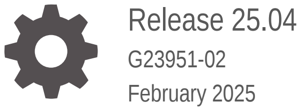
Oracle® Cloud

Oracle Visual Builder Studio Page Model Reference



Release 25.04
G23951-02
February 2025

Oracle Cloud Oracle Visual Builder Studio Page Model Reference, Release 25.04

G23951-02

Copyright © 2022, 2025, Oracle and/or its affiliates.

Primary Author: Oracle Corporation

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience vii
Documentation Accessibility vii
Diversity and Inclusion vii
Related Resources vii
Conventions viii

1 Understand the Page Model

Variables 1-1
- Object Variables 1-1
- Array Variables 1-2
- Metadata Variables 1-3
- Built-in Variables 1-3
- Types 1-6
- Built-in Extended Types 1-8
  - Service Data Provider 1-9
  - ServiceDataProviderFactory 1-66
  - Multi-Service Data Provider 1-67
  - MultiServiceDataProviderFactory 1-85
  - Array Data Provider 2 1-86
  - Array Data Provider (Legacy) 1-92
- Custom Extended Types 1-98
- InstanceFactory Types 1-102
- JET Dynamic UI Variable Types 1-107
- Default Values 1-109
- Expressions in Default Values 1-111
- Input Variables 1-111
- Persisted Variables 1-112
- rateLimit Variable Property 1-112

Constants 1-113
- Use Rules to Derive the Value for Interface Constants 1-115

JavaScript Action Chains 1-129
- JavaScript Actions 1-129

Assign Variable 1-129
Call Action Chain 1-129
Call Component 1-130
Call Function 1-130
Call REST 1-131
Call Variable 1-137
Code 1-138
Fire Data Provider Event 1-138
Fire Event 1-143
Fire Notification 1-144
For Each 1-145
Get Dirty Data Status 1-146
Get Location 1-147
If 1-148
Login 1-148
Logout 1-149
Navigate Back 1-149
Navigate To Application 1-149
Navigate To Flow 1-151
Navigate To Page 1-151
Open URL 1-152
Reset Dirty Data Status 1-153
Reset Variables 1-153
Return 1-154
Run in Parallel 1-154
Scan Barcode 1-156
Share 1-157
Switch 1-157
Try-Catch-Finally 1-158
JSON Action Chains 1-158
JSON Actions 1-158
Assign Variables Action 1-158
Call Action Chain Action 1-161
Call Component Action 1-162
Call Function Action 1-163
Call REST Action 1-163
Call Variable Method Action 1-169
EditorUrl Action 1-170
Fire Event Action 1-171
Fire Data Provider Event Action 1-172
Fire Notification Event Action 1-176
ForEach Action 1-176

Get Location Action 1-178
If Action 1-180
Login Action 1-180
Logout Action 1-181
Navigate Action 1-181
Navigate Back Action 1-187
Open URL Action 1-187
Reset Variables Action 1-189
Return Action 1-189
Run in Parallel / Fork Action 1-190
Scan Barcode Action 1-191
Share Action 1-192
Switch Action 1-193
Take Photo Action 1-194
Transform Chart Data Action (Deprecated) 1-196
Web Share Action 1-199
Action Chain Properties 1-200
Variable References in Action Chains 1-200
Action Chain Variables 1-202
Action Results 1-202
Flow 1-203
Flow Properties 1-204
Using Flows to Create Single-Page Applications 1-204
Represent the Flow State in the URL 1-205
Navigating Between Flows and Pages 1-206
Flow Lifecycle 1-206
Load Flow Resources 1-206
Use Flows Not in the Flows Folder 1-206
Shell Flow 1-207
Fragments 1-208
Define a Fragment Component 1-208
Fragment Scopes and Namespaces 1-210
Define Fragment Input Parameters 1-211
Write Back a Fragment Variable Value to the Parent Container 1-213
Deferred Rendering of a Fragment 1-214
Fragment Events 1-214
Referencing Fragments in Extensions 1-220
Extending a Fragment 1-224
Fragment Patterns 1-225
Components 1-228
HTML Source 1-228
VB Switcher Component 1-228

VB Switcher Navigation 1-229
VB Switcher Usage and Properties 1-229
VB Switcher Methods 1-230
VB Switcher Events 1-231
VB Switcher Examples 1-231
Imports 1-233
Import Custom Components 1-233
Import Custom Modules 1-233
Import Modules Using requireJS Path Mapping 1-233
Import Modules Using a Global Functions Resource Path 1-235
Import Custom CSS 1-246
Security 1-248
Security Configuration 1-248
Security Provider 1-249
User Information 1-250
Error Handling 1-250
Helper Utilities 1-251
REST Helper 1-251
Module Function Event Builder 1-254
Security Helper 1-255
Events 1-255
Declared Events 1-257
Lifecycle (Page and Flow) Events 1-258
Component Events 1-261
Fragment Events 1-263
Custom Events 1-263
System Events 1-264
Event Behavior 1-265
Variable ‘onValueChanged’ Events 1-266

2 Related Topics

Declarative RequireJS Path Mapping 2-1
Service Resolution 2-1
Service Transforms 2-6
Metadata Transforms 2-10
Translations 2-12

Preface

Oracle Visual Builder Page Model Reference describes the structure and components used in the Oracle Visual Builder page model.

Topics:

- Audience
- Documentation Accessibility
- Diversity and Inclusion
- Related Resources
- Conventions

Audience

Oracle Visual Builder Page Model Reference is intended for users who want to understand the structure and components used in visual application pages and application extensions.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `https://www.oracle.com/corporate/accessibility/`.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `https://support.oracle.com/portal/` or visit `Oracle Accessibility Learning and Support` if you are hearing impaired.

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 Public Cloud

  `http://cloud.oracle.com`
- Anatomy of Visual Applications in *Building Responsive Applications with Visual Builder Studio*
- What Is Oracle Visual Builder Studio? in *Using Visual Builder Studio*

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 Understand the Page Model

The page model consists of a JSON file. To work with the page model by hand, you should understand the structure and components of this JSON.

Applications built using Visual Builder typically have multiple flows, each containing multiple pages. Every `application` has a default `flow`, and every `flow` has a default `page`, and pages are what users see and interact with.

A Visual Builder `page` is backed by a model. This guide describes how to work with the metadata in the model's JSON file. Other containers like, `application`, `flow`, `page`, `fragment` and `layout`, also each use a similar model.

Note:

While most metadata defined in the page model applies to pages and other container types (for example, `variables` are supported in every container), the level of support for specific metadata may differ between container types. For details on the differences, refer to the specific container types.

Variables

Variables are the basic blocks of state management. Components and expressions in applications are bound to variables, and the variables and their structure must be defined to ensure that the design time and runtime work properly.

A variable must have a name and a type. Variables are in the `variables` namespace.

A variable can send an event when it changes. To add an event handler to a value change event, specify it in the 'onValueChanged' property of the variable. For details, see Variable 'onValueChanged' Events. See rateLimit Variable Property for information on setting a timeout value for the 'onValueChanged' property.

Object Variables

Variables may also be objects that contain properties.

In this case, the type of the variable should be an object that defines what properties are allowed in that object.

The following variable in JavaScript:

```
let nameOfVariable = {
  foo: "someString",
  bar: 10
}
```

could be defined like this:

```
"nameOfVariable": {
  "type": {
    "foo": "string",
    "bar": "number"
  }
}
```

Example 1-1   An Object Containing Another Object

This JavaScript object

```
let otherObject = {
  foo: {
    name: "myName"
  },
  bar: 10
}
```

can be described by the following structure:

```
"otherObject": {
  "type": {
    "foo": {
      "name": "string",
    },
    "bar": "number"
  }
}
```

Array Variables

Variables can represent arrays.

Arrays are defined the same way as objects. However, in this case, the object type is inside an array.

Arrays can have nested objects or arrays as well, and object types can also contain nested arrays.

Example 1-2   An Array Represented by a Variable

A JavaScript array

```
let myArray = [
  {
    foo: "someString",
    bar: 10
  },
  {
    foo: "someOtherString",
    bar: 11
  }
]
```

can be represented like this:

```
"nameOfVariable": {
  "type": [
    {
      "foo": "string",
      "bar": "number"
    }
```

```
  ]
}
```

Example 1-3 An Array of Strings

```
"nameOfVariable": {
  "type": "string[]"
}
```

Metadata Variables

Metadata variables are variables intended to represent metadata in specific cases. They are declared in a different "metadata" namespace (regular variables are in the "variables" namespace/declaration), and have slightly different behavior than regular variables. Metadata variables:

- do not have a "persisted" property
- do not have an "input" property (and cannot be used on a URL for navigation input, for example).
- are initialized after "variables" variables, and as such, "variables" declarations cannot have expressions dependent on their values.
- only specific types are supported; these are unique to "metadata" variables.

For a description of the "metadata" declarations used to provide metadata to JET Dynamic UI Components, see JET Dynamic UI Variable Types.

Built-in Variables

There are several built-in variables available.

currentPage

To access some of the current page's metadata, such as ID and title, there is a built-in variable named `currentPage` on the application object. The `currentPage` variable automatically updates as the current page changes during navigation. This can be used to update a navigation component with the currently selected page.

Name	Description
`$application.currentPage.id`	The path of the current page. The path describes the location of the page in the flow hierarchy.
`$application.currentPage.path`	The path of the current page for the application. The path describes the location of the page in the flow hierarchy.
`$application.currentPage.title`	The title of the current page. The title is formed by prepending all the titles of the shells in the flow hierarchy to the current page.
`$flow.currentPage`	The id of the current page for this flow.

currentFlow

If there is a `routerFlow` in the page, the `$page.currentFlow` variable can be used to retrieve the id of this flow.

Name	Description
`$page.currentFlow`	The id of the current flow.

current App UI

The current following App UI variables are available on the global object when using App UIs.

Name	Description
`$global.currentAppUi.id`	The id of the App UI
`$global.currentAppUi.urlId`	The id of the App UI as shown in the URL
`$global.currentAppUi.displayName`	The display name for the App UI
`$global.currentAppUi.description`	The description of the App UI
`$global.currentAppUi.defaultPage`	The default page of the App UI (if there is one)
`$global.currentAppUi.defaultFlow`	The default flow of the App UI (if there is one)
`$global.currentAppUi.applicationStripe`	The stripe of the custom App UI
`$global.currentAppUi.pillarTheme`	The pillar theme to use for the App UI
`$global.currentAppUi.pillarThemeMode`	The pillar theme mode to use for the App UI
`$global.currentAppUi.icon`	The icon of the custom App UI
`$global.currentAppUi.usage`	(This variable is reserved for Oracle Cloud Applications)
`$global.currentAppUi.menuDisplayName`	The name of the custom App UI
`$global.currentAppUi.extensible` (Boolean)	If this App UI can be extended

deployment

Use the `deployment` variable to distinguish between web, mobile, and progressive web applications that have been deployed from VB Studio.

Name	Description
`$application.deployment.appType`	Deprecated. The variable is always set to `web`.
`$application.deployment.pwa = 'enabled' || 'disabled'`	Used to indicate if application is configured as a progressive web applications (PWA).

For a web application, `$application.deployment.pwa` is always `'disabled'`, regardless of whether the web application is running in the Designer or deployed.

For a PWA, the value of `$application.deployment.pwa` is set to `'disabled'` when the application is in the VB Studio Designer and `$application.deployment.pwa` is set to `'enabled'` when the application is deployed.

path

The `path` variable is used to build the path to a resource, such as an image located in a folder in the application or in a flow.

Name	Description
`$application.path`	The path needed to retrieve a resource located in the application folder.
`$flow.path`	The path needed to retrieve a resource in the flow folder.
`$extension.path`	The path needed to retrieve a resource in the current extension.

user

The `user` variable is used to access information about the current user. It is based on the User Info returned by the Security Provider. It is possible to modify the set of user information by changing the implementation of the Security Provider. See Security.

Name	Description
`$application.user.userId`	The user id <string>.
`$application.user.fullName`	The user full name <string>.
`$application.user.email`	The user email <string>.
`$application.user.username`	The user name <string>.
`$application.user.roles`	The user roles (array of strings).
`$application.user.roles.`*`roleName`*	Returns true if *roleName* is a role of this user.
`$application.user.permissions`	User permissions (array of strings).
`$application.user.permissions.`*`permName`*	Returns true if *permName* is a permission of this user.
`$application.user.isAuthenticated`	Returns true if this user is authenticated.

translations

This is not a variable, but an API available for getting localized strings using `$<container>.translations.<bundlename>.key`, or `$container.<translations>.format(<bundlename>,<key>,args...)`.

This API exists for `$application`, `$flow`, and `$page`, but is only useful if you have defined translation bundles. If translation values are needed in JavaScript function modules, they must be passed as arguments to the function.

responsive

This is not directly a variable, but contains several Knockout Observables that represent JET Media Queries. The following are available, and are accessible via `$application.responsive.`*`XXX`* (for example, `$application.responsive.smUp`): `smUp, mdUp, lgUp, xlUp, smOnly, mdOnly, lgOnly`.

info

Some information from the application and page descriptor can be retrieved using the info keyword.

Name	Description
`$application.info.id`	The application id defined in `app-flow.json`
`$application.info.description`	The application description defined in `app-flow.json`
`$flow.info.id`	The flow id defined in *`flow-id`*`-flow.json`
`$flow.info.description`	The flow description defined in *`flow-id`*`-flow.json`
`$page.info.title`	The page title defined in *`page-id`*`-page.json`
`$page.info.description`	The page description defined in *`page-id`*`-page.json`
`$fragment.info.id`	This is the id set on the `oj-vb-fragment` component or the system generated stable id (if an id is not set on the component). The fragment id defined in *`fragment-id`*`-fragment.json`

Name	Description
`$fragment.info.title`	The fragment title defined in *`fragment-id`*-`fragment.json`.
`$fragment.info.description`	The fragment description defined in *`fragment-id`*-`fragment.json`.
`$layout.info.id`	The layout id.

components

This is not a variable, but contains utility methods for finding JET components on a page. These methods return an element that is a JET component. If no element is found, or if the element is not part of a JET component, these methods will return `null`.

Note:

These methods are not for finding general elements To find elements on the page, use methods such as `document.getElementById` and `document.querySelector`.

Name	Description
`$page.components.byId('myCard')` (deprecated)	Use `document.getElementById`, which returns a JET Component or `null`.
`$page.components.bySelector('#myCompId')` (deprecated)	Use `document.querySelector`, which returns a JET Component or `null`.

Types

Types define structure in much the same way as variables.

Types can be defined at the application, flow, and page level, and can be referenced by variables.

Types can be defined once at the application level in the application model. This can help you to avoid using the same structure repeatedly in different variables.

Example 1-4 Using Types in the Application Model

```
types: {
  "myType": {
    "foo": "string",
    "bar": "number"
  }
}
```

Example 1-5 Referencing Types in a Variable

To reference types in a variable, prefix the type with 'application:', for example:

```
"nameOfVariable": {
  "type": "application:myType"
}
```

Page

A page can access a type defined in itself, or the parent flow, or the application.

Definition	Result
`"nameOfVariable": { "type": "myType" }`	Uses the type named `myType` defined in the **page**.
`"nameOfVariable": { "type": "page:myType" }`	Uses the type named `myType` defined in the **page** (same as no prefix).
`"nameOfVariable": { "type": "flow:myType" }`	Uses the type named `myType` defined in the **flow** containing this page.
`"nameOfVariable": { "type": "application:myType" }`	Uses the type named `myType` defined in the **application**.

Flow

A flow can access a type defined in itself, or the application.

Definition	Result
`"nameOfVariable": { "type": "myType" }`	Uses the type named `myType` defined in the **flow**.
`"nameOfVariable": { "type": "flow:myType" }`	Uses the type named `myType` defined in the **flow** (same as no prefix).
`"nameOfVariable": { "type": "application:myType" }`	Uses the type named `myType` defined in the **application**.

Application

An application can access a type defined in itself.

Definition	Result

`"nameOfVariable": {`
`  "type": "myType"`
`}`
	Uses the type named `myType` defined in the **application**.

`"nameOfVariable": {`
`  "type": "application:myType"`
`}`
	Uses the type named `myType` defined in the **application** (same as no prefix).

Type References

An existing type can be used inside a type definition.

```
"types": {
  "region": {
    "facility": {
      "id": "string",
      "name": "string",
      "detail": "string"
    },
  "address": "flow:address",  <-- Use address defined in the parent flow
  "facilities": "facility[]"  <-- Use facility defined above
  }
}
```

Built-in Extended Types

VB provides a few built-in 'extended' types that extend from some base types provided by JET (for example, JET ArrayDataProvider) or implement an interface (JET DataProvider), and, most importantly, that use the VB Extended Type mechanism so that these types are VB aware. These VB types are generally used with a VB variable.

Authors can also use the same Extended Type mechanism to write Custom Extended Types.

VB provides these built-in extended types:

- Service Data Provider
  This built-in extended type represents a data provider that fetches data from a service endpoint and that can be bound to listView, table and other collection components that can bind to a DataProvider implementation. It encapsulates various capabilities such as filtering, sorting, pagination, and fetch and allows externalizing fetches to an actionChain.
- Multi-Service Data Provider
  JET components that bind to data providers like oj-combobox-one / oj-select-single (or the -many variants) often use different 'fetch' capabilities. Example, a oj-select-single component calls fetchFirst() (on the DataProvider implementation) to populate its options, in addition to fetchByKeys() to fetch data for selected value and fetchByOffset. This built-in

extended type is a dataProvider implementation that combines multiple ServiceDataProvider variables, each providing a unique fetch capability.

- Array Data Provider 2
  This extended builtin type is a data provider implementation where the data is available as an array. Generally with vb/ArrayDataProvider2 (similar to vb/ArrayDataProvider) all the data is set once, the data itself can fetched from a backend service (say a list of countries) as it is assumed that array once created is static, i.e. data changes infrequently or has limited/infrequent adds/updates and removes done to it.
- Array Data Provider (Legacy)
  This extended type uses the JET `oj.ArrayDataProvider` implementation, which is based on the DataProvider interface, and whose data is a plain array. The properties on the variable of type vb/ArrayDataProvider generally mirror the JET ADP's properties.

Service Data Provider

Service Data Provider represents a data provider that provides data by fetching it from a service or endpoint and that can be bound to components. It also allows externalizing fetches through an action chain.

The Service Data Provider can be used to fetch collections of data either implicitly using a configured endpoint, or externally by delegating to an action chain. Additionally, when Service Data Provider uses an Oracle Cloud Applications service, the built-in business object REST API transforms associated with the service automatically enable capabilities such as sorting, filtering, and pagination of the data. When used with endpoints not part of an Oracle Cloud Applications service, it's important for service authors to provide a custom transforms implementation that supports these capabilities. (It's worth noting that some functionality is controlled by the type of endpoint. For example, pagination properties such as `limit` and `offset` are available on a Get Many endpoint, but not a Get One endpoint.)

A variable that uses this built-in type can be bound to collection components like listView, table, combobox/select, chart, and other JET components that accept a data provider.

When the properties of the Service Data Provider variable change, it listens to the variable onValueChanged event, and notifies all its subscribers (such as components) to refresh (by raising a data provider event). Currently, UI components are the only listeners of this event.

Service Data Provider Properties

ServiceDataProvider (SDP) exposes properties that a variable of this type can use to configure. All properties are directly accessible through the variable. Expressions like `{{ $page.variables.incidentListTableSource.filterCriterion }}` can be used where expressions are supported, including component (markup) attributes.

endpoint

A string that is the REST endpoint in the format '*serviceName/endpointName*'. The endpoint is typically a GET endpoint that returns a collection, and is defined in the service model.

fetchChainId

A string that is the 'id' of the actionChain to use to fetch the results. See Implicit and Externalized Fetches for more information.

headers

An object of the names of one or more header properties, and the corresponding values. Any headers specified here are also set on the externalized REST action by the design time.

Alternatively, if a fetchChainId is not specified, headers are passed through to the internal REST calling mechanism by the ServiceDataProvider.

idAttribute

Supports composite keys, using multiple properties. It is a string or array that is the field or fields in the response data for each row, that represents the 'id', or key, field. Deprecated; use **keyAttributes** instead.

keyAttributes

A string or array, that is the field or fields in the response data for each row, that represent(s) the 'id' (or key) field. Can be:

- A property name - the key, in various contexts, will also be a string.
- An array of property names - the key will also be an array, of values.
- @value, use all properties - the key will also be an array, of values.
- @index, use the index as the key - the key will be an integer.

itemsPath

A string that is the path to the root of the actual collection in the response. Example 'result' if the response returned from the endpoint looks like `{count: 10, result: [...]}`

capabilities

An object that defines the capabilities supported by the ServiceDataProvider and the endpoint it uses. The capabilities object is defined by the JET DataProvider API.

This property serves as a hint for UI components bound to an SDP variable, to know about the capabilities the endpoint supports and use the correct fetch / sort / filter behaviors.

A variable of type `vb/ServiceDataProvider` generally defaults to a '`fetchFirst`' capability if no capability is specified. This means that the endpoint associated to the SDP is assumed to support a `fetchFirst` behavior. The same endpoint can support other 'fetch' capabilities as well.

For example, with business object REST API GETAll endpoints, the same endpoint can provide `fetchFirst` / `fetchByKeys` ('lookup') and `fetchByOffset` ('randomAccess') behaviors.

With third-party services it's important for authors to carefully consider the behaviors their endpoint supports before configuring the SDP property. For example if the third-party service endpoint provides optimal 'lookup' based `fetchByKeys`, and a 'randomAccess' based `fetchByOffset`, it's important that the author implements the appropriate transforms functions to support these capabilities. Refer to the section on Request Transformation Functions, particularly the '`paginate`' and '`fetchByKeys`' types for details.

If the same endpoint cannot be used to provide the other fetch behaviors then it might be required to use a Multi-Service Data Provider. In all other cases SDP will fallback to using the `fetchFirst` behavior to provide sub-optimal implementations of `fetchByKeys` and `fetchByOffset` behavior.

<table>
<tr><th>Key / Type</th><th>sub-key</th><th>Values</th><th>Example</th><th>Description</th></tr>
<tr><td>fetchFirst (optional) / object</td><td>impleme ntation</td><td><code>"iteration"</code></td><td></td><td>fetchFirst is not a capability supported by the JET DataProvider contract but is a new capability that SDP introduces.
Why this is needed?
SDP variables created prior to this enhancement will always assume the 'fetchFirst' capability, for backwards compatibility. New SDP variables created in DT 'may' choose to set this property to correctly reflect the capability the endpoint supports.</td></tr>
<tr><td>fetchByKeys (optional) / object</td><td>impleme ntation</td><td><code>"lookup"</code>
<code>"iteration"</code></td><td>getCustomers endpoint supports a lookup based fetchByKeys.
<pre>"customersSDP": {
  "type": "vb/ServiceDataProvider",
  "defaultValue": {
    "endpoint": "demo-data-service/getCustomers",
    "keyAttributes": "id",
    "itemsPath": "result",
    "capabilities": {
      "fetchByKeys": {
        "implementation": "lookup"
      }
    }
  }
},</pre></td><td>(see JET DataProvider API)
the "lookup" based implementation indicates the endpoint supports fetching key(s) data using a single request.
For business object REST API services, most GETAll endpoints that provide a fetchFirst capability also support querying for a key. So the default business object REST API transforms uses the fetchFirst endpoint to query for the key(s) and this property need not be set. In rare cases an entirely different endpoint might be required to fetch key data, in which case a Multi Service Data Provider might be needed.
For all other types of services, authors must ensure a lookup based 'fetchByKeys' transforms function is provided. If not an "iteration" based implementation is used.
when a "lookup" based implementation is not provided, SDP falls back to an "iteration" based implementation. This is non-performant because it uses fetchFirst / iteration to iterate over rows until the requested key(s) are located, before returning the requested key data in the form - FetchByKeysResults.</td></tr>
</table>

<table>
<tr><th>Key / Type</th><th>sub-key</th><th>Values</th><th>Example</th><th>Description</th></tr>
<tr><td></td><td>multiKey Lookup</td><td>"yes"
"no"</td><td><pre>"capabilities" : {
  "fetchByKeys": {
"implementation":
"lookup",
"multiKeyLookup":
"no"
  }
}</pre></td><td>Tells SDP whether endpoint can fetch multiple or a single key at a time. defaults to 'yes'. only available when implementation is 'lookup'. This is automatically supported for business object REST API services that use the provided business object REST API transforms.
when fetchByKeys() is called with more than one key and the capability only supports lookup by a single key, then as an optimization SDP makes multiple fetch calls against the endpoint, one per key and assembles the results</td></tr>
<tr><td>fetchByOffset (optional) / object</td><td>implementation</td><td>"iteration"
"randomAccess"</td><td>getIncidents endpoint supports a lookup based fetchByOffset.<pre>"incidentsSDP": {
  "type": "vb/
ServiceDataProvider",
  "defaultValue": {
    "endpoint":
"demo-data-service/
getIncidents",
    "keyAttributes":
"id",
    "itemsPath":
"result",
    "capabilities": {
"fetchByOffset": {
"implementation":
"randomAccess"
      }
    }
  }
}</pre></td><td>the "randomAccess" based implementation requires an endpoint that supports random access of requested page from an offset.
For business object REST API services, most GETAll endpoints support querying from a specified offset, and so the default business object REST API transforms uses this implementation automatically.
For all other types of services, authors must ensure a "randomAccess" based (paginate) transforms function is provided. If not an "iteration" based implementation is used.
when a "randomAccess" based implementation is not provided, SDP falls back to an "iteration" based implementation. This is non-performant because it uses fetchFirst / iteration to iterate over pages until the desired offset is reached.</td></tr>
</table>

Key / Type	sub-key	Values	Example	Description
filter / object	operators	array of supported operators	`"capabilities" : { "filter": { "operators": ["$eq", "$or"] } }`	a map of supported filter operators. Note: VB does not support Set types so use Array for operators This doc does not go into the details of wiring up the 'filter' and 'sort' capabilities, but when these are set the `getCapability()` method on the DataProvider will use the information defined here. For more on filter operators, see the JET documentation: oj.FilterCapability.html#operators It's a combination of attribute and compound operators. • For list of attribute operators - oj.AttributeFilterDef.html#op • For list of component operators - oj.CompoundFilterDef.html#op
	textFilter	any value	`"capabilities" : { "filter": { "textFilter": true } }`	any truthy value can be set for textFilter. By default SDP sets this to true. This value tells the consumer of the SDP that text filtering is enabled. For business object REST API endpoints, text filtering works by default with some minimal configuration, but the service author is expected to write a filter transforms function for any complex text filtering. See Filter Transform for details. For 3rd party endpoints the service author must write a filter transforms function that handles the text filter, or they can turn off the capability entirely.
sort / object			`"capabilities" : { "sort": { "attributes": "single" } }`	array of supported sort operators. For more on sort capabilities, see oj.SortCapability in the JET documentation.

responseType

The type of the response that is returned by the ServiceDataProvider. This can be an object or array. When provided it is used for two purposes:

1. To determine the fields to fetch (aka select fields) from the endpoint. A transforms author will be provided these fields via the 'select' transforms function, if they wish to edit it, but ultimately the final response is shaped by the ServiceDataProvider based on the responseType set on it (see point 2 below).
   a. When using an Oracle Cloud Application-based endpoint with ServiceDataProvider, the built-in business object REST API transforms are loaded automatically (vb/

BusinessObjectsTransform for Business Objects or business object REST API services), and the select transforms function creates a 'fields' query parameter with the desired fields, both scalar and objects (and recursively includes the object's fields, as well). This will both *include* and *expand* fields.

2. To automatically shape the response (from the fetch call) to match the responseType. Shaping a response to match the responseType usually means that missing data is 'fixed up'. This is done to ensure that binding expressions used in components work without issues.

   a. For example, an expression like `{{ $current.objectVar.propA }}` will fail if objectVar is missing.

> **Note:**
>
> Auto-shaping of response data is based on rules determined by the Visual Builder type system. If authors do not want the automatic shaping of data performed by ServiceDataProvider to introduce unexpected behavior, they must either ensure that the response data is 'complete', or they need to wrap binding expressions to guard against missing data. Response data can be made 'complete' either on the server-side, or the client can use a 'body' response transforms function to fix up incomplete data based on business rules.

Some additional things to consider:

When ServiceDataProvider externalizes data fetch

When author chooses to externalize the ServiceDataProvider fetch, the design-time often configures a chain with a RestAction, with most properties from the ServiceDataProvider on the action (RestAction and ServiceDataProvider configuration share similar properties). It also adds a 'hookHandler' property. There are certain properties that are best set on the ServiceDataProvider and not on the RestAction. Refer to the Externalized Fetch section for a list of properties that must be configured on the ServiceDataProvider variable.

It is recommended that 'responseType' always be configured on the ServiceDataProvider so that the 'select fields' are requested with the fetch call, and auto shaping of the response does not yield unexpected results (see note). The former is always determined by ServiceDataProvider.

> **Note:**
>
> For external fetches, if the RESTAction also has 'responseType' set, then it gets applied first to the response. Not only is this redundant and not performant, it's also problematic if the responseType on RestAction were to auto-shape the response to have fewer attributes than what the 'select fields' requested.

When ServiceDataProvider is used with dynamic components

Another reason for recommending that 'responseType' always be configured on the ServiceDataProvider is to address dynamic UI cases, where the responseType is not known at design-time, and 'select fields' are only provided at runtime (see note). In fact the responseType is often set to a wildcard type ('any' / 'any[]').

> **Note:**
>
> Dynamic collection components determine the list of attributes to fetch only at runtime. And this is provided via a fetchFirst() call to ServiceDataProvider (using the 'attributes' parameter) and not configured using the 'responseType' property (see JET FetchListParameters). When 'attributes' are provided, 'responseType' is ignored. There is also no default auto-shaping done when attributes are provided.

body

An object that represents the body for a fetch request, where the request is made to a POST based endpoint. Another example is where ElasticSearch based endpoints use a POST method to fetch search results, where the search criteria are set using the body.

uriParameters

An object that defines one or more properties that are parameters on the endpoint URL. For example, the FixitFast service has an endpoint to retrieve all incidents for a technician using the URL `http://.../incidents?technician={technician}`. Here 'technician' is a query parameter that will be defined under uriParameters like this:

```
"uriParameters": {
  "technician": "{{ $page.variables.appUser.name }}"
},
```

The uriParameters are used to perform a simple string replacement if the URL includes parameters that must be substituted before it's resolved. Otherwise the parameters are appended to the URL. The uriParameters are also passed to the query transform function (details below), so page authors can use the value of the above property to tweak the URI further if needed.

pagingCriteria

An object that defines the paging defaults if needed. Generally a paging component (like listView or table) will provide the data provider with size or offset or both. If the component does not provide either size or offset, the ServiceDataProvider will use the values set on this property as defaults. The pagingCriteria are then passed to the `paginate` transform function (see below). Supports the following properties.

- **size**: number of rows to fetch by default, when no size is provided by caller.
- **offset**: the offset to start the fetch from. Defaults to 0.
- **maxSize**: the default maximum number of rows to fetch when the caller (usually a component) requests that all rows be fetched. Some JET components, like oj-chart, often request all rows by setting `{ size: -1 }`. This property can be used to control the maximum number of rows to fetch, when it may not be performant to ask the service endpoint to return all rows. If this property is not set, then the `size: -1` property is passed through to the paginate transforms, and it may be necessary for transforms authors to handle -1 as the size.
- **iterationLimit**: the upper limit of the number of rows that can be fetched during iteration cycles. This is only used when `size` isn't provided and continuous iteration of rows is required. An example is when a list of values component tries to fetch labels for selected keys and the underlying multiServiceDataProvider is not configured with a 'lookup' based fetchByKeys capability. So the ServiceDataProvider reverts to using an optimized 'iteration' based implementation that is based on the fetchFirst capability. When this happens, there

could be numerous fetch requests hitting the endpoint. If the service or endpoint would like to limit this, it's important to set this value. This also gets used with the optimized fetchByOffset capability for its optimized iteration based implementation.

Page authors need to understand how the above properties are used by the ServiceDataProvider during a fetch call:

1. Generally, the page size used by a fetch can be defaulted using the pagingCriteria.size. This is only used when a component does not explicitly provide a size. The same is true for an offset.
2. When the size is provided by the caller (for example, components), this overrides the default pagingCriteria.size set on the ServiceDataProvider.

> **Note:**
>
> When components do ask for a specific number of rows, and the ServiceDataProvider returns more rows than were explicitly requested, some components can get in an indeterminate state. In such cases, to control the fetchSize, it's better to set this property on the component. Specifically, oj-list-view has a scrollPolicyOptions.fetchSize.

3. Some components do not support a fetchSize property. If this is the case, you can force the fetch to be a different value from what the component requested by writing a paginate transform function where the size can be tweaked. But you might then encounter the indeterminate state described in #2.
4. It is generally not recommended that you set endpoint-specific size and offset parameters using the uriParameters property directly (for example, the business object REST API supports 'limit' and 'offset' query parameters that are the equivalent of the pagingCriteria.size and offset). If you do, you are on your own to write a business object REST API transform that can merge/use the value set both in the uriParameters and pagingCriteria properties. And you are also likely run into the caveats explained in #3.

filterCriterion

An object representing a single attribute filter criterion with the properties `{ op, attribute, value }`, where 'op' is one of the supported JET attribute operators, and 'attribute' and 'value are the name and value of the attribute respectively. It may also represent a compound filter criterion `{op, criteria}`, where 'op' is a compound operator, and ‘criteria’ is an array of attributes or compound criterion.

Most complex filter expressions can be expressed using the JET filterCriterion structure. Sometimes you may need to externalize fetches to build your filter criteria for the REST action.

> **Note:**
>
> The business object REST API transforms shipped with Visual Builder support all attribute operators except $regex. They can transform a simple attribute filter or a compound filter that is an array of attribute filter criterion.

```
// attribute criterion
{
  "op": "$eq",
  "attribute": "empName",
```

```
  "value": "Lucy"
}

// In the business object REST API, the above criterion will become the
following query parameter:
//   "q=empName = 'Lucy'"


// compound criterion
{
  "op": "$or",
  "criteria": [
    {
      "op": "$gt",
      "attribute": "hireDate",
      "value": "2015-01-01"
    },
    {
      "op": "$le",
      "attribute": "hireDate",
      "value": "2018-01-01"
    }
  ]
}

// In the business object REST API, the above criterion will become the
following query parameter:
//   "q=hireDate > '2015-01-01' or hireDate <= '2018-01-01'"
```

Complex grouped criteria can be expressed in JSON using the filterCriterion API, but a transform function that can handle such grouped (or nested) criteria will need to be written by page authors for the business object REST API or for other external REST services, in order to build the appropriate query parameter.

```
{
  "op": "$and",
  "criteria": [
    {
      "op": "$sw",
      "attribute": "project",
      "value": "BUFF"
    },
    {
      "op": "$or",
      "criteria: [
        {
          "op": "$ge",
          "attribute": "label",
          "value": "foo"
        },
        {
          "op": "$le",
          "attribute": "label",
          "value": "bar"
        }
```

```
        ]
      }
    ]
}

// In the business object REST API, the above criterion will become the
following query parameter:
// "q=((project LIKE 'BUFF%') and ((label >=  'foo) or (label <= 'bar')))"
```

sortCriteria

An array of objects, where each object is an atomic sort expression of the form shown here. If you have more complex structures for representing sortCriteria, you can use the externalized fetch option to build sort criteria and provide it to the REST action. See Implicit and Externalized Fetches for details.

```
[{
  "attribute": "<name of the field>",
  "direction": "<'ascending' (default) or 'descending'>"
}]
```

When using multiple attributes for the sortCriteria, you specify them separated by commas:

```
[
 {
  "attribute": "col2",
  "direction": "ascending"
 },
 {
  "attribute": "col3",
  "direction": "ascending"
 }
]
```

mergeTransformOptions

This property allows a page author to set a callback to fix up or merge the final transforms options that are passed to the transform functions configured on the ServiceDataProvider. Let's say a sample endpoint, `GET /customers`, supports an 'ids' query parameter that can used to query customers by specific keys. For example:

```
/customers?ids=cus-101,cus-103
```

A component like oj-select-many might call the ServiceDataProvider requesting the customer data for specific keys by calling fetchByKeys() with these keys: `['cus-101', 'cus-103']`.

The ServiceDataProvider does not support a declarative way to automatically map these keys programmatically to the 'ids' query parameter on the URL. Therefore, it might be necessary for the page author to use this property to set a function callback that will fix up the query transforms option. For details on writing this function, see Merge Transform Options Function.

transformsContext

A context object passed to the transform functions for both request and response. For fetchFirst calls, the context will be available for all iterations using the same iterator. Authors

can manage this object as they wish. If this property is not set, an empty Object is provided by default to all transform functions. When a fetchMetadata property is provided as part of a fetch*() call, then this property is automatically set on the transformsContext Object and made available to transform functions.

- **fetchMetadata**
  For Elastic searches where the query can be arbitrarily complex, callers can send extra search metadata via the fetch call. This parameter can be used to tweak the body that is used as POST-body in the query.

  > **Note:**
  >
  > This is a Preview API and subject to change.

- **textFilterAttributes**
  See Filter Transform for details on this property.

totalSize

See getTotalSize

transforms

An object that has two properties for specifying `'request'` and `'response'` transform functions (callbacks).

Request transformation (or transform) functions are generally specified on the service (or endpoint) definition as it applies to all usages of the service. The transform functions specified here are only applicable for the current usage of the service or endpoint.

Request transform functions are primarily used to transform the URL or Request configuration before a request is sent to the endpoint.

Response functions can be used to process the response and return any additional state along with the response. Additional state is saved as internal state on the data source variable.

At design time, the page author will need to know whether the endpoint supports paging, sorting, filtering (or QBE), and the format/syntax for specifying these. Using the transform functions, the page author can tweak the Request to build a URL containing the paging, sorting, filtering params, and additional endpoint specific query params.

- `request`: An object whose properties refer to the type of the request transform functions, and the value the actual function. The following types are supported. See Request Transformation Functions for details.
  - **paginate**: a paginate function that implements code to transform the request for pagination (or iterating through record sets) specific to the endpoint.
  - **sort**: a sort function that implements code to transform the request for sorting, specific to the endpoint.
  - **filter**: a filter function. Note: Refer to the next section for details on how to use the transform functions.
  - **query**: a query function, to pre-process query parameters available through the uriParameters property.
  - **select**: a select (fields) function used to build the list of fields to fetch, if the endpoint supports it.

  - **body**: a body transform function that allows page authors to tweak the body if needed before the fetch call is made.
  - **fetchByKeys**: transforms function that allows a page author to take a key or Set of keys passed in via the options, and update the request to fetch requested keys.
- `response`: An object whose properties also refer to the type of the response transform function. See Response Transformation Functions for details.
  - **paginate**: This transform function is called immediately after the REST layer receives a response. It is called with the response so this function can process it and return an object with a group of properties set. The returned object is the primary way ServiceDataProvider obtains information about the paging state of the request:
    * `totalSize`: <optional> used to inform SDP what the totalSize of the result is.
    * `hasMore`: <generally requiredc> A boolean that indicates whether there are more records to fetch. Example in business object REST API usecases this would map to the hasMore boolean property commonly returned in the response. See explanation below for behavior of SDP when hasMore is not set.
    * `pagingState`: <optional> This can be used to store any paging state specific to the paging capability supported by the endpoint. In 1.0.0, this property can be used in the response paginate transform function, to set additional paging state. Which will then be passed 'as is' to the request paginate transform function, for the next fetch call.
  - **body**: This transform function is called immediately after the REST layer receives a response. It is a hook for authors to transform the response body, and is not guaranteed to be called in any specific order.

  The way this works is an iterating component will get the AsyncIterator from the dataProvider (like ServiceDataProvider) and keep iterating until there is no more data to fetch, or until the component viewPort is filled, or until its current scrollPosition is reached (this might be needed when a selected row is several pages down), whichever comes first. So it's extremely important for SDP to have the above information, to know when to stop iterating.

Missing 'hasMore' property in the paginate

In the event that service implementors may not have configured a paginate transform, we provide the following fallback behavior. If the first fetch request from by the SDP's AsyncIterator, has no 'hasMore' through the paginate response, SDP assumes there are no more records to fetch and iterator is marked as done. This behavior at least allows components to render some data without causing repetitive fetches. Of course this means scrolling through component will not fetch next set, if the endpoint did indeed have more rows to fetch.

Implicit and Externalized Fetches

When a ServiceDataProvider is configured with properties described in the Service Data Provider Properties section, it will, for the most part, manage fetching data and notifying components implicitly. The exception is the 'fetchChainId'.

Implicit Fetch

A typical configuration for an implicitly fetching ServiceDataProvider would look like this:

```
"incidentListDataProviderImplicit": {
  "type": "vb/ServiceDataProvider",
  "description": "configuration for implicit fetches",
```

```
  "input": "none",
  "defaultValue": {
    "endpoint": "ifixitfast-service/getIncidents",
    "headers": {},
    "keyAttributes": "id",
    "itemsPath": "result",
    "uriParameters": {
      "technician": "{{ $application.user.userId }}"
    }
  }
}
```

It is important to note that a ServiceDataProvider variable does not cache its data, just its configuration. The data is also not persisted to history, session or localStorage.

Since the data can be arbitrarily large data sets, it is recommended that page authors use other means to cache data on the client, such as the JET offline toolkit cache. This applies to externalized fetches as well.

