApp/scripts/hooks` directory to your migrating app's `scripts/hooks` directory.

   b. Copy the `hooks.json` scripting configuration file from the `tempApp/scripts/hooks` directory to your migrating app's `scripts/hooks` directory. The updated configuration file associates any new script template files with their corresponding build system hook point and allows the Oracle JET CLI to call your scripts.

7. In your app, open the `main.js` app bootstrap file and verify that it contains the following code.

   a. Verify that the `paths` property of the `requirejs.config` definition includes the opening and closing `//injector` and `//endinjector` comments. If the comments were removed, add them to your `requirejs.config()` definition, as shown.

```
requirejs.config(
  {
    baseUrl: 'js',

    paths:
    /* DO NOT MODIFY
    ** All paths are dynamicaly generated from the path_mappings.json
file.
    ** Add any new library dependencies in path_mappings json file.
    */
    //injector:mainReleasePaths
    {
       ...no need to revise...
    }
    //endinjector
```

```
    }
);
```

When you build or serve the app, the tooling relies on the presence of these injector comments to inject release-specific library paths in `main.js`. The updated `path_mapping.json` file (from the previous migration step) ensures that the migrated app has the correct library configuration paths for this release.

b. Verify that any modifications you made to `app.loadModule()` in your `main.js` file appear.

Starting in JET release 9.0.0, `app.loadModule()` is deprecated and has been removed from `main.js`, but your bootstrap code may continue to use the function, for example to change a path. Because migrating apps created prior to release 9.x.0 rely on `loadModule()`, the function should remain in your migrated `main.js` file, and your migrated `appController.js` file should contain the `loadModule()` definition.

In new apps that you create, starting in JET release 9.0.0, the `loadModule()` observable dependency is no longer used to support the deprecated `ojRouter` for use with the `oj-module` element. Starter app templates now use `CoreRouter` and work with `oj-module` though the `ModuleRouterAdapter`.

c. Remove the private function `_ojIsIE11`, if it appears in the `main.js` app bootstrap file. This function was included previously to detect whether an app was running in the IE11 web browser in order to load the required polyfills and the transpiled to ES5 version of Oracle JET. Starting in JET release 11.0.0, JET removed support for IE11 and no longer distributes the polyfills and other resources to run JET apps in the IE11 web browser. As a result, the `_ojIsIE11` function serves no purpose, and can be removed.

```
(function () {

   function _ojIsIE11() {
      var nAgt = navigator.userAgent;
      return nAgt.indexOf('MSIE') !== -1 || !!nAgt.match(/
Trident.*rv:11./);
   };
   var _ojNeedsES5 = _ojIsIE11();

   requirejs.config(
     {
      ...
     }
   );
}());
```

8. Enter the following commands to change to the app's top-level directory and upgrade the local NPM dependencies that you have updated in the app's `package.json` file:

```
cd appDir
npm install
```

9. In the app's `src` directory, replace any hardcoded references to a previous version.

a. Edit the `src/index.html` file and replace the existing CSS reference with version `v18.1.0`.

```
<link rel="stylesheet" href="css/libs/oj/v18.1.0/alta/oj-alta-min.css"
id="css" />
```

b. Search for other hardcoded references to a previous release version that may appear in `.html` and `.js` files and replace those with the current release version.

10. If present in the `src/index.html` file, replace the following script tags:

```
. . .
    <script type="text/javascript" src="js/libs/require/require.js"></
script>
    <script type="text/javascript" src="js/main.js"></script>
  </body>
</html>
```

With these script tag injector tokens:

```
 . . .
  <!-- This injects script tags for the main javascript files -->
    <!-- injector:scripts -->
    <!-- endinjector -->
  </body>
</html>
```

At build time, these tokens will automatically be replaced with the required scripts tags. During debug builds, the tokens will be replaced with script tags that load the `require.js` and `main.js` files. During release builds, the tokens will be replaced with script tags that load the `require.js` and `bundle.js` files. Because it is no longer used during release builds, the `main.js` file will be deleted at the end of the build. This means that if your app does not use the script tag injector tokens, it will have to include a script tag in the `appRootDir/src/index.html` file that loads `bundle.js` instead of `main.js`.