Externalized Fetch via an Action Chain

When a 'fetchChainId' property is present, the ServiceDataProvider delegates the fetch to the action chain. A typical configuration for a ServiceDataProvider variable (supporting a fetchFirst capability) that externalizes REST will look like the code below. These are the only properties that are allowed to be configured (or that are relevant):

- **capabilities**: when this property isn't set, the 'fetchFirst' fetch capability is assumed.
- **fetchChainId**
- **idAttribute** (deprecated) or **keyAttributes**
- **itemsPath**
- **mergeTransformOptions**: this property is defined on the ServiceDataProvider variable, because merging transform options only applies when an action chain (with a REST action) is called in the context of a data provider fetch call.
- **transformsContext**: Unlike most transforms-related properties, this property can only be defined on the SDP configuration. Most transforms-related properties can be defined on the REST action (requestTransformationOptions, requestTransformFunctions, responseTransformationFunctions).
- **responseType**

```
"variables": {
  "incidentListTableSource": {
    "type": "vb/ServiceDataProvider",
    "input": "none",
    "persisted": "session",
    "defaultValue": {
      "fetchChainId": "fetchIncidentListChain",
      "keyAttributes": "id",
      "itemsPath": "result",
      "responseType": "application:incidentsResponse"
    }
  }
},
"chains": {
  "fetchIncidentListChain": {
```

```
    ...
  },
}
```

The type definition of "application:incidentsResponse" used by the 'responseType' property can be seen in this example. This structure is similar to the one returned from a REST response. Note that itemsPath is always located within the 'body' property of the response that is returned.

For example, the `app-flow.json` file for the ServiceDataProvider configuration shown above could look like this:

```
"incidentsResponse": {
  "type": {
    "status": "string",
    "headers": "object",
    "body": {
      "result": "application:incidentSummary[]"
    }
  }
},
"incidentSummary": {
  "type": {
    "id": "string",
    "problem": "string",
    "priority": "string",
    "status": "string",
    "customer": "application:customer"
  }
},
```

A sample return value from the action chain would look like this:

```
{
  "status": "200",
  "headers": {},
  "body": {
    "result": [
      {
        "id": "incident_1",
        "problem": "heater broken",
        "priority": "high",
        "status": "open",
        "customer": {}
      }
    ]
  }
}
```

Generally, users externalize fetches to ensure full control over how the request and response are processed.

For example, users can connect custom sort and filter query parameters either in the service endpoint or in the REST action. This is the preferred configuration approach. If, however, properties like sortCriteria, filterCriterion, transforms, and so on, are defined on the

ServiceDataProvider, they will be ignored, and those configured on the REST action will be used when building the request. It's important to note that sortCriteria / filterCriterion passed in by the component / caller will always get used and (attempted to be) merged with the ones configured on RestAction. See Merge Transform Options Function property.

In the example below, the action chain 'fetchIncidentListChain' defined in the fetchChainId property of the ServiceDataProvider variable above has a typical chain configuration, one of which is a RestAction.

1. The 'hookHandler' property under configuration chain variable will be automatically generated at design time and is always set to vb/RestHookHandler. SDP implements a custom hookHandler that extends from this class.
2. If the REST response returns a structure that is exactly what the ServiceDataProvider expects, this can be returned directly (as in the example below). But if the REST response is different from the expected responseType, then an action that maps the REST response to the structure defined by 'responseType' on the SDP needs to be configured.
3. The last action in the chain will always be a ReturnAction whose payload resembles the REST response whose body resembles 'responseType'. The `incidentsResponse` response variable in the chain is provided for clarity but is not used by the chain.
4. If more fields are returned than what the responseType has, SDP will attempt to auto-map the result to the response type.
5. It's important to not set the 'returnType' property when a ReturnAction is already present in the chain for SDP, because this additionally coerces the response returned to the caller.

```
"chains": {
   "fetchIncidentListChain": {
    "variables": {
      "configuration": {
        "type": {
          "hookHandler": "vb/RestHookHandler"
        },
        "description": "the configuration for the rest action",
        "input": "fromCaller",
        "required": true
      },
      "response": {
        "type": "application:incidentsResponse"
      }
    },
    "root": "fetchIncidentList",
    "actions": {
      "fetchIncidentList": {
        "module": "vb/action/builtin/restAction",
        "parameters": {
          "endpoint": "ifixitfast-service/getIncidents",
          "uriParams": {
            "technician": "{{ $application.user.userId }}"
          },
          "hookHandler": "{{ $variables.configuration.hookHandler }}",
          "requestTransformOptions": {
            "sort": "{{ $page.variables.sortExpression }}",
            "filter": "{{ $page.variables.filterAtomicExpression }}"
          },
          "requestTransformFunctions": {
```

```
              "paginate": "{{ $page.functions.paginate }}",
              "query": "{{ $page.functions.query }}",
              "filter": "{{ $page.functions.filter }}",
              "sort": "{{ $page.functions.sort }}"
            },
            "responseTransformFunctions": {
              "paginate": "{{ $page.functions.paginateResponse }}"
            }
          },
          "outcomes": {
            "success": "returnSuccessResponse",
            "failure": "returnFailureResponse"
          }
        },
        "returnSuccessResponse": {
          "module": "vb/action/builtin/returnAction",
          "parameters": {
            "outcome": "success",
            "payload": "{{ $chain.results.fetchIncidentList }}"
          }
        },
        "returnFailureResponse": {
          "module": "vb/action/builtin/returnAction",
          "parameters": {
            "outcome": "failure",
            "payload": "{{ $chain.results.fetchIncidentList }}"
          }
        }
      }
    }
  }
}
```

Merge Transform Options Function

mergeTransformOptions function signature:

A page author can use the mergeTransformOptions function callback on the ServiceDataProvider fetch to fix up the transforms options that will be passed to the transform functions, if and when needed. The function will be passed two parameters: '`configuration`' and '`transformOptions`'.

The **configuration** object will contain one set of `{ capability, context, externalContext, fetchParameters }` set as a request is servicing one fetch capability.

For the **configuration** object, this table describes configuration parameters for the fetchByKeys capability.

Sub-property	Sub-property	Value	Description
capability	-	fetchByKeys	A hint that supplies the author the fetch capability.

Sub-property	Sub-property	Value	Description
context	• idAttribute • itemsPath • uriParameters • filterCriterion • sortCriteria • pagingCriteria • responseType • capabilities – fetchByKeys – keys – ...	-	Provides a snapshot of the ServiceDataProvider variable at the time the fetchByKeys() call is made. For external chains, the state may not include all properties listed here.
externalContext			If the fetch was externalized then the context setup on the RestAction
fetchParameters	keys	-	The original parameters passed in via the fetchByKeys call.

This table describes configuration parameters for the fetchByOffset capability.

Property	Sub-property	Value	Description
capability	-	fetchByOffset	A hint telling the author the fetch capability for the current request.
context	• idAttribute • itemsPath • uriParameters • filterCriterion • sortCriteria • pagingCriteria • responseType • capabilities – fetchByKeys – keys – ...		A snapshot of the value of the ServiceDataProvider variable at the time the fetchByOffset() call was made.
externalContext			If the fetch was externalized then the context setup on the RestAction
fetchParameters	• filterCriterion • size • offset • sortCriteria	-	The original parameters passed in via the fetchByOffset call.

This table describes configuration parameters for the fetchFirst capability.

Property	Sub-property	Value	Description
capability	-	fetchFirst	A hint telling that the request is a fetchFirst capability.

Property	Sub-property	Value	Description

value	• idAttribute
	• itemsPath
	• uriParameters
	• filterCriterion
	• sortCriteria
	• pagingCriteria
	• responseType
	• capabilities
	  – fetchByKeys
	  – keys
	  – ...
		-	A snapshot of the value of the ServiceDataProvider variable at the time the fetchFirst() call was made.

externalContext			If the fetch was externalized then the context setup on the RestAction
fetchParameters	• filterCriterion • size • sortCriteria	-	-

This table describes the properties for the transformOptions parameter.

Property	Description
• query • filter • paginate • sort • select	These are the properties when the ServiceDataProvider is configured for implicit fetch. When the ServiceDataProvider is configured to use an external fetch chain, the options configured on the RestAction 'requestTransformOptions' property will be made available here.

A sample endpoint, `GET /customers`, supports an 'ids' query parameter that can used to query customers by specific keys. For example: `customers?ids=cus-101,cus-103`.

For this to work, there is currently no easy way at design time to map the keys provided by the component programmatically to the 'ids' query parameter on the URL. It might be necessary for page authors to use this property to wire up a function that will merge the transforms option.

This should be configured as follows:

1. Configuring 'mergeTransformOptions' property
   - The ServiceDataProvider variable below defines a fetchByKeys capability.
   - The 'mergeTransformOptions' property is configured to point to a page function.

```
"customerSingleSDP_External": {
  "type": "vb/ServiceDataProvider",
  "defaultValue": {
    "endpoint": "demo-data-service/getCustomers",
    "keyAttributes": "id",
    "itemsPath": "result",
    "capabilities": {
      "fetchByKeys": {
        "implementation": "lookup"
      }
```

```
    }

    "mergeTransformOptions":
      "{{ $page.functions.processOptionsForGetCustomers }}"
  }
}
```

2. Implementing the function
   - The page author uses the function to fix up the 'query' transform options that will be passed to the query transform function.
   - The page function `"{{ $page.functions.processOptionsForGetCustomers}}"` will look like the following:

```
/**
 * fix up the query transform options.
 * When the fetchByKeys capability is set, the 'keys' provided via the
fetch call
 * can be be looked up via the configuration.fetchParameters. This can be
 * set/merged onto the 'query' transform options (1). This allows the
transform
 * function to then use the keys to build the final 'ids=' query param on
the url.
 * See queryCustomersByIds method.
 *
 * Note: (1) this is needed because there is no way through DT
configuration
 * to define a mapping of 'keys' that are provided via a fetch call, to
the 'ids'
 * query parameter.
 *
 * @param configuration a map of 3 key values. The keys are
 * - fetchParameters: parameters passed to a fetch call
 * - capability: 'fetchByKeys' | 'fetchFirst' | 'fetchByOffset'
 * - context: the context of the SDP when the fetch was initiated.
 *
 * @param transformOptions a map of key values, where the keys are the
names of
 *  the transform functions.
 * @returns {*}
 */
PageModule.prototype.processOptionsForGetCustomers =
    function (configuration, transformOptions) {
  var c = configuration;
  var to = transformOptions;
  var fbkCap = !!(c && c.capability === 'fetchByKeys');
  var keysToFetch = fbkCap ? (c && c.fetchParameters &&
c.fetchParameters.keys) : null;

  if (fbkCap && keysToFetch && keysToFetch.length > 0) {
    // join keys
    var keysToFetchStr = keysToFetch.join(',');
    to = to || {};
    to.query = to.query || {};
    // ignore ids set on the query options and instead use ones passed in
by
```

```
      //   fetchByKeys call
      to.query.ids = keysToFetchStr;
    }

    return to;
  };
```

3. • A query transform function is not needed in the above example because the query parameters are automatically appended to the final request URL if no additional transformation of the query options to the query parameter is needed.
   • A query transform function might be needed in more complex use cases.

Request Transformation Functions

A request transformation (or transform) function is generally specified on the service endpoint. It can also be specified on the ServiceDataProvider variable, which overrides the endpoint one.

A request transform function is called right before a request is made to the server/endpoint. It provides a chance for page authors to transform the options (paginate, filter, sort, and so on) and build the final (request) configuration. The ServiceDataProvider supports a predefined list of request transform function types, described in this section. Note that there are no guarantees of the order in which transform functions are called, except that `vbPrepare` is called first and `body` transform is called last.

Each request transformation function has the following signature (except in the case of `vbPrepare` and `fetchByKeys` transform):

```
function (configuration, options, transformsContext) {

    // process the options and update configuration object
    return configuration;
}
```

The parameters to the function are:

- **configuration**: An object that has the following properties:
  – **endpointDefinition**: The metadata pertaining to the endpoint.
  – **fetchConfiguration**: The configuration pertaining to this fetch call. If fetch was initiated by ServiceDataProvider, this includes the following properties:
    * **capability**: The fetch capability, like fetchByKeys, fetchFirst, fetchByOffset.
    * **context**: A snapshot of the ServiceDataProvider variable state at the time the fetch call was made.
    * **externalContext**: If the fetch was externalized to a chain, then the context setup on the RestAction in that chain.
    * **fetchParameters**: A snapshot of the original fetch parameters provided by the initiator of the fetch (such as a component). The parameters passed to the fetch call are defined by the JET Data Provider fetch API.
    * **transformsOptions**: The full set of transforms options that are passed to each transform function. These are computed using the parameters configured on the Service Data Provider, the RestAction (if applicable), and the input parameters provided by initiator (such as the component).

  - **initConfig**: Map of another configuration passed into the request. The 'initConfig' exactly matches the 'init' parameter of the request.
  - **parameters**: Path and query parameters. These are not writable.
  - **url**: Full URL of the request.
- **options**: An object that is relevant to the type of transformation function. For a filter function, for example, this would be the filterCriterion.
- **transformsContext**: A context object, set by the author (ServiceDataProvider, RestHelper, Call Rest action), that is then passed to every transform function to store or retrieve any contextual information for the current request lifecycle.

If transformations are needed for a specific data provider instance, these functions can be defined on the ServiceDataProvider variable under the 'transforms' property. For externalized fetch cases, the RestAction properties can be used for configuring transformations.

vbPrepare Transform

In order to fetch the data required by the application, clients are expected to use the VB *RestHelper* directly or, for example, via a *RestAction* or *ServiceDataProvider*. Regardless of the invocation mechanism, there are two main pieces of information that are usually provided for the request to happen: the identifier of the endpoint and, if relevant, the values of the endpoint "parameters" (such as server variables, path parameters, query parameters, and header parameters).

The *vbPrepare* request transform provides a hook that clients can use to programmatically modify the parameters, which can effectively change the URL of the request issued by Visual Builder.

Signature

The vbPrepare transform can be declared as follows:

```
Request.prototype.vbPrepare = function(configuration, options,
transformsContext) {
  // Clients can manipulate the parameters of the fetch via the
'options.parameters' object.
}
```

This transform is invoked before any other transform. Its arguments are also slightly different from the other transforms:

- The `configuration` parameter provides the information about the endpoint being fetched, including the endpoint identifier, the OpenAPI path for the endpoint, and the server details (including URL templates and server variables).
- The `options` parameter has a property `parameters` that exposes the object holding the parameters passed to the RestHelper. The value of `parameters` is a "live" object: in other words, changing the properties of `options.parameters` actually modifies the values used by the RestHelper.
- The `transformsContext` is an object that is set at the RestHelper, which is then passed to all transforms.

vbPrepare Request Transform Examples

The examples below illustrate the arguments passed to the vbPrepare transform, as well as the effect its code has on the fetch performed by the RestHelper.

The 'store' service

The examples below use a service `store`, defined as follows:

- The service is located on an extension *extA*, and is exposed to extensions that depend on *extA*.
- The server of the service refers to the backend `storeapi`, and has a server variable `storeId`:

```
"servers": [
  {
    "url": "vb-catalog://backends/extA:storeapi/{storeId}",
    "variables": {
      "storeId": {
        "default": "001"
      }
    }
  }
],
```

- The backend `storeapi` is defined in the *catalog.json* artifact of *extA*, and also has a server variable:

```
"backends" {
  "storeapi": {
    "extensionAccess": true,
    "transforms": {
      "path": "./storeapi.js"
    },
    "servers": [{
      "url": "https://www.mystore.com/{version}",
      "variables": {
        "version": {
          "default": "1.0"
        }
      }
    }]
  }
}
```

- The service has an operation listProduct with both a "path" and a "query" parameter:

```
"/products/{productId}": {
  "get": {
    "operationId": "listProduct",
    "description": "List a product",
    "parameters": [
      {
        "name": "productId",
        "in": "path",
        "description": "The ID of product.",
        "required": true,
        "schema": {
          "type": "string"
        }
      },
      {
```

```
            "name": "manufactureModel",
            "in": "query",
            "description": "Whether or not to use the manufacture's model.",
            "required": false,
            "schema": {
              "type": "boolean",
              "default": false
            }
          }
        ],
        "responses": {
          ...
        }
      }
    }
```

Transform Script

As indicated above, the transform script *storeapi.js* is provided by the `storeapi` backend, so that's the artifact that must contain the vbPrepare transform method.

If the service `store` itself had a transform script, the method for vbPrepare should be declared there.

Example 1-6 RestAction

Assume that the following *RestAction* is defined in an action chain.

```
"getProduct": {
  "module": "vb/action/builtin/restAction",
  "parameters": {
    "endpoint": "extA:store/listProduct",
    "uriParams": {
      "productId": "[[ $chain.variables.productId ]]"
    }
  },
```

The *getProduct* action provides only the value for the `productId` path parameter, which is required. The data fetch fails if the path is not specified.

The vbPrepare transform could be implemented in *storeapi.js* like this (this example uses the traditional, "function-prototype" design for transforms):

```
define([], function () {
  var Request = function () {};

  Request.prototype.vbPrepare = function(configuration, options) {
    /*
     * configuration = {
     *   endpointId: 'extA:store/listProduct',
     *   endpointPath: '/products/{productId}',
     *   serverUrlTemplates: [
     *     {
     *       // The url of the server of the store service.
     *       template: 'vb-catalog://backends/extA:storeapi/{storeId}',
     *
     *       // The value that VB would use for the 'storeId' server variable,
```

```
     *       // which in this case is the default value provided by the
     *       // OpenAPI definition.
     *       variables: {
     *         storeId: '001',
     *       },
     *     },
     *     {
     *       // The url of the server of the storeapi backend.
     *       template: 'https://www.mystore.com/{version}',
     *
     *       // The value that VB would use for the 'version' server variable,
     *       // which in this case is the default value provided by the
     *       // OpenAPI definition.
     *       variables: {
     *         version: '1.0',
     *       },
     *     },
     *   ],
     * }
     */

    /*
     * options = {
     *   // The resolved uriParams specified by the RestAction.
     *   parameters: {
     *     productId: 'tv001',
     *   },
     * }
     */

     // Changing both the 'productId' path parameter and the 'version' server
variable.
     options.parameters.productId = 'notebook003';
     options.parameters['server:version'] = '2.1';

     // Adding a query parameter that is not specified in the OpenAPI
definition.
     options.parameters.internalSKU = true;
  };

  var Response = function() {};

  return {
    request: Request,
    response: Response
  };
});
```

With the transform above, the request URL that is fetched is `https://www.mystore.com/2.1/001/products/notebook003?internalSKU=true`.

Example 1-7 RestHelper

Assume that the following code is defined in a script located in extension *extB* that depends on *extA*.

```
const restHelper = RestHelper.get('extA:store/listProduct', { extensionId:
'extB' });

restHelper.parameters({
  'server:storeId': 'To-001',
  manufactureModel: true,
  productId: 'tv001',
});

restHelper.transformsContext({
  myValue: 123,
});

return restHelper.fetch;
```

Also, assume that the web application defines the following value on the *index.html* artifact:

```
<script type="text/javascript">
  window.vbInitParams = {
    'services.catalog.common.version': 'untested',
  };
</script>
```

The vbPrepare transform could be implemented in *storeapi.js* like this (this examples uses an alternative, simpler design for transforms):

```
'use strict';

define([], () => ({
  request: {
    vbPrepare: (configuration, options, transformsContext) => {
      /*
       * configuration = {
       *   endpointId: 'extA:store/listProduct',
       *   endpointPath: '/products/{productId}',
       *   serverUrlTemplates: [
       *     {
       *       // The url of the server of the store service.
       *       template: 'vb-catalog://backends/extA:storeapi/{storeId}',
       *
       *       // The value that VB would use for the 'storeId' server
variable,
       *       // which in this case is provided via the
'RestHelper.parameter'
       *       variables: {
       *         storeId: 'To-001',
       *       }
       *     },
       *     {
       *       // The url of the server of the storeapi backend.
```

```
       *       template: 'https://www.mystore.com/{version}',
       *
       *       // The value that VB would use for the 'version' server
variable,
       *       // which in this case is provided via the 'vbInitParams' from
index.html.
       *       variables: {
       *         version: 'untested',
       *       }
       *     },
       *   ],
       * }
       */

      /*
       * options = {
       *   // The parameters set via 'RestHelper.parameters'.
       *   parameters: {
       *     'server:storeId': 'To-001',
       *      manufactureModel: true,
       *      productId: 'tv001',
       *   },
       * }
       */

      /*
       * // The value set via 'RestHelper.transformsContext'
       * transformsContext = {
       *   myValue: 123,
       * }
       */

      // Not setting the 'manufactureModel' query parameter to use the
server's
      // default value.
      delete options.parameters.manufactureModel;
    },
  },
}));
```

With the transform above, the request URL that is fetched is `https://www.mystore.com/untested/To-001/products/tv001`.

FetchByKeys Transform

A fetchByKeys transforms function allows the page author to take a key or Set of keys passed in via the options and tweak the URL, to fetch the data for the requested keys.

When the consumer of the SDP calls the `fetchByKeys()` method, if the transforms author has provided a 'fetchByKeys' transforms implementation, it is called over the other transforms. If no fetchByKeys transforms function is provided then the default transforms are called.

The built-in business object REST API transforms already provides a fetchByKeys transforms function implementation that appends the keys to the URL. This should suffice for most common cases and should result in at most one fetch request to the server. For third-party REST endpoints, the author can provide a custom fetchByKeys transforms implementation.

Signature

The fetchByKeys transform function can be declared like this:

```
const fetchByKeys = function (configuration, options, transformsContext) {

  var c = configuration;
  // use the keys provided to update the c.url

  return c;
}
```

This function has the following parameters:

- `configuration` an object with the following properties:
  - Refer to the signature in Request Transformation Functions for details on the various properties.
- `options` the keys to fetch
- `transformsContext` is an object that is set by the author (ServiceDataProvider, RestHelper, Call Rest action) to then be passed as is to all transforms for the current fetch cycle.

The function returns the updated `configuration` object.

Examples

The examples below illustrate the arguments passed to the *fetchByKeys* transform, as well as the effect its code has on the fetch performed by the RestHelper.

Example 1-8   FixItFast service

The examples below use a service `fixitfast` that has a GET endpoint to retrieve the list of customers.

ServiceDataProvider variable

Assume that the following variable of type *vb/ServiceDataProvider* is defined in a page that refers to the GET /customers endpoint.

```
{
  "variables": {
    "customersSDP": {
      "type": "vb/ServiceDataProvider",
      "defaultValue": {
        "endpoint": "fixitfast-service/getCustomers",
        "keyAttributes": "id",
        "itemsPath": "result",
        "responseType": {
          "result": "customerResponse[]"
        }
      }
    },
    "customersSDP2": {
      "type": "vb/ServiceDataProvider2",
      "constructorParams": [
        {
          "endpoint": "fixitfast-service/getCustomers",
```

```
          "keyAttributes": "id",
          "itemsPath": "result",
          "responseType": {
            "result": "customerResponse[]"
          }
        }
      ]
    }
  }
}
```

The above SDP variable *customersSDP*, an extended type variable, is bound to an endpoint that returns all customers. Another variable, *customersSDP2* is an instance factory type - vb/ServiceDataProvider2 that is bound to the same endpoint.

> **Note:**
>
> In the former case, the `fetchByKeys` capability may need to be set explicitly to values other than the defaults for optimal behavior. This is because the extended type vb/ServiceDataProvider does not automatically fetch the capabilities from the service metadata transforms, whereas this is not the case for the extended type. Example,
>
> ```
> {
>   "customersSDP": {
>     "type": "vb/ServiceDataProvider",
>     "defaultValue": {
>       // ...
>       "capabilities": {
>         "fetchByKeys": {
>           "implementation": "lookup",
>           "multiKeyLookup": "single"
>         }
>       }
>     }
>   }
> }
> ```
>
> Refer to the Service Data Provider docs for details.

In both cases, when a component bound to the SDP variable initiates a fetchByKeys calls it passes the Set of keys whose data needs to be fetched. In both cases, a RestHelper instance is created and the fetchByKeys request transforms alone is run prior to the fetch. The *fetchByKeys* request transforms function uses the Set of keys passed in to build a query param on the *configuration* url.

The `fetchByKeys` transform is implemented in the default service transforms as follows:

```
define([], function () {
  class Request {

    static fetchByKeys(configuration, options, transformsContext) {
      const c = configuration;
```

```
      const fetchConfig = c.fetchConfiguration;
      const fetchByKeysCap = !!(c && c.capability === 'fetchByKeys'); //
check if current fetch call is a fetchByKeys

      if (fetchByKeysCap) {
        const fetchKeys = options;
        if (fetchKeys && fetchKeys instanceof Set && fetchKeys.size > 0) {
          const keyAttribute = fetchConfig.context.keyAttributes ||
fetchConfig.context.idAttribute;
          const keysArr = Array.from(keys);

          // use the key values provided along with the keyAttribute to
update the c.url
        }
      }

      return c;
    }
  }

  class Response {};
  class Metadata {};

  // Note: as an example, the Request object is expanded to include just the
fetchByKeys property
  return {
    metadata: Metadata,
    request: { fetchByKeys: Request.fetchByKeys },
    response: Response
  };
});
```

Filter Transform

The filter request transform allows authors to take a filter criterion and turn into a filter query that is generally appended to the `configuration.url`.

The filter criterion provided via the `options` parameter is an object that has a single attribute filter criterion with properties { op, attribute, value }, or more complex criterion as defined by the JET Data Provider docs. Additionally, text filtering is also supported (similar to the JET TextFilterDef).

While the JET Data Provider Filter API defines many complex structures for representing a filter criterion only a subset of these definitions and capabilities are implemented by the Business Object based service transforms implementation.

The Business Objects transforms supports transforming filter criterion that use a simple form (similar to JET AttributeExprFilterDef - refer to the JET docs for details), or a compound filter that is an array of simple attribute filter criterion. Nested criterion structures as shown below are also supported. See examples shown below

When using an SDP also refer to the docs for the same on how `filterCriterion` property is configured.

Examples of Criterion Transforms

This example is a simple attribute criterion that transforms to `"q=empName = 'Lucy'"`:

```
{
  "op": "$eq",
  "attribute": "empName",
  "value": "Lucy"
}
```

This example is a compound criterion that tranforms to `"q=hireDate > '2015-01-01' or hireDate <= '2018-01-01'"`:

```
{
  "op": "$or",
  "criteria": [
    {
      "op": "$gt",
      "attribute": "hireDate",
      "value": "2015-01-01"
    },
    {
      "op": "$le",
      "attribute": "hireDate",
      "value": "2018-01-01"
    }
  ]
}
```

This example is a nested compound criterion that transforms to the following query parameter `"q=((foo LIKE 'foo%') and ((bar >= 'bar1') or (bar <= 'bar2')))"`:

```
{
  "op": "$or",
  "criteria": [
    {
      "op": "$and",
      "criteria": [
        {
          "attribute": "price",
          "op": "$gt",
          "value": 30
        },
        {
          "attribute": "price",
          "op": "$lt",
          "value": 40
        }
      ]
    }
  ]
}
```

This is an example of text filtering that transforms to `q=(PartyName LIKE 'Megha%')`. The term 'PartyName' is picked up by automatically looking up a special property on the

transformsContext property - `vb-textFilterAttributes: ['PartyName']`. If nothing is set then the keyAttribute is used as the attribute for the text search:

```
{
  "text": "Megha"
}
```

This is an example of text filtering that is combined with compound criterion that transforms to `q=((PartyName LIKE 'Megha%') and (Foo = 'bar1'))',"`:

```
{
  "op": "$and",
  "criteria": [
    {
      "text": "Megha"
    },
    {
      "op": "$or",
      "criteria": [
        {
          "op": "$eq",
          "attribute": "Foo",
          "value": "bar1"
        }
      ]
    }
  ]
}
```

For 3rd party services the author can provide a custom *filter* transforms implementation.

Signature

The filter transform function can be declared like this:

```
const filter = function(configuration, options, transformsContext) {

  var c = configuration;
  // use the filter criterion provided on 'options' parameter to generate the
filter query
  // update c.url as needed

  return c;
}
```

This function has the following parameters:

- `configuration` an object with the following properties:
  - Refer to the signature in Request Transformation Functions for details on the various properties.

- `options` the filter criterion to transform. Business Objects transforms supports the following compound and attribute operators in the filter criterion:

```
  '$co', // see JET AttributeFilterOperator.AttributeOperator.$co
  '$eq', // see JET AttributeFilterOperator.AttributeOperator.$eq
  '$ew', // see JET AttributeFilterOperator.AttributeOperator.$ew
  '$pr', // see JET AttributeFilterOperator.AttributeOperator.$pr
  '$gt', // see JET AttributeFilterOperator.AttributeOperator.$gt
  '$ge', // see JET AttributeFilterOperator.AttributeOperator.$ge
  '$lt', // see JET AttributeFilterOperator.AttributeOperator.$lt
  '$le', // see JET AttributeFilterOperator.AttributeOperator.$le
  '$ne', // see JET AttributeFilterOperator.AttributeOperator.$ne
  '$sw', // see JET AttributeFilterOperator.AttributeOperator.$sw

  '$and', // see JET CompoundFilterOperator.CompoundOperator.$and
  '$or',  // see JET CompoundFilterOperator.CompoundOperator.$or
]
```

- `transformsContext` is an object that is set by the author (ServiceDataProvider, RestHelper, Call Rest action) to then be passed as is to all transforms for the current fetch cycle.

The function returns the updated `configuration` object.

Usages

The examples below illustrate the arguments passed to the *filter* transform as well as the effect its code has on the fetch performed by the RestHelper.

The examples below use a service `fixitfast` that has a GET endpoint to retrieve the list of customers.

Example 1-9   ServiceDataProvider variable

Assume that the following variable of type *vb/ServiceDataProvider* is defined in a page that refers to the above GET /customers endpoint. The SDP variable includes a default filter criterion and also sets a property called `transformsContext` with a key `vb-textFilterAttributes` that is set to `['lastName']`. This is a hint to the filter request transform to use this attribute when building a text query.

```
{
  "variables": {
    "customersSDP": {
      "type": "vb/ServiceDataProvider",
      "defaultValue": {
        "endpoint": "fixitfast-service/getCustomers",
        "keyAttributes": "id",
        "itemsPath": "result",
        "responseType": {
          "result": "customerResponse[]"
        },
        "filterCriterion": {
          "op": "$eq",
          "attribute": "region",
          "value": "{{ $variables.customer.region || \"US\" }}"
        },
        "transformsContext": {
          "vb-textFilterAttributes": ["lastName"]
```

```
        }
      }
    }
  }
}
```

When a caller such as a component bound to the above SDP initiates a fetch call, it can also provide additional filter criterion along with a text value to search against if applicable (example oj-select-single provides a text filter in the form `{ text: '<some-text-to-search>' }`). These are combined with the configured filter criterion above and the merged filter criterion is provided to the filter transforms function. Refer to the Service Data Provider docs for details.

The *filter* request transforms function uses the filter criterion passed in via the options parameter to build a query param on the *configuration* url.

The `filter` transform is implemented in the service transforms as follows:

```
define([], function () {
  class Request {

    static filter(configuration, options, transformsContext) {
      const c = configuration;
      const tc = transformsContext;
      const textFilterAttributes = tc && tc[VB_TEXT_FILTER_ATTRS];

      // process options to build the query

      return c;
    }
  }

  class Response {};
  class Metadata {};

  // Note: as an example, the Request object is expanded to include just the 
filter property
  return {
    metadata: Metadata,
    request: { filter: Request.filter },
    response: Response
  };
});
```

Example 1-10 Rest Action

Assume that the following variable of type *vb/ServiceDataProvider* is defined in a page that delegates the fetch to an action chain (set via `'fetchChainId'` property).

The chain `"fetchCustomersChain"` in its `fetchCustomers` RestAction configuration sets a default filter criteria via the property `requestTransformOptions.filter`.

When a fetch is initiated by a component bound to the SDP, the filter criterion is automatically passed in to the `filter` transform function associated to the service, along with the ones provided by caller, via the `options` parameter. Refer to the Call Rest action docs for details.

```
{
  "variables": {
    "customersSDP": {
      "type": "vb/ServiceDataProvider",
      "defaultValue": {
        "fetchChainId": "fetchCustomersChain",
        "keyAttributes": "id",
        "itemsPath": "result"
      }
    }
  },
  "chains": {
    "fetchCustomersChain": {
      "variables": {
        "configuration": {
          "type": {
            "hookHandler": "vb/RestHookHandler"
          },
          "description": "the configuration for the rest action",
          "input": "fromCaller",
          "required": true
        }
      },
      "root": "fetchCustomers",
      "actions": {
        "fetchCustomers": {
          "module": "vb/action/builtin/restAction",
          "parameters": {
            "endpoint": "fixitfast-service/getCustomers",
            "hookHandler": "{{ $variables.configuration.hookHandler }}",
            "responseType": "customersComputedResponse",
            "requestTransformOptions": {
              "filter": {
                "op": "$eq",
                "attribute": "region",
                "value": "{{ $page.variables.customer.region || \"US\" }}"
              }
            }
          },
          "outcomes": {
            "success": "returnCustomersResponse",
            "failure": "returnFailureResponse"
          }
        }
      }
    }
  }
}
```

Write a Filter Transforms Function for Text Filtering

If your SDP binds to a business object REST API endpoint, you have the following options to get text filtering to work:

- Option 1: Configure the SDP to include a vb-textFilterAttributes property where the attributes to apply the text filter is specified. The built-in business object REST API transforms look for this property and automatically build a filter criterion using the text and turns it into a 'q' param.

  ```
  "transformsContext": {
      "vb-textFilterAttributes": ["lastName"]
  }
  ```

  For the above configuration example, if a user enters text 'foo' in select-single, the SDP generates `q=lastName LIKE 'foo%'`.

  By default, the operator used is 'startsWith' as this is considered to be more optimized for db queries than 'contains'.

- Option 2: If Option 1 doesn't meet your needs, then you can write a custom filter transform that massages the text filter and turns it into a regular filterCriterion.

If you use option 2, you could do something similar to the following example. In this example, resourcesListSDP uses the getall_resources endpoint. The (request) filter transforms property is a callback that is defined in the PageModule.

```
"resourcesListSDP": {
  "type": "vb/ServiceDataProvider",
  "defaultValue": {
    "endpoint": "crmRestApi11_12_0_0/getall_resources",
    "keyAttributes": "PartyNumber",
    "itemsPath": "items",
    "responseType": "page:getallResourcesResponse",
    "transformsContext": {
      "vb-textFilterAttributes": ["PartyName"]
    },
    "transforms": {
      "request": {
        "filter": "{{ $functions.processFilter }}"
      }
    }
  }
}
```

It's important to note that the transformsContext object is an argument to every transforms function, so transforms authors can read the attributes and build the query that way.

The transforms function below takes the text value provided by the component and turns into an attribute filter criterion using the attributes passed in:

```
define(['vb/BusinessObjectsTransforms'], function(BOTransforms) {
  'use strict';

  var PageModule = function PageModule() {};
```

```
   /**
   * The filter transform parses the text filter that may be part of the
options and replaces
   * it with an appropriate attribute filter criterion using the
textFilterAttrs.
   *
   * Note: select-single provides a text filter in the form { text:
'someTextToSearch' }.
   *
   * The processing of the resulting filterCriterion is delegated to the
Business Object REST API
   * transforms module, which takes the filterCriterion and turns it into the
'q' param.
   * @param textFilterAttrs
   * @return a transforms func that is called later with the options
   */
  PageModule.prototype.processFilter = function(config, options,
transformsContext) {
    const c = configuration;
    let o = options;
    let textValue;
    let isCompound;
    const tc = transformsContext;
    const textFilterAttributes = tc && tc['vb-textFilterAttributes];

    textValue = o && o.text;

    // build your regular filtercriterion and delegate to VB BO REST API
filter transforms

    return BOTransforms.request.filter(configuration, o);
  }
  return PageModule;
});
```

Note:

Page authors are discouraged from configuring the SDP with the 'q' parameter directly, for example by setting a 'q' parameter in the uriParameters property. It is recommended that authors always use filterCriterion property to define 'q' criteria. This is especially important when using text filtering because the components always provide a filterCriterion which is appended to any configured filterCriterion on the SDP It becomes especially difficult for VB to reconcile the 'q' defined in uriParameters with the filterCriterion and authors are on their own to merge the two.

It's also important to note that select-single calls fetchByKeys very often to locate the record(s) pertaining to the select keys. For this reason, a new fetchByKeys transforms function has been added. Refer to the fetchByKeys transforms function for details.

Paginate Transform

The *paginate* request transform allows authors to take a paging criteria and generate paging related query param that is then appended to the `configuration.url`.

The paging criteria is provided via the `options` parameter with properties { `size`, `offset` }.

Example of Paging Criterion Transform

This example is a paging criterion that transforms to `"?limit=5&offset=10`:

```
{
    "size": 5,
    "offset": 10
}
```

Signature

The paginate transform function can be declared like this:

```
const paginate = function(configuration, options, transformsContext) {

  var c = configuration;
  // use the paging criteria provided on 'options' parameter to generate the 
appropriate query param
  // update c.url as needed

  return c;
}
```

This function has the following parameters:

- `configuration` an object with the following properties:
  - Refer to the signature in Request Transformation Functions for details on the various properties.
- `options` the paging criteria to transform:
  - `size`: Specifies how many rows to fetch. If a size is not specified, or size is set to -1 (some JET components, such as JET chart components, often request all rows by setting `{ size: -1 }`. When this is the case, it might be necessary to handle -1 as size).
  - `offset`: Specifies which row to start the fetch from.
- `transformsContext` is an object that is set by the author (ServiceDataProvider, RestHelper, Call Rest action) to then be passed as is to all transforms for the current fetch cycle.

The function returns the updated `configuration` object.

Usages

Example 1-11 When size of -1 is provided

```
// Paginate function that limits fetched to a max size of 100
const paginate = (configuration, options, context) => {
```

```
  var c = configuration;
  var os = (options.size === -1 || options.size > 100) ? 100 : options.size;

  if (options) {
    c.url = appendToUrl(c.url, 'limit', os);
    c.url = appendToUrl(c.url, 'offset', options.offset);
  }
  return c;
}
```

Query Transform

The query request transform allows authors to take the uri parameters and modify or add new query parameters to the `configuration.url`.

Normally uriParameters configured on the Service Data Provider or the Call Rest action are appended to the URL automatically but there may be cases where user would want to tweak the query parameters some more.

Let's say the endpoint GET /incidents, supports a query parameter called "search", which does a semantic aka contextual search. If there is a special param that needs to always be appended before calling the endpoint, then the transform function could be used for that.

Signature

The paginate transform function can be declared like this:

```
Request.prototype.query = function(configuration, options, transformsContext)
{

  var c = configuration;
  // use the query criteria provided on 'options' parameter to generate the
query param
  // update c.url as needed

  return c;
}
```

This function has the following parameters:

- `configuration` an object with the following properties:
  - Refer to the signature in Request Transformation Functions for details on the various properties.
- `options` the uri parameters.
- `transformsContext` is an object that is set by the author (ServiceDataProvider, RestHelper, Call Rest action) to then be passed as is to all transforms for the current fetch cycle.

The function returns the updated `configuration` object.

Examples

The examples below illustrate the arguments passed to the *query* transform as well as the effect its code has on the fetch performed by the RestHelper.

The example below uses a service `fixitfast` that has a GET endpoint to retrieve the list of customers.

Example 1-12 ServiceDataProvider variable

Assume that the following variable of type *vb/ServiceDataProvider* is defined in a page that refers to the GET /incidents endpoint.

The SDP variable includes the `uriParameters` property as shown below.

```
{
  "variables": {
    "incidentListTableSource": {
      "type": "vb/ServiceDataProvider",
      "input": "none",
      "defaultValue": {
        "endpoint": "fixitfast-service/getIncidents",
        "keyAttributes": "id",
        "uriParameters": {
          "technician": "hcr",
          "searchIn": "{{ $page.variables.searchType }}"
        },
        "transforms": {
          "request": {
            "query": "{{ $page.functions.query }}"
          }
        }
      }
    }
  }
}
```

The `query` request transforms function uses the criteria passed in via the `options` parameter to build a query param on the *configuration* url.

The `query` transform is implemented in the page module JavaScript as follows:

```
define([], function () {
  class PageModule {

    query(configuration, options, transformsContext) {
      const o = options;
      const c = configuration;

      if (o && !o.searchIn) {
        let newUrl = c.url;
        newUrl = `${newUrl}&search=faq`; // appends faq search qp

        c.url = newUrl;
      }
      return c;
    }
  }

  return PageModule;
});
```

Select Transform

The *select* request transform allows authors to construct the query param, to include fields whose values need to be included, in the response. Example, the built-in Business Objects based transforms creates a 'fields' query parameter.

The select criteria provided via the `options` parameter is an Object with properties { attributes, type }. The type is the response type structure typically specified on the ServiceDataProvider and Call Rest action configurations. The attributes are provided by the caller, such as the component, through a fetch call. See JET Data Provider docs for details on the fetch methods and the parameters. particularly 'attributes' (JET FetchAttribute). When `attributes` and `type` are present, how these are combined is left to the discretion of the transforms' implementation.

Examples

select criteria with `type` transforms to `"fields=PartyId,PartyStatus,PartyType;PrimaryAddress:AddressId,FormattedAddress"`

```
{
  "attributes": null,
  "type": [
    {
      "PartyId": "number",
      "PartyStatus": "string",
      "PartyType": "string",
      "PrimaryAddress": [
        {
          "AddressId": "number",
          "FormattedAddress": "string"
        }
      ]
    }
  ]
}
```

select criteria with `attributes` transforms to `fields=a;b:b1,b2;c:c2;c.c1:c1a,c1b'`

```
{
  "attributes": [
    "a",
    {
      "name": "b",
      "attributes": [
        "b1",
        "b2"
      ]
    },
    {
      "name": "c",
      "attributes": [
        {
          "name": "c1",
          "attributes": [
            "c1a",
            "c1b"
```

```
          ]
        },
        "c2"
      ]
    }
  ],
  "type": null
}
```

select criteria with both type and attributes transforms to
`fields=PartyId,PartyStatus,PartyType,a;PrimaryAddress:AddressId,FormattedAddress,b;PrimaryAddress.FormattedAddress:c`

```
{
  "attributes": [
    "PartyId",
    {
      "name": "PrimaryAddress",
      "attributes": [
        "AddressId",
        {
          "name": "FormattedAddress",
          "attributes": [
            "c"
          ]
        },
        "b"
      ]
    },
    "a"
  ],
  "type": {
    "items": [
      {
        "PartyId": "number",
        "PartyStatus": "string",
        "PartyType": "string",
        "PrimaryAddress": [
          {
            "AddressId": "number",
            "FormattedAddress": "string"
          }
        ]
      }
    ]
  }
}
```

For 3rd party services the author can provide a custom *select* transforms implementation.