11. Test the migration and verify the look and feel.

a. Run `npx @oracle/ojet-cli build` and `npx @oracle/ojet-cli serve` with appropriate options to build and serve the app.

For a list of available options, enter the following command at a terminal prompt in your app's top-level directory: `npx @oracle/ojet-cli help`.

b. If your app uses a custom theme, you may encounter the following console error message if you do not recompile the custom theme after you update it from the previous to the current release:
`"Your CSS file is incompatible with this version of JET (18.1.0)"`

c. To make your custom theme compatible with the new JET version, run `npx @oracle/ojet-cli add sass` to enable Sass processing.

d. Run `npx @oracle/ojet-cli build` and `npx @oracle/ojet-cli serve` with appropriate options to build and serve the app.

For a list of available options, enter the following command at a terminal prompt in your app's top-level directory: `npx @oracle/ojet-cli help`.

If your app uses a custom theme, be sure to include the `--theme` option to recompile the CSS:

```
npx @oracle/ojet-cli build [options] --theme=myTheme
```

To specify multiple custom themes, use:

```
npx @oracle/ojet-cli build [options] --theme=myTheme --
themes=myTheme,myTheme1,myTheme2
```

12. Optional: When you are satisfied that your app has migrated successfully, remove the temporary app and delete the renamed `migrating-path_mapping.json` and `migrating-main.js` files from your migrated app. Should you find issues, you can re-run the JAF audit tool for an audit report.

After you migrate your app and ensure it runs with the Alta theme, you can follow an additional, separate process to migrate to the Redwood theme, as described in Migrate to the Redwood Theme CSS.

C

Migrate Oracle JET Legacy Components to Core Pack Components

You can use the Oracle JET Core Pack migrator to migrate Oracle JET legacy components (`oj-*`) to their equivalent Core Pack components (`oj-c-*`) without breaking your app's functionality. It also migrates Selenium WebDriver test files written for legacy components to work with the migrated Core Pack components.
The Core Pack migrator is packaged with the Oracle JET Audit Framework (Oracle JAF). To begin the Core Pack migration workflow, first download Oracle JAF and run audits on your app. Then use the information in the audit report to prepare your app's code for migration; there are specific issues that must be manually addressed in your app because they will either cause the migration to fail or be ignored by the migrator.

Once you've assessed your code and prepared it for migration, you can perform a dry-run migration that generates a `migrator.log` file where you can review the migrator's intended changes to your code.

You also have the option to create a migration configuration file that will tailor the behavior of the Core Pack migrator. You can run the migrator using its default settings, but you can use the `migrationConfig.json` file to modify its operations so that, for example, it migrates only test files or excludes certain components from migration.

When you are satisfied with your pre-migration code and the results of the dry run, you can perform the full migration.

> **Note:**
>
> Not all migration can be automated, and some components may need manual correction after migration in order to function. For a list of known migration issues and help resolving them, see Troubleshoot Core Pack Migration Issues.

After you successfully migrate your app's legacy components to Core Pack, you can migrate the components in its Selenium WebDriver test files. See Migrate WebDriver Test Files to Core Pack.

For information on installing and using Oracle JAF, see Install the Oracle JET Audit Framework. You can verify what versions of Oracle JAF and the Core Pack migrator are installed by running the commands `ojaf -v` and `ojaf -mig -v` in a terminal window.

Prepare Your Oracle JET App for Core Pack

Before you use the Oracle JET Core Pack migrator on your app's code or test files, run an Oracle JAF audit to inspect for issues that might impact migration. For example, deprecated APIs are not available in Core Pack components and will cause your migration to fail if they are present in your code.

Oracle JAF's built-in audit rules assist you in identifying and fixing invalid functionality.

Additionally, in the Oracle JET API documentation, there is a section labeled “Migration” for each legacy component that has a Core Pack equivalent to which you can migrate. This section facilitates the migration of a given legacy component to Core Pack by providing specific guidance and information on the differences between the two component versions. See, for example, the oj-button component’s migration section. Only legacy components with an equivalent Core Pack component are eligible for migration.

> **Note:**
>
> If you don’t see the API documentation for superseded components on the API page, you can select the “Show Maintenance Mode” checkbox in the sidebar to reveal it.