Signature

The filter transform function can be declared like this:

```
function(configuration, options, transformsContext) {

  var c = configuration;
  // use the select criteria provided on 'options' parameter to generate the
query param
  // update c.url as needed

  return c;
}
```

This function has the following parameters:

- `configuration` an object with the following properties:
  - Refer to the signature in Request Transformation Functions for details on the various properties.
- `options` select the fields whose data is requested:
  - `type`: an Object that is configured as `responseType` property on the Service Data Provider or Rest Action
  - `attributes`: a structure defined by JET that is an Array<(string|FetchAttribute)>. See JET docs for details.
- `transformsContext` is an object that is set by the author (ServiceDataProvider, RestHelper, Call Rest action) to then be passed as is to all transforms for the current fetch cycle.

The function returns the updated `configuration` object.

Usage

The examples below illustrate the arguments passed to the *select* transform, as well as the effect its code has on the fetch performed by the RestHelper.

The example below uses a service `fixitfast` that has a GET endpoint to retrieve the list of customers.

Example 1-13 ServiceDataProvider variable

Assume that the following variable of type *vb/ServiceDataProvider* is defined in a page that refers to the above GET /customers endpoint. The SDP variable includes a response type that it expects the response to be in, set via the property `responseType`.

```
{
  "types": {
    "customerResponse": {
      "PartyId": "number",
      "PartyStatus": "string",
      "PartyType": "string",
      "PrimaryAddress": [
        {
          "AddressId": "number",
          "FormattedAddress": "string"
        }
      ]
    }
```

```
  },
  "variables": {
    "customersSDP": {
      "type": "vb/ServiceDataProvider",
      "defaultValue": {
        "endpoint": "fixitfast-service/getCustomers",
        "keyAttributes": "PartyId",
        "itemsPath": "result",
        "responseType": {
          "items": "customerResponse"
        }
      }
    }
  }
}
```

When a caller such as a component bound to the above SDP initiates a fetch call, it can also provide additional attributes. These are combined with the configured `responseType` above and the combined select criteria is provided to the *select* transforms function.

The `select` request transforms function uses the criteria passed in via the `options` parameter to build a query param on the *configuration* url. The `select` transform is implemented in the service transforms as follows:

```
define([], function () {
  class Request {

    static select(configuration, options, transformsContext) {
      const c = configuration;
      const selectOpts = options;

      // process options to build the query param for the fields requested in
the response
      if (selectOpts && (selectOpts.type || selectOpts.attributes)) {
        // ...
      }

      // update the c.url
      return c;
    }
  }

  var Response = function() {};
  var Metadata = function() {};

  // Note: as an example, the Request object is expanded to include just the
select property
  return {
    metadata: Metadata,
    request: { select: Request.select },
    response: Response
  };
});
```

Sort Transform

The *sort* request transform allows authors to transform a sort criteria into a sort query param that is then appended to the `configuration.url`.

The sort criteria provided via the `options` parameter is an array that has one or more criteria with properties { attribute, direction }. See JET SortCriterion for details.

Examples of Sort criteria

This example shows a single sort criterion that transforms to `"orderBy=firstName:asc"`

```
[
  {
    "attribute": "firstName",
    "direction": "ascending"
  }
]
```

This example shows a compound sort criteria that transforms to `"orderBy=firstName:asc,age:desc"`

```
[
  {
    "attribute": "firstName",
    "direction": "ascending"
  },
  {
    "attribute": "age",
    "direction": "descending"
  }
]
```

For 3rd party services the author can provide a custom *sort* transforms implementation.

Signature

The sort transform function can be declared like this:

```
const sort = function(configuration, options, transformsContext) {

  var c = configuration;
  // use the sort criteria provided on 'options' parameter to generate the
query param
  // update c.url as needed

  return c;
}
```

This function has the following parameters:

- `configuration` an object with the following properties:
  - Refer to the signature in Request Transformation Functions for details on the various properties.

- `options` the sort criteria to transform.
- `transformsContext` is an object that is set by the author (ServiceDataProvider, RestHelper, Call Rest action) to then be passed as is to all transforms for the current fetch cycle.

The function returns the updated `configuration` object.

Usage

The examples below illustrate the arguments passed to the *sort* transform, as well as the effect its code has on the fetch performed by the RestHelper.

The example below uses a service `fixitfast` that has a GET endpoint to retrieve the list of customers.

Example 1-14 ServiceDataProvider variable

Assume that the following variable of type *vb/ServiceDataProvider* is defined in a page that refers to the above GET /customers endpoint. The SDP variable includes a default sort criteria set via the property `sortCriteria`.

```
{
  "variables": {
    "customersSDP": {
      "type": "vb/ServiceDataProvider",
      "defaultValue": {
        "endpoint": "fixitfast-service/getCustomers",
        "keyAttributes": "id",
        "itemsPath": "result",
        "responseType": {
          "result": "customerResponse[]"
        },
        "sortCriteria": [
          {
            "attribute": "lastName",
            "direction": "ascending"
          }
        ]
      }
    }
  }
}
```

When a caller such as a component bound to the above SDP initiates a fetch call, it can also provide additional sort criteria. These are combined with the configured cort criteria above and the merged sort criteria is provided to the *sort* transforms function.

The

```
sort
```

request transforms function uses the criteria passed in via the

```
options
```

parameter to build a query param on the *configuration* url. The

```
sort
```

transform is implemented in the service transforms as follows:

```
define([], function () {
  class Request {

    static sort(configuration, options, transformsContext) {
      const c = configuration;
      const sortCriteria = options;

      // process options to build the query param for the sort
      if (sortCriteria && Array.isArray(sortCriteria) && sortCriteria.length
> 0) {
        sortCriteria.forEach((sc) => {
          const dir = sc.direction === 'descending' ? 'desc' : 'asc';
          const attr = sc.attribute || "";
          if (attr) {

            // build sort criteria and append to url
          }
        })
      }

      // update the c.url
      return c;
    }
  }

  class Response {};
  class Metadata {};

  // Note: as an example, the Request object is expanded to include just the
sort property
  return {
    metadata: Metadata,
    request: { sort: Request.sort },
    response: Response
  };
});
```

Example 1-15   REST Action

Assume that the following variable of type *vb/ServiceDataProvider* is defined in a page that delegates the fetch to an action chain (set via `'fetchChainId'` property).

The chain `"fetchCustomersChain"` in its `fetchCustomers` RestAction configuration sets a default sort criteria via the property `requestTransformOptions.sort`.

When a fetch is initiated by a component bound to the SDP, the sort criteria is automatically passed in to the `sort` transform function associated to the service, via the `options` parameter. Refer to the Call Rest action docs for details.

```
{
  "variables": {
    "customersSDP": {
      "type": "vb/ServiceDataProvider",
      "defaultValue": {
        "fetchChainId": "fetchCustomersChain",
        "keyAttributes": "id",
        "itemsPath": "result"
      }
    }
  },
  "chains": {
    "fetchCustomersChain": {
      "variables": {
        "configuration": {
          "type": {
            "hookHandler": "vb/RestHookHandler"
          },
          "description": "the configuration for the rest action",
          "input": "fromCaller",
          "required": true
        }
      },
      "root": "fetchCustomers",
      "actions": {
        "fetchCustomers": {
          "module": "vb/action/builtin/restAction",
          "parameters": {
            "endpoint": "fixitfast-service/getCustomers",
            "hookHandler": "{{ $variables.configuration.hookHandler }}",
            "responseType": "customersComputedResponse",
            "requestTransformOptions": {
              "sort": [
                {
                  "attribute": "lastName",
                  "direction": "desc"
                }
              ]
            }
          },
          "outcomes": {
            "success": "returnCustomersResponse",
            "failure": "returnFailureResponse"
          }
        }
      }
    }
  }
}
```

Body Transform

(Required) <Enter a short description here.>

The *body* request transform allows the author to modify the body payload for the fetch request. Some *getAll* endpoints in any type of service, be it Business Object / BOSS / ElasticSearch, may use a POST operation with a body (example, where the search criteria is set on the 'body' of the request payload) in order to retrieve search results. The body of the payload is generally provided by the caller of the request, which can be modified using this request transform function.

The *body* transform function is called after all other transforms are run. This is to allow additional contextual information to be added by the previous transforms (to the `transformsContext` parameter) that need to be included in the body.

Signature

The body transform function can be declared like this:

```
Request.prototype.body = function(configuration, options, transformsContext) {
  // Clients can manipulate the options of the fetch that contains the body
of the payload to send.
}
```

This function has the following parameters:

- `configuration` an object with the following properties:
  - Refer to the signature in Request Transformation Functions for details on the various properties.
- `options` the body for the POST request.
- `transformsContext` is an object that is set by the author (ServiceDataProvider, RestHelper, Call Rest action) to then be passed as is to all transforms for the current fetch cycle.

The function returns the updated `configuration` object.

Usage

The examples below illustrate the arguments passed to the *body* transform, as well as the effect its code has on the fetch performed by the RestHelper.

The example below uses a service `fixitfast` that has a POST endpoint to retrieve the list of incidents currently open in the system.

- The service has an operation *postToGetIncidents* similar to what is shown below. Please refer to the VB Docs on creating a Service Connection on how to add/configure POST endpoints and also set up a custom transforms for the same.

```
{
  "/incidents": {
    "post": {
      "operationId": "postToGetIncidents",
      "responses": {
        "default": {
          "description": "Default response"
        }
```

```
      },
      "x-vb": {
        "transforms": {
          "path": "./fixitfast-post-transforms.js",
          "disabled": {
            "request": [
              "sort",
              "filter",
              "query"
            ]
          }
        },
        "headers": {
          "Content-type": "application/json",
          "Accept": "application/json"
        }
      }
    }
  }
}
```

Transform Script

As indicated above, the transform script `fixitfast-post-transforms.js` is specified in the fixitfast service catalog, along with the endpoint definition. This artifact contains the *body* transform method.

If the service `fixitfast` had a transform script, the method for *body* could be declared there as long as the transforms code applies to this POST endpoint.

Example 1-16   ServiceDataProvider variable

Assume that the following variable of type *vb/ServiceDataProvider* is defined in a page that refers to the above POST endpoint.

```
{
  "variables": {
    "userFilter": {
      "type": "object",
      "defaultValue": {
        "technician": "hcr",
        "role": "tech"
      }
    },
    "searchCriteria": {
      "type": "object",
      "defaultValue": {
        "searchLevel": "allReports"
      }
    },
    "incidentsList": {
      "type": "vb/ServiceDataProvider",
      "defaultValue": {
        "endpoint": "fixitfast-service/postToGetIncidents",
        "keyAttributes": "id",
        "itemsPath": "result",
        "body": {
```

```
          "userFilter": "{{ $page.variables.userFilter }}",
          "searchCriteria": "{{ $page.variables.searchCriteria }}"
        }
      }
    }
  }
}
```

The above SDP variable incidentsList is bound to listview component that initiates a fetch to retrieve all incidents. This creates a RestHelper instance and subsequently the request transforms to be run. The *body* request transforms function is called, and the body is updated as needed, on the *configuration* object, before the POST request is made.

The body transform is implemented in `fixitfast-post-transforms.js` as follows:

```
define([], function () {
  class Request {
    static body(configuration, options, transformsContext) {
      const c = configuration;

      /*
       * options = {
       *   userFilter: {
       *     technician: 'hcr',
       *     role: 'tech'
       *   },
       *   searchCriteria: {
       *     searchLevel: 'allReports'
       *   }
       * }
       */

      // Update the body values if needed
      const body = c.initConfig.body || {};

      return c;
    };
  }

  class Metadata {};
  class Response {};

  // Note: as an example, the Request object is expanded to include just the
body property
  return {
    metadata: Metadata,
    request: { body: Request.body },
    response: Response
  };
});
```

Response Transformation Functions

Response transformation (transform) functions are called right after a request returns successfully, and allow a page author to transform / augment the response to a form expected by the caller.

The ServiceDataProvider supports a predefined list of response transformation function types, described in this section. Note that there are no guarantees of the order in which transform functions are called.

Signature

A response transformation function has the following signature: `function (result)`. It can be defined on the service endpoint, but can also be overridden on the variable.

```
function(configuration, transformsContext) {

    // process the contents and return the result appropriate for the
response transform
    return result;
}
```

Generally these functions are implemented by a service author and associated to the service, but the individual functions can also be overridden on the Service Data Provider variable or the Call Rest action.

The parameters to the function are:

- **configuration**: An object that has the following properties:
  - **headers**: The response headers.
  - **body**: The body returned in the response.
  - **fetchConfiguration**: the configuration pertaining to this fetch call. If fetch was initiated by ServiceDataProvider this includes the following properties:
    - **capability**: The fetch capability, like fetchByKeys, fetchFirst, fetchByOffset
    - **context**: a snapshot of the ServiceDataProvider variable state at the time the fetch call was made.
    - **externalContext**: if the fetch was externalized to a chain, then the context setup on the RestAction in that chain
    - **fetchParameters**: a snapshot of the original fetch parameters provided by the initiator of the fetch (such as a component). The parameters passed to the fetch call are defined by the JET Data Provider fetch API.
    - **transformsOptions**: these are full set of transforms options that are passed to each transform function. These are computed using the parameters configured on the Service Data Provider, the RestAction (if applicable), and the input parameters provided by initiator (such as the component).
- **transformsContext**: a context object that is passed to every transform function to store/ retrieve any contextual information for the current request lifecycle.

The function returns a `configuration` object appropriate for the response transform type. See the following common response transform types (paginate transform, body transform) below for details on the returned responses.

Paginate Transform

This transformation function is called immediately after the fetch returns with a response. The paginate response transform function can process the response and return an object with the following properties set.

The returned Object is the primary way in which callers like Service Data Provider know about the paging state.

- `totalSize`: A number tracking for the (canonical) total size of the result is. See JET Data Provider Docs for details.
- `hasMore`: generally required. A boolean that indicates whether there are more records to fetch. Example in Business Objects based services, this would map to the hasMore boolean property commonly returned, in the response. Iterating components can use this information to keep iterating until there is no more data to fetch, or until certain UI conditions are met (this might be needed when a selected row is several pages down).
- `pagingState`: This can be used to store any paging state specific to the paging capability supported by the endpoint. This additional paging state will then be passed 'as is' to the request paginate transform function, for the next iteration.

Body Transform

This transform function is called immediately after the REST call returns with a response, and is a hook for authors to transform the response body, if needed. This function is not guaranteed to be called in any specific order.

Example

A ServiceDataProvider variable that is configured with a custom body response transform is shown below. While it overrides the `body` response transforms whereas, the `paginate` response transforms function used to process the response returned by fetch call, is the default transforms associated to the service.

```
{
  "variables": {
    "incidentsList": {
      "type": "vb/ServiceDataProvider",
      "defaultValue": {
        "endpoint": "ifixitfast-openapi3/getCustomers",
        "keyAttributes": "id",
        "itemsPath": "items",
        "responseType": {
          "items": "customerResponse[]",
          "extraResult": "extraResponse"
        },
        "transforms": {
          "response": {
            "body": " {{ $page.functions.bodyResponse }}"
          }
        }
      }
    }
  }
}
```

The default `paginate` response transforms functions for a Business Object based service returns an object with the properties `{ totalSize, hasMore }` as shown below:

```
define([], function () {
  class Response {
    /**
     * Called after response returns from a fetch call, this is a good place 
to process
     * response, to provide pagination info such as totalSize and hasMore.
     *
     * @param configuration - a Map containing the following properties
     * - headers: response header
     * - body: body of the response
     * - fetchConfiguration: the configuration that triggered this fetch call.
     *
     * @param transformsContext transforms context
     *
     * @returns {{}}
     */
    static paginateResponse(configuration, transformsContext) {
      const ps = {};
      const tr = {};

      if (configuration.body) {
        const rb = configuration.body;

        if (rb.totalCount) {
          tr.totalSize = rb.totalCount;
        }
        if (rb.totalCount > 0) {
          tr.hasMore = !!rb.hasMore;
        } else {
          tr.hasMore = false;
        }
      }
      return tr;
    };
  }

  var Response = function() {};
  var Metadata = function() {};

  // Note: as an example, the Request object is expanded to include just the 
sort property
  return {
    metadata: Metadata,
    request: Request,
    response: { paginate: Response.paginateResponse },
  };
});
```

The custom body response transforms function configured in the SDP variable is defined in the PageModule JS. It appends extra results to the return value.

```
define([], function () {
  class PageModule {
    /**
     * Fix up response data and extract other info and return a transformed
result body.
     * The object returned must have the body that the caller is configured
for
     *
     * @param configuration - a Map containing the following properties
     * - headers: response header
     * - body: body of the response
     * - fetchConfiguration: the configuration that triggered this fetch call.
     *
     * @param transformsContext transforms context
     *
     * @returns {*}
     */
    static bodyResponse = (configuration, transformsContext) => {
      const tr = {};
      const c = configuation;

      if (c.body) {
        // fix up result.body from REST if needed and set the new body in tr
        tr.items = c.body.items;

        // you can also store additional data.
        tr.extraResult = { foo: 'bar' };
      }

      return tr;
    };
  };

  return PageModule;
});
```

Methods

ServiceDataProvider implements most methods from `oj.DataProvider`, except for the isEmpty method.

Most ServiceDataProvider methods, such as fetchFirst, fetchByKeys, fetchByOffset, containsKeys, and getCapabilities, are called by the component that interfaces with the DataProvider implementation and will rarely need to be used directly. The getTotalSize method is an exception to this general rule.

getTotalSize method

The getTotalSize method returns a Promise that resolves to the total size of data available on the service endpoint. If a positive number is not set in the response transforms, a size of -1 is returned. Generally the returned value is the canonical size of the (endpoint) fetch when no search criteria is applied. In other words, this value is meant to be the same every time a fetch is called against the endpoint.

Because page authors often want the convenience of binding the totalSize on the page, vb/ServiceDataProvider supports a totalSize property that is a number. This can be used instead of the getTotalSize method, which is used by JavaScript callers.

For example, a page author can use the totalSize property of the ServiceDataProvider in markup as follows:

```
<oj-bind-text id="totalIncRows"
  value="[[ $variables.incidentListDataProvider.totalSize ]]"></oj-bind-text>
```

Features and Capabilities

Page authors generally need not be concerned with this, but it's generally useful to understand the features and capabilities that SDP supports. For details refer to JET DataProvider#getCapability.

At design time, a page author may need to know what features and capabilities the endpoint supports, and they may need to configure the correct properties and transforms.

Events

At design time, a page author may need to know what features and capabilities the endpoint supports, and they may need to configure the correct properties and transforms.

Events

The events raised by the data provider are defined by contract for `oj.DataProvider`. These events are fired at appropriate times to notify UI components. Page authors may need to force the variable to fire some of the DataProvider events, including 'add', 'remove', 'refresh', and 'update'.

vbDataProviderNotification Event Listener

Page authors can register an event listener of this type in order to be notified of catastrophic errors that may occur when something goes wrong during an implicit fetch. For an externalized fetch, where the fetch is externalized to a action chain, the current mechanism of handling failure outcomes can continue to be used.

For example, on the page, the listeners property can have this definition:

```
"vbDataProviderNotification": {
  "chains": [
    {
      "chainId": "someChainX"
    }
  ]
}
```

The event payload available to the listener is an object that has the following properties:

- **severity**: a string
- **detail**: any details of the error, such as REST failure details
- **capability**: an object with the capabilities configured on the ServiceDataProvider
- **fetchParameters**: an object with the parameters passed to the fetch

- **context**: an object representing the state of the ServiceDataProvider at the time the fetch was initiated
- **id**: uniqueId, a string, the id of the ServiceDataProvider instance
- **key**: since the event can be fired multiple times, this identifies the event instance

Page authors can use this to display an error message.

Example 1-17 Firing a DataProvider event by using a fireDataProviderEvent action

A page is configured to have a master list and detail form showing the details of the current selected row on the list. Suppose that the form is wired to PATCH to a different endpoint than the one configured on the list. When the user updates the form data, it's desirable for the same actionChain to also raise the 'update' event on the ServiceDataProvider so it can show the changes to the current row. To configure the page:

```
<!-- list view bound to page variable incidentListTableSource -->
<oj-list-view id="listview"
              data="{{$variables.incidentListTableSource}}"
...
</oj-list-view>

<!-- form UI fields bound to page variable currentIncident -->
<div class="oj-form-layout"
  <div class="oj-form"
    <div class="oj-flex"
      <div class="oj-flex-item"
        <oj-label for="problem"Problem</oj-label>
      </div>
      <div class="oj-flex-item"
        <oj-input-text id="problem"
                       value="{{$variables.currentIncident.problem}}"
                       required=true</oj-input-text>
      </div>
    </div>
...

<!-- Save button bound to componentEvent handler 'saveIncident' -->
<oj-button href="#" id='saveButton'
           label='Save'
           on-dom-click='[[$componentEvents.saveIncident]]'</oj-button>


// saveIncident calls the actionChain 'saveIncidentChain', which
// (1) defines 2 variables - incidentId and incidentPayload
// (2) then calls a REST action to put/patch payload
// (3) then it takes the result from (2) and assigns to incidentsResponse
chain
//     variable,
// (4) calls an actionChain to fire a data provider event to refresh the SDP
page
//     variable
// (5) an update event payload passed to the action chain
"saveIncidentChain": {
  "variables": {                                                   // (1)
    "incidentId": {
      "type": "string",
```

```
        "description": "the ID of the incident to update",
        "input": "fromCaller",
        "required": true
      },
      "incidentPayload": {
        "type": "object",
        "description": "the payload of the incident data",
        "input": "fromCaller",
        "required": true
      },
      "incidentsResponse": {
        "type": "application:incidentsResponse"
      }
    },
    "root": {
      "id": "saveIncidentToRest",                              // (2)
      "module": "vb/action/builtin/restAction",
      "parameters": {
        "endpoint": "ifixitfast-service/putIncident",
        "uriParams": {
          "id": "{{ $variables.incidentId }}"
        },
        "body": "{{ $variables.incidentPayload }}"
      },
      "outcomes": {
        "success": "assignVariables_incidentsResponse"
      }
    },
    "assignVariables_incidentsResponse": {
      "module": "vb/action/builtin/assignVariablesAction",
      "parameters": {
        "$variables.incidentsResponse.result": {
          "source": "{{ $chain.results.saveIncidentToRest.body }}" // (3)
        }
      },
      "outcomes": {
        "success": "updateIncidentList"
      }
    },
    "updateIncidentList": {
      "module": "vb/action/builtin/callChainAction",
      "parameters": {
        "id": "fireDataProviderMutationEventActionChain",         // (4)
        "params": {
          "payload": {
            "update": {                                           // (5)
              "data": "{{ $variables.incidentsResponse }}"
            }
          }
        }
      }
```

```
  }
}


"fireDataProviderMutationEventActionChain": {
  "variables": {
    "payload": {
      "type": "application:dataProviderMutationEventDetail",
      "input": "fromCaller"
    }
  },
  "root": "fireEventOnDataProvider",
  "actions": {
    "fireEventOnDataProvider": {
      "module": "vb/action/builtin/fireDataProviderEventAction",
      "parameters": {
        "target": "{{ $page.variables.incidentListDataProvider }}", // SDP
variable
                                                     // on which the event is
fired
        "add": "{{ $variables.payload.add }}",
        "remove": "{{ $variables.payload.remove }}",
        "update": "{{ $variables.payload.update }}" // has the updated record
details
      }
    }
  }
},
```

ServiceDataProviderFactory

Some times it's desirable to create a standalone VB type instance programmatically by passing an initial state. Here the instance is not backed by a variable, that is, its state is not stored in redux. Instead the instance and/or the caller manages its state essentially.For such cases VB publishes a contract for a TypeFactory that any type author can implement. See Custom Extended Types.

The TypeFactory contract is provided in the `vb/types/factories/typeFactory.js`. VB provides TypeFactory implementations for creating a ServiceDataProvider instance. Refer to the ServiceDataProviderFactory for details. (`vb/types/factories/serviceDataProviderFactory.js`)

Methods

createInstance

Returns an instance of the ServiceDataProvider. Refer to the JSDocs for the parameters supported on this method. The instance returned supports all methods from the DataProvider contract.

- options, object used to instantiate the ServiceDataProvider with, usually contains these properties
  - dataProviderOptions, its initial or 'default' state.
    - state properties are same as what a regular ServiceDataProvider variable takes

    - serviceOptions, optional configuration needed by the RestHelper to locate the endpoint details. This can be skipped if the dataProviderOptions includes an 'endpoint' property
        * properties:
            * `url <string>`
            * operationRef <string>

caller can create an instance as follows:

```
ServiceDataProviderFactory.createInstance({ dataProviderOptions: { endpoint:
"foo/getBars", responseType: "barType[]", keyAttributes: "id"} })
  .then((sdpInstance) => {
    const iter = sdpInstance.fetchFirst();
    iter.next().then((results) => {
      // process results
    });
  });
```

Multi-Service Data Provider

The vb/MultiServiceDataProvider built-in type is a data provider implementation that combines multiple vb/ServiceDataProvider variables, each providing a unique fetch capability.

Often components that bind to data providers, like oj-combobox-one and oj-select-single (or the -many variants), require or use different 'fetch' capabilities on the data provider implementation.

For example, an oj-select-single component might call fetchFirst() (on the DataProvider implementation) to populate its options, and then call fetchByKeys() to fetch data for selected value, and fetchByOffset() to fetch items from an offset. Often the endpoint configured on a ServiceDataProvider may provide multiple capabilities - for example, most GETAll endpoints for business object REST API services also allow fetching data for specific keys, and from an offset, on the same endpoint. However, on rare occasions authors might require different endpoints to support different fetch capabilities. A MultiServiceDataProvider can be used for this purpose.

Design Time Assumptions

At design time, a service author can identify different endpoints that provide the fetchByKeys and fetchByOffset capabilities, in addition to the current fetch all (fetchFirst capability). When there are different endpoints a page author must pick different endpoints for each (fetch) capability when configuring a variable of a type vb/MultiServiceDataProvider. It is common for the same REST endpoint to support multiple capabilities.

- **for fetchByKeys**
    - For example, the same endpoint can fetch all territories and a set of territories that match a set of territory codes (fetchByKeys): `GET /fndTerritories? and /fndTerritories?q=TerritoryCode in ('US', 'AE')`
    - the same endpoint can be used to fetch all customers, or to fetch customers by specific keys using the same endpoint but different query parameters: `GET /customers and GET /customers?ids=cus-101,cus-103`.
- **for fetchByOffest**
    - an Oracle Cloud application endpoint can fetch all territories, and territories at a given offset - GET `/fndTerritories` and `/fndTerritories?offset=50&size=10`

Properties

A variable of the built-in type vb/MultiServiceDataProvider can be configured with the `dataProviders` property using the following sub-properties.

dataProviders Sub-property	Type	Example	Description
fetchFirst	"vb/ServiceDataProvider"	`{ "variables": { "activitiesMultiSDP": { "type": "vb/MultiServiceDataProvider", "defaultValue": { "dataProviders": { "fetchFirst": "{{ $variables.listSDP }}" } } } } }`	A MultiServiceDataProvider is needed only when more than one fetch capability needs to be configured.

dataProviders Sub-property	Type	Example	Description
fetchByKeys	"vb/ ServiceDataProvider"	`{ "variables": { "activitiesMultiSDP": { "type": "vb/MultiServiceDataProvider", "defaultValue": { "dataProviders": { "fetchFirst": "{{ $variables.listSDP }}" "fetchByKeys": "{{ $variables.detailSDP }}" } } } } }`	A reference to the vb/ ServiceDataProvider variable.

dataProviders Sub-property	Type	Example	Description
fetchByOffset	"vb/ServiceDataProvider"	`{ "variables": { "activitiesMultiSDP": { "type": "vb/MultiServiceDataProvider", "defaultValue": { "dataProviders": { "fetchFirst": "{{ $variables.listSDP }}" "fetchByOffset": "{{ $variables.listSDP }}" } } } } }`	A reference to the vb/ServiceDataProvider variable.

Behavior

- A variable of type vb/MultiServiceDataProvider must have at least one fetch capability defined. Otherwise an error is flagged.
- When a fetchFirst capability is not defined, a no-op fetchFirst capability is used. The JET DataProvider contract requires a fetchFirst implementation to be provided.
- All fetch capabilities must point to a variable of type vb/ServiceDataProvider.
- A MultiServiceDataProvider cannot reference another MultiServiceDataProvider variable.

Usage

Here are some of the common ways service endpoints might provide their fetch capabilities.

Usage: When a service provides unique endpoints for different fetch capabilities

When a service has unique endpoints for each fetch capability, we will require one variable of type '`vb/ServiceDataProvider`' per fetch API, and a variable of type '`vb/MultiServiceDataProvider`' variable that combines the individual ServiceDataProvider variables together. The list-of-values component will then bind to a variable of type `vb/MultiServiceDataProvider`.

Let's consider this third-party REST API that is used to get information about countries.

- fetchFirst capability: to get a list of all countries and their info, where the alpha3Code is the primary key
  - service/endpoint: rest-service/getAllCountries
  - `GET https://restcountries.eu/rest/v2/all`
- fetchByKeys capability (with multi key lookup): to get a list of countries by their three-letter alpha code
  - service/endpoint: rest-service/getCountriesByCodes
  - `GET https://restcountries.eu/rest/v2/alpha?codes=usa;mex`

In order for the list-of-values component to use the above endpoints, the design time will need to create three variables:

- One vb/MultiServiceDataProvider variable that references two ServiceDataProvider variables, one for each fetch capability
- Two vb/ServiceDataProvider variables

vb/MultiServiceDataProvider Configuration

At design time, a variable using this type will be created that looks like this:

```
1  {
2    "variables": {
3      "countriesMultiSDP": {
4        "type": "vb/MultiServiceDataProvider",
5        "defaultValue": {
6          "dataProviders": {
7            "fetchFirst": "{{ $page.variables.allCountriesSDP }}
8            "fetchByKeys": "{{ $page.variables.countriesByCodesSDP }}"
9          }
10       }
11     }
12   }
13 }
```

- Line 3: countriesMultiSDP is a variable of type vb/MultiServiceDataProvider. This defines two properties: 'fetchFirst' and 'fetchByKeys'.
- Line 7: The fetchFirst property allows the MultiServiceDataProvider to call fetchFirst() on the referenced ServiceDataProvider variable.
- Line 8: The fetchByKeys property allows the MultiServiceDataProvider to call fetchByKeys() on the referenced ServiceDataProvider variable.

vb/ServiceDataProvider Variables Configuration

For the above use case, the referenced ServiceDataProvider variables will be configured as follows:

<table>
<tr><th>Configuration</th><th>Description</th></tr>
<tr><td>

```
1  {
2    "variables": {
3      "allCountriesSDP": {
4        "type": "vb/
ServiceDataProvider",
5        "defaultValue": {
6          "endpoint": "rest-service/
getAllCountries",
7
   "keyAttributes": "alpha3Code"
8        }
9      },
10     "countriesByCodesSDP": {...}
11   }
12 }
```

</td><td>

Line 3: defines the ServiceDataProvider variable with a fetchFirst capability.

- When a capabilities property is not specified, it's assumed that the ServiceDataProvider supports a fetchFirst capability.
- When a capabilities property is present but no fetch capability is defined (that is, only the filter and sort capabilities are defined), fetchFirst is assumed.

Line 6: defines the endpoint to use the getAllCountries operation to fetch all countries.

</td></tr>
</table>

Configuration	Description

`1 {`
`2   "variables": {`
`3     "allCountriesSDP": {`
`4    "countriesByCodesSDP": {`
`5       "type": "vb/ServiceDataProvider",`
`6       "defaultValue": {`
`7         "endpoint": "rest-service/getCountriesByCodes",`
`8        "keyAttributes": "alpha3Code",`
`9         "capabilities": {`
`10           "fetchByKeys": {`
`11            "implementation": "lookup",`
`12            "multiKeyLookup" : 'no'`
`13           }`
`14         },`
`15        "mergeTransformOptions": "{{ $functions.fixupTransformOptions }}"`
`16       }`
`17     }`
`18   }`
	Line 4: defines the ServiceDataProvider variable that supports a fetchByKeys capability.
	Line 7: uses the getCountriesByCodes operation to fetch a list of countries by their codes.
	Line 9: a '`capabilities`' property is added to ServiceDataProvider that has a '`fetchByKeys`' property object. See next section for details.
	• '`implementation`' property is set to "`lookup`"
	• '`multiKeyLookup`' property set to "no"
	Line 15: the '`mergeTransformOptions`' property is set to a page function.
	• this is needed so page author can map the keys set programmatically to be turned into the query parameters '`?codes=`'
	 Note:
	Normally fetchByKeys() is called by a JET component programmatically with one or more keys.
	• When keys are provided programmatically, ServiceDataProvider will use a best-guess heuristic to map keys to the appropriate transform options. But when this is not easily decipherable by ServiceDataProvider, page authors can use a '`mergeTransformOptions`' property that maps to a function, to fix up the list of the 'query' options. This function will be passed in all the info it needs to merge the final transform options.
	 Note:
	In this example the keys need to map to the codes uriParameters, and such a mapping cannot be represented in the page model using an expression.
	• When no keys are provided, ServiceDataProvider will throw an error.

<table>
<tr><th>Configuration</th><th>Description</th></tr>
<tr><td>

```
1  /**
2   * fix up the query transform
options.
3   * When the fetchByKeys
capability is set, the 'keys'
provided via the fetch call
4   * can be be looked up via
configuration.fetchParameters.
5   * This can be used to set a
'codes' property on the 'query'
transform options
6   * whose value is the keys
provided via a fetch call.
7   *
8   * @param configuration a map of
3 key values, The keys are
9   *   - fetchParameters:
parameters passed to a fetch call
10  *   - capability: 'fetchByKeys'
| 'fetchFirst' | 'fetchByOffset'
11  *   - context: the context of
the SDP when the fetch was initiated.
12  *
13  * @param transformOptions a map
of key values, where the keys are the
14  *         names of the transform
functions.
15  * @returns {*}
16  */
17
PageModule.prototype.fixupTransformOp
tions =
18     function (configuration,
transformOptions) {
19   var c = configuration;
20   var to = transformOptions;
21   var fbkCap = !!(c &&
c.capability === 'fetchByKeys');
22   var keysToFetch = fbkCap ?
23                     (c &&
c.fetchParameters &&
c.fetchParameters.keys) : null
24
25   if (fbkCap && keysToFetch &&
keysToFetch.length > 0) {
26     // join keys
27     var keysToFetchStr =
keysToFetch.join(';');
28     to = to || {};
29     to.query = to.query || {};
30
```

</td><td>Line 17: function that fixes up the transform options that will be sent to the transform functions.

Line 33: set a new 'codes' query parameter, whose value is a ';' separated list of country alpha codes.</td></tr>
</table>

Configuration	Description
31 // ignore codes set on the query options and instead use ones passed in 32 // by fetchByKeys call 33 to.query.codes = keysToFetchStr; 34 } 35 36 return to; 37 };	

Configuring a JET Combo/Select at Design Time

To configure a list-of-values field that uses the above, the design time needs to create three variables:

- One vb/MultiServiceDataProvider variable
- Two vb/ServiceDataProvider variables

The MultiServiceDataProvider variables are bound to the combo/select components as follows.

- Line 2 points to a variable of type `vb/MultiServiceDataProvider`.

```
1  <oj-combobox-one id="so11" value="{{ $variables.selectedActivities }}"
2                        options="[[ $variables.countriesMultiSDP ]]"
3                         options-keys.label='[[ "name" ]]'
4                        options-keys.value='[[ "alpha3Code" ]]'
5  </oj-combobox-one>
```

A distinct vb/ServiceDataProvider variable is needed for each unique service/endpoint. Often authors want to provide different default filterCriterion, sortCriteria or uriParams, or even write different transforms for each capability. Isolating each capability to a unique ServiceDataProvider variable allows for this separation.

Any individual vb/ServiceDataProvider variables might externalize its fetch, or allow an actionChain to assign values to its properties directly via expressions. They can also allow a fireDataProviderEventAction to reference the Service Data Provider variable directly. First class variables are the easiest way to give page authors access.

Usage: When a service provides unique endpoints for different fetch capabilities, but the fetchByKeys endpoint only supports a single-key-based lookup

In this use case, the service supports a fetchFirst capability that fetches all rows, and a fetchByKeys capability that returns a single row by its key. There is no endpoint that can return rows by multiple keys.

To understand this usecase further let's take the example of the sample ifixitfast service - and the incidents endpoints that is used to get information about incidents.

- fetchFirst capability: to get a list of all incidents for the selected technician,
  - service/endpoint: `fixitfast-service/getIncidents`
  - `GET https://.../ifixitfaster/api/incidents?technician=hcr`

- fetchByKeys capability (with single key lookup): to get a single incident it its 'id'
  - service/endpoint: `fixitfast-service/getIncident`
  - `GET https://.../ifixitfaster/api/incidents/inc-101`

In order for the list-of-values component to use the above endpoints, the design time will need to create three variables:

- One vb/MultiServiceDataProvider variable that references two ServiceDataProvider variables, one for each fetch capability
- Two vb/ServiceDataProvider variables

vb/MultiServiceDataProvider Variable Configuration

The configuration for the `vb/MultiServiceDataProvider` variable is similar to the previous examples.

```
1  {
2    "variables": {
3       "countriesMultiSDP": {
4         "type": "vb/MultiServiceDataProvider",
5         "defaultValue": {
6          "dataProviders": {
7             "fetchFirst": "{{ $page.variables.allIncidentsSDP }}"
8              "fetchByKeys": "{{ $page.variables.incidentBySingleKeySDP }}"
9          }
10       }
11      }
12    }
13  }
```

vb/ServiceDataProvider Variables Configuration

For the previous use case, the referenced ServiceDataProvider variables will be configured as follows.

<table>
<tr><th>Configuration</th><th>Description</th></tr>
<tr><td>some-page.json

<pre>
1   {
2   "variables": {
3   "allIncidentsSDP": {
4   "type": "vb/
ServiceDataProvider",
5   "defaultValue": {
6
"endpoint": "fixitfast-service/
getAllIncidents",
7   "keyAttributes": "id",
8   "itemsPath": "result",
9   "uriParameters": {
10
"technician": "{{ $application.user.u
serId }}"
11   }
12   }
13   },
14   "incidentBySingleKeySDP":
{...}
15   }
16   }
</pre></td><td>

- Line 3: defines the ServiceDataProvider variable with the fetchFirst capability.
- Line 6: defines the endpoint that uses the getAllIncidents operation to fetch all incidents.
</td></tr>
</table>

<table>
<tr><th>Configuration</th><th>Description</th></tr>
<tr><td>

```
1  {
2    "variables": {
3      "allIncidentsSDP": {...},
4      "incidentBySingleKeySDP": {
5        "type": "vb/
ServiceDataProvider",
6        "defaultValue": {
7          "endpoint": "fixitfast-
service/getIncident",
8          "keyAttributes": "id",
9          "uriParameters": {
10           "id":
"{{ $variables.incidentId }}"
11         }
12         "capabilities": {
13           "fetchByKeys": {
14             "implementation":
"lookup",
15             "multiKeyLookup" :
'no'
16           }
17         }
18       }
19     }
20   }
```

</td><td>

Line 4: defines the ServiceDataProvider variable with the fetchByKeys capability. The ServiceDataProvider variable is configured for an implicit fetch.

Line 7: uses the getIncident operation to fetch a single incident by its id.

Line 9: maps the 'id' key in the 'uriParameters'.

- At runtime the 'id' key value is substituted in the path parameter of the URL.
- For example, if the 'id' value is "inc-101", the request URL goes from https://.../incidents/{id} → http://.../incidents/inc-101

Line 12: a new 'capabilities' property is added to ServiceDataProvider that has a 'fetchByKeys' key object.

- The 'implementation' property is set to "lookup".
- The 'multiKeyLookup' property is set to "no", as the endpoint only supports lookup using a single key at a time.

Notice that a 'mergeTransformOptions' property is not set.

- This is because Service Data Provider uses a simple heuristic to map the 'keys' provided programmatically to the 'id' sub-property of the 'uriParameters'.
  - It can do this because ServiceDataProvider sees that the keyAttributes value "id" is the same attribute key set on 'uriParameters'.
  - Also, this is only possible when ServiceDataProvider is configured to use implicit fetch (that is, it does not use an external action chain to do a fetch).
- In some cases the ServiceDataProvider cannot easily decipher the mapping (as seen in the previous example), and this is when page authors can use a 'mergeTransformOptions' property to map the keys to the right transform options.
- When multiple keys are provided by the caller, ServiceDataProvider as an optimization calls the single endpoint a single key at a time, assembles the result, and returns this to caller.