To prepare your app for migration, install the latest version of Oracle JAF and run an audit on your code:

1. Open a terminal window and enter the following command to install Oracle JAF globally:

   ```
   npm install -g @oracle/oraclejet-audit
   ```

2. After you install Oracle JAF, navigate in your terminal window to your app’s root directory, then initialize a default JAF configuration:

   ```
   ojaf --init
   ```

   Oracle JAF tooling scaffolds a default JAF configuration file named `oraclejafconfig.json` in your app’s root directory.

3. In your terminal window, enter the command to run an audit:

   ```
   ojaf
   ```

The audit generates a report in your terminal window. Using the report, resolve any issues flagged in your code. For example, you should fix issues raised that include the word “deprecated.” Deprecated APIs are not available in Core Pack components, and your migration will fail if your components use a deprecated API.

> **Note:**
>
> Oracle JET Core Pack components do not support the Alta theme. If your app uses Alta, then you must migrate to Redwood before you can use these components in your app. See Oracle JET App Migration to Current Release.

After auditing your code and verifying that all issues have been resolved, you can proceed with your migration.

Migrate Legacy Components Using the Core Pack Migrator

After preparing your app for migration, perform a dry-run migration to understand the migrator’s intended changes to your code and find any issues that should be addressed before or after migration. If there are aspects of the migrator’s behavior that you would like to modify, you can create a migration configuration file that you can use to customize its operations.

Follow the steps in the migration procedure below as a guide.

1. Perform a dry-run migration. The dry-run migration does not update or migrate code but generates a detailed log of all code that would get migrated.

   Open a terminal window in your app's `./src` directory and enter the command `ojaf -mig -dr`.

   A message will appear once the dry run finishes, which provides information on how long the migration took and how many files and components were updated. It also points you to the `migrator.log` file, which was created in the app's `./src` directory and contains details on the files and components that would be affected by migration.

2. Review the results of the dry run in the `migrator.log` file and decide whether to use the migrator's default settings or to modify its behavior using a migration configuration file.

   To learn more about the migration configuration file, see Customize the Core Pack Migrator's Behavior.

   If you create or modify a migration configuration file, then you can test out the changes to the migrator in a dry run by executing the following command in a terminal window, replacing the path with the location of your own configuration file: `ojaf -mig -dr -c ./migrationConfig.json`

3. When you're satisfied with your code and migration configuration, run the migration command from a terminal window in your app's `./src` directory.

   - If you have modified the configuration file, run the following command with the configuration option pointing to the location of your configuration file:

     ```
     ojaf -mig -c ./migrationConfig.json
     ```

   - To run the migrator with its default configuration, execute the following command:

     ```
     ojaf -mig
     ```

Customize the Core Pack Migrator's Behavior

You can fine-tune how the Core Pack migrator operates by using a migration configuration file. The `migrationConfig.json` file supports a range of properties that allow you to configure how the Core Pack migrator behaves. For example, you can specify that the migration only affects certain components or subsets of components, rather than your entire app.

The migration configuration file contains properties that you can use to specify:

- If your migration is for only WebDriver test files and what extension they use.
- What folders, files, and components should be skipped during migration or, conversely, should be the only targets of migration.
- Whether to force the migration of specific legacy components to Core Pack.
- Whether to exclude from migration any components with tags that contain attributes with an expression as a value.
- The base folder where the migration will execute.
- The name and location of the migrator log file.

Any name can be provided to the `migrationConfig.json` file. Therefore, a user can have different sets of migration rules defined by multiple configuration files.

To begin, create the `migrationConfig.json` file in your chosen directory, typically in the directory from which you will run the migrator, such as your app's `./src` directory.

The following sample `migrationConfig.json` file displays the available configuration options that you can include.

```
{
  "testFiles": "",
  "testFilesExtension": "",
  "excludeFolders": [],
  "excludeFiles": [],
  "excludeComponents": ["legacyTag1", "legacyTag2"],
  "includeFiles": [],
  "includeFolders": [],
  "includeComponents": ["legacyTag1", "legacyTag2"],
  "forceMigration": ["legacyTag1", "legacyTag2"],
  "excludeAttributeExpressions": {
    "legacyTag1": ["attributes"],
    "legacyTag2": ["attributes"]
  },
  "baseMigrationDir": "",
  "logFileName": "",
  "logFilePath": ""
}
```

Note that all configuration settings are optional; you can choose to include only those needed for your migration.

The following table describes the available migration configuration options and how to use them.

Table C-1 Properties in the migrationConfig.json File

Property Name	Value Type	Description
`testFiles`	Boolean	Set this value to `true` if you intend to migrate only WebDriver test files and the directory you will migrate contains only test files.
`testFilesExtension`	String	Tells the migrator the specific extension it can use to locate test files.
`excludeFolders`	Array	Contains folder names that the migrator will skip. Provide the full path for each folder name.
`excludeFiles`	Array	Contains file names that the migrator will skip. Provide the full path for each file name.
`excludeComponents`	Array	Contains legacy component tags that the migrator will skip.
`includeFiles`	Array	Contains the names of the only files that the migrator will process for migration. Provide the full path for each file name.
`includeFolders`	Array	Contains the names of the only folders that the migrator will process for migration. Provide the full path for each folder name.
`includeComponents`	Array	Contains the only legacy component tags that the migrator will process for migration.
`forceMigration`	Array	Contains legacy component tags for which the `IgnoreRuleSet` will be ignored and the tags will be migrated to Core Pack.

Table C-1 (Cont.) Properties in the migrationConfig.json File

Property Name	Value Type	Description
`excludeAttributeExpressions`	Object	Contains component-specific attributes that should not be migrated if they contain an expression as a value.
`baseMigrationDir`	String	Determines the root directory that the migrator will run from. By default, the migrator runs from the calling directory if not specified. You can use a relative or absolute path.
`logFileName`	String	Determines the name of the log file generated during migration. If not specified, the name `migrator.log` will be used by default.
`logFilePath`	String	Determines the output location for the migration log file. If not specified, the calling directory will be used by default.

Migrate WebDriver Test Files to Core Pack

The Oracle JET Core Pack migrator helps you migrate your WebDriver test files written for apps with legacy components so that they work with Core Pack components.

To migrate your test files, first follow the pre-migration steps found in Prepare Your Oracle JET App for Core Pack, which include initializing an Oracle JAF configuration in your app, running audits on your test files, and resolving any issues you find.

> **Note:**
>
> Before running audits, make sure that your `oraclejafconfig.json` file is configured so that Oracle JAF can locate your test files. Include any paths to your test files within the `files` property as string values, such as `"./src/**/*.spec.ts"`.
>
> ```
> "files": [
>   "./src/**/*.html",
>   "./src/**/*.js",
>   "./src/**/*.ts",
>   "./src/**/component.json",
>   "./src/css/**/*.css",
>   "./src/**/*.spec.ts"
> ],
> ```

Once you have audited your test files and prepared them for migration, open or create your `migrationConfig.json` in your chosen directory, typically your app's `./src` directory, and configure it as follows.

1. Set the `testFiles` property in the `migrationConfig.json` file to `true`.
2. If your test files have a unique extension, such as `.spec.ts` or `.test.ts`, then add it to the migration configuration file as a string value to the `testFilesExtension` property.

   If you do not add an extension for test files to the `testFilesExtension` property, then all files in a directory that will be migrated must be test files.

3. Review the rest of your configuration to ensure that it meets your requirements for migration. For instance, verify that the `baseMigrationDir` property, which affects the directory that the migrator runs from, is correct.

   At a minimum, your `migrationConfig.json` file should resemble the following example.

   ```
   {
     "testFiles": "true",
     "testFilesExtension": ".spec.ts"
   }
   ```

4. Open a terminal window in the same directory as your migration configuration file. Execute the dry-run migration command while pointing to the configuration file:

   ```
   ojaf -mig -dr -c ./migrationConfig.json
   ```

5. Review the results of the dry run in the `migrator.log` file. Once you have made any necessary changes and are satisfied with your code and configuration, run the actual migration:

   ```
   ojaf -mig -c ./migrationConfig.json
   ```

When the migration is complete, try running your test files against the app that uses the Core Pack components they are meant to test. You may need to make manual corrections to get your tests to pass.