</td></tr>
</table>

<table>
<tr><th>Configuration</th><th>Description</th></tr>
<tr><td><pre>
1 {
2   "variables": {
3   "allIncidentsSDP": {...},
4
"incidentBySingleKeySDP_External": {
5   "type": "vb/
ServiceDataProvider",
6   "defaultValue": {
7   "fetchChainId":
"fetchSingleIncidentChain",
8   "keyAttributes": "id",
9   "mergeTransformOptions":
"{{ $page.functions.fixupTransformOpt
ions }}",
10   "capabilities": {
11   "fetchByKeys": {
12   "implementation":
"lookup",
13   "multiKeyLookup": "no"
14   }
15   }
16   }
17   },
18   "chains": {}
19 }
</pre></td><td>Line 4: defines the ServiceDataProvider variable with a fetchByKeys capability.
The Service Data Provider variable uses an action chain to fetch data. See the next section for the action chain configuration.
Line 9: sets a mergeTransformOptions function.
This function is used by the page author to fix up the 'query' transform options to use the key passed in via the fetch call.</td></tr>
</table>

Configuration	Description

<pre>/**
 * Process the transform options.
 * When ServiceDataProvider uses
external fetch chain, it doesn't
 * have all the information to build
the final transform options
 * to use with the transform
functions. In such cases the page
 * author can use this method to
build the final list of options.
 * Replaces id set via configuration
with the value passed in by caller.
 *
 * @param configuration an Object
with the following properties
 *   - capability: 'fetchByKeys' |
'fetchFirst' | 'fetchByOffset'
 *   - fetchParameters: parameters
passed to the fetch call
 *   - context: the context of the
Service Data Provider variable at
 *   the time the fetch
call was made
 *
 * @param transformOptions a map of
key values, where the keys are the
 *   names of the transform
functions.
 *
 * @returns {*} the transformOptions
either the same one passed in or
 *   the final fixed up transform
options
 */
PageModule.prototype.fixupTransformOp
tions = function (configuration,
transformOptions) {
  var c = configuration;
  var to = transformOptions || {};
  var fetchByKeys = !!(c &&
c.capability === 'fetchByKeys');
 	
  if (fetchByKeys) {
    var key =
c.fetchParameters.keys[0];
    if (key &&
        (!to.query || (to.query &&
to.query.id !==
c.fetchParameters.keys[0]))) {
      to.query = to.query || {};
      to.query.id = key;</pre>
	mergeTransformOptions function

Configuration	Description

<pre>   }
  }
  return to;
};</pre>	

<table>
<tr><th>Configuration</th><th>Description</th></tr>
<tr><td>

```
1  {
2    "variables": {},
3    "chains": {
4      "fetchSingleIncidentChain": {
5        "variables": {
6          "configuration": {
7            "type": {
8              "hookHandler": "vb/
RestHookHandler"
9            },
10           "description": "the
configuration for the rest action",
11           "input": "fromCaller",
12           "required": true
13         },
14         "uriParameters": {
15           "type": "object",
16           "defaultValue": {
17             "id":
"{{ $page.variables.incidentId }}"
18           }
19         }
20       },
21       "root":
"fetchSingleIncidentAction",
22       "actions": {
23
"fetchSingleIncidentAction": {
24           "module": "vb/action/
builtin/restAction",
25           "parameters": {
26             "endpoint":
"fixitfast-service/getIncident",
27             "hookHandler":
"{{ $variables.configuration.hookHand
ler }}",
28             "uriParams":
"{{ $variables.uriParameters }}",
29             "responseType":
"flow:incident",
30
"requestTransformFunctions": {
31               "query":
"{{ $page.functions.queryIncidentById
 }}"
32             }
33           },
34           "outcomes": {
35             "success":
"returnIncidentResponse",
36             "failure":
```

</td><td>

The external fetch action chain is configured as follows.

Line 4: the action chain used by the ServiceDataProvider.

Line 23: the RestAction, the chain calls to fetch a single incident by id.

Line 28: the 'uriParams' property of the RestAction is set to the page variable "incidentId".

- The value of the "incidentId" variable might be different from what the caller passes in.
- The mergeTransformOptions function above builds the query options containing the final id value.

Line 31: the requestTransformFunction.query maps to a query transform function that substitutes the endpoint URL with the final id value.

</td></tr>
</table>

Configuration	Description

<pre>"returnFailureResponse"
37   }
38   },
39   }
40   }
41   }
42 }</pre>	

<pre>/** * query transform function that takes the id provided in the options * and expands the URL. * @param configuration * @param options * @returns {*} */ PageModule.prototype.queryIncidentByI d = function (configuration, options) { const c = configuration; if (options && options.id) { var result = URI.expand(c.endpointDefinition.url, { id: options.id }); var newUrl = result.toString(); if (newUrl !== c.url) { console.log(`typesDemo sample: replacing ${c.url} with ${newUrl}`); } c.url = newUrl; } return c; };</pre>	Query transform function

Usage: When the same endpoint supports multiple fetch capabilities

Most list-of-value objects fall into this category. For example, to fetch both a list of territories and to fetch a subset of territories by their ids, the same endpoint is used:

- fetchFirst capability:
  - service/endpoint: fa-crm-service/getTerritories
  - GET /fndTerritories?finder=EnabledFlagFinder;BindEnabledFlag=Y
- fetchByKeys capability:
  - GET /fndTerritories?finder=EnabledFlagFinder;BindEnabledFlag=Y**&q=TerritoryCode IN ('AE', 'AD', 'US')**

In this case, a single ServiceDataProvider variable of type `vb/ServiceDataProvider` that multiplexes different fetch capabilities is the recommended approach. The ServiceDataProvider variable can then be used to bind to the list-of-values component.

Note:

It is recommended that service authors ensure that the service is configured to use the default business object REST API transforms.

vb/ServiceDataProvider Variables Configuration

The data returned by the service endpoint will look something like this:

```
{
  "items": [
    {
      "TerritoryCode": "AE",
      "AlternateTerritoryCode": "ar-AE",
      "TerritoryShortName": "United Arab Emirates",
      "CurrencyCode": "AED"
    },
    ...
  ],
  "count": 25,
  "hasMore": false,
  "limit": 25,
  "offset": 0,
}
```

The ServiceDataProvider variables for the fetchFirst and fetchByKeys capabilities will be configured as follows

sample-page.html	Description

`"territoriesSDPVar": {`
`  "type": "vb/ServiceDataProvider",`
`  "defaultValue: {`
`    "endpoint": "fa-crm-service/`
`getTerritories",`
`    "keyAttributes": "TerritoryCode",`
`    "itemsPath": "items",`
`    "uriParameters": {`
`      "finder":`
`"EnabledFlagFinder;BindEnabledFlag=Y"`
`    }`
`  }`
`}`
	A finder query parameter is applied to all queries going against the endpoint.
	When no capabilities are set, the ServiceDataProvider variable is assumed to support a fetchFirst capability

Configuring a JET Select-Single in Design Time

- Line 1: the value is bound to a variable that is an array of selected TerritoryCode keys.

- Line 2: the data attribute is bound to the ServiceDataProvider variable.

```
1 <oj-select-single id="so11" value="{{ $variables.selectedTerritories }}"
2               data="[[ $variables.territoriesSDPVar ]]"
3               item-text='[[ "TerritoryShortName" ]]'
4 </oj-select-single>
```

Usage: When a service provides a fetchByKeys capability, and DataProvider.containsKeys is called

The containsKeys() method can be called by components bound to a ServiceDataProvider variable that supports the 'fetchByKeys' capability. The default implementation of containsKeys() will call fetchByKeys() and return a oj.ContainsKeysResult object, as defined by the JET DataProvider contract. This implementation addresses the most common usecase.

MultiServiceDataProviderFactory

Some times it's desirable to create a standalone VB type instance programmatically by passing an initial state. In this case, the instance is not backed by a variable, that is, its state is not stored in redux. Instead the instance and/or the caller manages its state essentially. For such cases VB publishes a contract for a TypeFactory that any type author can use. See Custom Extended Types.

The TypeFactory contract is provided in the `vb/types/factories/typeFactory.js`. VB provides TypeFactory implementations for creating a ServiceDataProvider instance. Refer to the MultiServiceDataProviderFactory for details. (`vb/types/factories/multiServiceDataProviderFactory.js`)

Methods

createInstance

Returns an instance of the MultiServiceDataProvider. Refer to the JSDocs for the parameters supported on this method. The instance returned supports all methods from the DataProvider contract.

- options. The object used to instantiate the ServiceDataProvider usually contains these properties:
  - dataProviderOptions. This is its initial or 'default' state.
    - state properties are similar to the properties of a regular MultiServiceDataProvider variable
- serviceOptions. This optional configuration is needed by the RestHelper to locate the endpoint details.
- – `vbContext`. This optional configuration is needed by the RestHelpers to locate the service of an endpoint. Typically this object should be obtained from a Visual Builder API or via a callback mechanism.
  If not available, clients should pass in an object with a string property 'extensionId'. The property's value is the id of the extension executing this code (for example, the id of the extension that contains the action chain using the MultiServiceDataProvider).

Here is an example of how a caller can create an instance

Example 1-18 Create SDP

```
// create SDP
ServiceDataProviderFactory.createInstance({ dataProviderOptions: { endpoint:
"foo/getBars", responseType: "barType[]", keyAttributes: "id"} })
  .then((sdpInstance) => {
    // use SDP to create MDP instance
    MultiDataProviderFactory.createInstance({ dataProviderOptions:
{ dataProviders: { fetchFirst: sdpInstance  } } })
    .then((mdpInstance) => {
      const iter = mdpInstance.fetchFirst();
      iter.next().then((results) => {
        // process results
      });
    });
  });
```

Array Data Provider 2

Like the legacy Array Data Provider, the built-in Array Data Provider 2 can be bound to collection components.

Like ArrayDataProvider, this built-in type is a data provider implementation where the data is available as an array. All the data is set once, and the data itself can fetched from a backend service (say a list of countries), but it is assumed that array once created is static, that is, data changes infrequently or has limited and infrequent adds, updates and removes done to it.

The vb/ArrayDataProvider2 can be bound to collection components such as listView and table components. Operations on the data, such as sorts, adds, removes, and updates, are managed by the vb/ArrayDataProvider2 itself. This is different from the vb/ServiceDataProvider, where all operations generally are processed in the back end via REST calls.

ArrayDataProvider2 behaves differently from the legacy ArrayDataProvider in the following ways:

- Writes to individual properties of the ArrayDataProvider2.data are NOT allowed, and users will see an error when this occurs. Usually this happens when components use writable binding expressions that write directly to properties within individual data (array) items.
- ArrayDataProvider2 SUPPORTS using the fireDataProviderEventAction to mutate data, in addition to the assignVariablesAction.
- ArrayDataProvider2 tracks mutations to data made using fireDataProviderEventAction and notifies listeners (that is, components) of just the changes. This has the benefit of only updating the necessary parts of the UI.

A variable of this type is generally defined on the page, using the built-in type `vb/ArrayDataProvider2`.

```
{

  "variables": {
    "productListADPA": {
      "type": "vb/ArrayDataProvider2",
      "defaultValue": {
        "itemType": "application:productSummary",
        "keyAttributes": "id"
```

```
        }
      }
    }
    ...
```

ArrayDataProvider2 has several properties available.

data

The static array of data that the ArrayDataProvider2 wraps. The data property is set once when the page loads. The implicitSort criteria that the data is pre-sorted with is also set once the page loads.

keyAttributes

A string or array of string field names that represent the primary key for each row. Can be one of:

- a field name - the *key* value is a primitive or whatever the field value represents.
- an *array* of field names - the *key* will also be an array of values. For example, for keyAttributes: ['id'], when data is [{id: 'ie', name: "IE"}, {id: 'chrome', name: "Chrome"}], the corresponding keys will be [['ie'], ['chrome']]
- @value, use all properties - the *key* will also be an array of all values.
- @index, use the index as the key - the *key* will be an integer.

implicitSort

The implicit sort criteria by which the data is pre-sorted. This is an array of objects, where each object is an atomic sort expression of the form:

```
{
  "attribute": "<name of the field>",
  "direction": "<'ascending' (default) or 'descending'>"
}
```

itemType

The type of each item in the data array. This is usually a string that points to an application type or to a definition.

sortComparators

An optional object with a 'comparators' property that is either an array of arrays where each inner array has 2 items - name of the attribute that the sortCriteria applies to, and a comparator function callback that is used by ADP to sort the attribute (column), or is a Map of attribute to comparator function. This API is similar to the JET SortComparator API.

Here are some examples of configuration for array or arrays.

```
"sortComparators": {
  "comparators": [
    [
      "Category", "{{ $page.functions.alphaSort }}"
    ],
    [
      "Product", "{{ $page.functions.alphaSort }}"
    ]
```

```
  ]
}
```

Using a Map:

```
sortComparators: {
  comparators: "{{ new Map([['name', $page.functions.alphaSort]]) }}",
}
```

The comparator function will look like this:

```
var alphaSort = function (a, b) {
  return a.localeCompare(b);
}
```

textFilterAttributes

An array of attributes to filter on. See the JET documentation for ArrayDataProvider textFilterAttributes.

```
"customerListADP": {
  "type": "vb/ArrayDataProvider2",
  "defaultValue": {
    "keyAttributes": "id",
    "itemType": "flow:customer",
    "textFilterAttributes": [
      "lastName", "firstName"
    ]
  }
}
```

Features and Capabilities

ArrayDataProvider2 supports the same capabilities as the legacy ArrayDataProvider:

sort

- `{capabilityName: 'full', attributes: 'multiple}` means the endpoint has support for sorting results by one or more fields.
- null means the endpoint has no support for sorting.

Data Mutation and Refresh Events

vb/ArrayDataProvider2 notifies components when the underlying data mutates or is changed in a way that requires a refresh. The events currently supported by any iterating data providers are the 'mutate' ('add', 'remove' and 'update') event and 'refresh'. See **Assigning Data** for details.

Variable Events

All variables including vb/ArrayDataProvider2 raise the variable onValueChanged event when any of its properties change. ArrayDataProvider2 in particular will detect which of its data has changed, and will automatically notify subscribers of just the change (these are typically components that are bound to the ArrayDataProvider2 variable and have registered a listener).

Assigning Data

The data property of the vb/ArrayDataProvider2 variable is set once, when the page or component loads. The implicitSort criteria that the data is pre-sorted with is also set once the page or component loads.

After the initial load, a page author can mutate the data **either** by directly manipulating the data array using the 'assignVariablesAction' action or by using the 'fireDataProviderEventAction'.

Using a fireDataProviderEventAction, authors can mutate data property, and also notify components in one shot. When the mutation events 'add', 'remove' and 'update' are called the vb/ArrayDataProvider2 implementation will automatically mutate the underlying data, so users are not required to mutate the ArrayDataProvider2.data prior to raising this event, say, using an assignVariablesAction. This is a convenience offered only by the vb/ArrayDataProvider2 implementation, not by vb/ArrayDataProvider. See Fire Data Provider Event Action for details.

Often the mutation to the data is triggered by the UI or some other app logic, which might require the use of assignVariablesAction. This is another way to update the ArrayDataProvider2.data, in which case It's not required to use the fireDataProviderEventAction. See Assign Variables Action for details.

Note:

ADP data in a JSON file needs to be assigned a valid JSON value. ADP data that is assigned a value from the result of a previous action (for example, a call module action or REST action), must also be valid JSON. When a non-JSON value (such as JavaScript values like NaN or Infinity) is provided, you should choose the correct JSON value that should be used and then replace it. For example, the JavaScript value "NaN" can be replace by "0", which is an accepted JSON value.

Example 1-19 Where the data is literally inlined

In this example, the ArrayDataProvider2 variable productsADPB has its initial data inlined.

```
"variables": {
  "productsADPB": {
    "type": "vb/ArrayDataProvider2",
    "description": "mutations are done on 'data' property using
assignVariables",
    "defaultValue": {
      "itemType": "ProductType",
      "keyAttributes": "id",
      "data": [{
        "Amount": 30,
        "CurrencyCode": "USD",
        "Quantity": 3,
        "RegisteredPrice": 30,
        "Type": "Literal",
        "Product": "Product-Literal",
        "id": 30
      }]
    }
  }
}
```

To remove an item from the above ArrayDataProvider2 data you can use an assignVariablesAction.

- Line 16: filters the data array of productsADPB by removing the item with the matching key

```
1  "removeProductsADPB": {
2    "root": "removeFromProductsADPB",
3    "description": "",
4    "variables": {
5      "key": {
6        "type": "number",
7        "required": true,
8        "input": "fromCaller"
9      }
10   },
11   "actions": {
12     "removeFromProductsADPB": {
13       "module": "vb/action/builtin/assignVariablesAction",
14       "description": "splice returns the removed item, so filter is used 
instead, which mutates and returns the original array",
15       "parameters": {
16         "$page.variables.productsADPB.data": {
17           "source": "{{ $page.variables.productsADPB.data.filter((p) => 
p.id !== $chain.variables.key) }}",
18           "reset": "empty",
19           "auto": "always"
20         }
21       }
22     }
23   }
24 }
```

When the data is inlined or is assigned from a vbEnter action chain, you can add or update items to the array using the assignVariablesAction.

- Line 1: shows an example action where the product is updated directly
- Line 12: shows an example action where the new product is added to the tail end of the data array

```
1  "updateProductsADPB": {
2    "module": "vb/action/builtin/assignVariablesAction",
3    "description": "directly updating ADP2.data item is possible when data 
has no expression",
4    "parameters": {
5      
"$page.variables.productsADPB.data[$page.variables.productsADPB.data.findInde
x(p => p.id === $chain.variables.key)]": {
6        "source": "{{ $chain.variables.product }}",
7        "auto": "always",
8        "reset": "empty"
9      }
10   }
11 }
12 "addToProductsADPBTail": {
13   "module": "vb/action/builtin/assignVariablesAction",
```

```
14   "parameters": {
15
"$page.variables.productsADPB.data[$page.variables.productsADPB.data.length]":
 {
16       "source": "{{ $chain.results.generateNewProduct }}"
17     }
18   }
19 }
```

Example 1-20   Where the productsADPC is updated via a fireDataProviderEventAction

In this example, productsADPC has its data coming from another variable.

```
"productsADPC": {
  "type": "vb/ArrayDataProvider2",
  "description": "mutations on data can be done on the referenced 'products'
or on "
    + "the 'data' property directly. The latter will disconnect the
reference",
  "defaultValue": {
    "data": "{{ $page.variables.products }}",
    "itemType": "ProductType",
    "keyAttributes": "id"
  }
}
```

To update a specific product, you can use the fireDataProviderEventAction to set the target, data and keys properties.

- Line 28: set the event payload using the fireDataProviderEventAction

```
1  "updateProductsADPC": {
2    "root": "updateProduct",
3    "description": "updates productsADPC using data provider mutation event",
4    "variables": {
5      "product": {
6        "type": "page:ProductType",
7        "required": false,
8        "input": "fromCaller"
9      }
10   },
11   "actions": {
12     "updateProduct": {
13       "module": "vb/action/builtin/assignVariablesAction",
14       "parameters": {
15         "$chain.variables.product": {
16           "source": {
17             "Amount": "{{ $chain.variables.product.Amount *
(1+Math.floor(Math.random() * Math.floor(5))) }}",
18             "Quantity": "{{ $chain.variables.product.Quantity *
(1+Math.floor(Math.random() * Math.floor(5))) }}"
19           },
20           "reset": "none",
21           "auto": "always"
22         }
```

```
23        },
24        "outcomes": {
25          "success": "fireEventProductsADPC"
26        }
27      },
28      "fireEventProductsADPC": {
29        "module": "vb/action/builtin/fireDataProviderEventAction",
30        "parameters": {
31          "target": "{{ $page.variables.productsADPC }}",
32          "update": {
33            "keys": "{{ [ $chain.variables.product.id ] }}",
34            "data": "{{ [ $chain.variables.product ] }}"
35          }
36        }
37      }
38    }
39 },
```

Array Data Provider (Legacy)

The built-in legacy array data provider could be bound to collection components in previous versions. It should not be used in new applications.

This legacy built-in type is a data provider implementation based on the JET oj.ArrayDataProvider implementation, where the data is static. A static source of data can be fetched from a backend service, but it is assumed that it does not change frequently and only allows infrequent adds/updates and removes. This data provider can be bound to collection components such as listView and table components. Operations on the data, such as sorts, adds, removes, or updates are managed by the vb/ArrayDataProvider itself. This is different from the vb/ServiceDataProvider, where all operations generally are processed in the back end via REST calls.

New applications should use vb/ArrayDataProvider2.

The ArrayDataProvider behaves as follows:

- Writes to individual properties of the ArrayDataProvider.data are allowed. Usually this happens when components use writable binding expressions that write directly to properties within individual data (array) items.

> **Note:**
>
> It's important to remember that when you use a writable binding expression, the component writes the new value to the bound ADP.data property. This causes the ADP variable to change and the table or listview component bound to the ADP variable to refresh. If this behavior is not desired, use vb/ArrayDataProvider2 and the proper editable table / list-view patterns. (The recommended patterns are documented in the Oracle blogs.)

- ArrayDataProvider does not support using the fireDataProviderEventAction to mutate data. Instead, use the assignVariablesAction.

A variable of this type is generally defined on the page, using the built-in type `vb/ArrayDataProvider`.

```
{

  "variables": {
    "productListADPD": {
      "type": "vb/ArrayDataProvider",
      "defaultValue": {
        "itemType": "application:productSummary"
      }
    }
  }
  ...
```

The ArrayDataProvider has several properties available.

data

The static array of data that the ArrayData Provider wraps. The data property is set once when the page or component loads. The implicitSort criteria that the data is pre-sorted with is also set once the page or component loads.

idAttribute

A string or array of string field names that represent the primary key for each row. Deprecated: use **keyAttributes** instead.

keyAttributes

A string or array of string field names that represent the primary key for each row.

- a field name - the *key* value is a primitive or whatever the field value represents.
- an array of field names - the *key* will also be an array of values.
- @value, use all properties - the *key* will also be an array of all values.
- @index, use the index as the key - the *key* will be an integer.

implicitSort

The implicit sort criteria by which the data is pre-sorted. This is an array of objects, where each object is an atomic sort expression of the form:

```
{
  "attribute": "<name of the field>",
  "direction": "<'ascending' (default) or 'descending'>"
}
```

itemType

The type of each item in the data array. This is usually a string that points to an application type or to a definition.

Features and Capabilities

The ArrayDataProvider provides a sort feature:

- `{capabilityName: 'full', attributes: 'multiple}` means the endpoint has support for sorting results by one or more fields.

- null means the endpoint has no support for sorting.

Data Mutation and Refresh Events

vb/ArrayDataProvider notifies components when the underlying data mutates or is changed in a way that requires a refresh. The only way to mutate ArrayDataProvider data is via the 'assignVariablesAction' event. The 'fireDataProviderEventAction' is a no-op when it comes to updating the data property but can be used to notify just the listeners of the ArrayDataProvider (components) of the change. But the latter is not needed when assignVariablesAction is used, because it does both.

Variable Events

All variables including vb/ArrayDataProvider raise the variable onValueChanged event when any of its properties change. ArrayDataProvider in particular will detect which of its data has changed, and will automatically notify subscribers of just the change (these are typically components that are bound to the ArrayDataProvider variable and have registered a listener).

Assigning Data

The data property of the vb/ArrayDataProvider variable is set once, when the page or component loads. The implicitSort criteria that the data is pre-sorted with is also set once the page or component loads.

After the initial load, a page author can mutate the data by directly manipulating the data array using the assignVariablesAction action. Typically, the mutation to the data is triggered by the UI or some other application logic. In either circumstance, the ArrayDataProvider data needs to be manually updated. When the data property mutates, ArrayDataProvider automatically detects the change and notifies all listeners/components of the change, so that they can re-render. If the data is mutated directly, it's not required to use the fireDataProviderEvent action with the ArrayDataProvider.

Example 1-21 Where the data refers to a constant

Here the ArrayDataProvider variable productADPE gets its initial data from a constant, **productsConstant**. The ArrayDataProvider data array is initialized with one item.

```
"constants": {
  "productsConstant": {
    "type": "ProductType[]",
    "defaultValue": [{
      "Amount": 10,
      "CurrencyCode": "USD",
      "Quantity": 1,
      "RegisteredPrice": 10,
      "Type": "Constant",
      "Product": "Product-C1",
      "id": 10
    }]
  }
},
"productsADPE": {
  "type": "vb/ArrayDataProvider",
  "description": "mutations on data have to be done directly to the 'data'
property",
  "defaultValue": {
    "data": "{{ $page.constants.productsConstant }}",
    "itemType": "ProductType",
    "keyAttributes": "id"
```

```
  }
},
```

In order to add a new item to the above ArrayDataProvider data you can use an **assignVariablesAction**:

- Line 12: action that generates a new product item
- Line 22: assigns a new array with the new item appended to the existing data

It is currently not possible to add to a specific index of the array using assignVariablesAction, when the array references a constants expression.

```
 1 "addProductsADPE": {
 2  "description": "adds the generated product to the end",
 3  "variables": {
 4    "detail": {
 5      "required": true,
 6      "type": "any",
 7      "input": "fromCaller"
 8    }
 9  },
10  "root": "generateNewProduct",
11  "actions": {
12    "generateNewProduct": {
13      "module": "vb/action/builtin/callModuleFunctionAction",
14      "parameters": {
15        "module": "{{ $page.functions }}",
16        "functionName": "generateNewProduct"
17      },
18      "outcomes": {
19        "success": "assignToADPData"
20      }
21    },
22    "assignToADPData": {
23      "module": "vb/action/builtin/assignVariablesAction",
24      "parameters": {
25        "$page.variables.productsADPE.data": {
26          "source":
"{{ $page.variables.productsADPE.data.concat([$chain.results.generateNewProduc
t]) }}",
27          "reset": "empty"
28        }
29      }
30    }
31  }
32 }
```

Example 1-22 Where the data refers to another variable

In this example the ArrayDataProvider variable productADPF gets its initial data from the variable **products**. The ArrayDataProvider data array is initialized with one item.

```
"variables": {
  "products": {
    "type": "ProductType[]",
```

```
    "defaultValue": [{
      "Amount": 20,
      "CurrencyCode": "USD",
      "Quantity": 2,
      "RegisteredPrice": 20,
      "Type": "Variable",
      "Product": "Product-V1",
      "id": 20
    }]
  },
  "productsADPF": {
    "type": "vb/ArrayDataProvider",
    "description": "mutations on data can be done on the referenced 
'products' or "
     + "on the 'data' property directly. The latter will disconnect the 
reference",
    "defaultValue": {
      "data": "{{ $page.variables.products }}",
      "itemType": "ProductType",
      "keyAttributes": "id"
    }
  },
```

In order to update an item of the above ArrayDataProvider data, you can use an **assignVariablesAction**:

- Line 5: the action chain gets the updated product item
- Line 22: assign a new array to productsADPF with the updated product

```
 1 "updateProductsADPF": {
 2  "root": "assignToADPData",
 3  "description": "",
 4  "variables": {
 5    "updatedProduct": {
 6      "type": "page:ProductType",
 7      "required": true,
 8      "input": "fromCaller"
 9    },
10    "key": {
11      "type": "number",
12      "required": true,
13      "input": "fromCaller"
14    }
15  },
16  "actions": {
17    "assignToADPData": {
18      "module": "vb/action/builtin/assignVariablesAction",
19      "description": "assigning to specific item in ADP.data is not 
possible, so we replace entire array",
20      "parameters": {
21        "$page.variables.productsADPF.data": {
22          "source": "{{ $page.variables.productsADPF.data.map(p => (p.id 
=== $chain.variables.key ? $chain.variables.updatedProduct : p)) }}",
23          "reset": "empty"
24        }
```

```
25        }
26      }
27    }
28}
```

Example 1-23 Where the data is literally inlined

In this example the ArrayDataProvider variable productADPG has its initial data inlined.

```
"variables": {
  "productsADPG": {
    "type": "vb/ArrayDataProvider",
    "description": "any mutations are done on 'data' property directly",
    "defaultValue": {
      "itemType": "ProductType",
      "keyAttributes": "id",
      "data": [{
        "Amount": 30,
        "CurrencyCode": "USD",
        "Quantity": 3,
        "RegisteredPrice": 30,
        "Type": "Literal",
        "Product": "Product-Literal",
        "id": 30
      }]
    }
  }
}
```

In order to remove an item from the above ArrayDataProvider data you can use an **assignVariablesAction**. Line 16 filters the data array of productsADPG by removing the item with the matching key.

```
 1 "removeProductsADPG": {
 2   "root": "removeFromProductsADPG",
 3   "description": "",
 4   "variables": {
 5     "key": {
 6       "type": "number",
 7       "required": true,
 8       "input": "fromCaller"
 9     }
10   },
11   "actions": {
12     "removeFromProductsADPG": {
13       "module": "vb/action/builtin/assignVariablesAction",
14       "description": "splice returns the removed item, so filter is used
instead, which mutates and returns the original array",
15       "parameters": {
16         "$page.variables.productsADPG.data": {
17           "source": "{{ $page.variables.productsADPG.data.filter((p) =>
p.id !== $chain.variables.key) }}",
18           "reset": "empty",
19           "auto": "always"
20         }
```

```
21        }
22      }
23    }
24 }
```

When the data property is a literal value, to add or update items to the array it is possible to assign to a specific item of the array:

- Line 1: shows an example action where the product is updated directly
- Line 12: shows an example action where the new product is added to the tail end of the data array

```
 1 "updateProductsADPG": {
 2   "module": "vb/action/builtin/assignVariablesAction",
 3   "description": "directly updating ADP.data item is possible when data
has no expression",
 4   "parameters": {
 5
"$page.variables.productsADPG.data[$page.variables.productsADP3.data.findInde
x(p => p.id === $chain.variables.key)]": {
 6       "source": "{{ $chain.variables.product }}",
 7       "auto": "always",
 8       "reset": "empty"
 9     }
10   }
11 }
12 "addToProductsADPGTail": {
13   "module": "vb/action/builtin/assignVariablesAction",
14   "parameters": {
15
"$page.variables.productsADPG.data[$page.variables.productsADPG.data.length]":
 {
16       "source": "{{ $chain.results.generateNewProduct }}"
17     }
18   }
19 }
```

Custom Extended Types

Page authors can implement a Visual Builder type class using either the Extended Type mechanism (that extends from the `vb/types/extendedType` class module) or use the Instance Factory mechanism. The latter is much simpler to use since authors can simply plug their type into a Visual Builder variable without writing any extra JavaScript code (which was needed with the Extended Type system).

At runtime the instance of the custom type class can automatically make use of the redux framework to store its 'value' (state). Visual Builder variables generally have a type that points to a class or a type definition or can be a JavaScript primitive or an object. The Visual Builder runtime discovers built-in types and custom types by detecting a forward slash in the type name (for example, `my/ComicStripType`). The type is assumed to be a require path to a type module and loads it.

An example:

```
"myVariable": {
  "type": "my/ComicStripType",
  "defaultValue": {}
}
```

Reserved Properties

value

The state of an extended type is generally referred to as its value and its default value can be specified using the 'defaultValue' property of a variable. For example, the comicStripType specifies its default value, an Object, by providing defaults for 'name', 'publicationType' etc. Also note that charactersADP is a reference to a variable of type vb/ArrayDataProvider2.

The type of the value is defined via the '`getTypeDefinition`' function (see below). In this example, this would be the properties in the defaultValue object: `name`, `publicationType`, `publications`, etc.

In order to make the value accessible in expressions via '`<$scope>.variables.comicStripVar.value`' where `$scope` is `$page`/`$flow` etc., and '`comicStripVar`' is the type instance of the custom type that is created, '`value'` is a special property defined on the extended type instance and for this reason, will overlay any local 'value' property defined in your implementation. For this reason, take care not to use this property internally! Property accessors to read (see `getValue()` method) and write (see `setValue()` method) the value are provided.

```
"comicStripVar": {
  "type": "vb/sample/types/comicStripType",
  "defaultValue": {
    "name": "flowPage-Calvin & Hobbes",
    "publicationType": "flowPagePublicationType",
    "publications": [
      {
        "publication": "Universal Press Syndicate",
        "volumes": 24,
        "author": "Bill Watterson",
        "title": "The Doghouse",
        "year": 1987,
        "launchDate": "1985-11-18T08:00:00.000Z"
      },
      {
        "publication": "United Feature Syndicate",
        "volumes": 250,
        "author": "Bill Watterson",
        "title": "Calvin and Hobbes",
        "year": 1990,
        "launchDate": "1990-06-01T08:00:00.000Z"
      }
    ],
    "charactersADP": "{{ $variables.flow1SecondComicCharactersAdpVar }}"
  }
}
```

internalState

In addition to 'value', extended type instances are provided an 'internalState' property. Custom types can externalize their internal state so that it can be captured in redux by using this 'internalState' property. More specifically they can use property accessors to read (see `getInternalState()` method) and write (see `setInternalState()` method) the internal state are provided.

Methods

getTypeDefinition

As stated before, the type definition for the value of an extended type must be provided via the '`getTypeDefinition`' function. This method is called at the time the type instance is created. The example below returns the type definition of the state (value) of comicStripType. `name`, `publicationType`, `publications` and `charactersADP` represent its state.

```
class ComicStripExtendedType extends ExtendedType {
  getTypeDefinition(variableDef, scopeResolver) {
    let publicationsDef = 'any';
    if (variableDef.defaultValue && variableDef.defaultValue.publicationType)
{
      // responseType is specified in the defaultValue
      const { publicationType } = variableDef.defaultValue;

       if (typeof publicationType === 'string') {
        publicationsDef = `${publicationType}[]`;
      }
    }
    return {
      type: {
        name: 'string',
        publicationType: 'string',
        publications: TypeUtils.getType(`${this.getId()}:${publicationsDef}`,
          { type: publicationsDef }, scopeResolver),
        charactersADP: 'vb/ArrayDataProvider2',
      },
      resolved: true, // because we are pre-resolving type references
    };
  }
}
```

hoistValueObjectProperties
As a convenience, if the type of this variable as defined in 'getTypeDefinition' is 'object', all root properties of the values will be hoisted to the root variable type instance. This allows these properties to be accessible via expressions like '`$scope.variables.theInstance.property`'. If this is not desired, return `false` from 'hoistValueObjectProperties'.

init / activate / dispose (lifecycle methods)
A Visual Builder variable goes through various lifecycle stages. Extended type instances will be notified of these stages via the `init`, `activate` and `dispose` methods.

- **activate**
  The 'activate' method is called when this and other variables in the current scope have been created and its initial (default) values determined. This method is called right before the 'vbEnter' event and the value of the variable, and can be a good time for types to do other setup using the resolved value. It is important to note that at the time '`activate`' is

called, any value assigned, to the extended type variable or the variables it depends on, in the vbEnter action chains will not be available.

- **dispose**
  The 'dispose' method is called when the current scope is being torn down and all variables, including this variable is being disposed. This would be a good time to cleanup state for the extended type. It is important to note that any outstanding async tasks that are pending, would be the responsibility of the extended type to wind down gracefully.

handlePropertyValueChanged
When the value of an extended type variable changes (say via assignVariablesAction) it will be notified of the change via this method.

invokeEvent
Additionally, custom type implementations have the ability to fire a custom event using 'invokeEvent', providing a name, payload. For example, 'comicStripUpdate' is an event fired by the ComicStripType in the sample provided below.

getType
Custom extended types can retrieve the exploded type structure given a type definition, using the 'getType' method.

Sample Extended Type - ComicStripType

Implementation

```
'use strict';
  define(['vb/types/extendedType', 'vb/types/typeUtils'], (ExtendedType,
TypeUtils) => {
  class ComicStripType extends ExtendedType {
      getTypeDefinition(variableDef, scopeResolver) {
      let publicationsDef = 'any';
      if (variableDef.defaultValue &&
variableDef.defaultValue.publicationType) {
        const { publicationType } = variableDef.defaultValue;
          if (typeof publicationType === 'string') {
          publicationsDef = `${publicationType}[]`;
        }
      }
      return {
        type: {
          name: 'string',
          publicationType: 'string',
          publications: TypeUtils.getType(`${this.getId()}:$
{publicationsDef}`,
            { type: publicationsDef }, scopeResolver),
          charactersADP: 'vb/ArrayDataProvider2',
        },
        resolved: true, // because we are pre-resolving type references
      };
    }

    activate() {
      console.log('activate called on variable', this.id);
        const value = this.getValue();
      const { name } = value;
      const { publicationType } = value;
      const { publications } = value;
```

```
      const { charactersADP } = value;
      let charactersADPVValue;
      if (charactersADP) {
        charactersADPVValue = charactersADP.getValue();
      }

      const initialValue = {
        name, publications, publicationType, charactersADPVValue,
      };

      this.setInternalState('opStatus', 'not-started');
      console.log('initial evaluated value for variable', this.id, 'is',
finalValue);
    }

    handlePropertyVariableChangeEvent(e) {
      if (e.name.endsWith('value')) {
        if (e.diff) {
          if (e.diff.publications) {
            // process value change here
          }
        }
      }
    }

    /**
     * a sample method provided by this type that fakes a async op and
updates the internalState
     * @returns {Promise<T>}
     */
    callAsyncMethod() {
      this.setInternalState('opStatus', 'started');
        return Promise.resolve().then(() => {
        // call some other async method; set some internalState and fire an
event
        callAnotherAsyncMethod().then((res) => {
          const result = res;
          this.setInternalState('opStatus', 'completed');
          this.invokeEvent('comicStripUpdate', { status: 'success', result });
        });
      });
    }
  }

  return ComicStripType;
});
```

InstanceFactory Types

vb/InstanceFactory

With an InstanceFactory type, authors can declaratively plug in any JET type or a custom type, and use it with a special Visual Builder variable (instance factory variable). The InstanceFactory type:

- Supports creating immutable, or re-creatable (type) classes.

- Many constructs in JET are immutable classes that are then assigned to component properties. As a framework, Visual Builder facilitates the (re)creation of these classes, and reassignment when the configuration of these classes change .
- The `vb/InstanceFactory` variable takes in the JS (type) class, as well as the parameters to the constructor. When bound, this variable provides an 'instance' of the class (along with the 'constructorParams').
- When the constructor parameters change, the InstanceFactory variable will automatically create a new instance of the class.
- Like regular variables, a VB 'valueChanged' event is fired when an InstanceFactory variable changes. The event payload will have the old and new values containing the two properties `constructorParams` and `instance`.
- The 'constructorParams' of the variable alone will be serialized and persisted, not the instance. If the constructorParams includes a property that references another InstanceFactory variable, then that variable needs to be marked 'persisted', if author wants to persist the full tree. It's preferable that authors always update the 'constructorParams', so the instance is created automatically. If the instance is updated separately from constructorParams, the persisted state may not accurately reflect the correct state.

Associate a type with a variable to create an instance of that type

This example shows how you can do this.

<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td><pre>
1 "variables": {
2   "customersADP": {
3   "type" : "ojs/
ojarraydataprovider",
4   "constructorParams": []
5   }
6 },
7 "types": {
8   "ojs/ojarraydataprovider": {
9   ...
10   "constructorType": "vb/
InstanceFactory"
11   }
12 }
</pre></td><td>

Line 3: use the JET array data provider type ojs/ojarraydataprovider. This module is automatically loaded when the variable is created, because a require mapping for ojs already exists.
you must use a '/' in its name
Line 8: types declaration for ojs/ ojarraydataprovider
Line 10: indicates that instance of JET ADP is created using a vb/InstanceFactory.
An author can use the short convention, in which case the type name is assumed to be the require JS module, Or use the longer convention "<code>vb/InstanceFactory<ojs/ojarraydataprovider></code>", if the typename is different than the actual require path.

</td></tr>
</table>

Specify an array of params using the 'constructorParams' property

In this example, a JET ADP takes a data array as its first param and an options Object as its second param.

Code	Description

`1  "customersADP": {`
`2    "type" : "ojs/`
`ojarraydataprovider",`
`3    "constructorParams": [`
`4`
`"{{ $page.variables.customersData }}"`
`,`
`5      {`
`6        "keyAttributes": "id",`
`7        "textFilterAttributes": [`
`8          "lastName",`
`9          "firstName"`
`10        ]`
`11      }`
`12    ]`
`13 }`
	• Line 4: customersData is the data array
	• Line 5: options object

Create an instance of the type, when the variable 'customersADP' is created

The variable has two properties that are stored in redux.

instance

This holds the constructed ADP instance.

A component that wants an ADP instance can use it this way.

```
<oj-select-single id="ss11"
 value="{{ $variables.customerId }}"

 data="[[ $variables.customersADP.instance ]]"

 item-text='[[ $page.functions.getItemText ]]'>
</oj-select-single>
```

constructorParams

- the array of params passed to the constructor of the type
- the constructorParams can be used in EL expressions as well for readonly expressions
- `$variables.customersADP.constructorParams`

> **Note:**
>
> The properties defined on the instance can be mutated directly, and will be reflected on the instance stored in redux.
>
> The methods available on the instance can be called directly.
>
> The properties and methods supported on the instance are assumed to be declared by the type author using typescript or at design-time. This information is not relevant for runtime purposes.

To change the 'constructorParams'

Variable properties can be changed in several ways.

Using assignVariables action

For an InstanceFactory variable that is defined like this:

```
"incidentsList": {
  "type": "vb/ServiceDataProvider2",
  "constructorParams": [
    {
      "endpoint": "demo-data-service/getIncidents",
      "keyAttributes": "id",
      "itemsPath": "result",
      "uriParameters": "{{ $variables[\"technicianURIParams\"] }}"
    }
  ]
},
"incidentsListView": {
  "type": "ojs/ojlistdataproviderview",
  "constructorParams": [
    "{{ $page.variables.incidentsList.instance }}", // SDP2
    {
      "sortCriteria": [
        {
          "attribute": "priority",
          "direction": "ascending"
        }
      ]
    }
  ]
}
```

The assignVariablesAction below adds a filterCriterion property on the constructorParams of the JET ListDataProviderView variable. This assignment will cause the variable to create a new instance based on the new values.

```
"setFilterCriterion": {
  "module": "vb/action/builtin/assignVariablesAction",
  "parameters": {
    "$page.variables.incidentsListView.constructorParams[1]": {
      "source": {
```

```
          "op": "$eq",
          "attribute": "status",
          "value": "accepted"
        },
        "mapping": {
          "$target.filterCriterion": {
            "source": "$source",
            "reset": "empty"
          }
        },
        "reset": "none"
      }
    }
}
```

Using resetVariables action

For an InstanceFactory variables that is defined like above, the resetVariablesAction looks like below to reset the view variable.

```
"resetVariables": {
  "module": "vb/action/builtin/resetVariablesAction",
  "parameters": {
    "variables": [
      "$page.variables.incidentsListView"
    ]
  }
}
```

Using component writeback via EL bindings

Note:

This option is not supported.