For instance, a property that is called using a function in a test file may not have been properly migrated, or perhaps your test files were properly migrated and a legacy component in your app's code contains an attribute that is not supported in Core Pack.

Use the migration report in the `migrator.log` file to troubleshoot issues. To assist you, the `migrator.log` file has links to migration sections in the API documentation for each legacy component that has a Core Pack equivalent.

Troubleshoot Core Pack Migration Issues

Some components may need manual correction after migration. For guidance on migrating specific legacy components to their Core Pack equivalents, read their migration sections in the Oracle JET API docs, which are accessible at the end of the `migrator.log` file. You can also refer to the Core Pack section of the JET API docs for more information.

Here is a list of instances that require manual migration:

- Formatting changes

  Users will see formatting changes in files as a result of migrating. Run formatters post-migration in order to ensure files remain in the correct formats.

- Deprecated APIs

  Deprecated APIs are not available in Core Pack components and must be removed before migration. If a deprecated API is present in your code, then your migration will fail.

- Components that switched to a data-driven approach

  There are some legacy components that took children that are instead data-driven in their Core Pack equivalents. These will be ignored by the migrator and must be migrated manually. See the section on data-driven components in the Core Pack API documentation, as well as the component-specific migration section.

- CSS

CSS will no longer work as expected with some components. See the section on CSS in the Core Pack API documentation, as well as the component-specific migration section.

- Components where a property type has changed

  If a component's property is set, but the property type has changed, then the legacy instance can't be migrated.

  For example, on form controls, such as `oj-input-text` and `oj-input-number`, the Core Pack `converter` type and the legacy `converter` type don't match. Therefore, if the `converter` property is set, then the instance won't be migrated. When this is the case, the `migrator.log` file will have a variation of the following message: `Rule to ignore: "oj-input-text" filtered out due to attribute "converter".`

- Class attributes with an expression as a value

  Class migration will only happen if the value is a string, otherwise migration will not modify anything in the expression.

- References to strongly typed properties

  In TypeScript code, there may be cases where you want to type variables or properties based on the existing types declared by the JET components you are consuming.

  An example of this is when you define a custom component and want to define a chroming property on that component that can be passed directly down to a legacy `oj-button` component that you use as part of that component implementation. This type of dependency must be migrated as well if you are migrating the legacy component to its Core Pack equivalent

  When using the JET legacy components, you would typically use the `IntrinsicProps` of the component that is exported:

  ```
  import { ButtonIntrinsicProps} from 'ojs/ojbutton';
  ...
  const localChroming:ButtonIntrinsicProps['chroming']
  ```

  In Core Pack components, the `props` type is not re-exported from the Core Pack class because there is a more generic way to access the same information using the Preact `ComponentProps` convenience type. Instead, the code would be migrated as follows:

  ```
  import { ComponentProps } from 'preact';
  import 'oj-c/button';
  ...
  type ButtonProps = ComponentProps<'oj-c-button'>
  ...
  const localChroming:ButtonProps['chroming'] = ...
  ```

- References to strongly typed event payloads

  When coding in TypeScript, you can strongly type events emitted from JET legacy components by referencing the event map for the component:

  ```
  import 'ojs/ojbutton';
  import { ojButtonEventMap } from 'ojs/ojbutton';
  ...
  private async onChatActionClick(event: ojButtonEventMap['ojAction']){...}
  ```

With Core Pack components, rather than import the `eventMap`, you can directly reference the type from the component interface:

```
import 'oj-c/button';
import { CButtonElement } from 'oj-c/button';
...
private async onChatActionClick(event: CButtonElement.ojAction){...}
```

D

Oracle JET References

Oracle JET includes third-party libraries and tools that are referenced throughout the guide. Oracle also provides optional tools and libraries to assist with app development.

Oracle Libraries and Tools

Oracle provides optional tools and libraries to use in conjunction with Oracle JET. Use this reference to locate additional information about the Oracle products referenced throughout the guide.

Name	Description	Additional Information
Voluntary Product Accessibility Template (VPAT)	Documentation of Oracle's degree of conformance with applicable accessibility standards	Voluntary Product Accessibility Template (VPAT)
CSS Variable Theme Builder	An interactive demo app that you can use to try out custom Redwood themes by downloading the sample app.	CSS Variable Theme Builder

Third-Party Libraries and Tools

Use this reference to locate additional information about the third-party libraries and tools used by Oracle JET and referenced throughout the guide.