Call methods on the instance using an action

You can use 'callVariableMethodAction' to call any method, including async methods. It's important to remember that because actions in a chain are intrinsically synchronous, a method that returns a Promise waits for the Promise to resolve before executing the next action.

```
"callGetCapabilityChain": {
  "root": "getCapabilityOnLDPV",
  "actions": {
    "getCapabilityOnLDPV": {
      "module": "vb/action/builtin/callVariableMethodAction",
      "parameters": {
        "variable": "$page.variables.incidentsListView",
        "method": "getCapability",
        "params": [
          "sort"
        ]
      }
```

```
      }
    }
  }
```

Update the instance and constructorParams together

You can use the assignVariablesAction and a built-in function to update the instance and constructorParams together.

In the following example, Line 6 uses a built-in utils called 'assignmentUtils' that provides an assignValue method. This allows authors to provide both the updated instance, and the associated constructorParams.

```
1  "assignInstanceAndCPToListViewVar": {
2    "module": "vb/action/builtin/assignVariablesAction",
3    "description": "update variable instance and constructorParams
declaratively",
4    "parameters": {
5      "$page.variables.incidentsListView": {
6        "module": "{{ $application.builtinUtils.assignmentUtils }}",
7        "functionName": "assignValue",
8        "params": [
9          {
10           "instance":
"{{ $chain.results.setFilterCriterion_priorityLow.instance }}",
11           "constructorParams":
"{{ $chain.results.setFilterCriterion_priorityLow.constructorParams }}"
12         }
13       ]
14     }
15   }
16 }
```

JET Dynamic UI Variable Types

These 'specific' variable types, specific to each JET metadata provider type, hide the 'factory' detail from the declaration.

The (requireJS) prefix 'oj-dynamic' **must** be mapped to the root of the components/ providers. Typically, this would be done using the declarative "requirejs" syntax in app-flow.json.

There is no "options" property; all properties are top-level "defaultValue" properties.

Binding Syntax

The binding for these variables is different than typical Visual Builder variables; each of these variables expose the JET metadata provider as a 'provider' property of the variable.

For example, see the "metadata" attribute below:

```
<oj-dynamic-form id="myForm" class="oj-flex-item oj-sm-12 oj-md-12"
 value="{{$page.variables.formData}}"
 metadata="[[$page.metadata.activities.provider]]">
```

vb/DynamicLayoutMetadataProviderDescriptor

The following parameters are mutually exclusive:

Parameter	Description
endpoint	A standard Visual Builder endpoint ID, in the form of <service ID> / <operationID>, in an OpenAPI3 document with appropriate JSON Schema type information.
path	A path to a JSON file, which contains a (JET-defined) JSON descriptor for the data.

```
"metadata": {
  "employee": {
      "type": "vb/DynamicLayoutMetadataProviderDescriptor",
      "defaultValue": {
        "endpoint": "sales/getAllSales"
      }
  },
  "department": {
    "type": "vb/DynamicLayoutMetadataProviderDescriptor",
      "defaultValue": {
        "path": "dynamicLayouts/some/path",
        "operationId": "get_Chickens",
      }
  }
}
```

vb/ContainerMetadataProviderDescriptor

There is no `defaultValue`.

```
"metadata": {
  "myContainerLayoutVar": {
    "type": "vb/ContainerMetadataProviderDescriptor"
  },
```

vb/HeterogeneousMetadataProviderDescriptor

Parameter	Description
discriminator	The field in the data that contains the options that can be used to determine which metadata provider to use for each new provider.

```
"metadata": {
  "incidentsProvider": {
```

```
    "type": "vb/HeterogeneousMetadataProviderDescriptor",
    "defaultValue": {
      "discriminator": "discriminatorField"
    }
  }
},
```

vb/ServiceMetadataProviderDescriptor

Parameter	Description
endpoint	A standard VB endpoint ID, in the form of <service ID> / <operationID>, in an OpenAPI3 document with appropriate JSON Schema type information.

```
"metadata": {
  "employee": {
      "type": "vb/ServiceMetadataProviderDescriptor",
      "defaultValue": {
        "endpoint": "sales/getAllSales"
      }
  }
}
```

vb/JsonMetadataProviderDescriptor

Requires that 'oj-dynamic' prefix be (requireJS) mapped to the root of the Dynamic UI Components.

The following parameters are mutually exclusive:

- **path** - path to a JSON file
- **data** - a (JS) object

```
"metadata": {
  "employee": {
      "type": "vb/JsonMetadataProviderDescriptor",
      "defaultValue": {
        "path": "path/to/some.json"
      }
  }
}
```

Default Values

Variables (but not types) may have default values.

To specify a default value:

```
"nameOfVariable": {
  "type": "string",
  "defaultValue": "someString"
},
"someOtherVariable": {
  "type": "boolean",
  "defaultValue": true"
```

```
},
"yetAnotherVariable": {
  "type": "number",
  "defaultValue": 10
}
```

Example 1-24 Object Variables

Object variables can also have default values:

```
"nameOfVariable": {
  "type": {
    "foo": "string",
    "bar": "number"
  },
  "defaultValue": {
    foo: "myDefaultFoo"
  }
}
```

Example 1-25 Object Variables That Reference An Application Type

Object variables that reference an application type can also have a default value for their properties:

```
"nameOfVariable": {
  "type": "application:myType",
  "defaultValue": {
    "foo": "myDefaultValue"
  }
}
```

Example 1-26 Arrays

Arrays can also have a default value for their properties:

```
"nameOfVariable": {
  "type": "application:myArrType",
  "defaultValue": [
    {
      "foo": "myDefaultValue"
    }
  ]
}
```

The following table shows how a variable is initialized, based on its type, when no default value is provided.

Type	Initial Value
String	Undefined
Number	Undefined
Boolean	Undefined
Any	Undefined
Object	{ }
Array	[ ]
Custom type	An empty object with all properties initialized according to this table

Expressions in Default Values

Default values may contain expressions.

When a default value contains an expression, note that expressions can also use other variables. You can reference a variable with the following syntax:

Scope	Variable Syntax
Application	$application.variables.<variableName>
Page	$page.variables.<variableName>
Action Chain	$chain.variables.<variableName>

Expressions must be wrapped in expression syntax :`{{ expr }}`. and the expression must be the entire value. Expressions can also call external functions via the page function module.

To reference another variable in a default value, you can do the following:

```
"nameOfVariable": {
  "type": "application:myType",
  "defaultValue":  {
    "foo": "{{ $application.variables.someOtherVariable }}"
  }
}
```

Since these are expressions, you can also add simple Javascript code to the values:

```
"myOtherVariable": {
  "type": {
    "someBoolProperty": "boolean"
  },
  "defaultValue": {
    "someBoolProperty": {{ $application.variables.someOtherVariable === true }}"
  }
}
```

Input Variables

Variables can also be inputs to the page.

There are two types of input. The first consists of inputs that come from the URL. The second type consists of inputs that are passed internally by the framework. To mark a variable as an input, you can use the following properties:

```
"nameOfVariable": {
  "type": "string",
  "input" "fromCaller/fromUrl"
  "required": true
}
```

Here the input is either "fromCaller" or "fromUrl". If it is "fromCaller", it will be passed internally using the params property of the navigate action. If it is "fromURL", it will be passed via the URL request parameter of the same name, like `?myVar=someValue`. If the "required" property is true, the variable value will be required to be passed during a navigation or page load.

The implicit object `$parameters` is used to retrieve the input parameter values inside the vbBeforeEnter event handler. Input variables do not exist until the vbEnter event.

In this example, the input `regionName` is retrieved using `$parameters.regionName` in the vbBeforeEnter handler and using `$page.variables.regionName` in the vbEnter handler.

```
"eventListeners": {
  "vbBeforeEnter": {
    "chains": [
      {
        "chainId": "checkForRegionName",
        "parameters": {
          "regionName": "{{ $parameters.regionName }}"
        }
      }
    ],
  },
  "vbEnter": {
    "chains": [
      {
        "chainId": "initializeVariables",
        "parameters": {
          "regionName": "{{ $page.variables.regionName }}",
          "facilityId": "{{ $page.variables.facilityId }}"
        }
      }
    ]
  }
},
```

Persisted Variables

The value of a variable can be persisted on the history, for the current session or across sessions.

If you set "persisted" to "history", the variable value is stored in the browser history. When navigating back to a page in the browser history using the browser back button or when refreshing the page, the value of the variable is restored to its value at the time the application navigated away from this page.

If you set "persisted" to "session", the variable is stored in the browser session storage as long as the browser is open. To store a variable across sessions, use "device" instead of "session".

If you set "persisted" to "device", the variable is stored in the browser local storage, so it is persisted on the device where the application is running even if the browser is closed.

To remove a variable from storage, set its value to null.

Example 1-27 Using a Persisted Variable

```
"variables": {
  "sessionToken": {
    "type": "string",
    "persisted": "session"
  }
}
```

rateLimit Variable Property

A variable can set a rateLimit property that limits how often the onValueChanged event is fired.

Specify the rateLimit property, with a timeout property in milliseconds, to limit how often the onValueChanged event is fired on that variable. For example:

```
"pageVar": {
  "type": "string",
  "onValueChanged": {...},
  "rateLimit": {
    "timeout": 1000 // in milliseconds
  }
}
```

The default is to wait for the timeout to expire after all changes stop before firing the change event.

Constants

Constants are scoped like variables, but their values can't be changed through assignment.

Constants have the following properties and restrictions:

- The scope of a constant can be page, flow, application, or action chain. The value of a constant is defined declaratively in the descriptor using the `constants` property.
- The value of a constant can be an expression. The expression can refer to previously-defined constants and variables in the current scope or application/flow.
- Constants are evaluated first, so expressions in variables can refer to constants.
- The name of a constant cannot be used by a variable in the same scope.
- Constants can be used in action chains.
- A constant can be an input parameter to a page or action chain.
- A constant cannot be of a built-in type.
- A constant holds a value that is immutable (contrary to JavaScript). For instance, in the case where the content is an object, this means the object's contents (for example, its properties) cannot be altered.
- Constants do not dispatch change events, since their values never change.

```
"constants": {
  "myConstant": {
    "type": "string",
    "description": "A useful constant",
    "defaultValue": "This string"
  }
}
```

Type

Constant type is the same as for variable except it cannot be a built-in type.

Default Value

Static Default Value. Constants hold a value that is immutable (unlike JavaScript). For instance, in the case where the content is an object, this means the object's contents (for example, its properties) **cannot** be altered. The value of a constant can be overridden in an

extension during initialization, but once the value is set, it cannot be changed. (discussed below).

Dynamic Default Value. A constant's default value can be an expression that contains variables. In this this case, the constant will change when the variable value changes. That change triggers a valueChange event that can be listened to using the `onValueChanged` property:

```
"constants": {
  "fullName": {
    "defaultValue": "{{ $variables.firstName + ' ' + $variables.lastName }}",
    "onValueChanged": {
      "chains": [
        {
          "chainId": "fullNameChanged"
        }
      ]
    }
  }
}
```

Default Value from rules. (Extensions only) Interface constants defined on certain containers, like a page or a fragment, can define a rules model that provides the values for all its interface constants. For more, see Use Rules to Derive the Value for Interface Constants.

Input

Constant input is the same as for variable.

Extension

Like variables, constants can be accessed by downstream or dependent extensions if they are defined in the `interface` section of the base container.

```
"interface": {
  "constants": {
    "extendableConstant": {
      "type": "string",
      "description": "A constant visible to extensions",
      "defaultValue": "A string"
    }
  }
}
```

Additionally, when extending a container with an interface constant, the (base) value of the constant can be changed on the extending container, using the **`defaultValue`** property, in the `extensions` section:

```
"extensions": {
  "constants": {
    "extendableConstant": {
      "defaultValue": "Value from the extension"
    }
  }
}
```

Note that the `onValueChanged` can also be overwritten. In that case, the chain(s) defined in the extension will be invoked instead the one(s) in the base object.

Use Rules to Derive the Value for Interface Constants

For interface constants defined on a flow, page, or fragment **in an extension**, you can use rules to define the constant's default value. This is similar to how business rules are used in Layouts.

To illustrate how this feature can be helpful, let's look at a real-world example. Let's say a page author is designing a shopping cart, and wishes to calculate the sales tax for an item added to a cart. To store the sales tax, the author defines a constant `salesTax`.

Determining the sales tax is a complex calculation that depends on several variables, for example, the country, city, or state where the item is bought, and the tax rates applied to different types of goods. This makes calculating the correct sales tax something that can be challenging to express using a simple expression. A page author could use an action chain that reacts to the 'valueChanged' property for the variables to perform the calculations, but action chains like this can be time consuming to configure. Action chains are also not extendable, making it difficult to delegate their evaluation to a downstream customer extension.

This is where having a separate rules engine provides a simple and convenient way to define rules that:

- Determine the value for one or more constants, and
- Establish a predictable lifecycle where the rules are run as needed, in response to state changes.

The rules are run in order of priority, such that values from higher priority rules take precedence over lower priority rules. First, each rule condition is tested to see whether it will provide a value for one or more constants during the current run. The participating rules (the rules whose conditions evaluated to true) are then executed, and the values for the interface constants that are provided by rules are gathered. If multiple rules provide values for the same constant, then the value from a higher priority rule is selected over a lower priority one. Rules defined in a customer extension have higher priority over the rules defined in the base (or factory) extension. Within the same rules model, the rules are listed from highest to lowest priority.

Using rules model to define values for constants

A page author may choose to use rules to determine the values for all the interface constants defined in the base. To do this, the author should:

- Define a top-level property `options` on the page model with the sub-property `constantsRules` with the value set to `on`. This informs the framework that rules are enabled for the current container's interface constants, as long as the container allows rules. For example, layout containers do not support constants rules.
- Define a `-rules.json` file (described below).
- Remove the `defaultValue` property set on all interface constants, and instead configure rules that return the same value.

> **Note:**
>
> An error will be reported if a `defaultValue` property is configured and rules are turned "on". The constant can have other properties configured as shown below.

For example, the interface constants in a base extension's `page.json` with rules might look like this:

```
{
  "pageModelVersion": "2504.0.0",
  "title": "allRules-page",

  "options": {
    "constantsRules": "on"
  },
  "interface": {
    "constants": {
      "baseStateTax": {
        "type": "number"
      },
      "salesTax": {
        "type": "number"
      },
      "CATax": {
        "type": "number"
      }
    }
  }
}
```

A customer extending the base page above can do the following if they want to define rules for the constants in their extension:

- Define a top-level property `options` on the page model with the sub-property `constantsRules` with the value set to `on`. This informs the framework that rules are enabled for the current container's interface constants, as long as the container allows rules. For example, layout extensions do not support constants rules but page-x and fragment-x do.
- Define a `-rules-x.json` file (described below).
- Remove all constant definitions in the sub-property `constants` under `extensions` of the page-x model, and instead define default value rules for the extended constants. For example, *page*`-x.json` in a customer's extension might look like the example below, where no 'extensions constants' are defined.

> **Note:**
>
> The empty object in this example is shown for clarity, and is not required.

```
{
  "pageExtensionModelVersion": "2501.10.0",
  "description": "Extension page",

  "options": {
    "constantsRules": "on"
  },

  "extensions": {
    "constants": {}
```

```
  }
}
```

> **Note:**
>
> Each extension can choose to configure rules or keep the current `defaultValue` configuration.

Defining and Running Rules

An example of `cart-page.json` used in a shopping cart page is defined below. In this example there are three interface constants: `baseStateTax`, `salesTax`, and `CATax`.

```
{
  "options": {
    "constantsRules": "on"
  },

  "interface": {
    "constants": {
      "baseStateTax": {
        "type": "number"
      },
      "salesTax": {
        "type": "number"
      },
      "CATax": {
        "type": "number"
      }
    }
  },
  "variables": {
    "address": {
      "type": {
        "addressLine": "string",
        "city": "string",
        "county": "string",
        "state": "string",
        "zip": "number"
      }
    },
    "city": {
      "type": "string",
      "defaultValue": "{{ $variables.address.city }}"
    },
    "county": {
      "type": "string",
      "defaultValue": "{{ $variables.address.county }}"
    },
    "state": {
      "type": "string",
      "defaultValue": "{{ $variables.address.state }}"
    },
```

```
    "purchaseDate": {
      "type": "string",
      "description": "the date when items are bought"
    }
  }
}
```

Their values are provided by `cart-page-rules.json`, which is shown below. The suffix '-rules', identifies it as a rules artifact associated with the cart-page.

> **Note:**
>
> For the most part, the structure of the rule definition below mirrors that for layout business rules.

```
{
  "rules": [
    {
      "id": "salesTaxRuleWithDate",
      "description": "calculates state tax if purchase date falls earlier
than 2020",
      "condition": {
        "expression": "{{ $modules.utils.checkYear($variables.purchaseDate) <
2020 }}",

        "referencedContext": {
          "generated": [
            "$variables.purchaseDate"
          ],
          "extra": []
        }
      },

      "overlay": {
        "$constants.salesTax":
"{{ $modules.utils.getSalesTaxForPrior($variables.state, $variables.county, $v
ariables.city) }}"
      }
    },
    {
      "id": "salesTaxRule",
      "description": "calculates state tax for when state or county or city
properties change",
      "condition": {
        "expression": "{{ $variables.state || $variables.county
|| $variables.city  }}",
        "referencedContext": {
          "generated": [
            "$variables.state",
            "$variables.county",
            "$variables.city"
          ],
          "extra": []
```

```
          }
        },

        "overlay": {
          "$constants.baseStateTax":
"{{ $modules.utils.getBaseTaxForState($variables.state) }}",
          "$constants.salesTax":
"{{ $modules.utils.getSalesTax($variables.state, $variables.county, $variables
.city) }}"
        }
      },
      {
        "id": "taxDefaultsRule",
        "description": "provides base values",

        "condition": {
          "referencedContext": {
            "generated": [
              "$constants.CATax"
            ]
          }
        },
        "overlay": {
          "$constants.baseStateTax": "{{ $constants.CATax }}",
          "$constants.salesTax": "{{ $constants.CATax }}"
        }
      },
      {
        "id": "CATax-defaultValueRule",
        "description": "default value expression in rules for CATax",

        "overlay": {
          "$constants.CATax": 6.25
        }
      }

  ]
}
```

Rules property

Multiple rules are defined and listed in order of priority, from highest to lowest, under the `rules` property, and the rules are executed from lowest priority to highest priority. Each rule is identified by an "id" (for example, `salesTaxRuleWithDate`, `salesTaxRule`). Each rule is run until values for all constants have been determined. A single rule has the following properties:

- `id`: a string identifier
- `description`: a string description
- `condition`: an object containing two properties:
  - `expression`: optional boolean, whether the rule can be executed or not. The conditional expression can call a module function, but it cannot return a Promise.
  - `referencedContext`: array of variables and constants that cause a rule to be part of a rules engine execution when their values change. This array, when set, can be a fast way to include a rule for the current run.

    * `generated` is generally set by the design time.
    * `extra` is provided by the rules author.

- `overlay`: an object containing the interface constants (static values or expressions) as its properties. In the example above, some rules provide values for `$constants.salesTax`, whereas others for `$constants.CATax`. When multiple values for the same constant are provided, then the value from the higher priority rule is used. The source expressions (the rhs expressions) used in the overlay can call a module function, but it cannot return a Promise, or be async methods. For example, make a Rest fetch.

> **Note:**
>
> If an "overlay" has multiple constants mapping, where the value of one constant depends on the other, it is recommended that separate rules be defined as the order that constants would be evaluated within a single overlay cannot be guaranteed.

The following table describes the rules in the example above.

Rule	Description
`CATax-defaultValueRule`	This rule provides a default value for the `CATax` constant. This rule has no conditions set, so it is always run.
`taxDefaultsRule`	This rule provides values for the constants `baseStateTax` and `salesTax`, using the current value for `CATax`. It has a "referencedContext" property set, that depends on another constant `CATax`. This means that any time the value of the interface constant changes, this rule is likely to run. The value for `CATax` happens to be provided by a lower priority rule (`CATax-defaultValueRule`) in the current run. In such cases, the value determined by this rule for the `CATax` constant is used by *taxDefaultsRule*, in determining the values for `baseStateTax` and `salesTax`.
`salesTaxRule`	This rule is slightly more complex where the "condition" is tested (for boolean true), before the values for `salesTax` and `baseStateTax` constants are returned via "overlay". Let's say that during init the "address" variableis empty. When this is the case then the condition evaluates to false, and the rule is skipped. When the "address" variable updates, causing `city` or `state` or `county` to be updated, then this rule is executed because one of its listed `referencedContext` variables has changed, meaning the condition now evaluates to true. Because tax calculations are complex, module methods `getBaseTaxForState()` and `getSalesTax()` are used. They use the state, county, and city information to determine the right tax values.

Rule	Description
`salesTaxRuleWithDate`	This rule uses the purchase date in addition to the address to determine the tax values. It defines a "condition" that uses a module function method to validate the purchase date. If the provided date is earlier than 2020, the rule is run to determine the `salesTax` constant, and returned via "overlay".

Rules execution behavior

Generally, rules are executed during initialization at the beginning of the lifecycle, and every time the 'rules context' changes. Supported containers are flow, page, and fragment.

During init, the sub-property `expression` under `condition` alone is checked before a rule is selected for execution. `condition` can also be a plain string expression or boolean.

After the initial values are determined, any time a referenced variable or constant that is listed in the `referencedContext` sub-property of `condition` changes, then the rules engine is re-executed, and the new list of participating rules determined, by running each rule in order of priority.

At the end of a run, the final values of (interface) constants are updated on the associated scope.

Note:

During execution, both the `referencedContext` and condition `expression` are checked to see whether the rule participates. For instance, if the valueChange is for a variable that is not present in the `referencedContext` the rule is skipped. Otherwise, the condition is checked before a rule is considered for the run. As stated before, when multiple participating rules provide values for the same constant(s), then the value evaluated by the rule with higher priority is used over the value provided by a lower priority rules.

What context is available within rules?

VB business rules can use all context properties that are generally available to a container. For example, `$variables`, `$constants`, `$modules` from the current scope, or properties from an outer scope such as `$flow`, and `$application` can be used.

Note:

This is done for backwards compatibility with existing customer extensions pages that may have `defaultValue` expressions already defined that refer to any available scope properties. If the customer chooses to combine the `defaultValue` expressions from all interface constants into rules, they must be able to do so without having to redefine new expressions.

In addition, rules associated with a customer's extension page can access the interface constants or variables from its base (via `$base.constants`, `$base.variables`).

Each rules engine run is synchronous, so at the beginning of the run, a snapshot of the current context is used when evaluating expressions in rules.

Evaluating Rules at Init time

Generally, values returned by the rules, defined both in factory and customer extensions, are taken as the initial values for the constants.

When a customer extension has a `defaultValue` property set on its extended constants (meaning, there are no rules defined), then these values win over any rule values from the base because customer overrides in an extension always have priority over the base. This ensures current pages are backwards compatible.

Evaluating Rules after ValueChange

Post-init, the rules engine executes every time the value for any of the `referencedContext` variables or constants changes. In the example below, ExtA defines the interface constants `mode`, `greeting`, `name`, and `anonUser`, whose values are provided by rules. `$variables user`, along with other scope properties for the current page, are also accessible from rules.

ExtA: page.json	ExtA: page-rules.json

ExtA: page.json

```
{
  "options": {
    "constantsRules": "on"
  },

  "interface": {
    "constants": {
      mode: {
        type: "string"
      },
      salutation: {
        type: "string"
      },
      anonUser: {
        type: "string"
      },
      "greeting": {
        "type": "string"
      }
    },
    "variables": {
      "user": {
        "type": "userType",
        "defaultValue":
"{{ $application.variables.user }}",
        "mode": "readOnly"
      }
    }
  },
  "types": {
    "userType": {
      "name": "string",
      "title": "string",
      "role": "string"
    }
  }
}
```

ExtA: page-rules.json

```
{
  "rules": [
    {
      "id": "extA/anonRule",
      "condition": {
        "expression":
"{{ $variables.user.name === ''
|| $variables.user.name ===
undefined }}",
        "referencedContext": {
          "generated": [
            "$variables.user"
          ]
        }
      },
      "overlay": {
        "$constants.mode": "oracle-
public",
        "$constants.greeting":
"{{ $constants.salutation + ' '
+ $constants.anonUser + '!' }} "
      }
    },

    {
      "id": "extA/nonAdminRule",
      "condition": {
        "expression":
"{{ $variables.user.name
&& $variables.user.role !==
'admin' }}",
        "referencedContext": {
          "generated": [
            "$variables.user"
          ]
        }
      },
      "overlay": {
        "$constants.mode": "oracle-
internal",
        "$constants.greeting":
"{{ $constants.salutation + ' '
+ $functions.toTitleCase($variables.u
ser.name) + '!' }} "
      }
    },

    {
      "id": "extA/adminRule",
      "condition": {
        "expression":
```

<table>
<tr><th>ExtA: page.json</th><th>ExtA: page-rules.json</th></tr>
<tr><td></td><td><pre>
"{{ $variables.user.name
&& $variables.user.role ==
'admin' }}",
        "referencedContext": {
          "generated": [
            "$variables.user"
          ]
        }
      },
      "overlay": {
        "$constants.mode": "oracle-
restricted",
        "$constants.greeting":
"{{ $constants.salutation + ' '
+ $functions.toTitleCase($variables.u
ser.name) + '!' }} "
      }
    },

    {
      "id": "extA/defaultValuesRule",
      "overlay": {
        "$constants.anonUser": "Jane
Doe",
        "$constants.salutation":
"Hello"
      }
    }
  ]
}
</pre></td></tr>
</table>

Extending Rules in Extensions

An extension page can already extend (interface) constants from its base extension. Now, an extension container (page-x) can additionally define its own rules that provide values for the extended constants. Factory rules cannot themselves be extended by a customer extension. Instead, new rules can be introduced that tweak the values of the extended constants.

Any constants, variables, and generally all scope properties that are available to extensions can be used in rules.

Rules created in extensions are run before the base rules are.

To take the example above, a customer extension page extends the interface constant `salutation`, providing a locale-specific value. (For example, "Hola Señora (| Señor | "). The correct salutation is based on the user's gender.

Either the `defaultValue` property on the constant can be set, or the extension can define a rules model to provide the value. Both configurations are shown, and the behavior is the same if the expression or value configured on the constant's `defaultValue` is used as is in the rules model.

ExtB: `page-x.json` - with only defaultValue properties and rules disabled

```
{
  "extension": {
    "constants": {
      "salutation": {
        "defaultValue":
"{{ $functions.getLocaleSalutation($base.variables.user) }}"
      }
    }
  }
}
```

A customer extension that wishes to use rules instead, will have the configurations below:

ExtB: `page-x.json` - with extended constants defined in rules

```
{
  "options": {
    "constantsRules": "on"
  },

  "extension": {},
  ...
}
```

and

ExtB: `page-rules-x.json`

```
{
  "rules": [
  {
      "id": "extB/defaultsRule",
      "condition": {
        "referencedContext": {
          "generated": [ "$base.variables.user" ]
        }
      },
      "overlay": {
        "$base.constants.salutation":
"{{ $functions.getLocaleSalutation($base.variables.user) }}"
      }
    }
  ]
}
```

In this example, the rules file defines a `condition` with a `referencedContext` sub-property that includes the variable `user` referenced in the expression (that was set as the `defaultValue` property). This is needed if it is required for the rule to participate every time the `user` variable changes.

Additionally, the `overlay` object assigns a value to the base constant `$base.constants.salutation` from ExtA.

The table below assumes that rules are configured for the page on both extensions. When the page loads, the user is not logged in, so an anonymous user, Jane Doe, is used. Then Mia, a sales rep, logs in, causing the variable `user`, defined on extA, to change. Then she changes her role to 'admin'.

Event	Constant	Evaluated Value	Rule Value

`User not logged in at init. $variables.user is not set`	`mode`
	`salutation`
	`anonUser`
	`greeting`
		`oracle-public`
		`Hola`
		`Jane Doe`
		`Hola Jane Doe!`
			`from extA`
			`from extB. Overrides rule value in extA`
			`from extA`
			`from extA`

`User Mia logs in. $variables.user is updated and rules are re-run`	`mode`
	`name`
	`salutation`
	`greeting`
		`oracle-internal`
		`mia`
		`Hola Senora`
		`Hola Senora Mia!`
			`from extA`
			`from extA`
			`from extB. rule amends value based on user's gender`
			`from extA`

`Mia changes $variables.user.role = "admin". Rules are re-run`	`mode`
	`salutation`
	`greeting`
		`oracle-restricted`
		`Hola Senora`
		`Hola Senora Mia!`
			`from extA`
			`from extB. rule runs again as a condition is not set`
			`from extA`

Note:

Generally, while evaluated rule values are made available to higher priority rules, in the example above, it is noteworthy that the higher priority rule from extB (defaultsRule) provides the value for the constant `salutation`, this value is available to a lower priority rule (extA/adminRule) that is lower in the run order, as an optimization. All final rule values for the interface constants are assigned to the (page) constants at the end, after the rules have run.

Items to remember about configuring rules in interface constants:

- Only interface constants are eligible for rules evaluation.
- All interface constants defined on the base container, must either have its values come from rules, or have the `defaultValue` property set. An error is flagged if both are present.
- An extension container that extends one or more interface constants must also either configure a `defaultValue` property on all constants, or provide values for the same from a rules model.
- An extension container may choose to provide values (for constants) from rules even if the base container does not have rules.

Usage Scenarios

Let's consider these simple examples involving rules and their behavior.

> **Note:**
>
> For readability, the definitions are condensed.

In this example, ExtC depends on ExtB, which depends on ExtA.

ExtA	ExtB	ExtC

page.json
 	
`{`
`  "options/`
`constantsRules": "on",`
`  "interface/`
`constants": {`
`    foo: { "type":`
`"string" },`
`    bar: { "type":`
`"string" },`
 	
`    lucy: { "type":`
`"string" },`
`    charlie: { "type":`
`"string" }`
`  }`
`}`
	page-x.json
	
	`{`
	`  "options/`
	`constantsRules": "on",`
	`  "extension/`
	`constants": {`
	`    foo: { "type":`
	`"string" },`
	`    bar: { "type":`
	`"string" },`
	
	`    charlie: { "type":`
	`"string"  }`
	`  }`
	`}`
		page-x.json
		
		`{`
		`  "extension/`
		`constants": {`
		`    foo:`
		`{ "defaultValue":`
		`"override from c" },`
		
		`    lucy:`
		`{ "defaultValue": "" }`
		`  }`
		`}`

<table>
<tr><th>ExtA</th><th>ExtB</th><th>ExtC</th></tr>
<tr><td>page-rules.json
<pre>
{
    "rule-a1": {
      "foo": "r1 from a"
    },
    "rule-a2": {
      "foo": "r2 from a",
      "bar": "r2 from a"
    },
    "rule-a3": {
      "foo": "r3 from a",
      "bar": "r3 from a",
      "lucy": "r3 from a"
    },
    ...
    "rule-a10": {
      "condition": false
      "lucy": "r10 from a"
    }
}
</pre></td><td>page-rules-x.json
<pre>
{
  "rule-b1": {
    condition: false,
    bar: "r1 from b"
  },
  "rule-b2": {
    foo: "r2 from b"
  },
  "rule-b3: {
    condition: false,
    lucy: "r3 from b" // Error!
  },
  "rule-b4": {
    charlie: "r4 from b"
  }
}
</pre></td><td>No page-rules-x.json present</td></tr>
</table>

In the table below, the final value/expression for the interface constants defined above when evaluated in base and customer extensions are marked in **bold**.

A constant value from the leaf extension (ExtC) has higher priority over a value provided by a lower extension. The value itself can come from either the `defaultValue` configuration or from rules. Rules are executed progressively from lowest priority to highest priority, so a value from a leaf extension gets a higher priority.

- ExtC provides a value for constants (`$base.constants.foo`, `$base.constants.lucy`) that overrides rule values from extension ExtB.
- ExtC does not provide values for other constants (`bar` and `charlie`), so values from lower priority extensions are used.

Expression	Rules Values in ExtA	Rules Values in ExtB	Rules Values in ExtC	Final Value	Why?
$base.constants.foo	"r1 from a"	"r2 from b"	"override from c"	**"override from c"**	ExtB value overwrites ExtA
$base.constants.bar	"r2 from a"	-		**"r2 from a"**	
$base.constants.lucy	"r3 from a"	-	"	**"**	topmost `defaultValue` is always used it for backwards compatibility.

Expression	Rules Values in ExtA	Rules Values in ExtB	Rules Values in ExtC	Final Value	Why?
$base.constants.charlie	-	"r4 from b"	-	**"r4 from b"**	

JavaScript Action Chains

A JavaScript action chain is a sequence of actions started by an event. When a given event occurs in a page, the event listener listening for that event kicks off the action chain.

For information about JavaScript action chains, see *Work with JavaScript Action Chains* in one of these guides:

- Extending Oracle Cloud Applications with Visual Builder Studio
- Building Responsive Applications with Visual Builder Studio

JavaScript Actions

This section lists the built-in JavaScript actions that are available in Visual Builder for creating JavaScript action chains.

Assign Variable

This action is used to assign a value to a local, page, flow, or application variable. It can also be used to create a local variable.

When using code to reference a variable, the scope must be specified, unless it's defined in the current scope. If the scope isn't specified, it means it's the current scope. For example, on a page, the page's variables would be referenced as `$variables.myVar` instead of `$page.variables.myvar`.

Call Action Chain

This action is used to start an action chain that has been defined in the same page, flow, or application.

Note:

You can call a JSON action chain from a JavaScript action chain using this action; however, you can't call a JavaScript action chain from a JSON action chain.

To call an action chain, you need to pass the following parameters:

Parameter Name	Description
chain	The name of the action chain to call. No prefix is required for page level action chains, but application level ones need to be prefixed with `application:` and flow level ones with `flow:`.
params	An expression that maps to an array of parameters.

Here's an example of a call to an action chain with 2 input parameters:

```
const callChainResult = await Actions.callChain(context, {
        chain: 'MyActionChainToCall',
        params: {
              ip1: $application.variables.var1,
              ip2: $application.variables.var2
        },
});
```

Return Values

The call returns a result if the called action chain returns a result.

Call Component

A Call Component action provides a declarative way to call methods on JET components.

Here are details about this action's parameters:

Parameter Name	Description
selector	The component on the page that is to be called. A component must have its ID parameter specified for it to show up in the drop-down list. You can also use the DOM method `document.getElementById` to locate a JET element/component.
method	The name of the component method to call.
params	Array of parameters to pass to the method, if it takes arguments. Primitives, objects, and array parameters are passed by value and not by reference. Instances are still sent as references.

Here's an example of a call to the Call Component action:

```
      const callTableComponentRefreshRowResult = await
Actions.callComponentMethod(context, {
        selector: '#tableComponent',
        method: 'refreshRow',
        params: $variables.rowID,
      });
```

Call Function

This action is used to call a function defined for the current flow, page, or application (web app). For extensions, it's used to call a function defined for the current flow, page, App UI, or extension. These functions are referred to as *module functions*, and they're created and edited using the JavaScript editor for a particular scope.

Using code, the syntax for specifying a module function that isn't a page function is:

```
$<scope>.functions.someFunction();
```

For page functions, the scope isn't specified:

```
$functions.someFunction();
```

Example
Suppose this function is defined for a page:

```
sum(num1, num2) {
   return num1 + num2;};
};
```

You call the function and assign its result like this:

```
const sumResult
= $functions.sum($variables.firstNum_pv, $variables.secondNum_pv);
```

For extensions, you can create global functions, which would be available to all App UIs in the extension, and referenced using this syntax, `<artifact-scope>.modules.<module-id>.<function-name>`. For example:

```
$application.modules.calculator.add
```

For more about using global functions in extensions, see Add Global Functions.

Return Values

The result payload is equivalent to whatever the function returns (which may be undefined if there is no return). If the function returns a promise, the result payload will be whatever is resolved in the promise.

Call REST

The call REST action is used to make a REST call in conjunction with the service definitions.

Internally, this action uses the REST Helper, which is a public utility. Its parameters are as follows.

Parameter Name	Description
endpoint	The endpoint ID as defined in the service configuration.
uriParams	A key/value pair map that will be used to override path and query parameters as defined in the service endpoint.
body	A structured object that will be sent as the body.
requestType	The content-type of the request, either 'json', 'form', or 'url'. **Note:** Note that this is deprecated. Instead, use 'contentType' and 'fileContentType'.
headers	An object; each property name is a header name and value that will be sent with the request.
contentType	An optional string value with an actual MIME type, which will be used for the "content-type" header. When used with "fileContentType", this is also used as the type for the File blob.

Parameter Name	Description
responseType	If set, the specified type is used to do two things at run-time: • Generate a fields parameter for the REST URI to limit the attributes fetched; • Automatically map the fetched response to the response type (when used with the built-in vb/BusinessObjectsTransform). This applies to standard action chains. See the definition for "responseType" in Service Data Provider Properties for details on how the assigned type is used in that context.
filePath	An optional path to a file to send with the request. If "contentType" is set, that is used as the type for the File contents. If “contentType” is not set, a lookup of common file extensions will be used.
filePartName	Optional, used with filePath to allow override of the default name ("file") for the FormData part.
fileContentType	An optional string, used in combination with "contentType", "multipart/form-data", and "filePath".
hookHandler	Used primarily by vb/ServiceDataProvider when externalizing data fetches. See Service Data Provider for details.
requestTransformOptions	A map of values to pass to the corresponding transform, as the "options" parameter.
requestTransformFunctions	A map of named transform functions, called before making the request, where the function is: *fn(configuration, options)*
responseTransformFunctions	A map of named transform functions, called before making the response, where the function is: *fn(configuration, options)*
responseBodyFormat	A string that allows an override of the standard Rest behavior, which normally looks for a “content-type” header to determine how to read and parse the response. Possible values are "text", "json", "blob", "arrayBuffer", "base64", "base64Url", and "formData".
responseFields	This is an "advanced" field, for use specifically with JET Dynamic Forms. The value would typically be a variable that is bound to the `<oj-dynamic-form>` "rendered-fields" attribute. This is how a calculated layout can tell the Rest Action call which fields to fetch. **Note**: The vb/BusinessObjectsTransform transform is necessary to create a query from this value. **Note**: When "responseFields" is provided, "responseType" is ignored.

Using multipart/form Data

If you have set "contentType" to "multipart/form-data", the Call REST action interprets your request "body" object as the form parts. Each property of the body object is a form part, which is a key-value pair with its own content type and disposition.

If "filePath" is also set, it is added as an additional part using the lookup of common file extension types.

If "filePath" is also set, it is added as an additional part using the sample simple file extension type association. The name of this part is "file", or can be specified using "filePartName".

You may optionally override the file type by using "fileContentType" for the file part.

For more about working with the multipart/form-data format, refer to this Oracle blog, Consuming REST APIs in VB - multipart/form-data.

Parameters Typically Required per Endpoint Type

These are the typically required parameters for each endpoint type:

- POST:
  - `body` parameter is set to the variable containing the new record's data.
  - `uriParams` parameter is used to provide any required input parameters.

  Here's an example POST endpoint call:

```
const callRestCreateIncident = await Actions.callRest(context, {
  endpoint: 'fixitfast/putIncident',
  body: $variables.incidentPayload ,
  uriParams: {
    id: $constants.incidentId,
  },
});
```

- GET:
  - `uriParams` parameter is used to provide any required input parameters, such as an ID input parameter to get a single record.
    Here's an example of a GET endpoint call to get a single record. The `empIDToGet_ip` variable is an input parameter that passes the record's ID to the action chain that contains this Call REST call:

```
const getEmployeeResult = await Actions.callRest(context, {
  endpoint: 'businessObjects/get_Employee',
  uriParams: {
    'Employee_Id': empIDToGet_ip,
  },
});
```

- DELETE:
  - `uriParams` parameter is used to provide the ID of the record to delete.
    Here's an example of a DELETE endpoint call to delete a record. The `empIDToDelete_ip` variable is an input parameter that passes the record's ID to the action chain that contains this Call REST call:

```
      const callRestBusinessObjectsDeleteEmployeeResult = await
Actions.callRest(context, {
        endpoint: 'businessObjects/delete_Employee',
        uriParams: {
          'Employee_Id': empIDToDelete_ip,
        },
      });
```

- PATCH:
  - `body` parameter is set to the variable containing the record with the updated data.
  - `uriParams` parameter is used to provide the ID of the record to update.

  Here's an example PATCH endpoint call:

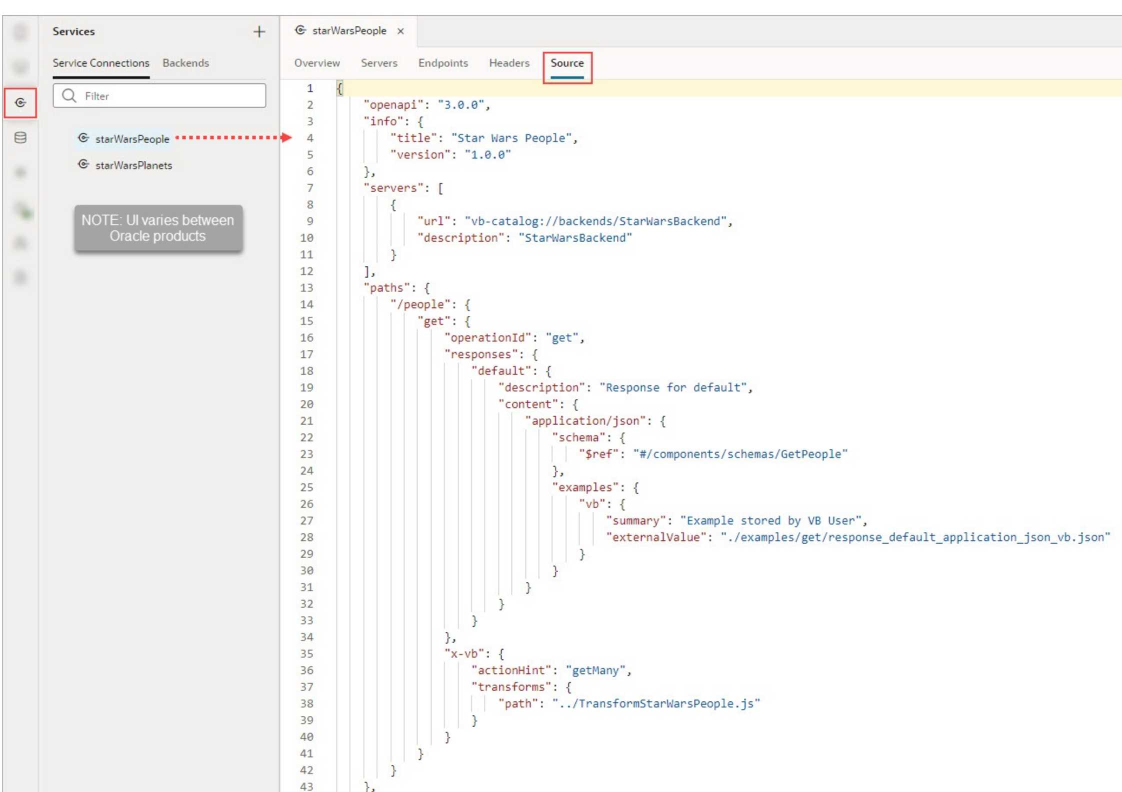
```
const updateEmployeeResult = await Actions.callRest(context, {
  endpoint: 'businessObjects/update_Employee',
  uriParams: {
    'Employee_Id': $variables.empID_pv,
  },
```

```
      body: $variables.EmpUpdatedData_pv,
    });
```

Service Definitions

If your service connection details are static, the details, such as the server, path, and schema of the request and response, are stored in the `openapi3.json` file for the service connection.

To view or edit a service's definition, select the service connection in the **Services** pane, then open the **Source** tab. The editor uses the OpenAPI3 specification and JSON format.



Transforms

The *requestTransformOptions, requestTransformFunctions*, and *responseTransformFunctions* can be used to modify the request and response. Some built-in service endpoints have built-in transform functions for 'sort', 'filter', 'paginate', and 'select', so options for these transform functions can be defined using the same name via the *requestTransformOptions* property. For third party services, the options set are based on the type of transform functions supported.

When using the Rest Action, the transform names have no semantic meaning and all request and response transforms are called.

Request and response transform functions have the following signatures.

Transform Type	Parameters	Return Value

Request	/**
	 * configuration: {
	 * url:
	 * initConfig: {
	 * method: // string with http method
	 * body: // request body, if any
	 * credentials: // string see (fetch) Request
	 * headers: // object, map of strings
	 * }
	 * },
	 *
	 * options: provided by the application
	 *
	 * context: an empty object, which exists for the
	 * lifetime of one REST call, a set of
	 * transforms share this.
	 **/
	
	mytransform(configuration, options, context)
		Configuration object; see "Parameters".
		Typically, returns the same object passed in, or a modified one.

Response	/** * response: { body, headers } * * context: an empty object, see "Request transforms" * */ myresponsetransform(response, context);	The return value is application-defined. The value is returned as the 'transformResults' of the REST call result: /** * { * response: The (fetch) Response object. Note that the body has already * been read, so the functions (ex. json()) cannot be called. * * body: the result of the json()/text()/etc. * * transformResults: a map of return values from Response Transforms * } */

Example 1-28 A Simple Transform Function

You would assign request and response transform functions to a Call REST action when interacting with a third-party service or when you need to override a business object's auto-generated transforms (`vb/BusinessObjectTransforms`). Otherwise, you should assign transform functions to a backend, service connection, or endpoint.

In this example, the transform functions are defined in the page module and assigned to a Call REST action. The arguments are automatically passed to the module functions.

```
const allCities = await Actions.callRest(context, {
  endpoint: 'businessObjects/getall_City',
  requestTransformOptions: {
  sort: "?orderBy=cityName:desc",
  },
  requestTransformFunctions: {
    sort: $page.functions.sort2
  },
  responseTransformFunctions : {
    transform: $page.functions.transformCityNames
  },
});
```

pass options parameter to sort module function

Here are the transform functions defined in the page module, located on the page's JavaScript tab:

```
class PageModule {
    transformCityNames(result) {
        let tr = {};
        if (result.body) {
            tr = result.body.items;
            for (let i = 0; i < tr.length; i++) {
              tr[i].cityName = tr[i].cityName + " (city)";
            }
        }
    }

    sort2(configuration, options) {
      configuration.url = configuration.url + options;
      return configuration;
    }
}
```

Error Handling and Return Values

If the underlying REST API request returns a status code, the error object is returned for you to handle the error yourself, otherwise an auto-generated error notification is shown.

The object returned by the Call REST action returns these results:

Result	Relevant Properties of Returned Object	Returned Object

Success	If the returned object's ok property is set to true, indicating success, these are the object's relevant properties:
	• body: object with results from the call (scalar, object, array, etc.)
	• headers: Headers object
	• ok: boolean, set to true
	• status: number, set to 200
	• statusText: string, set to "OK"
		{
		  body {},
		  error: null,
		  headers: Headers {},
		  message: {summary: ' '},
		  ok: true,
		  status: 200
		  statusText: "OK",
		}
		
		*If a single record is returned, it is contained in the body{} object; if multiple records are returned, they are contained in the body{} object's results parameter.

Error	If the returned object's ok property is set to false, indicating failure, these are the object's relevant properties:
	• error: error object or null
	• message: object with error summary
	• ok: boolean, set to false
	• status: number
	• statusText: string showing type of error
		{
		  body: null,
		  error: null,
		  headers: Headers {},
		  message: {summary: '<error summary>'},
		  ok: false,
		  status: <status number>,
		  statusText: '<error type>'
		}

For details about working with business objects, refer to Accessing Business Objects Using REST APIs.

Call Variable

This action is used to call a method of an InstanceFactory variable that has been defined in the current scope (flow, page, or application). Using this action with any other type results in an error.

You can call any method on the current instance associated with the InstanceFactory variable, including asynchronous ones. However, since actions are by design synchronous, this action will wait for the asynchronous call to resolve before proceeding to the next action in the chain.

Here's an example of a call to an InstanceFactory variable's method:

```
const getRangeResult = $variables.myBook.instance.getRange($variables.range);
```

Return Values

The result payload is equivalent to whatever the function returns (which may be undefined if there is no return). If the function returns a promise, the result payload will be whatever is resolved in the promise.

Code

In the Design editor, the Code action is used to add JavaScript code to an action chain. To do so, add the **Code** action from the Action pallet to the action chain and enter the code in the Properties pane.

In the Code editor, you can use this action to create a local function.

Fire Data Provider Event

The Fire Data Provider Event action causes the DataProvider specified via the `target` parameter to dispatch an `oj.DataProvider` event as a way to notify all listeners registered on that DataProvider to react to changes to the underlying data. For example, a component using a particular ServiceDataProvider may need to render new data because new data has been added to the endpoint used by the ServiceDataProvider.

The action can be called either with a mutation or a refresh event. The refresh event is used to re-fetch and re-render all data, and the mutation event is used to specify which changes to show.

> **Note:**
>
> This action is not necessary for a VB Array Data Provider variable, since the data array of an ADP variable, exposed via the `data` property, can be updated directly using the Assign Variable action. Assigning the data array is automatically detected by Visual Builder, and all listeners are notified of this change. Users will be warned of this when the fireDataProviderEvent is used with an ADP, prior to mutating the `data` property directly.

A mutation event can include multiple mutation operations (add, update, remove) as long as the ID values between operations do not intersect. This behavior is enforced by JET components. For example, you cannot add a record and remove it in the same event, because the order of operations cannot be guaranteed.

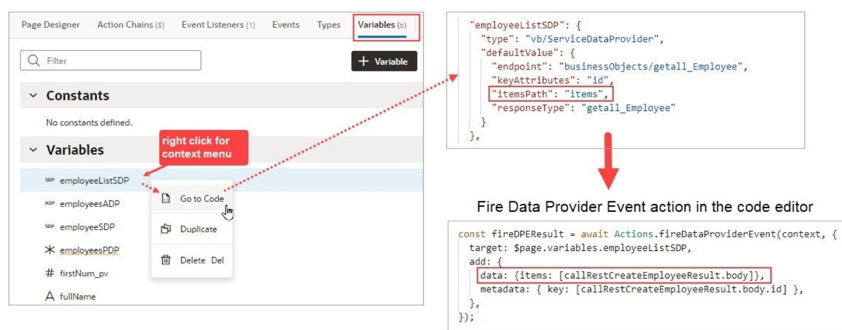
This table provides details about the parameters for the Fire Data Provider Event action. For further details, see DataProviderOperationEventDetail in Oracle JET API Reference.

Name	Type	Description
target	string	Target of the event, usually a variable of type vb/SDP. Example: `target: $variables.employeeSDP`

<table>
<tr><th>Name</th><th>Type</th><th>Description</th></tr>
<tr><td>refresh</td><td>null</td><td>Indicates a data provider refresh event needs to be dispatched to the data provider identified by the target. A null value is specified because the refresh event does not require a payload.
Example:<pre>
await Actions.fireDataProviderEvent(context, {
  target: $variables.employeeListSDP,
  refresh: null,
});
</pre>For further details, see DataProviderRefreshEventDetail in Oracle JET API Reference.</td></tr>
</table>

Name	Type	Description
add	object	The following properties may be present in the payload:

- **data**: `Array<Object>`; required. Passes the added records from the add operation's returned result. If you are using an SDP variable, the structure of the data passed to this parameter must match the structure specified by the `itemsPath` parameter of the SDP variable's definition:



The SDP's `itemsPath` property specifies where the added records are in the response payload, relative to the root of the response. Here are three different structures for an add operation's response and the corresponding `itemsPath` specification:

1. Added records (just one in this example) are provided as an array, at the root of the response:

```
[
    {
      "id": 149,
      "firstName": "Qinqin",
      "lastName": "Han"
    }
]
```

The `itemsPath` specification for this case is: `"itemsPath": ""`

Here's an example of the `data` parameter for the Fire Data Provider Event action for this case:

```
      //Add new employee record
      const callRestCreateEmployeeResult = await
Actions.callRest(context, {
         endpoint: 'businessObjects/create_Employee',
         body: $variables.newEmpData,
      });

      const fireDPEResult = await
Actions.fireDataProviderEvent(context, {
         target: $variables.employeeListSDP,
         add: {
           data: [callRestCreateEmployeeResult.body],
           keys:
[callRestCreateEmployeeResult.body.id],
```

Name	Type	Description

		`metadata: [{key: callRestCreateEmployeeResult.body.id,}], }, });`
		2. Added records (just one in this example) are provided in an array, which is in an object's property, such as this object's `items` property:
		`{ "items": [ { "id": 149, "firstName": "Qinqin", "lastName": "Han" } ], "count": 1, "hasMore": false, "offset": 0 }`
		The `itemsPath` specification for this case is: `"itemsPath": "items"`
		Here's an example of the `data` parameter for the Fire Data Provider Event action, for this case:
		`//Add new employee record const callRestCreateEmployeeResult = await Actions.callRest(context, { endpoint: 'businessObjects/create_Employee', body: $variables.newEmpData, }); const fireDPEResult = await Actions.fireDataProviderEvent(context, { target: $variables.employeeListSDP, add: {` **`data: {items: [callRestCreateEmployeeResult.body]},`** `keys: [callRestCreateEmployeeResult.body.id], metadata: [{key: callRestCreateEmployeeResult.body.id,}], }, });`

Name	Type	Description

3. Added records (just one in this example) are provided as an array in a nested structure that matches the `itemsPath` property. In this example, the added records are in the `bar` property of this object's `foo` property:

```
{
  "foo" : {
    "bar" : [
        {
          "id": 149,
          "firstName": "Qinqin",
          "lastName": "Han"
        }
      ]
  }
}
```

The `itemsPath` specification for this case is: `"itemsPath": "foo.bar"`

Here's an example of the `data` parameter for the Fire Data Provider Event action code for this case:

```
      //Add new employee record
      const callRestCreateEmployeeResult = await
Actions.callRest(context, {
         endpoint: 'businessObjects/create_Employee',
         body: $variables.newEmpData,
      });

      const fireDPEResult = await
Actions.fireDataProviderEvent(context, {
         target: $variables.employeeListSDP,
         add: {
            data: {foo: {bar:
[callRestCreateEmployeeResult.body]}},
            keys:
[callRestCreateEmployeeResult.body.id],
            metadata: [{key:
callRestCreateEmployeeResult.body.id,}],
         },
      });
```

- **keys**: `Set<*>`; required for optimal performance. Ensure that the `keyAttributes` parameter is set for the SDP variable. Here's an example value for this parameter:

  ```
  keys: [callRestCreateEmployeeResult.body.id],
  ```

- **metadata**: `Array.<`*`ItemMetadata.<KeyValue>`*`>`; required for optimal performance. Passes the key values of the added records. Here's an example value for this parameter:

  ```
  metadata: [{key: callRestCreateEmployeeResult.body.id,}],
  ```

Name	Type	Description
		• **addBeforeKeys**: `Array<keys>`; optional. Array of keys for items located after the items involved in the operation. They are relative to the data array, after the operation was completed, and not to the original array. If null and the index is not specified, then insert at the end. • **indexes**: `Array<number>`; optional. Indexes of items involved in the operation, relative to after the operation completes and not to the original dataset. Indices are with respect to the DataProvider with only its implicit sort applied. For further details, see DataProviderAddOperationEventDetail in Oracle JET API Reference.

remove		Only the **keys** parameter is required to identify the records. For details about the keys parameter, refer to the add event above.
		Example:
		`await Actions.fireDataProviderEvent(context, {`
		`        target: $variables.employeeSDP,`
		`        remove: {`
		`           keys: [$variables.productId],`
		`        },`
		`      });`
		For further details, see DataProviderMutationEventDetail in Oracle JET API Reference.

update		The update event's payload is similar to that of the add event, except `addBeforeKeys` is not present.
		Example:
		`await Actions.fireDataProviderEvent(context, {`
		`        target: $variables.employeeSDP,`
		`        update: {`
		`          data: {items: [callRestUpdateEmployeeResult.body]},`
		`          keys: [callRestCreateEmployeeResult.body.id],`
		`          metadata: [{key: callRestCreateEmployeeResult.body.id,}],`
		`        },`
		`      });`
		For further details, see DataProviderMutationEventDetail in Oracle JET API Reference.

Fire Event

This action allows you to fire a custom event that has been defined in your application, flow, page or fragment, using the Events tab. A custom event can carry a payload that you define when you create the event, and the payload is passed to the event using the Fire Event action.

Here's a quick overview of how a custom event and the Fire Event action are used:

1. Create a custom event, defining parameters if required.
2. Create an event listener, which can start more than one action chain:
   a. Assign it the custom event
   b. Create a new action chain for the event, which is launched when the event is triggered. Create the action chain through the Event Listener tab, because if the listener's custom event has input parameters, the action chain is created with an `event` input parameter. This `event` object will contain the custom event's input parameters (example:

`event.param1`, `event.param2...`), and the `event` object is automatically passed to the new action chain..

3. In the action chain that will trigger the event, use the Fire Event action to trigger the custom event, providing any parameters defined for the event.

This table describes the parameters for the Fire Event action:

Parameter Name	Description
event	Name of custom event, defined in your application, that you want to invoke.
payload	Event's payload; source can be a page variable, a specific value or an expression.

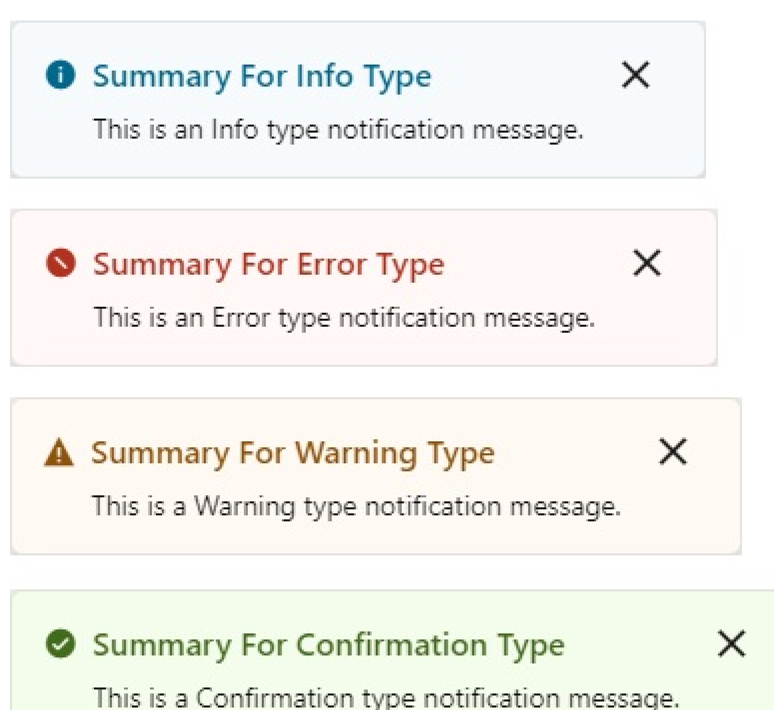
Here's an example of a call to the Fire Event action:

```
const fireApplicationEventNavigateToItemResult = await
Actions.fireEvent(context, {
    event: 'application:customEventToFire,
    payload: {
       item: $application.variables.varForCustomEvent,
    },
});
```

Fire Notification

This action is used to fire a "vbNotification" event to display a message to the user in the browser.

There are four types of notifications: Info, Error, Warning, and Confirmation. They display a summary and a message underneath:



"vbNotification" events are just like custom events, except that they have a defined name and a suggested use. Notifications are generally intended to help implement a flexible message display, but the specific use can be defined by the application. See Custom Events for details.

Here are details about this action's parameters:

Parameter Name	Description
summary	Summary/title to display in large, bold letters for notification.
message	Message to display in notification.
displayMode	Display mode: persist or transient
type	Notification type: info, error, warning, or confirmation

Here's an example of a call to the Fire Notification action:

```
await Actions.fireNotificationEvent(context, {
     message: $variables.message,
     summary: $variables.summary,
     displayMode: 'persist',
     type: 'error',
});
```

For Each

This action lets you execute one or more actions for each item in an array.

Here are details about this action's parameters:

Parameter Name	Description
items	An expression that evaluates to the array that is to be looped.
item	The default alias for the current item in the array; can be changed as desired.
index	The default alias for the index position; can be changed as desired.
mode	Defines whether the actions are run serially (default) or in parallel. Regardless of the mode, the For Each action does not complete until the actions for each item in the items array are complete.

The "mode" parameter allows for serial or parallel action. The default is serial, for which each "actionId" call is only made for an item when any previous item's "actionId" call finished (meaning, any Promise returned from the last action resolves). Using "parallel" means that each "actionId" call does not wait for the previous call to finish (useful for Rest Action calls, etc). Using either mode, the For Each action does not finish until all Promises returned from the "actionId" chain resolve (if no Promise is returned, it is considered resolved on return).

The following table describes additional properties injected into the available contexts that the called action ('callee') can reference in its parameter expressions:

Parameter Name	Description
$current.data	The current array item.
$current.index	The current array index.
alias.data	An alternate syntax for `$current.data`, which allows a reference to `$current` from nested contexts.

Parameter Name	Description
alias.index	An alternate syntax for `$current.index`, which allows a reference to `$current` from nested contexts.

Return Values

On success, an array is returned with each element containing the return value from the last action in the loop, from each iteration. For instance, if the loop contains two actions that return results, actionA → actionB, and the loop iterates 5 times, the returned array will have 5 elements, each corresponding to an iteration and containing actionB's result from that iteration.

Get Dirty Data Status

The Get Dirty Data Status action is used to check if any of the values have changed for the tracked variables within a particular scope (application, page, fragment, layout, flow), within any contained pages, fragments, layouts, or flows, or within any extensions of them. If the value of one of the tracked values changes, the Dirty Data status for the variable's scope changes from '`notDirty`' to '`dirty`'. The Dirty Data status is returned for the scope that this action is used in.

When checking the dirty data status of a particular scope and its subscopes, it’s the scope from which the action chain is called that matters, not the scope in which the action chain is defined. For instance, if a page event initiates a flow or a page action chain that has a Get Dirty Data Status action, the Get Dirty Data Status action returns that page's dirty data status, because the action chain is called from the page.

This action has no parameters to set. Also, this functionality works with all of the data types, except Service Data Providers (SDPs). Currently, you'll have to handle the tracking of value changes for SDPs.

To set a variable to be tracked for value changes, go to the relevant Variables tab, select the variable, and in the Properties pane, set its Dirty Data Behavior property to '`Track`'.

To reset the scope's Dirty Data status back to '`notDirty`', use the Rest Dirty Status action.

Here's a sample action chain that uses this action, which is started by a vbBeforeExit event listener for the page:

```
    async run(context) {
      const { $page, $flow, $application, $constants, $variables } = context;

      const getDirtyDataStatusResult = await
Actions.getDirtyDataStatus(context, {
      });

      if (getDirtyDataStatusResult.status === 'dirty') {
        // Warn the user if there are unsaved changes
        await Actions.fireNotificationEvent(context, {
          summary: 'You have unsaved changed. Please Save or Cancel',
          displayMode: 'transient',
          type: 'error',
        });

        // Stay on the page
        return { cancelled: true };
```

```
      }

      /* Navigation from this page can be canceled by returning an object
with the property cancelled set to true.
        This is useful when the page state is dirty and navigation should not
be allowed before saving.*/
        return { cancelled: false };
    }
```

Get Location

The Get Location action provides a declarative access to geographical location information associated with the hosting device. This action requires the user's consent. As a best practice, it should only be fired on a user gesture, so as to associate the permission prompt with the action they just initiated.

Here are details about this action's parameters:

Parameter Name	Description
maximumAge	A positive long value indicating the maximum age in milliseconds of a possible cached position that is acceptable to return. If set to **0**, it means that the device cannot use a cached position and must attempt to retrieve the real current position. If set to **Infinity**, the device must return a cached position regardless of its age.
timeout	A positive long value representing the maximum length of time, in milliseconds, that the device is allowed to take in order to return a position. The default value is **Infinity**, meaning that `getCurrentPosition()` won't return until the position is available.
enableHighAccuracy	A boolean that indicates the application would like to receive the best possible results. If true, and if the device is able to provide a more accurate position, it will do so. This can result in slower response times or increased power consumption. If **false** (the default value), the device can save resources by responding more quickly or using less power. On mobile devices, **enableHighAccuracy** should be set to true in order to use GPS sensors.

If the geolocation API is supported in the browser, geolocationAction returns a JSON Position object that represents the position of the device at a given time.

Return Type	Description	Example

Object	The Position interface represents the position of the concerned device at a given time. The position, represented by a Coordinates object, comprehends the 2D position of the device, on a spheroid representing the Earth, but also its altitude and its speed.
	• *Position.coords* returns a Coordinates object defining the current location.
	• *Position.timestamp* returns a DOM timestamp representing the time at which the location was retrieved.
		Latitude and longitude can be accessed from the Position's coordinates as follows:
		`[[results.getCurrentLocation.coords.latitude ]]`
		`[[results.getCurrentLocation.coords.longitude ]]`
		where *getCurrentLocation* is a geolocationAction.

If geolocation is not supported by the browser, or a parameter with a wrong type is detected, an error is returned by `results.getCurrentLocation.error`. If a PositionError occurs when

obtaining geolocation, a `PositionError.code` payload is returned. Possible `PositionError.code` values are:

- PositionError.**PERMISSION_DENIED**
- PositionError.**POSITION_UNAVAILABLE**
- PositionError.**TIMEOUT**

For every failure, a descriptive error message can be obtained from the action chain, such as *[[results.getCurrentLocation.error.message]]*.

Here's an example of using the Get Location action:

```
const getLocationResult = await Actions.geolocation(context, {
  timeout: 0,
  maximumAge: Infinity,
});

if (getLocationResult.getCurrentLocation.error != null)  {
  await Actions.assignVariable(context, {
    variable: '$variables.coords_pv',
    value: getLocationResult.coords,
  });
} else {
  await Actions.fireNotificationEvent(context, {
    message: getLocationResult.getCurrentLocation.error.message,
    summary: 'Error',
  });
}
```

If

The If action is used to add conditions.

Login

This action launches the login process as defined in the Security Provider implementation.

It invokes the `handleLogin` function on the Security Provider with the `returnPath` argument.

This table describes the parameters for the Login action:

Parameter Name	Description
returnPath	The path of the page to go to after a successful login. If not defined, uses the default page of the application.

The behavior of the default implementation of the Security Provider `handleLogin` function is:

- Navigate to the login URL specified by the Security Provider configuration.
- If `returnPath` is not defined, use the default page of the application.
- Convert the page `returnPath` to a URL path and add it to the login URL.

Here's an example of a call to the Login action:

```
await Actions.login(context, {
        returnPath: '/loginpage',
      });
```

Logout

This action launches the logout process as defined in the Security Provider implementation.

It invokes the `handleLogout` function on the Security Provider with the `logoutUrl` argument.

This table describes the parameters for the Logout action:

Parameter Name	Description
logoutUrl	The URL to navigate to in order to logout. If not defined, uses the logout URL of the Security Provider configuration.

The behavior of the default implementation of the Security Provider `handleLogout` function is:

- Navigate to the URL defined by the `logoutURL` parameter.
- If the `logoutUrl` parameter is not defined, use the logout URL of the Security Provider configuration.
- After the user is logged out, the application continues to the default page of the application.

Here's an example of a call to the Logout action:

```
await Actions.logout(context, {
     logoutUrl: $variables.logoutURL_pv,
});
```

Navigate Back

The Navigate Back action is used to return to the previous page in a browser's history.

This table describes the parameters for the Navigate Back action:

Parameter Name	Description
params	A key/value pair map that will be used to pass parameters to the page.

Here's an example of a call to the Navigate Back action, in which two parameters are passed:

```
await Actions.navigateBack(context, {params: {
     inParam: $variables.var1,
     inParm1: $variables.var2,
   },
});
```

Navigate To Application

The Navigate To Application action is used to navigate to a navigable page or flow in a specified App UI, and if required, to pass parameters to the page or flow. For a page or flow to

be navigable, meaning you can navigate to it from a different App UI, that page or flow must be set as navigable, as will be explained for this action's `flow` and `page` parameter.

This table describes the parameters for the Navigate To Application action:

Parameter Name	Description
application	The application (App UI) to navigate to.
history	Set the effect on the browser history. Allowed values are: `replace`, `skip` and `push`. If set to `replace`, the current browser history entry is replaced, meaning that the Back button won't go back to that URL. If the value is set to `skip`, the URL is not modified. Default is `push`.
flow	The flow within the selected App UI to navigate to. Only flows that have their "Let other App UIs navigate to this flow" setting enabled are available in the dropdown list.
page	The page within the selected flow to navigate to. Only pages that have their "Let other App UIs navigate to this page" setting enabled on their Settings tab can be navigated to
params	An object with the parameters to pass to the application, if required.

Here's an example of a call to the Navigate To Application action, in which an input parameter is passed to the application:

```
      const navToEmployeeAppResult = await
Actions.navigateToApplication(context, {
        application: 'employee-app',
        history: 'replace',
        flow: 'empflow',
        page: 'empflow-main',
        params: {
          empID: empID,
        },
      }, { id: 'navToEmpApp' });
```

Navigate to the Same Application with Different Input Parameters
Navigating to the same application but with different input parameters is considered a valid navigation, and since the input parameters change, the `onValueChanged` event is triggered.

The navigation is pushed to the browser history, so pressing the browser's Back button restores the previous values of the input parameters.

Navigate To Flow

For base apps (web apps), this action is used to navigate to a flow in the current application, and if necessary, to pass parameters to the flow.

For App UIs, this action is used to navigate to a flow in the current App UI, and if necessary, to pass parameters to the flow. To navigate to a flow in a different App UI, use the Navigate to Application action.

This table describes the parameters for the Navigate To Flow action:

Parameter Name	Description
target	Specifies if the flow is for the current or parent page.
flow	Flow within the current App UI to navigate to.
page	Page within the flow to navigate to.
params	An object with the parameters to pass to the flow, if required.
history	Define the effect on the browser history. Allowed value are 'replace', 'skip' or 'push'. If the value is 'replace', the current browser history entry is replaced, meaning that back button will not go back to it. If the value is 'skip', the URL is not modified. Default is 'push'.

Here's an example of a call to the Navigate To Flow action, in which two input parameters are passed to the flow:

```
const navFlowResult = await Actions.navigateToFlow(context, {
    target: 'parent',
    flow: 'main',
    params: {
      inParam: $variables.var1,
      inParm1: $variables.var2,
    },
    page: 'main-start',
    history: 'push',
});
```

Navigate To Page

For base apps (web apps), this action is used to navigate to a page in the current application, and if necessary, to pass parameters to the page.

For App UIs, this action is used to navigate to a page in the current App UI, and if necessary, to pass parameters to the page. To navigate to a page in a different App UI, use the Navigate to Application action..

This table describes the parameters for the Navigate To Page action:

Parameter Name	Description
page	The page within the current application (web apps) or App UI to navigate to.
params	An object with the parameters to pass to the page, if required.
history	Define the effect on the browser history. Allowed value are 'replace', 'skip' or 'push'. If the value is 'replace', the current browser history entry is replaced, meaning that back button will not go back to it. If the value is 'skip', the URL is not modified. Default is 'push'.

Here's an example of a call to the Navigate To Page action, in which an input parameter is passed to the page:

```
const navigateResult = await Actions.navigateToPage(context, {
    page: 'main-display-results',
    params: {
      calResults: $variables.calculationResults,
    },
    history: 'push',
});
```

Navigate to the Same Page with Different Input Parameters
Navigating to the same page but with different input parameters is considered a valid navigation, and since the input parameters change, the `onValueChanged` event is triggered.

The navigation is pushed to the browser history, so pressing the browser's Back button restores the previous values of the input parameters.

Open URL

The Open URL action is used to navigate to an external URL. In a web app, this action opens the specified URL in the current window or in a new window using the `window.open()` API. In a native mobile app, this action can open local file attachments as well as remote resources.

In a native mobile app, this action supports opening local file attachments as well as remote resources. Allowed file types for the `url` parameter are:

- `.pdf`
- `.doc`
- `.txt`
- `.text`
- `.ppt`
- `.rtf`
- `.xls`
- `.mp3`
- `.mp4`
- `.csv`

The very first time, the user gets an option to select which application to use for opening a given file type. If no application is available to open such a file, this action fails with the appropriate error. After a file is first opened, it will always be opened with the same application across all Visual Builder installed apps on the device.

If the specified file is not local or if the file extension is not recognized, this action will use Cordova's plugin `cordova-plugin-inappbrowser` to open the specified URL.

This table describes the parameters for the Open URL action:

Parameter Name	Description
url	The URL to navigate to.
params	A key/value pair map that will be used as query parameters to the URL

Parameter Name	Description
hash	The hash entry to append to the URL.
history	Defines the effect on the browser history. Allowed values are 'replace' or 'push'. If the value is 'replace', the current browser history entry is replaced, meaning that the back button will not go back to it. Default is 'push'.
windowName	A name identifying the window as defined in the `window.open()` API (optional). If not defined, the URL opens in the current window. Otherwise, refer to the `window.open()` API documentation. In a mobile app, there are 3 possible values: `_self`, `_blank`, or `_system`. The default is `_self`. Refer to the documentation for `cordova-plugin-inappbrowser`. For local file types, this parameter is ignored.

Once on the URL location, the browser back button will re-enter the last page, if you specified a value for the `windowName` parameter that opens the URL in the current window. The page input parameters will be remembered, even if their type is 'fromCaller'.

Here's an example of a call to the Open URL action, in which one parameter is passed:

```
await Actions.openUrl(context, {
    url: $variables.urlToOpen_pv,
    params: {
      inParam: $variables.itemID,
    },
    hash: $variables.hashPart_pv,
    history: 'push',
    windowName: '_self',
});
```

Reset Dirty Data Status

The Reset Dirty Data Status action is used to reset the Dirty Data status of the scope (application, fragment, flow, page) that the action is used in to '`notDirty`'. The Dirty Data status of a scope changes from '`notDirty`' to '`dirty`' when one of its tracked variables has its value changed.

This action takes no parameters, and it is used with the Get Dirty Data Status action.

Here's an example of a call to this action:

```
await Actions.resetDirtyDataStatus(context, { });
```

Reset Variables

The Reset Variables action is used to reset variables to their default values, as defined in their variable definitions.

This table describes the parameters for the Reset Variables action:

<table>
<tr><th>Parameter Name</th><th>Description</th></tr>
<tr><td>variables</td><td>An array of variables. Here is an example:
<code>["$page.variables.var1", "$page.variables.var2"]</code>

 Note:
If a single variable expression is provided instead of an array, it is implicitly treated as an array of one variable.

Each expression in the array has to resolve to a variable or variable property, and variables must be prefixed with their scope:
• <code>$application.variables</code>
• <code>$page.variables</code>
• <code>$chain.variables</code>
Each expression should be followed by a variable name or a path to a variable property. For example:
• <code>$application.variables.a</code>
• <code>$page.variables.a.b</code>
• <code>$variables.a.b.c</code> (which is shorthand for <code>$chain.variables.a.b.c</code>)</td></tr>
</table>

Here's an example of a call to the Reset Variables action, in which two variables are to be reset:

```
await Actions.resetVariables(context, {
   variables: [
     '$page.variables.firstNum_pv',
     '$page.variables.secondNum_pv',
   ],
 }, { id: 'resetFirstAndSecondNum' });
```

Return

The Return action is used to return a payload for an action chain and to return control back to where the action chain was called. For instance, action chain A can call action chain B, which returns a value, then action chain A can use that returned value for further processing.

The Return action can also be used to exit an action chain early due to an exception, such as an invalid value, or some other condition. If no value is returned by the Return action, the value of undefined is returned by default.

For the Run In Parallel action, which uses `aysc()` functions to run blocks of code in parallel, the Return action can be used to return a value for a block of code. For further details, see Run in Parallel.

Run in Parallel

The Run in Parallel action is used to run multiple action chains in parallel, and it can also be used to wait for their results to produce a combined result.

The actions to run for each sequence are placed within an `asyn()` method, and the value returned by the `asyn()` method is put into the array returned by the Run in Parallel action. The first element of the returned array contains the result from the first `asyn()` method, the second element contains the result from the second `asyn()` method, and so on.

Here's an example of the Run in Parallel action, which returns its results in an array named `empInfo`. In parallel, the action makes REST calls to get an employee's office location, department, and team. The employee's information is then displayed:

```
async run(context, { office_ip = 1, department_ip = '1', team_ip = 2 }) {
      const { $page, $flow, $application, $constants, $variables } = context;

      const empInfo = await Promise.all([
        async () => {

          const callRestGetOfficesResult = await Actions.callRest(context, {
            endpoint: 'businessObjects/get_Offices',
            uriParams: {
              'Offices_Id': office_ip,
            },
          });

          return callRestGetOfficesResult.body.location;
        },
        async () => {

          const callRestGetDepartmentResult = await Actions.callRest(context,
{
            endpoint: 'businessObjects/get_Department',
            uriParams: {
              Department_Id: department_ip,
            },
          });

          return callRestGetDepartmentResult.body.name;
        },
        async () => {
          const callRestBusinessObjectsGetTeamResult = await
Actions.callRest(context, {
            endpoint: 'businessObjects/get_Team',
            uriParams: {
              'Team_Id': team_ip,
            },
          });

          return callRestBusinessObjectsGetTeamResult.body.name;
        },
      ].map(sequence => sequence()));

      await Actions.fireNotificationEvent(context, {
        summary: 'Employee Info',
        message: 'LOCATION: ' + empInfo[0] + ' DEPARTMENT: ' + empInfo[1] + '
TEAM: '+ empInfo[2],
      });
    }
```

Return Values

This action returns an array (`empInfo`) with the first element (index 0) containing the value returned from the first `asyn()` method, the second element containing the value from the second `asyn()` method, and the third element containing the value from the third `asyn()` method.

Scan Barcode

Use the Scan Barcode action in your mobile application to scan QR codes and barcodes for details such as URLs, Wi-Fi connections, and contact information.

The parameters for this action are:

Parameter Name	Description
image	An image object, which can be a CanvasImageSource, Blob, ImageData, or an `<img>` element
formats	Optional: A series of barcode formats to search for, for example, one or more of the following: `['aztec', 'code_128', 'code_39', 'code_93', 'codabar', 'data_matrix', 'ean_13', 'ean_8', itf', 'pdf417', 'qr_code', 'upc_a', 'upc_e']` Note that all formats may not be supported on all platforms. If `formats` is not specified, the browser will search all supported formats, so limiting the search to a particular subset of supported formats may provide better performance.
convertBlob	Optional: A boolean that enables you to automatically convert a Blob to an ImageBitmap when using the Scan Barcode action to process the outcome of the Take Photo action. If `true`, the Blob object is converted as an ImageBitmap before being passed to the Scan Barcode action. If `false` (default), the Blob object is left as is. You'll need to manually do the conversion, for example, by adding a function to your application and calling the function using the `callModuleFunctionAction` in your action chain.

Here's an example of a call to the Scan Barcode action, in which a bitmap returned by a module function is used for the Image parameter:

```
      const scanCreateImageBitmapResultResult = await
Actions.barcode(context, {
        image: createImageBitmapResult,
        formats: [
          'qr_code',
        ],
      });
```

Return Values

On success, a DetectedBarcode object is returned using the auto-generated variable shown by the **Store Results In** parameter. If the browser does not support the Shape Detection API or if a specified format is not supported, an exception is thrown.

Share

The Share action is used to invoke the native sharing capabilities of the host platform in order to share content with other applications, such as Facebook, Twitter, Slack, SMS and so on.

Invoke this action following a user gesture, such as a button click. Also, we recommend that the Share action's UI only be shown if `navigator.share` is supported by the given browser, as in this HTML code:

```
<oj-button disabled="[[!navigator.share]]">Share</oj-button>
```

This table describes the parameters for the Share action:

Parameter Name	Description
title	Represents the title of the document being shared. This value may be ignored by the target.
text	Text that forms the body of the message being shared. Can be specified with or without a URL.
url	URL string that refers to the resource being shared. Any URL can be shared, not just URLs under website's current scope.

Here's an example of a call to the Share action:

```
     await Actions.webShare(context, {
        title: document.querySelector('h1').textContent,
        text: 'Check out this cool new app!',
        url: document.querySelector('link[rel=canonical]') &&
document.querySelector('link[rel=canonical]').href || window.location.href,
     });
```

Switch

Use the Switch action to select the actions to execute for a specific case value. If a case value is not matched, the "default" case is executed.

Here's an example of a Switch code block that returns a language's three letter code:

```
switch (language) {
  case 'English':
    return 'eng';
    break;
  case 'Chinese':
    return 'chn';
    break;
  case 'Spanish':
    return 'spn';
    break;
  default:
    return 'error';
    break;
}
```

Try-Catch-Finally

This action is used to add Try, Catch, and Finally blocks in order to gracefully handle errors and avoid program crashes.

JSON Action Chains

A JSON action chain is a sequence of actions started by an event. When a given event occurs in a page, the event listener listening for that event kicks off the action chain. Each JSON action chain is contained within its own JSON file, which is created and edited using the Action Chain editor.

JSON Actions

A list of built-in actions, JSON based, available in Visual Builder for applications

Note:

Action definitions minimally have a "module" property that specifies the action implementation. Actions can also have an optional "label" property, which is user-friendly.

Assign Variables Action

This action is used to assign values to a set of variables.

This action has two forms. The first is metadata-driven, where you can specify how assignment should be performed by using metadata. The second supports calling out to a custom assign variable function. This custom assign variable function can perform a transformation on the source value before assignment.

```
"myActionChain": {
  "root": "myAssignVariableAction",
  "actions": {
    "myAssignVariableAction": {
      "module": "vb/action/builtin/assignVariablesAction",
      "parameters: {
        "$page.variables.target1": { "source": "{{ $page.variables.source1 }}" },
        "$page.variables.target2": { "source": "{{ $page.variables.source2 }}" }
      }
    }
  }
}
```

Metadata-Driven Variable Assignment

This action is used to assign values to a set of variables using metadata.

Metadata-driven variable assignment lets you use metadata to specify how assignment should be performed.

This form takes a map of target expression and assignment metadata pairs. For example, if the target expression is a structure, it has to resolve to a variable or to a variable's property. The target expression has to be prefixed with one of the following:

- `$application.variables`
- `$page.variables`
- `$chain.variables`
- `$variables`

This should be followed by a variable name or a path to a variable property, such as the following:

- `$application.variables.a`
- `$page.variables.a.b`
- `$variables.a.b.c`

Note that `$variables.a.b.c` is a shortened form of `$chain.variables.a.b.c`.

The expression can be arbitrarily complex as long as it is a valid JavaScript expression and satisfies the above constraints.

The assignment metadata has the following format:

```
{
  "source": "some expression",
  "reset": "none", // default to " toDefault"
  "auto": "always", // default to "ifNoMappings"
  "mapping": { ... }
}
```

The "source" expression can be an arbitrary expression that evaluates to a primitive value, an object or an array.

The "reset" option can be one of the following:

- "toDefault" - reset the target to its default value before assignment. This is the default.
- "empty" - clear the target before assignment. If the target has an object type, the target will be reset to an empty object of that type. If the target is an array, the target will be reset to an empty array.
- "none" - overwrite the existing target value

The "auto" option controls whether to auto-assign all properties from the source to the corresponding properties of the target. It can be set to one of the following:

- "always" - auto-assignment will always be performed first before any mapping is applied.
- "ifNoMapping": auto-assignment will only be performed if no mapping is provided. This is the default.