Name	Description	Additional Information
Third-party licenses	A list of the third party libraries that Oracle JET uses can be obtained in the THIRDPARTYLICENSE.txt file that Oracle JET includes with its distribution.	https://github.com/oracle/oraclejet/blob/master/THIRDPARTYLICENSE.txt
Chai	An assertion library for NodeJS and the browser.	https://www.chaijs.com/
CSS (Cascading Style Sheets)	Used for adding style to web apps	http://www.w3.org/Style/CSS
HTML (Hypertext Markup Language)	Core language of the World Wide Web	http://www.w3.org/TR/html5
JavaScript	Scripting language used in client-side apps	https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript
Jest	A testing framework that you can use to test Oracle JET VComponent-based web components.	https://jestjs.io/
jQuery	JavaScript library designed for HTML document traversal and manipulation, event handling, animation, and Ajax	http://jquery.com
jQuery UI	JavaScript library built on top of jQuery for UI development. Oracle JET includes the UI Core download only.	http://www.jqueryui.com
Karma	Test runner that runs the code against code browsers.	https://karma-runner.github.io/latest/index.html
Knockout	JavaScript library used for two-way data binding	http://www.knockoutjs.com

Name	Description	Additional Information
Mocha	Test framework running on Node.js and in the browser, making asynchronous.	https://mochajs.org/
Node.js	Open source, cross-platform runtime environment for developing server-side web apps, used by Oracle JET for package management.	https://nodejs.org
Preact	The JavaScript library that Oracle JET uses to manage the virtual DOM and update the live DOM of apps and web components that you develop using Oracle JET.	https://preactjs.com/
proj4js	JavaScript library designed for transforming point coordinates from one coordinate system to another, including datum transformations.	http://proj4js.org/
RequireJS	JavaScript file and module loader used for managing library references and lazy loading of resources	http://www.requirejs.org
RequireJS CSS	RequireJS CSS plugin that allows to load CSS files.	http://requirejs.org/docs/faq-advanced.html
Selenium WebDriver	A tool that drives a browser natively, as a real user would, either locally or on remote machines, to automate web app testing.	http://docs.seleniumhq.org/projects/webdriver
TypeScript	A strongly typed programming language that builds on JavaScript.	https://www.typescriptlang.org/
Visual Studio Code	Microsoft Integrated Development Environment (IDE) with available Oracle JET Core Components extension for app development.	https://marketplace.visualstudio.com/items?itemName=Oracle.oracle-jet-core

E

Properties in the oraclejetconfig.json File

The `oraclejetconfig.json` file supports a range of properties that you can configure to determine the behavior of your Oracle JET project.

> **Note:**
>
> Where `Property` is `<prop>.<subprop>` it indicates that `<subprop>` is a subproperty of `<prop>`. For example, `paths.components` means `"paths": { "components": "value" }`.

Table E-1 Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes
architecture	String	`mvvm` or `vdom`	mvvm	Type of app architecture.
components	Object	component name/version value pairs		Component name/version value pairs for components to be restored from the component exchange upon `ojet restore`. Similar format to a `package.json`. For example: `"components": { "oj-doceg-double-picker": "^2.0.0" }`
bundleName	String	simple file name with a .JS extension	bundle.js	Allows an override of the default name used for an optimized app.
bundler	String	`webpack` | `<any>`		In release 11.0.0, JET introduced bundler-only support for Webpack. If `webpack` was specified as the value for the `bundler` property, the `before_webpack` hook was used to bundle the app. Otherwise, the `before_optimize` hook managed the RequireJS-based app bundling. Webpack-based bundler applied only to the app bundling. Custom component optimization continued to use RequireJS-based bundling, and could be configured with the `before_component_optimize` hook. In release 12.0.0, JET introduced end-to-end Webpack support. With the `--webpack` argument in an `ojet create` command, Oracle JET creates an `ojet.config.js` file where you configure Webpack usage. No `bundler` property is configured in the `oraclejetconfig.json` file.
defaultBrowser	String	browser name	chrome	Sent to Apache Cordova when serving hybrid mobile apps as `--target` when the destination is `browser`.
defaultTheme	String	redwood, redwood-notag, stable	redwood	Name of theme to use as the default in the app.