The "mapping" is a piece of metadata used to provide fine-grained control over what gets assigned from the source to the target. When no "mapping" is used to control the assignment, there are two possible schemes for assignment depending on the target type, *auto* and *direct*.

Auto Assign Source to Target

If the target has a concrete type, the assign action will auto-assign the source to the target. If the target type is an object type, auto-assignment will recursively assign each property in the source object to the corresponding property in the target object based on the target type. If the target is an array, the source will be treated as an array if it is not one already. For each item of the source array, an empty item will be created using the target's array item type and appended to the target array. The source item is then auto-assigned to the target item.

If the target property is an object and the source property is a primitive or vice versa, no assignment will be made. For primitive types, the source value will be coerced into the target type before assignment. For boolean type, the coercion will be based on whether the source value is truthy except for "false" (case-insensitive) and "0" which will be coerced to false.

Direct Assign Source to Target
If the target has a wildcard type, e.g., any, any[], object or object[], direct assignment will be performed. The behavior may differ depending on the wildcard type:

- any - the source value is directly assigned to the target
- any[] - the source value is turned into an array if not one already and then directly assigned to the target
- object - same as any except the source value has to be an object. Otherwise, no assignment is performed.
- object[] - same as any[] except the items in the source array have to be objects. Otherwise, no assignment is performed.

Example: Metadata-driven assignment takes a map of target expression and assignment metadata pairs.

```
"myActionChain": {
  "root": "myAssignVariableAction",
  "actions": {
    "myAssignVariableAction": {
      "module": "vb/action/builtin/assignVariablesAction",
      "parameters: {
        "$page.variables.target1": { "source": "{{ $page.variables.source1 }}" },
        "$page.variables.target2": { "source": "{{ $page.variables.source2 }}" }
      }
    }
  }
}
```

Example

```
"$page.variables.target": {
  "source": "{{ $page.variables.source }}",
  "mapping": {
    "$target.a": "$source.b",
    "$target.b.c": "$source.c.b"
  }
}
```

Example

```
"$page.variables.target": {
  "source": "{{ $page.variables.source }}",
  "mapping": {
    "$target.a": "$source.b",
    "$target.b": {
      "source": "$source.c"
      "mapping": {
        "$target.c": "$source.b"
      }
    }
  }
}
```

Assign Variables With a Custom Function

This action uses a custom function to assign values to a set of variables.

A custom assign variable function can perform a transformation on the source value before assignment.

The AssignVariablesAction will first look up the function referenced by "functionName" from the page's functions module and call it with the current available scopes. It will then assign the return value of the function call to the target variable. The custom function should have the following signature:

```
PageModule.prototype.myAssignVariableFunction = function (helper, targetDefaultValue)
```

The "targetDefaultValue" is the default value for the target which can be used to emulate the "toDefault" reset option.

The "helper" is an utility object that can be used to retrieve values for variables within the current scope and perform auto-assignment. It has the following interface:

```
class AssignmentHelper {
  /**
   * Gets a variable from a scope by its string representation, e.g.,
   * helper.get("$page.variables.myVar")
   */
  get(expr);

  /**
   * Assigns properties from one or more sources to the target if and
   * only if the property already exists on the target. The sources
   * are processed in the order they are defined.
   *
   * If target is null, any empty target value will be created based
   * on the target's type. If the target is not null, it will be cloned
   * and the sources will be assigned into the clone. In either case,
   * this value will be returned as the result.
   */
   pick(target, ...sources) {
}
```

Example: an assign variable function that resets the target value to its default value and auto-assign the source to the target:

```
PageModule.prototype.myAssignVariableFunction = function (helper, targetDefaultValue) {
  var source = helper.get("$page.variables.source");
  var result = helper.pick(targetDefaultValue, source);
  return result;
}
```

Call Action Chain Action

The action module for this action is "vb/action/builtin/callChainAction".

Note:

You can call a JSON action chain from a JavaScript action chain using this action; however, you can't call a JavaScript action chain from a JSON action chain.

To call an action chain, you need to pass the following parameters:

Parameter Name	Description
id	The ID of the action chain to call. Action chains need to be prefixed with `application:` for an application chain and `flow:` for a flow chain.
params	An expression that maps to an array of parameters.

The outcome and result will be the outcome and result of the last action executed in the called action chain.

Call Component Action

The action module for this action is "vb/action/builtin/callComponentMethodAction". This provides a declarative way to call methods on JET components.

Parameters

Parameter Name	Description

component	The component on the page. Use the DOM method `document.getElementById` to locate a JET element/component.
	The following deprecated utility methods are provided in the $page scope to get JET components, but will be removed in a future release:
	`$page.components.byId('myCard')`
	`$page.components.bySelector('#myCompId')`
	 Note:
	These two methods will return `null` if no element is found, or if the element is not part of a JET component.

method	The name of the component method to call.
params	Array of parameters to pass to the method, if it takes arguments. Primitives, objects, and array parameters are passed by value and not by reference. Instances are still sent as references.

For this sample composite component, the 'flipCard' method takes two parameters: 'model', which is unused (null below), and 'event', which we construct with a 'type' property:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "flipCardMethodCall": {
      "label": "Flip the Card",
      "module": "vb/action/builtin/callComponentMethodAction",
      "parameters": {
        "component": "{{ document.getElementById('myCard') }}",
        "method": "flipCard",
        "params": ["{{ null }}", { "type": "click" }]
      }
    }
  }
}
```

Call Function Action

The action module for this action is "vb/action/builtin/callModuleFunctionAction".

To call a module function, you need to pass the following parameters.

Parameter Name	Description
module	The module to call the function on. This could be `"$page.functions"`, `"$application.functions"`, or `"$flow.functions"`.
functionName	The name of the function to call.
params	An array of parameters. Note that a single parameter must still be passed as a single item array.

The outcome is either 'success' if the function call was successful, or 'error' otherwise. The result payload is equivalent to whatever the function returns (which may be undefined if there is no return). If the function returns a promise, the result payload will be whatever is resolved in the promise.

Suppose there is a function defined in the page functions module as follows:

```
PageModule.prototype.sum = function(one, two) {
  return one + two;
}
```

You can call that function with the following action:

```
"myActionChain": {
  "root": "mySumAction",
  "actions": {
    "myAction": {
      "label": "call my sum function",
      "module": "vb/action/builtin/callModuleFunctionAction",
      "parameters": {
        "module": "{{$page.functions}}",
        "functionName": "sum",
        "params": ["3", "4"]
      }
    }
  }
}
```

After this action call, `$chain.results.mySumAction` should be set to 7.

Call REST Action

The action module for this action is "vb/action/builtin/restAction".

The call REST action is used to make a REST call in conjunction with the service definitions.

Internally, this action uses the Rest Helper, which is a public utility. Its parameters are as follows.

Parameter Name	Description
endpoint	The endpoint ID as defined in the service configuration.
uriParams	A key/value pair map that will be used to override path and query parameters as defined in the service endpoint.
body	A structured object that will be sent as the body.
requestType	The content-type of the request, either 'json', 'form', or 'url'. **Note:** Note that this is deprecated. Instead, use 'contentType' and 'fileContentType'.
headers	An object; each property name is a header name and value that will be sent with the request.
contentType	An optional string value with an actual MIME type, which will be used for the "content-type" header. When used with "fileContentType", this is also used as the type for the File blob.
responseType	If set, the specified type is used to do two things at run-time: • Generate a fields parameter for the REST URI to limit the attributes fetched; • Automatically map the fetched response to the response type (when used with the built-in vb/BusinessObjectsTransform). This applies to standard action chains. See the definition for "responseType" in Service Data Provider Properties for details on how the assigned type is used in that context.
filePath	An optional path to a file to send with the request. If "contentType" is set, that is used as the type for the File contents. If "contentType" is not set, a lookup of common file extensions will be used.
filePartName	Optional, used with filePath to allow override of the default name ("file") for the FormData part.
fileContentType	An optional string, used in combination with "contentType", "multipart/form-data", and "filePath".
hookHandler	Used primarily by vb/ServiceDataProvider when externalizing data fetches. See Service Data ProviderServiceDataProvider for details.
requestTransformOptions	A map of values to pass to the corresponding transform, as the "options" parameter.
requestTransformFunctions	A map of named transform functions, called before making the request, where the function is: *fn(configuration, options)*
responseTransformFunctions	A map of named transform functions, called before making the response, where the function is: *fn(configuration, options)*

Parameter Name	Description
responseBodyFormat	A string that allows an override of the standard Rest behavior, which normally looks for a “content-type” header to determine how to read and parse the response. Possible values are "text", "json", "blob", "arrayBuffer", "base64", "base64Url", and "formData".
responseFields	This is an "advanced" field, for use specifically with JET Dynamic Forms. The value would typically be a variable that is bound to the `<oj-dynamic-form>` "rendered-fields" attribute. This is how a calculated layout can tell the Rest Action call which fields to fetch. **Note**: the vb/BusinessObjectsTransform transform is necessary to create a query from this value. **Note**: When "responseFields" is provided, "responseType" is ignored.

Using multipart/form Data

If you have set "contentType" to "multipart/form-data", the Action will interpret your request "body" object as the form parts. Each property of the body object will be a form part. If "filePath" is also set, it will be added as an additional part using the lookup of common file extension types.

If "filePath" is also set, it will be added as an additional part using the sample simple file extension type association. The name of this part will be "file", or can be specified using "filePartName".

You may optionally override the file type by using "fileContentType" for the file part.

Defining Services

In order to use a REST API, it should be first defined.

In this example, the following endpoint is registered for the 'foo' service:

```
{
  "openapi": "3.0",
  "info": {
    "version": "1.1",
    "title": "ifixitfast",
    "description": "FIF",
  },
  "host": "exampledomain.com",
  "basePath": "/services/root",
  "schemes": [
    "http"
  ],
  "paths": {
    "/foo/{id}": {
      "get": {
        "summary": "get a specific Foo object",
        "operationId": "getBar",
        "parameters": [
           {
            "name": "id",
            "in": "path",
            "required": true,
```

```
              "type": "string"
            }
          ],
          "responses": {
            "200": {
              "description": "",
              "schema": {}
            }
          }
        }
      }
    }
  }
}
```

You can invoke that endpoint with the following, passing in a value for the 'id' path parameter from a page parameter:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "myAction": {
      "module": "vb/action/builtin/restAction",
      "parameters": {
        "endpoint": "foo/getBar",
        "uriParams": {
          "id": "{{ $page.variables.myId }}"
        }
      }
    }
  }
}
```

Declaring Services in the Application

Service definitions are referenced in declarations in the application or in flows. The service name and path are defined by a "services" section in an app-flow.json or xxx-flow.json model. Service declarations support two syntaxes: a string (path), or an object with "path" and "headers":

```
"services": {
  "fooService": "./demo-data-service.json",
  "barService": {
     "path":  "./service-def.json",
     "headers": {
        "Accept": "application/vnd.oracle.openapi3+json"
     }
  }
}
```

Transforms

The *requestTransformOptions, requestTransformFunctions*, and *responseTransformFunctions* can be used to modify the request and response. Some built-in service endpoints have built-in transform functions for 'sort', 'filter', 'paginate', and 'select', so options for these transform functions can be defined using the same name via the *requestTransformOptions* property. For third party services, the options set are based on the type of transform functions supported.

When using the Rest Action the transform names have no semantic meaning, and all request and response transforms are called.

Request and response transform functions have the following signatures.

<table>
<tr><th>Transform Type</th><th>Parameters</th><th>Return Value</th></tr>
<tr><td>Request</td><td><pre>/**
 * configuration: {
 * url:
 *   initConfig: {
 *   method: // string with
http method
 *   body: // request body, if
any
 *   credentials: // string
see (fetch) Request
 *   headers: // object, map
of strings
 *   }
 * },
 *
 * options: provided by the
application
 *
 * context: an empty object,
which exists for the
 *   lifetime of one REST call,
a set of
 *   transforms share this.
 **/

mytransform(configuration,
options, context)</pre></td><td>Configuration object; see "Parameters".
Typically, returns the same object passed in, or a modified one.</td></tr>
</table>

Transform Type	Parameters	Return Value

Response	`/**`
	` * response: { body, headers }`
	` *`
	` * context: an empty object, see "Request transforms"`
	` *`
	` */`
	`myresponsetransform(response, context);`
		The return value is application-defined. The value is returned as the 'transformResults' of the REST call result:
		
		`/**`
		` * {`
		` *  response: The (fetch) Response object. Note that the body has already`
		` *     been read, so the functions (ex. json()) cannot be called.`
		` *`
		` *  body: the result of the json()/text()/etc.`
		` *`
		` *  transformResults: a map of return values from Response Transforms`
		` * }`
		` */`

Example 1-29 A Simple Transform Function

One request transform function and one response transform function for a third party service or endpoint might look like this example. Here, the transform functions are defined in the page module and are configured on the RestAction directly. More commonly, transform functions are defined in the service definition and do not need to be mapped on the RestAction.

```
"fetchIncidentList": {
  "module": "vb/action/builtin/restAction",
  "parameters": {
    "endpoint": "ifixitfast-service/getIncidents",
    "requestTransformOptions": {
      "sort": "{{ $page.variables.sortExpression }}",
    },
    "requestTransformFunctions": {
      "sort": "{{ $page.functions.sort }}"
    },
    "responseTransformFunctions": {
      "paginate": "{{ $page.functions.paginateResponse }}"
    }
  },
  "outcomes": {
    "success": "returnSuccessResponse",
    "failure": "returnFailureResponse"
  }
},
```

The corresponding module functions would be:

```
PageModule.prototype.sort = function (configuration, options) {
   /// some code here to modify 'configuration'
```

```
   return configuration;
}

PageModule.prototype.paginateResponse = function (configuration) {
   /// some code here to modify 'configuration'
   return configuration;
}
```

Outcomes

The Call REST action has the following outcomes.

Outcome	Description	Result Payload

success	If the response code is within the 200 range (or 'ok' in fetch API terms).
	• status: number
	• headers: Headers object
	• body: the result of the call (scalar, obejct, array, etc).
		{
		  status: <responseCode>,
		  headers: <responseHeaders>,
		  body: <result body>
		}

failure	If the response code is outside of the 200 range (an error response).
	• message
	  – summary: string
	• error: Error object, or null
	• payload
	  – status: number
	  – headers: Headers object
	  – body: the result of the call (scalar, obejct, array, etc).
		{
		  message: {
		    summary: <rt message>
		  },
		  error: <Error, or null>,
		  payload: {
		    status: <responseCode>,
		    headers: <responseHeaders>,
		    body: <result body>
		  }
		}

Call Variable Method Action

The action module for this action is `vb/action/builtin/callVariableMethodAction`. This action is used to call a method on a variable of InstanceFactory type only. Using it with any other variable will report an error.

Here is an example.

```
"callGetCapabilityChain": {
  "root": "getCapabilityOnLDPV",
  "actions": {
      "getCapabilityOnLDPV": {
          "module": "vb/action/builtin/callVariableMethodAction",
          "parameters": {
            "variable": "$page.variables.incidentsListView",
            "method": "getCapability",
            "params": [
              "sort"
            ]
          }
```

```
      }
    }
  }
```

Where `incidentsListView` is an InstanceFactory variable defined like this:

```
"incidentsListLDPV": {
    "type": "ojs/ojlistdataproviderview",
    "constructorParams": [
        "{{ $page.variables.incidentsList.instance }}",
        {
          "sortCriteria": [
            {
              "attribute": "priority",
              "direction": "ascending"
            }
          ]
        }
    ],
    "persisted": "session"
}
```

To call a variable method, we need to pass the following parameters:

Parameter Name	Description
variable	The variable path
method	The name of the method to call
params	(optional) An array of parameters. Note that a single parameter must still be passed as a single item array.

The outcome is either 'success' if the function call was successful, or a 'failure' outcome. An error is thrown for configuration errors.

The result payload is equivalent to whatever the function returns (which may be undefined if there is no return). If the function returns a promise, the result payload will be whatever is resolved in the promise.

EditorUrl Action

This action is used to build the URL of the Visual Builder editor from an application at runtime. It gathers multiple pieces of information and returns a URL with request parameters representing various contextual info needed by the editor.

The action module for this action is vb/action/builtin/editorUrlAction.

Note:

This action should not be used for mobile applications.

The base URL pointing to the editor location is either passed as an argument to the action or has to be defined in the EDITOR_URL property of the `vbInitConfig` global object. If this value is not available, the action will abort with an error. Depending if the dynamicLayout request

parameter is defined, the editor will either edit the current page or the ruleset of a specific dynamic component.

Here is an example of editorUrlAction usage:

```
"openEditor": {
  "variables": {
    "componentId": {
      "type": "string",
      "input": "fromCaller",
      "required": true
    }
  },
  "root": "editorUrl",
  "actions": {
    "editorUrl": {
      "module": "vb/action/builtin/editorUrlAction",
      "parameters": {
        "componentId": "{{ $variables.componentId }}"
      },
      "outcomes": {
        "success": "openEditor"
      }
    },
    "openEditor": {
      "module": "vb/action/builtin/openUrlAction",
      "parameters": {
        "url": "{{ $chain.results.editorUrl }}",
        "windowName": "VB_EDITOR"
      }
    }
  }
}
```

Parameter	Description
editorUrl	URL of the VB Extension editor (optional). If not defined, use the value of `vbInitConfig.EDITOR_URL`.
componentId	The id of the component to use to retrieve the dynamic layout. (optional)

Fire Event Action

The action module for this action is "vb/action/builtin/fireCustomEventAction".

This action allows you to fire application-defined events.

```
"actions": {
  "fireEvent": {
    "module": "vb/action/builtin/fireCustomEventAction",
    "parameters": {
      "name": "application:customEventToFire",
      "payload": "{{ $variables.payload }}"
    }
  }
}
```

Fire Data Provider Event Action

The action module for this action is "vb/action/builtin/fireDataProviderEventAction".

This causes the DataProvider specified via the 'target' parameter to dispatch an oj.DataProvider event as a way to notify all listeners registered on that DataProvider to react to changes to the underlying data. For example, a component using a particular ServiceDataProvider may need to render new data because new data has been added to the endpoint used by the ServiceDataProvider.

The action can be called either with a mutation event or a refresh but not both. Generally a mutation event is raised when items have been added, updated, or removed from the data that the ServiceDataProvider represents.

Note:

This action can be used with a vb/ArrayDataProvider2. It does not need to be used with a legacy vb/ArrayDataProvider because the 'data' is already exposed as a property on the variable. This allows page authors to directly mutate the data array using the assignVariables action. This assignment is automatically detected by Visual Builder, and all listeners of this change are notified, removing the need to use a fireDataProviderEventAction. Users will be warned when the fireDataProviderEventAction is used with a legacy ArrayDataProvider, prior to mutating the 'data' property directly.

A mutation event can include multiple mutation operations (add, update, remove) as long as the id values between operations do not intersect. This behavior is enforced by JET components. For example, you cannot add a record and remove it in the same event, because the order of operations cannot be guaranteed.

The action can return either success or failure. Success returns null, while failure returns the error string.

Table 1-1 Parameters

Name	Type	Description	Example
target	string	Target of the event, usually a variable of type vb/SDP or vb/ADP.	`target: "{{ $page.variables.incidentList }}"`
refresh	null	Indicates a data provider refresh event needs to be dispatched to the data provider identified by the target. A null value is specified because the refresh event does not require a payload.	`refresh: null`

Table 1-1 (Cont.) Parameters

Name	Type	Description	Example

add	object	The following properties may be present in the payload:
		• **data**: Array<Object>; the results of the 'add' operation. Note there can be more than one rows added. If data alone is present in the payload, and the target has a keyAttributes property specified, then the 'keys' are built for you. The structure of the data returned must be similar to the responseType specified on the target variable of type vb/ServiceDataProvider (respecting the "itemsPath", if any), or the itemType specified on the vb/ArrayDataProvider
		• **keys**: optional Set<*>. the keys for the rows that were added. If a ServiceDataProvider variable is configured with a keyAttributes property, this can be determined by the ServiceDataProvider itself from the data, if data is present.
		• **metadata**: optional Array<ItemMetadata<Object>>. Since the ServiceDataProvider variable is configured with 'keyAttributes', this can be determined by the ServiceDataProvider itself.
		• **addBeforeKeys**: Optional Array of keys for items located after the items involved in the operation. They are relative to the data array, after the operation was completed and not the original array. If null and index are not specified, then insert at the end.
		• **afterKeys**: Deprecated: use **addBeforeKeys** instead. Optional Set<*>; a Set that is the keys of items located after the items involved in the operation. If null and index not specified then insert at the end.
		• **indexes**: optional Array<number>, identifying insertion point.
			`"add": {`
			`"data": "{{ $chain.results.saveProduct.body }}",`
			`"indexes": [0]`
			`}`
			
			An example with ServiceDataProvider, where "itemsPath": "items":
			
			`"updateList": {`
			`"module": "vb/action/builtin/fireDataProviderEventAction",`
			`"parameters": {`
			`"target": "{{ $page.variables.personList }}",`
			`"add": {`
			`"data": {`
			`"items": "{{ [$chain.results.createPersonPost.body] }}"`
			`}`
			`}`
			`}`
			`}`

remove		The payload for the remove event is similar to add above except 'afterKeys'/'addBeforeKeys' are not present.	`"remove": {`
			`"keys": "{{ [ $page.variables.productId ] }}"`
			`}`

Table 1-1 (Cont.) Parameters

Name	Type	Description	Example
update		Same as **remove**.	`"update": { "data": "{{ $page.variables.currentIncidentResponse }}" }`

The action can return two outcomes:

- The name of the outcome can be 'success' or 'failure'.
- The result of a failure outcome is the error string, and the result of a success outcome is null.

Example 1-30 Example 1

Configuring a refresh event to be dispatched to a ServiceDataProvider:

```
(1) activityListDataProvider is the name of the
page variable that is of type vb/ServiceDataProvider
(2) refresh has a null value

"fireDataProviderRefreshEventActionChain": {
  "variables": {
    "payload": {
      "type": {
        "target": "activityListDataProvider"                           // (1)
      }
    }
  },
  "root": "fireEventOnDataProvider",
  "actions": {
    "fireEventOnDataProvider": {
      "module": "vb/action/builtin/fireDataProviderEventAction",
      "parameters": {
        "target": "{{ $page.variables[$variables.payload.target] }}",
        "refresh": null                                               // (2)
      }
    }
  }
},
```

Example 1-31 Example 2

Configuring a remove event to be dispatched to a ServiceDataProvider:

```
(1) deleteProductChain deletes a product and ends up calling
another chain that fires a remove event on the ServiceDataProvider
(2) deletes the product from the backend service via a RestAction
(3) calls fireDataProviderEventAction
(4) on a variable of type vb/ServiceDataProvider
(5) with a remove payload

"variables": {
  "productListSDP": {
    "type": "vb/ServiceDataProvider",
    "defaultValue": {
      "keyAttributes": "id",
```

```
        "responseType": "application:productSummary[]"
      }
    },
  }
  "chains": {
    "deleteProductChain": { // (1)
      "variables": {
        "productId": {
          "type": "string",
          "description": "delete a single product",
          "input": "fromCaller",
          "required": true
        }
      },
      "root": "deleteProduct",
      "actions": {
        "deleteProduct": { // (2)
          "module": "vb/action/builtin/restAction",
          "parameters": {
            "endpoint": "ifixitfast-service/deleteProduct",
            "uriParams": {
              "productId": "{{ $page.variables.productId }}"
            }
          },
          "outcomes": {
            "success": "refreshProductList"
          }
        },
        "refreshProductList": {
          "module": "vb/action/builtin/callChainAction",
          "parameters": {
            "id": "fireDataProviderMutationEventActionChain",
            "params": {
              "payload": {
                "remove": {
                  "keys": "{{ [ $page.variables.productId ] }}"
                }
              }
            }
          }
        }
      }
    },
    "fireDataProviderMutationEventActionChain": {
      "variables": {
        "payload": {
          "type": "application:dataProviderMutationEventDetail",
          "input": "fromCaller"
        }
      },
      "root": "fireEventOnDataProvider",
      "actions": {
        "fireEventOnDataProvider": {
          "module": "vb/action/builtin/fireDataProviderEventAction", //
(3)                   // (2)
          "parameters": {
            "target": "{{ $page.variables.productListSDP }}",    // (4)
            "remove": "{{ $variables.payload.remove }}" // (5)
          }
        }
      }
```

```
    }
  },
```

Fire Notification Event Action

The action module for this action is "vb/action/builtin/fireNotificationEventAction". This action is used to fire "vbNotification" events.

"vbNotification" events are just like custom events, except that they have a defined name and a suggested use. Notifications are generally intended to help implement a flexible message display, but the specific use can be defined by the application. See Custom Events for details.

```
"actions": {
  "fireNotification": {
    "module": "vb/action/builtin/fireNotificationEventAction",
    "parameters": {
       "summary": "[[ $page.variables.summary ]]",
       "message": "[[ $page.variables.message ]]",
       "displayMode": "persist",
       "type": "info"
    }
  }
}
```

ForEach Action

This action lets you execute another action for each item in an array.

The ForEach action takes an 'items' and 'actionId', and adds a `$current` context variable for the called action, or 'Callee', in order to access the current item. The parameters are as follows:

Parameter Name	Description
as	An optional alias for `$current`. Used to name the context so that it can be referenced in nested Callees.
actionId	An ID in the current action chain.
items	An expression that evaluates to an array.
mode	"serial" (default) or "parallel".

The "mode" parameter allows for serial or parallel action. Prior to this parameter, the behavior was "serial"; each "actionId" call was made for an item only when any previous item's "actionId" call finished (meaning, any Promise returned from the last action resolves). Using "parallel" means that each "actionId" call does not wait for the previous call to finish (useful for Rest Action calls, etc).Using either mode, the ForEach action does not finish until all Promises returned from the "actionId" chain resolve (if no Promise is returned, it is considered resolved on return).

The following table describes additional properties injected into the available contexts that the called action ('callee') can reference in its parameter expressions:

Parameter Name	Description
$current.data	The current array item.
$current.index	The current array index.

Parameter Name	Description
alias.data	An alternate syntax for `$current.data`, which allows a reference to `$current` from nested contexts.
alias.index	An alternate syntax for `$current.index`, which allows a reference to `$current` from nested contexts.

The outcome of the action is either:

- "success", with an array containing the return value of the last action's results; in other words, an array of the return of the "sub-chain" ("chainlet"?) called for each item in the loop,
- or "failure" if there is some exception/error.

Note: Except for the return value for the last action, the results of each Action are not accessible outside of the sub-chain; for example, if the sub-chain is "actionA" → "actionB", the result of the ForEach will contain an array of "actionB" return values, and not "actionA"'s.

ForEach "as" Alias

By default, the ForEach Action ID in the declaration will be used for the *alias* to `$current`.

Note that if an action has an "as" alias, then the value will be used as the alias instead. For example, for as="foo", you can also create expressions that reference "foo.data" and "foo.index".

Example 1-32 Example 1

In this example, `$current.data` and `forEachCurrent.data` are equivalent.

```
actions: {
   "forEach": {
     "module": "vb/action/builtin/forEachAction",
        "parameters": {
           "items": "{{ $variables.testArray }}",
           "actionId": "someAction",
           "as": "forEachCurrent",
      },
    },
    "someAction": {
       "module": "someRandomAction",
         "parameters": {
             "outcome": "{{ $current.data.foo }}",
             "payload": {
               "text": "{{ forEachCurrent.data.bar }}",
               "index": "{{ $current.index }}' }"
          }
       }
    }
}
```

Example 1-33 Example 2

This example demonstrates the use of “as”.

```
"actions": {
  "forEachOuter": {
    "label: 'the outer-most action, a ForEach',
    "module": "vb/action/builtin/forEachAction",
    "parameters": {
```

```
      "items": ["a", "b"],
      "actionId": "forEachInner"
    }
  },
  "forEachInner": {
    "label": "the inner-most action, a ForEach, called by a ForEach",
    "module": "vb/action/builtin/forEachAction",
    "as": "inner",
    "parameters": {
      "items": [1, 2],
      "actionId": "someAction",
    }
  },
  "someAction": {
    "label": "a custom action",
    "module": "countToTwoAction",
    "parameters": {
      "someParam": "{{ forEachOuter.data }}",
      "anotherParam": "{{ inner.data }}"
    }
  }
}
```

Get Location Action

The action module for this action is "vb/action/builtin/geolocationAction".

This action provides a declarative access to geographical location information associated with the hosting device. This action requires the user's consent. As a best practice, it should only be fired on a user gesture. Doing so will allow users to more easily associate the system permission prompt for access with the action they just initiated.

Parameter Name	Description
maximumAge	A positive long value indicating the maximum age in milliseconds of a possible cached position that is acceptable to return. If set to **0**, it means that the device cannot use a cached position and must attempt to retrieve the real current position. If set to **Infinity**, the device must return a cached position regardless of its age.
timeout	A positive long value representing the maximum length of time, in milliseconds, that the device is allowed to take in order to return a position. The default value is **Infinity**, meaning that `getCurrentPosition()` won't return until the position is available.
enableHighAccuracy	A boolean that indicates the application would like to receive the best possible results. If true, and if the device is able to provide a more accurate position, it will do so. This can result in slower response times or increased power consumption. If **false** (the default value), the device can save resources by responding more quickly or using less power. On mobile devices, **enableHighAccuracy** should be set to true in order to use GPS sensors.

If the geolocation API is supported in the browser, geolocationAction returns a JSON Position object that represents the position of the device at a given time.

Return Type	Description	Example

Object	The Position interface represents the position of the concerned device at a given time. The position, represented by a Coordinates object, comprehends the 2D position of the device, on a spheroid representing the Earth, but also its altitude and its speed.
	• *Position.coords* returns a Coordinates object defining the current location.
	• *Position.timestamp* returns a DOM timestamp representing the time at which the location was retrieved.
		Latitude and longitude can be accessed from the Position's coordinates as follows:
		`[[$chain.results.getCurrentLocation.coords.latitude ]]`
		`[[$chain.results.getCurrentLocation.coords.longitude ]]`
		where *getCurrentLocation* is a geolocationAction.

If geolocation is not supported by the browser, or a parameter with a wrong type is detected, a failure outcome is returned. If a PositionError occurs when obtaining geolocation, a failure outcome with a PositionError.code payload is returned. Possible PositionError.code values are:

- PositionError.**PERMISSION_DENIED**
- PositionError.**POSITION_UNAVAILABLE**
- PositionError.**TIMEOUT**

For every failure, a descriptive error message can be obtained from the action chain, such as *[[$chain.results.getCurrentLocation.error.message]]*.

An example of using the geolocation action:

```
"chains": {
  "getCurrentLocation": {
    "root": "geolocation1",
    "description": "",
    "actions": {
      "geolocation1": {
        "module": "vb/action/builtin/geolocationAction",
        "parameters": {
          "timeout": 50000,
          "maximumAge": "{{Infinity}}"
        },
        "outcomes": {
          "failure": "fireNotification1",
          "success": "assignVariables1"
        }
      },
      "fireNotification1": {
        "module": "vb/action/builtin/fireNotificationEventAction",
        "parameters": {
          "summary": "[[ $chain.results.geolocation1.error.message ]]",
          "type": "error",
          "displayMode": "persist"
        }
      },
      "assignVariables1": {
        "module": "vb/action/builtin/assignVariablesAction",
```

```
          "parameters": {
            "$page.variables.coords": {
              "source": "{{ $chain.results.geolocation1.coords }}",
              "auto": "always"
            }
          }
        }
      }
    }
  },
```

If Action

The action module for this action is "vb/action/builtin/ifAction".

This action will evaluate an expression and return a 'true' outcome if the expression evaluates to true, and a 'false' outcome otherwise.

Parameter Name	Description
condition	The expression to evaluate.

For example:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "myAction": {
      "module": "vb/action/builtin/ifAction",
      "parameters": {
        "condition": "{{ $chain.results.myRestAction.code === 404 }}"
      },
      "outcomes": {
        "true": "...",
        "false": "..."
      }
    }
  }
}
```

Login Action

This action launches the login process as defined in the Security Provider implementation.

The action module for this action is "vb/action/builtin/loginAction". It invokes the handleLogin function on the Security Provider with the returnPath argument.

Parameter Name	Description
returnPath	The path of the page or flow to go to when login is successful.

The behavior of the default implementation of the Security Provider handleLogin function is:

- Navigate to the login URL specified by the Security Provider configuration.
- If returnPath is not defined, use the default page of the application.
- Convert the page returnPath to a URL path and add it to the login URL.

Example 1-34 Example

An example of a chain using the loginAction:

```
"signInChain": {
  "root": "signInAction",
  "actions": {
    "signInAction": {
      "module": "vb/action/builtin/loginAction"
    }
  }
}
```

Logout Action

This action launches the logout process as defined in the Security Provider implementation.

The action module for this action is "vb/action/builtin/logoutAction". It invokes the handleLogout function on the Security Provider with the logoutUrl argument.

Parameter Name	Description
logoutUrl	The URL to navigate to in order to log out.

The behavior of the default implementation of the Security Provider handleLogout function is:

- Navigate to the URL defined by the logoutURL parameter.
- If the logoutUrl parameter is not defined, uses the logout Url of the Security Provider configuration.
- After the user is logged out, the application continues to the default page of the application.

Example 1-35 Example

An example of a chain using the logoutAction:

```
"logoutChain": {
  "root": "logout",
  "actions": {
    "logout": {
      "module": "vb/action/builtin/logoutAction"
    }
  }
}
```

Navigate Action

The action module for this action is "vb/action/builtin/navigateAction".

This action will navigate the user to a page and also pass any parameters to activate that page. Parameters for this action are:

Parameter Name	Description
page	The path to the destination page. The path can be a single page ID, or a path starting with a page ID. It can be an absolute path starting at the application or relative to the current page. When used with 'flow' or 'application', the path cannot be absolute; it navigates to the page relative to the flow or App UI.
flow	ID of the destination flow, used to change the content of the flow displayed in the current page. When used with 'page', navigates to the page in that flow.

Parameter Name	Description
target	Target of the destination flow, used with 'flow' to change the content of the parent flow instead of the nested flow. Values are 'parent' or 'self' (default).
application	ID of the destination App UI, used to change the content of the App UI displayed in the base application. When used with 'page' and 'flow', navigates to the page or flow in that App UI.
params	A key/value pair map that will be used to pass parameters to a page (optional)
history	Defines the effect on the browser history. Values are 'replace', 'skip' or 'push'. If the value is 'replace', the current browser history entry is replaced, meaning that the back button will not go back to it. If the value is 'skip', the URL is not modified. (optional and default is 'push')

Page input parameters are page variables with the **Input Parameter** enabled. You can use the Navigate action to set the value for these input parameters. But if a page parameter was a path to a deeply nested page, like `/shell/main/other`, you'll see a list of all input parameters from each page/flow in the path (that is, input parameters for the shell page, the main flow, as well as other pages). Name collisions across flows/pages are not accounted for—something you'll need to keep in mind when defining input parameters.

Here's an example of the `navigate` action:

```
"myActionChain": {
  "root": "navigate",
  "actions": {
    "navigate": {
      "module": "vb/action/builtin/navigateAction",
      "parameters": {
        "page": "myOtherPage",
          "params": {
            "id": "{{ $page.variables.myId }}"
          }
        }
      }
    }
  }
```

This returns the outcome 'success' if there was no error during navigation. If navigation completed successfully, returns the action result **true**, otherwise **false**. Returns the outcome **fail** with the error in the payload if there was an error.

Navigating to the same page

Navigating to the same page with different input params is considered as a valid navigation. Since the current page is not changing, only the page input variable value will change and the `onValueChanged` event will be triggered. When navigating to the same page, the events `vbBeforeEnter`, `vbEnter`, `vbBeforeExit`, and `vbExit` are not triggered because the page never transitioned to an enter or exit state.

The navigation is pushed into the browser history, so pressing the browser's Back button will restore the previous values of the input variables.

Navigating between App UIs or from a page extension

Navigating to the default landing page of an App UI is always allowed, but navigation to any other pages is restricted to pages where navigation to them has been exposed. Exposing a

page is done by marking it with the navigation `fromExternal` property set to '**enabled**' in the page descriptor and the parent flow descriptor.

If you expose a page or flow to navigation, it becomes part of your extension API. That means that you can no longer rename, delete, or change the required input parameters for this page or flow, as extensions may depend on them.

When navigating from a page in one App UI to a page in another App UI, and the page in the second App UI is not exposed, will throw the following error, and navigation will be canceled:

```
Navigation from page main/main-start to appUI2/main/main-other is not enabled
```

Property	Description
`fromExternal`	Defines if navigation to this page or flow is allowed from another App UI or from a page extension. For navigation to be allowed to a page, the entire hierarchy of containers (parent flow, application) need to have this property set to '**enabled**'.This property is not required for the default page of an application. The default for this property is '**disabled**'.
`embeddable`	Defines if this page or flow can be embedded in an `oj-vb-switcher` component. For the page to be allowed to be embedded in a switcher, the entire hierarchy of containers (parent flow, application) need to have this property set to '**enabled**'. The default for this property is '**disabled**'.

Example 1-36 Page or flow descriptor with navigation fromExternal property set to enabled

```
"navigation": {
  "fromExternal": "enabled"
}
```

Navigating from a page extension

When navigation is executed from a page extension, only pages marked with the `fromExternal` property set to '**enabled**' are valid destination. If it is not enabled for the page, the following error is thrown, and navigation is canceled:

```
Navigation from page extension main/main-start to main/main-other is not enabled
```

Navigation with the `page` parameter

The 'page' parameter is the ID of a sibling page or a path starting with a sibling page's ID (like `pageId/flowId/...`). It cannot be or start with a flow ID.

Example 1-37 Navigate to a sibling of the current page

To navigate to page `other`, a sibling of the current page:

```
"parameters": {
  "page": "other"
}
```

Example 1-38 Navigate to a sibling page and change content of the nested flow

To navigate to flow `main`, which is defined under the sibling page `other`:

```
"parameters": {
  "page": "other/main"
}
```

Example 1-39 Navigate to the root application

To navigate to the root of the application:

```
"parameters": {
  "page": "/"
}
```

Example 1-40 Navigate to the current flow's default page

To navigate to the current flow's default page:

```
"parameters": {
  "page": ""
}
```

Example 1-41 Navigate to a deeply nested page relative to the application root

To navigate to a deeply nested page relative to the root of the application:

```
"parameters": {
  "page": "/shell/main/other"
}
```

Navigation with the `flow` parameter

The 'flow' parameter can only be the ID of a flow defined below the current page or an empty string.

Example 1-42 Navigate to a specific flow

To change the content of the flow displayed in the current page to the flow `main`:

```
"parameters": {
  "flow": "main"
}
```

Example 1-43 Navigate to a page in a specific flow

To change the content of the flow displayed in the current page to the flow `main` and navigate to the page `other` or the flow `main`:

```
"parameters": {
  "flow": "main",
  "page": "other"
}
```

Example 1-44 Navigate to the current page's default flow

To navigate to the current page's default flow:

```
"parameters": {
  "flow": ""
}
```

Example 1-45 Navigate the parent flow to a specific flow

To change the parent flow to the flow `main`:

```
"parameters": {
  "target": "parent",
  "flow": "main"
}
```

Example 1-46 Navigate the parent flow to the default flow

To change the parent flow to the default flow:

```
"parameters": {
  "target": "parent",
  "flow": ""
}
```

Example 1-47 Navigate to any page in a sibling flow

To change the parent flow to the flow `main` and navigate to page `other` in the flow `main` (note that page can be a path):

```
"parameters": {
  "target": "parent",
  "flow": "main",
  "page": "other"
}
```

Navigation with the `application` parameter

The 'application' parameter can only be the ID of an App UI defined in the current host application or an empty string.

When working with App UIs, the flow to be used is defined by the `defaultFlow` property in `app.JSON`.

- When the flow is defined, it replaces the App UI's default flow.
- When both flow and page are defined, the flow is always applied first; page navigation is relative to the flow.
- When only the page is defined, page navigation is relative to the default flow.
- When the page is a path, it's considered as being relative to the flow.
- When page starts with a backslash (`/`), it is ignored.

Example 1-48 Navigate to the default flow of a specific App UI

To navigate to the default flow of App UI `appUi2`:

```
"parameters": {
  "application": "appUi2"
}
```

Example 1-49 Navigate to a page in a specific App UI

To navigate to the page `other` within the default flow in App UI `appUi2`:

```
"parameters": {
  "application": "appUi2",
  "page": "other"
}
```

Example 1-50 Navigate to a deeply nested page in an App UI

To navigate to a deeply nested page `next` in App UI `appUi2`:

```
"parameters": {
  "application": "appUi2",
  "page": "other/main/next"
}
```

where `other` is a page within the default flow of `appUi2`, `main` is a flow, and `next` is a page.

Example 1-51 Navigate to a flow in an App UI

To navigate to a flow `main` in App UI `appUi2`:

```
"parameters": {
  "application": "appUi2",
  "flow": "main"
}
```

When `main` is a sibling of the default flow, the default flow is replaced with the `main` flow.

Example 1-52 Navigate to a page in another flow in an App UI

To navigate to a page `other` in flow `main` in App UI `appUi2`:

```
"parameters": {
  "application": "appUi2",
  "flow": "main",
  "page": "other"
}
```

Example 1-53 Navigate to the current App UI's root

To navigate to the root of the current App UI (the default page of the current App UI):

```
"parameters": {
  "application": ''
}
```

Example 1-54 Navigate to a deeply nested page relative to the current App UI's default flow

To navigate to a deeply nested page `next` relative to the current App UI's default flow:

```
"parameters": {
  "application": '',
  "page": "other/main/next"
}
```

where `other` is a page in current App UI's default flow, `main` is a flow, and `next` is a page.

Navigate Back Action

The action module for this action is "vb/action/builtin/navigateBackAction".

This action will go back one step in browser history. It has a single 'success' outcome and can return a payload by specifying values for the input parameters.

Parameter Name	Description
params	An optional key/value pair map that will be used to pass parameters to a page.

When a parameter is not specified, the original value of the input parameter on the destination page is used. When a parameter is specified, it has precedence over fromUrl parameters.

Open URL Action

The action module for this action is "vb/action/builtin/openUrlAction".

In a web app, this action opens the specified URL in the current window or in a new window using the `window.open()` API.

In a native mobile app, this action supports opening local file attachments as well as remote resources. Allowed file types for the url parameter are as follows:

- `.pdf`
- `.doc`
- `.txt`
- `.text`
- `.ppt`
- `.rtf`
- `.xls`

- `.mp3`
- `.mp4`
- `.csv`

The very first time, the user will get an option to select which application to use for opening a given file type. If no application is available to open such a file, this action will fail with the appropriate error. Once the given file has been opened once, it will always be opened with the same application across all Visual Builder installed apps on the device.

If the specified file is not local or if the file extension is not recognized, this action will use Cordova's plugin `cordova-plugin-inappbrowser` to open the specified URL.

Parameter Name	Description
url	The url to navigate to (required)
params	A key/value pair map that will be used as query parameters to the url (optional)
hash	The hash entry to append to the URL. (optional)
history	Defines the effect on the browser history. Allowed values are 'replace' or 'push'. If the value is 'replace', the current browser history entry is replaced, meaning that the back button will not go back to it. (optional, and default is 'push')
windowName	A name identifying the window as defined in the `window.open()` API (optional). If not defined, the URL opens in the current window. Otherwise, refer to the `window.open()` API documentation. In a mobile app, there are 3 possible values: `_self`, `_blank`, or `_system`. The default is `_self`. Refer to the documentation for `cordova-plugin-inappbrowser`. For local file types, this parameter is ignored.

Once on the URL location, the browser back button will re-enter the last page if you specified a value for the `windowName` parameter that opens the URL in the current window and the page input parameters will be remembered, even if their type is 'fromCaller'.

Example 1-55 Open a new window in the browser with the given URL

To open a URL:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "myAction": {
      "module": "vb/action/builtin/openUrlAction",
      "parameters": {
        "url": "http://www.example.com",
        "params": {
          "id": "{{ $page.variables.myId }}"
        },
        "windowName": "myOtherWindow"
      }
    }
  }
}
```

Reset Variables Action

Use this action to reset variables to their default values defined in their variable definitions.