Table E-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes

dependencies	Object	component name/object or version number pairs		Names of potential component or pack dependencies used to check whether certain pre-minified components should be excluded from the `ojet build --release` bundling process. For example:
				`"dependencies": { "oj-pack-comp": { "version": "2.0.0"} }`
				Or
				`"oj-comp": "2.0.0"`

enableDocGen	Boolean		true/false	When `true`, Oracle JET generates API doc for VComponent-based web components in the `.appRootDir/web/components/`*`component-name`*`/docs` directory after you build the component using the `ojet build component` command if you have previously run the `ojet add docgen` command that installs the packages and templates to generate API doc. If you have not installed the packages and templates for API doc, the `ojet build component` command fails. When `false`, Oracle JET does not generate API doc for VComponent-based web components.
exchange-url	String	URL		Component exchange instance for publishing components. For example: `https://exchange.url.com/api/0.2.0` **Note:** This setting can also be inherited (if not present) from a global value defined through `ojet configure --exchange-url=<addr> --global` which will be stored centrally (for example, `.ojet/exchange-url.json`)
fontUrl	String	URL		Link so that an Oracle JET CLI `build` command can insert icon fonts into Oracle JET templates. This property is associated with the `injector:font` token in the `appRootDir/src/index.html` file of your app.
generatorVersion	String	Oracle JET CLI version		Deprecated. Historical information about the version of JET that was first used to create the project. Not used by the CLI.
installer	String	`yarn` or `npm`	npm	If specified, an alternate installer to run instead of the default npm for `npm install` type commands.

Table E-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes
localComponents Support	boolean	true/false		Indicates whether the component exchange backend supports the `local components` extension. The value will be recorded by the CLI in `oraclejetconfig.json`. If a user wants to opt out of the local components support, they can set this value to `false` deliberately.
paths.components	String	path	jet-composites	Path where locally-created components are stored relative to a root that is dependent on the scaffolded project type: 1. Project created with `--vdom` or `--template=basic-vdom` template. Root will be `src/` . 2. Project created with `--typescript`. Root will be `src/ts/` . 3. Default project. Root will be `src/js`. **Note:** In a a project created with the vdom `basic` template or `--vdom` option, this value will be pre-set to just `components` rather than the default `jet-composites`.
paths.exchangeComponents	String	path	exchange_components	Folder where components added from the exchange are stored for new virtual DOM apps. Non-virtual DOM apps (MVVM) use the older `jet_components` when this is not set. This path must be a simple folder name which will be created in the root of the project as a peer of the `src/` folder.
paths.source.common	String	path	src	Simple folder name relative to the project root. A folder hierarchy cannot be used here. Other settings such as `paths.components` will be relative to this location.
paths.source.hybrid	String	path	src-hybrid	Simple folder name relative to the project root. A folder hierarchy cannot be used here.
paths.source.javascript	String	path	js	Simple folder name relative to the defined `src` folder. A folder hierarchy cannot be used here. Other settings such as `paths.components` may be relative to this location in the relevant project type.
paths.source.styles	String	path	css	Simple folder name relative to the defined `src` folder. A folder hierarchy cannot be used here.
paths.source.themes	String	path	themes	Simple folder name relative to the defined `src` folder. A folder hierarchy cannot be used here.

Table E-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes
paths.source.tsconfig	String	path		If specified, this subproperty enables the relocation of the `tsconfig.json` file from its default location at the app root. Simple folder name relative to the defined `src` folder. A folder hierarchy cannot be used here.
paths.source.typescript	String	path	ts	Simple folder name relative to the defined `src` folder. A folder hierarchy cannot be used here. Other settings such as `paths.components` may be relative to this location in the relevant project type.
paths.source.web	String	path	src-web	Simple folder name relative to the defined `src` folder. A folder hierarchy cannot be used here.
paths.staging.hybrid	String	path	hybrid	Path where the hybrid build products are generated.
paths.staging.themes	String	path	staged-themes	Path where themes are staged.
paths.staging.web	String	path	web	Path where the web build products are generated.
sassVer	String	semver-style version number	1.80.5	Dart Sass (`sass`) NPM package version that will be installed if sass is added
stripList	Array of strings	path strings		List of `.gitignore`-style paths to strip when `ojet strip` is executed. This bypasses the list in the `.gitignore` file.

unversioned	boolean	true/false	false	When `true`, Oracle JET generates components in a directory path without the component version number when you run `ojet build` or `ojet serve`. For example:
				`appRootDir/web/ts/jet-composites/my-cca-component-pack/my-widget-1`
				Without the `unversioned` property or with `"unversioned": false` (the default), the output path is:
				`appRootDir/web/ts/jet-composites/1.0.0/my-cca-component-pack/my-widget-1`
				The `unversioned` entry in the `oraclejetconfig.json` file takes precedence over the Oracle JET command-line argument (`--omit-component-version`). That is, if the `oraclejetconfig.json` file includes `"unversioned": false` (include component version number in the directory path) and you build your Oracle JET app using the following command, Oracle JET includes the component version in the generated directory path:
				`ojet build --omit-component-version`

watchInterval	String	Number of milliseconds	1000	Configure the interval at which the live reload feature polls the Oracle JET project for updates by configuring a value for this property. The default value is `1000` milliseconds.

F

Oracle JET CLI API for CI/CD

Oracle JET provides a public, programmatic API for the Oracle JET CLI.
This API enables execution of the following tasks in CI/CD pipelines, such as the pipelines provided by Oracle Visual Builder Studio.

> **Note:**
>
> In most circumstances calling the Oracle JET CLI as a shell task in the pipeline should provide sufficient functionality, but this API is available for advanced use cases.

1. `ojet build`
2. `ojet restore`
3. `ojet strip`
4. `ojet package`
5. `ojet publish`

You create a new instance of the API as follows:

```
const Ojet = require("@oracle/ojet-cli");
const ojet = new Ojet({ cwd: "path/to/invoke/ojet/from" });
```

> **Note:**
>
> Use `npm link @oracle/ojet-cli` from your project if you are using a globally-installed Oracle JET CLI (`npm install -g @oracle/ojet-cli`).

The following options are supported when you create a new instance of the API.

Option	Type	Description
`cwd`	string	Path from where you invoke the Oracle JET CLI.
`logs`	boolean	Controls ojet logging.

Properties

Name	Type	Description
version	string	Version of @oracle/ojet-cli

Method

The Oracle JET CLI API for CI/CD pipelines exposes one method, `execute`, that executes a CLI task and returns a promise which resolves to undefined on success and failure. The `execute` method supports the following options.

Option	Type	Description
`task`	`"build" | "restore" | "strip" | "package" | "publish"`	Name of the task to execute.
`scope`	`"app" | "component" | "pack"`	Scope of the task to execute.
`parameters`	`string[]`	Parameters of the task to execute.
`options`	`object`	Options to execute the task with.

Examples

Example	Code
`ojet build`	`try { await ojet.execute({ task: "build" }); } catch {}`
`ojet build --release`	`try { await ojet.execute({ task: "build", options: { release: true } }); } catch {}`
`ojet restore`	`try { await ojet.execute({ task: "restore" }); } catch {}`
`ojet strip`	`try { await ojet.execute({ task: "strip" }); } catch {}`

Example	Code
`ojet package component <component>`	`try { await ojet.execute({ task: "package", scope: "component" parameters: ["<component>"] options: { "pack": "<pack>" } }); } catch {}`
`ojet package pack <pack>`	`try { await ojet.execute({ task: "package", scope: "pack" parameters: ["<pack>"] }); } catch {}`
`ojet publish component <component> --username=<username> --password=<password> --secure=<true|false> --path=<path>`	`try { await ojet.execute({ task: "publish", scope: "component" parameters: ["<component>"], options: { username: "<username>", password: "<password>", secure: <true|false>, path: "<path> } }); } catch {}`

Example	Code
`ojet publish pack <pack> --username=<username> --password=<password> --secure=<true|false>`	`try { await ojet.execute({ task: "publish", scope: "pack" parameters: ["<pack>"], options: { username: "<username>", password: "<password>", secure: <true|false> } }); } catch {}`