The action module for this action is **vb/action/builtin/resetVariablesAction**.

Parameter Name	Description
variables	An array of variables. Here is an example. `["$page.variables.var1", "$page.variables.var2"]` **Note:** If a single variable expression is provided instead of an array, it will be implicitly treated as an array of one variable. Each expression in the array has to resolve to a variable or variable property. It has to be prefixed with one of the following: • `$application.variables` • `$page.variables` • `$chain.variables` Each expression should be followed by a variable name or a path to a variable property. For example: • `$application.variables.a` • `$page.variables.a.b` • `$variables.a.b.c` (which is shorthand for `$chain.variables.a.b.c`)

Return Action

The action module for this action is "vb/action/builtin/returnAction".

This action (which should be the terminal action of a chain) allows you to control the outcome and payload of that chain when necessary. Parameters for this action are as follows:

Parameter Name	Description
payload	The payload to return from this action. Useful in a 'callChainAction" to control the resulting payload from calling that action chain. This can be an expression.
outcome	The outcome to return from this action. Useful in a 'callChainAction" to control the resulting outcome from calling that action chain. This can be an expression.

An example that uses the return action on a chain that makes a REST call, but returns a simpler value:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "someRestCall": {
      "module": "vb/action/builtin/callRestAction",
      "parameters": {...},
      "outcomes": {
        "success": "myReturnAction"
      }
    }
    "myReturnAction": {
      "module": "vb/action/builtin/returnAction",
      "parameters": {
        "outcome": "success",
        "payload": 
"{{ $chain.results.someRestCall.body.somewhere.inthe.payload.isa.string }}"
      }
    }
  }
}
```

This will return a simple string on a successful REST call if this action chain was called via the 'callChainAction'.

Run in Parallel / Fork Action

The action module for this action is "vb/action/builtin/forkAction".

This action allows multiple action chain paths to run in parallel, then wait for their responses and produce a combined result. Normally, if you do not care what your action chains return, you can chain multiple action chains on the event handler. If you want to wait for the result, and take action once everything is complete, you can use this action instead.

A fork action has an arbitrary set of actions whose action sub-chains will run in parallel. A special outcome, 'join', will be followed once all the sub-chains complete processing. The outcome of the fork action is always 'join', and the result is a mapping from the outcome id's of the sub-chains to their outcome/result payload.

This action takes one parameter, "actions", which is a map of an action alias, to an Action ID in the chain. The *alias* is the property name used in the results of the Fork action results (an alias allows the same Action to be called multiple times in the same Fork Action).

Example 1-56 Example

To make two REST calls, then do some assignments only after they both complete:

```
"myActionChains": {
  "root": "myAction",
  "actions": {
  "myForkAction": {
    "module": "vb/action/builtin/forkAction",
    "parameters": {
      "orcl": "orcl",
      "crm": "crm",
    },
    "outcomes": {
      "join": "join"
    },
  "orcl": {
```

```
    "module": "vb/action/builtin/restAction",
    "parameters": {
      "endpoint": "stock/get-stock-quote",
      "uriParams": { "stock": "ORCL" }
    }
  }
  "crm": {
    "module": "vb/action/builtin/restAction",
    "parameters": {
      "endpoint": "stock/get-stock-quote",
      "uriParams": { "stock": "CRM" }
     }
   },
   "join": {
      "module": "vb/action/builtin/assignVariablesAction",
      "parameters": {
        "$page.variables.orcl": { "source": "{{ ''
+ $chain.results.getAllStockQuotes.orcl.result.body }}" },
        "$page.variables.crm": { "source": "{{ ''
+ $chain.results.getAllStockQuotes.crm.result.body }}" }
      }
    }
  }
}
```

Scan Barcode Action

Use this action in your mobile application to scan QR codes and barcodes for details such as URLs, Wi-Fi connections, and contact information.

The action module for this action is `vb/action/builtin/barcodeAction`. Parameters for this action are:

Parameter Name	Description
`image`	An image object, which can be a CanvasImageSource, Blob, ImageData, or an `<img>` element
`formats`	Optional: A series of barcode formats to search for, for example, one or more of the following: `['aztec', 'code_128', 'code_39', 'code_93', 'codabar', 'data_matrix', 'ean_13', 'ean_8', itf', 'pdf417', 'qr_code', 'upc_a', 'upc_e']` Note that all formats may not be supported on all platforms. If `formats` is not specified, the browser will search all supported formats, so limiting the search to a particular subset of supported formats may provide better performance.
`convertBlob`	Optional: A boolean that enables you to automatically convert a Blob to an ImageBitmap when using the Scan Barcode action to process the outcome of the Take Photo action. If `true`, the Blob object is converted as an ImageBitmap before being passed to the Scan Barcode action. If `false` (default), the Blob object is left as is. You'll need to manually do the conversion, for example, by adding a function to your application and calling the function using the `callModuleFunctionAction` in your action chain.

Here's an example of the `barcodeAction`'s metadata used to read QR code from an HTML image element:

```
"fromImage": {
          "module": "vb/action/builtin/barcodeAction",
          "parameters": {
             "image":  "[ document.querySelector('#qrcode') ]",
             "formats": "[[ [ 'qr_code' ] ]]"
           },
           "outcomes": {
             "failure": "showError",
             "success": "openUrl"
           }
         }
```

Here's another example, using the `barcodeAction` to process the outcome of the Take Photo action as a QR code:

```
"qrCodeFromFile": {
          "module": "vb/action/builtin/barcodeAction",
          "parameters": {
            "image": "[[ $chain.results.takePhoto.file ]]",
            "formats": "[[ [ 'qr_code' ] ]]",
            "convertBlob": true
          },
          "outcomes": {
            "failure": "showError",
            "success": "openUrl"
          }
        }
```

A success outcome will include the DetectedBarcode object as a result. DetectedBarcode (https://wicg.github.io/shape-detection-api/#detectedbarcode) has a `rawValue` property that corresponds to the decoded string. A failure outcome will be returned if the browser does not support Shape Detection API, or if a specified format is not supported.

Share Action

Use this action in mobile applications to invoke the native sharing capabilities of the host platform and share content with other applications, such as Facebook, Twitter, Slack, SMS and so on.

The action module for this action is "vb/action/builtin/webShareAction".

Invoke this action following a user gesture, such as a button click. Also, we recommend that the share UI should only be shown if `navigator.share` is supported in the given browser, as in this HTML code:

```
<oj-button disabled="[[!navigator.share]]">Share</oj-button>
```

Parameter Name	Description
title	Optional. Represents the title of the document being shared. This value may be ignored by the target.
text	Optional. Text that forms the body of the message being shared. Can be specified with or without a URL.
url	Optional. URL string that refers to the resource being shared. Any URL can be shared, not just URLs under website's current scope.

Although all parameters are individually optional, you must specify at least one parameter.

Example:

```
"share": {
  "module": "vb/action/builtin/webShareAction",
  "parameters": {
    "text": "Check out this cool new app!",
    "title": "[[document.querySelector('h1').textContent]]",
    "url": "[[ document.querySelector('link[rel=canonical]') &&
document.querySelector('link[rel=canonical]').href || window.location.href]]",
  },
  "outcomes": {
    "failure": "handleShareError"
  }
}
```

A success outcome is returned when the user completes a share action. A failure outcome is returned when the browser does not support the Web Share API or a parameter error is detected.

Switch Action

The action module for this action is "vb/action/builtin/switchAction".

This action will evaluate an expression and create an outcome with that value as the outcome name. An outcome of "default" is used when the expression does not evaluate to a usable string.

Parameter Name	Description
caseValue	This value is used as the outcome value. If null or undefined, the outcome is "default".
possibleValues	Optional. Array of strings, representing the allowed outcomes. If caseValue evaluates to something not in this array, the outcome is "default".

Example:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "myAction": {
      "module": "vb/action/builtin/switchAction",
      "parameters": {
        "caseValue": "{{ $chain.variables.myCase }}",
        "possibleValues": ["case1", "case2"]
```

```
        },
        "outcomes": {
          "case1": "...",
          "case2": "...",
          "default": "..."
        }
      }
    }
  }
```

Take Photo Action

The action module for this action is `vb/action/builtin/takePhotoAction`. Use this action in a mobile application to take photos or choose images from the system's image library. **The takePhotoAction is deprecated for web applications. Use the JET file upload component, or the camera component in the Components palette which uses the JET file upload component.**

The behavior of this action depends on the type of application that you use it in:

- iOS application: Prompts user with multiple options, such as Camera, Browse, or Like
- Android application: Prompts user with options, such as Camera, Browse, or Cancel
- Progressive web apps on Android and iOS: Prompts user with multiple options, such as Camera, Browse, or Like

Parameter Name	Description
mediaType	Set to `image` by default. The `video` type is also supported. Clear the `image` input value from the **Media Type** drop-down list if you want your mobile application to use the deprecated Take Photo action implementation from pre-19.1.3 releases. The pre-19.1.3 Take Photo action can only be used in Android and iOS applications.

If mediaType is set to `video`:

- For iOS Native apps, options to record video using the Camera or to select video files will be provided.
- For Android Native apps, only file selection is allowed. Recording using the Camera is not supported.
- For PWA apps on iOS and Android, options to record video using the Camera or to select video files will be provided.

Example 1-57 Example

The outcome of this action is a binary data object (blob) duck-typed as File. The outcome name is `file`.

```
// To use the outcome file in images, use the URL.createObjectURL and
URL.revokeObjectURL
// methods, as in the following example
const blobURL = URL.createObjectURL(fileBlob);

// Release the BLOB after it loads.
document.getElementById("img-712450837-1").onload = function () {
```

```
    URL.revokeObjectURL(blobURL);
};

// Set the image source to the BLOB URL
document.getElementById("img-712450837-1").src = blobURL;


// To upload the selected/captured image or video, use restAction and set the 
body of
// restAction to the outcome file of takePhotoAction.
"takePhoto1": {
    "module": "vb/action/builtin/takePhotoAction",
    "parameters": {
        "mediaType": "image"
    },
    "outcomes": {
        "success": "callTakePhotoSuccess",
        "failure": "callTakePhotoFailed"
    }
},
"callRestEndpoint1": {
    "module": "vb/action/builtin/restAction",
    "parameters": {
        "endpoint": "OracleCom/postUpload",
        "body": "{{ $chain.results.takePhoto1.file }}", // <- File is set as 
body of restAction
        "contentType": "image/jpeg"
    },
    "outcomes": {
        "success": "callUploadSuccess",
        "failure": "callUploadFailed"
    }
},
"callUploadFailed": {
    "module": "vb/action/builtin/callModuleFunctionAction",
    "parameters": {
        "module": "{{$page.functions}}",
        "functionName": "uploadFailed",
        "params": [
            "{{ $chain.results.callRestEndpoint1.body }}"
        ]
    }
},
"callUploadSuccess": {
    "module": "vb/action/builtin/callModuleFunctionAction",
    "parameters": {
        "module": "{{$page.functions}}",
        "functionName": "uploadSuccess",
        "params": [
            "{{ $chain.results.callRestEndpoint1.body }}"
        ]
    }
},
```

Transform Chart Data Action (Deprecated)

The action module for this action is `vb/action/builtin/transformChartDataAction`. **The transformChartDataAction is deprecated. Data should be set directly on the chart instead.**

Transforms a JSON array with a particular structure into a JSON object containing (array) properties that JET chart component expects.

Page Authors can use this action to take the response from a REST action, turn into a format that this action expects, and use the result returned by this action to assign to a variable bound to the chart component.

The action supports the following parameter.

Parameter Name	Type	Description	Example

source	Array<Object>	An array of objects, or data points, where each data point has one of the two structures below. The first is used with charts that show groups of data for one or more series, such as bar and pie. The second is used with charts that show three dimensions of data, such as bubble.
		
		// Structure 1
		{
		  group: '<group-name>',
		  series: '<series-name>',
		  value: '<value-number>'
		}
		
		// Structure 2
		{
		  group: '<group-name>',
		  series: '<series-name>',
		  valueX: '<valueX-number>',
		  valueY: '<valueY-number>',
		  valueZ: '<valueZ-number>'
		}
			// JSON for Structure 1
			[{
			  group: 'bob',
			  series: 'Feb',
			  value: 5
			}, {
			  group: 'joe',
			  series: 'Feb',
			  value: 2
			}]
			
			// JSON for Structure 2
			[{
			  group: 'bob',
			  series: 'Feb',
			  valueX: 5,
			  valueY: 1,
			  valueZ: 3
			}, {
			  group: 'joe',
			  series: 'Feb',
			  valueX: 6,
			  valueY: 2,
			  valueZ: 4
			}]

The action returns a JSON object with the following properties.

Return Type	Description	Example

Object	The Object has two properties. The properties differ based on the structure that's passed in.
	• groups: {Array} of one or more group names
	• series: {Array} of objects where each object has 2 properties: name and items
	  – name: {String} name of the series
	  – items:
	    * {Array} of numbers when the input resembles the Structure 1 above; or
	    * {Array} of objects, when the input resembles the second structure above, with each object containing the following properties:
	      * x: {Number}
	      * y: {Number}
	      * z: {Number}
		`// Return Value for Structure 1`
		`{`
		`  groups: ['bob', 'joe'],`
		`  series: [{`
		`    name: 'Feb',`
		`    items: [5, 2]`
		`  }]`
		`}`
		
		`// Return Value for Structure 2`
		`{`
		`  groups: ['bob', 'joe'],`
		`  series: [{`
		`    name: 'Feb',`
		`    items: [{`
		`      x: 5,`
		`      y: 1,`
		`      z: 3`
		`    }, {`
		`      x: 6,`
		`      y: 2,`
		`      z: 4`
		`    }]`
		`  }]`
		
		`}`

The example below shows a chain called "fetchTechnicianStatsChain" with four actions chained together to take a REST response and turn the JSON response into a form that can be used by a Chart UI component. The four actions are:

1. Use a Call REST endpoint action to fetch technician stats.
2. Use an Assign Variables action to map the response from (1) to a form that the Transform Chart Data action expects. If the REST response is so deeply nested that a simple transformation of source to target using an Assign Variables action is not possible, page authors can use a page function (using a Call Function action) to transform the data into a form that the Transform Chart Data action expects.
3. Use a Transform Chart Data action to take the response from (2) and turn it into a form that a Chart component can consume.
4. Use an Assign Variables action to store the return value from (3) in a page variable.

```
"actions": {
    "fetchTechnicianStatsChain": {
        "variables": {
            "flattenedArray": {
```

```
                "type": [
                    {
                        "group": "string",
                        "series": "string",
                        "value": "string"
                    }
                ],
                "description": "array of data points",
                "input": "none"
            }
        },
        "root": "fetchTechnicianStats",
        "actions": {
            "fetchTechnicianStats": {                                   // (1)
                "module": "vb/action/builtin/restAction",
                "parameters": {
                    "endpoint": "ifixitfast-service/getTechnicianStats",
                    "uriParams": {
                        "technician": "{{ $page.variables.technician }}"
                    }
                },
                "outcomes": {
                    "success": "flattenDataForBar"
                }
            },
            "flattenDataForBar": {                                      // (2)
                "module": "vb/action/builtin/assignVariablesAction",
                "parameters": {
                    "$chain.variables.flattenedArray": {
                        "source":
"{{ $chain.results.fetchTechnicianStats.body.metrics }}",
                        "reset": "toDefault",
                        "mapping": {
                            "$target.group": "$source.technician",
                            "$target.series": "$source.month",
                            "$target.value": "$source.incidentCount"
                        }
                    }
                },
                "outcomes": {
                    "success": "transformToBarChartData"
                }
            },
            "transformToBarChartData": {                                // (3)
                "module": "vb/action/builtin/transformChartDataAction",
                "parameters": {
                    "source": "{{ $chain.variables.flattenedArray }}"
                },
                "outcomes": {
                    "success": "assignToPageVariable"
                }
            },
            "assignToPageVariable": {                                  // (4)
                "module": "vb/action/builtin/assignVariablesAction",
                "parameters": {
                    "$page.variables.incidentChartDS": {
                        "source": "{{ $chain.results.transformToBarChartData }}",
                        "reset": "toDefault"
                    }
                }
            }
        }
```

```
    }
}
```

Web Share Action

The action module for this action is "vb/action/builtin/webShareAction".

The Web Share action allows mobile and web applications to share content with other applications, such as Facebook, Twitter, Slack, and SMS, by invoking the native sharing capabilities of the host platform.

Note:

Web apps require the web browser running the app to support the Web Share action. Currently, not all browsers support this native feature.

This action should only be invoked following a user gesture (such as a button click). It is a good idea to only enable share UI based of feature detection:

```
<oj-button disabled="[[!navigator.share]]">Share</oj-button>
```

Web Share action parameters correspond to Web Share API options:

The action supports the following parameters.

Parameter Name	Description
title	Title of the document being shared. May be ignored by the handler/target.
text	An arbitrary text that forms the body of the message being shared.
url	A URL string referring to a resource being shared.

All parameters are individually optional, but at least one parameter has to be specified. Any url can be shared, not just urls under website's current scope. Text can be shared with or without a url.

The example below illustrates an action's parameters one would specify to share the current page's title and url:

```
"share": {
  "module": "vb/action/builtin/webShareAction",
  "parameters": {
    "text": "Check out this cool new app!",
    "title": "[[document.querySelector('h1').textContent]]",
    "url": "[[ document.querySelector('link[rel=canonical]') &&
document.querySelector('link[rel=canonical]').href ||
window.location.href]]",  },
  "outcomes": {
    "failure": "handleShareError"
  }
}
```

A success outcome is returned once user has completed a share action. A failure outcome is returned when browser does not support Web Share API or a parameter error is detected.

Action Chain Properties

An action chain has two properties: the set of variables it can use, and the root action.

Action chains are defined under the 'chains' property of the page model. An action chain always has a root action. This root action will always be called when the action chain is invoked.

This action chain will call the 'myAction' action:

```
"chains": {
  "myActionChain": {
    "root": "myAction",
    "actions": {
      "myAction": {
        "label": "My action!",
        "module": "vb/action/builtin/someAction",
        "parameters": {
          "key": "value"
        }
      }
    }
  }
}
```

Each action has an outcome. Usually, an action supports the "success" or "error" outcomes. Some actions may also support other outcomes. Actions can be chained by connecting an additional action to a previous action's outcome.

To perform another action if the previous action succeeds, and handle error cases if it does not succeed, you could do the following:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "myAction": {
      "module": "vb/action/builtin/someAction",
      "parameters": {
        "key": "value"
      },
      "outcomes": {
        "success": "mySuccessAction",
        "error": "myErrorAction"
      }
    },
    "mySuccessAction": {
      "module": "vb/action/builtin/someAction"
    },
    "myErrorAction": {
      "module": "vb/action/builtin/someAction"
    }
  }
}
```

Variable References in Action Chains

Variables can be referenced for the parameter values of an action.

The runtime will automatically evaluate parameter values as expressions. Similar to the default value syntax of variables, variables can be referenced directly into an action parameter's value:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "myAction": {
      "label": "some action",
      "module": "vb/action/builtin/someAction",
      "parameters": {
        "key": "{{ $page.variables.myVariable }}"
      }
    }
  }
}
```

Simple JavaScript code can be added to the values:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "myAction": {
      "label": "some action",
      "module": "vb/action/builtin/someAction",
      "parameters": {
        "key": "{{ $page.variables.myVariable === 'yellow' }}"
      }
    }
  }
}
```

Non-expressions are entered in JSON:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "myAction": {
      "module": "vb/action/builtin/someAction",
      "parameters": {
        "myString": "somestaticvalue",
        "myNumber": 1
        "myBoolean": true
      }
    }
  }
}
```

Map and array values are also expressed in JSON:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "myAction": {
      "module": "vb/action/builtin/someAction",
      "parameters": {
        "key": {
          "key1": "static value",
          "key2": "{{ $page.variables.something }}"
        }
      }
    }
  }
}
```

Action Chain Variables

An action chain can also have variables. These are defined and used in the same way as page parameters.

Unlike page parameters, input variables only support the 'fromCaller' or 'none' type. Input variables must be specified by event handlers calling into action chains.

```
"myActionChain": {
  "variables": {
    "id": {
      "type": "string",
      "description": "the ID of something to update",
      "input": "fromCaller",
      "required": true
    }
  },
  "root": "myAction",
  "actions": {
    "myAction": {
      "module": "vb/action/builtin/someAction"
    }
  }
}
```

Action chain variables can be assigned to or read from using the syntax `$chain.variables.`*varName* and are only accessible within an action chain. They can also be referenced by the shorthand `$variables.`*varName* within the chain.

Action Results

Actions in an action chain can return a result that can be used by subsequent actions in the chain.

After an action implementation is run, it may return a result. The type of these results are specific to an implementation of an action. This result will be stored in a special variable, `$chain.results`. The results of a previous action are contained within `$chain.results`.<actionId>.

Example 1-58 Accessing a Previous Action`s Results

To access a previous action's results:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "myAction": {
      "module": "vb/action/builtin/someAction",
      "outcomes": {
        "success": "someOtherAction"
      }
    },
    "someOtherAction": {
      "module": "vb/action/builtin/someAction",
      "parameters": {
        "myKey": "{{ $chain.results.myAction }}"
```

```
      }
    }
  }
}
```

Example 1-59 Action Chain Return Type and Outcomes

You can specify a return type and an array of outcomes. If a return type is specified, the result of the final outcome will be auto-mapped into the return type. If "outcomes" is specified, the name of the final outcome must match one of the possible outcomes. Otherwise, the action chain will fail. Here is an example:

```
"myActionChain": {
  "root": "myAction",
  "actions": {
    "myAction": {
      "module": "vb/action/builtin/someAction"
    }
  },
  "returnType": "application:someType",
  "outcomes": ["success", "failure"]
}
```

Action "failure" Outcomes

Actions return a standard object shape when returning a "failure" outcome. The result will be an object with the following properties:

- "message": may contain one optional "summary" string property
- "error": may contain an Error object
- "payload": may contain any Action-specific additional information about the failure

Flow

A flow is a way to organize your application in independent and shareable units of work that are building blocks for a single-page application.

The structure of a flow is the same as the structure of an application. A flow has a descriptor (`<name>-flow.json`) and a functions module file (`<name>-flow.js`), and contains pages and possibly other flows. The id of a flow is the name of the folder that contain the flow structure.

The following example shows an application that contains two flows: `main` and `other`.

```
app-flow.[json|js]
  pages/
    app-page.[json|js|html]
  flows/
    main/
      main-flow.[json|js]
      pages/
        start-page.[json|js|html]
      flows/
        ...
    other/
      other-flow.[json|js]
      pages/
        start-page.[json|js|html]
```

Flow Properties

A flow can define a default page, variables, chains, functions, listeners and types.

Here is an example of the descriptor for the flow `other`:

```
{
  "flowModelVersion": "18.1.5",
  "id": "other",
  "description": "Flow other",
  "defaultPage": "start",
  "types": {}
  "variables": {},
  "chains": {},
  "eventListeners": {}
}
```

All pages of a flow can access variables, chains, functions, listeners and types defined in the flow. Defining these elements in the flow allows you to share definition and objects that are used across multiple pages in the flow.

Property	Description
defaultPage	The `defaultPage` property is used to define which page should be the current page of a flow, when the default page is not specified by the navigation. In the example application above with the flows `main` and `other`, the path `app/other` will navigate to the flow `other`, and display the flow's default page. In the descriptor for the flow `other` above, the `defaultPage` property defines `start` as the flow's default page.
types	Pages can address a flow type using the `"flow:typeName"` syntax, where `typeName` is a type defined in the flow.
variables	Flow variables can be addressed in any expression in the page using `$flow`.
chains	In a `callChainAction`, pages can address a flow chain using the `"flow:chainId"` syntax.

Using Flows to Create Single-Page Applications

You use the `<oj-vb-content>` component to nest flows into a page. By nesting a flow into a page, the page can display multiple pages within a single page in your application (single-page application). The `<oj-vb-content>` component has the same API as the JET `<oj-module>` element. The following example shows how to nest a flow in a page:

```
<oj-vb-content config="[[vbRouterFlow]]"></oj-vb-content>
```

The content of the *current page* of the *current flow* is displayed in the page at the location of this component tag in the view (HTML). The *currentFlow* and *currentPage* are managed by

Visual Builder using a hierarchy of routers. When navigating in the application, the router changes the value of the `currentPage` of a flow, or the `currentFlow` of a page, and this determines the content of the `oj-vb-content` element. The router also manages the URL to reflect the `currentFlow` and the `currentPage`.

For example, when navigating using the path `app/flow-a`, the current flow for page `app` is `flow-a`, and the content of the default page of `flow-a` is inserted at the location of the `oj-vb-content` tag.

Nesting content at a specific locations in the page allows you to build page templates or shells.

Note:

A flow should only be nested in page. Nesting a flow in a dialog will not work properly.

Using the page routerFlow property

When the navigation does not specify which flow to use, the `routerFlow` property of the page descriptor is used to determine the default router flow.

In the following example, when navigating to page `app`, the flow `main` will be used as the current flow. It will be the flow displayed in page `app` when no flow is specified in the navigation. The following example shows the `routerFlow` property in a page descriptor:

```
{
  "pageModelVersion": "18.1.5",
  "description": "Application Page",
  "routerFlow": "main",
  "variables": {
  ...
}
```

Represent the Flow State in the URL

There are two strategies for the router to represent the state in the URL: `query` and `path`.

- **query** (default): the current page path is stored in the URL using a query parameter like this:

  ```
  http://myApp/?page=app&app=main
  ```

- **path**: the current page path is stored in the URL using a path segment like this:

  ```
  http://myApp/vp/app/main
  ```

  Notice the marker `vp` added to the URL. It is needed in order for Visual Builder to recognize where the path to the current page starts.

To change the strategy, use the `routerStrategy` property in `app-flow.json`:

```
{
  "applicationModelVersion": "18.1.5",
  "id": "flowDemo",
```

```
  ...
  "routerStrategy": "path"
}
```

When using the `path` router strategy, the server where the application is deployed needs to be able to handle these URLs in a special way to ensure that browser refreshes and bookmarks work properly.

Navigating Between Flows and Pages

Flows and pages are loaded on demand at the time the application navigates to them. All pages located in the `pages/` folder are contained by this flow and it is possible to navigate from one page to an other page using the `navigateToPageAction` using the path `<pageId>`. It is also possible to navigate between flows within the same page, in this case the path to use is `<pageId>/<flowId>`, or just `<flowId>`.

Flow Lifecycle

While navigating between flows, for example from `flow-a` to `flow-b`, the *current flow* for a page changes from `flow-a` to `flow-b`. When this change happens, two events notifying the change are dispatched: 1) `flow-b` is entered, then 2) `flow-a` is exited.

The following table describes the two flow events:

Name	Description
vbEnter	Dispatched when entering a flow after all the flow scoped variables have been added and initialized.
vbExit	Dispatched when exiting a flow before disposing of flow resources.

Load Flow Resources

Two built-in variables are used to address local resources (for example, images): `$application.path` and `$flow.path`.

Each variable is used to build a path relative to the location of the flow:

```
<!-- Display an image located in the resource folder in this application -->
<img alt="photo" :src="[[$application.path + 'resources/images/tools.png']]"/>

<!-- Display an image located in the resource folder in this flow -->
<img alt="photo" :src="[[$flow.path + 'resources/images/tools.png']]"/>
```

Use Flows Not in the Flows Folder

A flow's id is the folder name in the `flows` folder. For example, if the `flows` folder contains a folder named `main`, the flow's id would be `main`. If you want to use a flow located in another location, you can use the `flows` property in the flow descriptor. The `flows` property is a map of paths keyed by the id given to the flow:

```
app-flow.json
{
```

```
  "id": "Main Application"
  ...
  "flows": {
    "crm": "some-nested-path/flows/crm",
    "flow2": "/flows/flow2",
    "flow3": "http://host:port/special/location/of/myFlow"
  }
}
```

The path is relative to the current flow location. If the path starts with ***/***, the path is absolute, meaning that it is relative to the application directory (the directory where `app-flow.json` is located). You can also use a URL for the path. When the path is a URL, the flow will be loaded from the URL.

Shell Flow

A shell flow is a special flow with only one page. The purpose of a shell flow is to define the shell page of an application, or of another flow. To make an application use a shell flow, you enter the id of the shell flow for the `defaultPage` property of the application. When you do this, the application will use the default page of the shell flow as the default page of the application:

```
{
  "applicationModelVersion": "18.2.3",
  "id": "flowDemo",
  "description": "An Application to demonstrate the use of flow",
  "defaultPage": "shellFlowId",
  ...
}
```

When using a flow for the default page, the flow id is not included in the URL. The flow id is hidden from the URL and from the path used for navigation.

> **Note:**
>
> Shell flows have the following limitations:
>
> - Only one page can be defined in a shell flow.
> - The page cannot make reference to artifacts such as variables or types defined in the shell flow metadata.

Defining the default flow of a shell

The default flow of a shell page is defined using the `routerFlow` property. The default flow can be defined externally by specifying a path in the `defaultPage` entry. This is so that the same shell flow can be re-used for multiple applications:

```
{
  "applicationModelVersion": "18.2.3",
  "id": "flowDemo",
  "description": "An Application to demonstrate the use of flow",
  "defaultPage": "shellFlow/crmFlow",
```

```
  ...
}
```

In the example above, the flow with the id "crmFlow" will be used as the default flow of the shell page.

Fragments

Fragments encapsulate a reusable piece of UI, model and code (HTML, JSON and JavaScript) that can be shared across pages in an application.

Fragments can be added and reused in pages and other fragments in applications, extensions and app UIs. A fragment can also be used multiple times in the same page, for example, providing different sets of input parameters to the same fragment, as shown here:

```
<oj-vb-fragment id="editProd1" name="edit-product">
    <oj-vb-fragment-param name="products"
             value="[[ $page.variables.productListDynamic ]]"></oj-vb-
fragment-param>
</oj-vb-fragment>

<oj-vb-fragment id="editProd2" name="edit-product">
    <oj-vb-fragment-param name="products"
             value="[[ $page.variables.productListStatic ]]"></oj-vb-fragment-
param>
</oj-vb-fragment>
```

A fragment can also be 'nested' in another fragment. However, when looking at the structure of applications and extensions, every fragment in a page, no matter how deeply it's 'nested', is an independent unit that encapsulates its state and 'logic, and is not 'aware' of its container.

Define a Fragment Component

To include a fragment in a page or other component, you use the `<oj-vb-fragment>` component, specifying the name of the fragment.

A fragment with the name "incident-list-fragment" could be written like this:

```
<oj-vb-fragment id="incLF1" name="incident-list-fragment"></oj-vb-fragment>
```

When the component above is rendered, it starts loading the fragment identified by the 'name'. The fragment instance created for the page is identified by the 'id'. The component can have the following properties:

Property Name	Description
id	Optional. A <string> unique to the container where the fragment is included. A fragment id must be unique, whether it's generated automatically or set by the author. This id is accessible within the fragment scope using `$fragment.info.id`. This can be used within expressions set on the 'id' property of components, and even the id of a "nested" fragment. **Note:** The id property need not be set on the oj-vb-fragment component. When an id is not provided a system generated unique id will be used. Though unique for the container consuming the fragment, this id is not considered "stable" and cannot be used for persisting variable values. If you want to persist variable values in a fragment, you'll need to provide an id, ensuring that it is both unique and stable, particularly when fragments are used inside of stamping components.
name	Required. A <string> name of fragment to load. The component loads the physical fragment artifacts using the 'name' property. This needs to be statically defined and cannot be an expression.

bridge	Required within a VDOM. Also works within a component.
	This property allows the fragment to discover the current context and establish a bridge between the component and Visual Builder eco-system. It's value is always `"vbBridge"`.
	The example below shows the 'bridge' property configured on an oj-vb-fragment. The same configuration can be used with oj-dynamic-form component as well.
	`<oj-vb-fragment id="incLL" name="incidentsListLayout" bridge="[[ vbBridge ]]">`
	`  <oj-vb-fragment-param name="userId" value="[[ $page.variables.technician.id ]]"></oj-vb-fragment-param>`
	`  <oj-vb-fragment-param name="filterCriterion" value="[[ $page.variables.filterCriterion ]]"></oj-vb-fragment-param>`
	` </oj-vb-fragment>`

- oj-vb-fragment-param (sub-component)	For each input parameter a fragment defines via the 'input' property on a variable, this sub-component can be used to provide the values for the input parameters. Parameters marked as "required" in the fragment must be provided. • `name`: string, name of param • `value`: any, value of the input param A fragment model (the descriptor JSON) can tag its variables with these properties to declare the input parameters (see Define Fragment Input Parameters below): • `input` ("*fromCaller*"), • `required`, that can be *true* (if the caller has to pass a value) or *false*, and • `writeback`, that can be set to *true* to allow the fragment variable value to be automatically written back to the input parameter variable.

Example 1-60 Include fragments in a page or another fragment

In this example of fragments in a page, the tab bar (`oj-tab-bar`) selection drives the fragment that is to be loaded.

incidentsShell-page.html	Notes
1 <oj-tab-bar selection="{{ $variables.incidentsLayout }}"> 2 3 <li id="list">List 4 <li id="map">Map 5 <li id="schedule">Schedule/li> 6 7 </oj-tab-bar> 8 <oj-switcher value="[[$variables.incidentsLayout]]"> 9 <div slot="list"> 10 <oj-vb-fragment id="incLL" name="incidentsListLayout"></oj-vb-fragment> 11 </div> 12 <oj-defer slot="map"> 13 <oj-vb-fragment id="incML" name="incidentsMapLayout"></oj-vb-fragment> 14 </oj-defer> 15 <oj-defer slot="schedule"> 16 <oj-vb-fragment id="incSL" name="incidentsScheduleLayout"></oj-vb-fragment> 17 </oj-defer> 18 </oj-switcher>	*Line 10, 13, 16*: `<oj-vb-fragment>` uses the 'name' property to specify a static fragment to load. The 'id' property is expected to be unique to the current page. Lines 13, 16: component is wrapped in an `oj-defer` (see Deferred Rendering of a Fragment).

Fragment Scopes and Namespaces

As a fragment is designed to be scope-agnostic, it is unaware of the parent container scope and associated properties. This means that the fragment cannot access its parent's scopes (such as `$page` or `$flow`), call its chains, and so on. However, a fragment can access some scopes: `$global` (unified app in extensions), `$application` (for the older style of apps), and `$extension` (in extensions).

Within a fragment, there is a new local `$fragment` scope can be used within the fragment (this can be particularly useful when writing expressions):

- `$variables` / `$fragment.variables` can be used to refer to the local variable in a fragment. `$fragment.variables` is needed for action chains.

Namespaces

Namespaces are used when referencing types in other scopes. The namespaces are similar to the scopes: `global:` / `application:` (for base apps), and `fragment:` for local scope.

Define Fragment Input Parameters

A fragment can define parameters that are required or optional. Callers must provide a value for each of the required input parameters, and may provide values for optional ones. Additionally, parameters that are marked for 'writeback' will cause the fragment to automatically writeback the changed/updated value in its variable to the source variable, that was set via the `<oj-vb-fragment-param>` tag. For details, see Write Back a Fragment Variable Value to the Parent Container.

Input parameters can be reapplied on a fragment after the fragment is loaded. When input parameter values change "mid-cycle", the fragment receives this value automatically, so the fragment can react to the change, as determined by the fragment author. This can be useful, for example, when the input parameter is an expression involving a page variable, and the variable's value changes.

Once a fragment is loaded using input parameters provided by the outer page, if the page variable's value changes, the updated value is automatically picked up by the fragment parameters. This means that the 'live' behavior of variables, where the value change of a 'live' variable triggers changes in other variables, will also automatically update the input parameters that use the same variable.

The following special properties can be applied to fragment variables to determine behavior:

Property	Description
`"input": "fromCaller"`	Identifies the variable or constant as a fragment input parameter, which the caller of a fragment can provide.
`"required": true`	Identifies that the variable must be provided by the caller.
`"writeback": true`	Identifies that the variable's value will be automatically written back to the input parameter variable.

Example 1-61 Fragment where the userId and fragFilterCriterion variables are set as required and fromCaller.

The incidentsSDP variable can use the input param values to initialize its state.

A page that loads a fragment can provide the parameters like this:

incidentShell-page

```
<oj-vb-fragment id="incLL" name="incidentsListLayout">

  <oj-vb-fragment-param name="userId"
value="[[ $page.variables.technician.id ]]"></oj-vb-fragment-param>
  <oj-vb-fragment-param name="fragFilterCriterion"
value="[[ $page.variables.filterCriterion ]]"></oj-vb-fragment-param>

</oj-vb-fragment>
```

The page JSON defines the variables above like this:

incidentsShell-page

```
"technician": {
  "type": "object",
  "defaultValue": {
    "id": "rosie",
    "name": "Rosie Riveter"
  }
},
"filterCriterion": {
  "type": "object",
  "defaultValue": {
    "op": "$ne",
    "attribute": "status",
    "value": "closed"
  }
}
```

In the 'incidentShell-page', the page variables `userId` and `filterCriterion` are passed in as parameters to the fragment ("incidentsListLayout-fragment"). In this example, when the `filterCriterion` *page* variable changes, it updates the fragment parameters, which in this example is the `fragFilterCriterion` *fragment* variable. As a local SDP variable on the fragment references `fragFilterCriterion`, a re-fetch is triggered by the SDP using the new criteria.

incidentsListLayout-fragment

```
"userId": {
    "type": "string",
    "input": "fromCaller",
    "required": true
},
"fragFilterCriterion": {
    "type": "object",
    "input": "fromCaller",
    "required": true
},
"incidentsSDP": {
    "type": "vb/ServiceDataProvider",
    "defaultValue": {
      "endpoint": "ifixitfast-service/getIncidents",
      "keyAttributes": "id",
      "uriParameters": {
        "technician": "{{ $variables.userId }}"
      },
      "filterCriterion": "{{ $variables.fragFilterCriterion }}"
    }
}
```

Example 1-62 Fragment where a list-view of incidents is bound to an SDP variable

In this example, the fragment defines an SDP variable as required by the caller using the (`input: fromCaller`) property.

Note:

Though this is unusual, the caller can pass a reference to the SDP variable defined by the outer page to a fragment. The reference the fragment variable holds can be used only for the purposes of rendering. Using an action that mutates the state of the variable (such as `assignVariables` or `resetVariables`) is not allowed, and will throw errors. **Use with extreme caution.**

incidentsListLayout-fragment

```
{
  "variables": {
    "incidentsSDP": {
      "type": "vb/ServiceDataProvider",
      "input": "fromCaller",
      "required": true
    }
  }
}
```

The page that loads the fragment above will provide the input parameters using the oj-vb-fragment-param sub-component:

incidentsShell-page

```
<oj-vb-fragment id="incLL" name="incidentsListLayout">

   <oj-vb-fragment-param name="incidentsSDP"
value="[[ $page.variables.incidentsSDP ]]">
   </oj-vb-fragment-param>

 </oj-vb-fragment>
```

Write Back a Fragment Variable Value to the Parent Container

A fragment variable whose value is provided by the caller (`"input": "fromCaller"` property), can additionally be marked as supporting "writeback" (`"writeback":true`). This allows the fragment variable value to be automatically set / written back to the input parameter variable of the page. You can set the `writeback` property on fragment variables with the following types: primitive, array and object. If an input parameter value is already passed in by reference (for example, an SDP or `$dynamicLayoutContext`), the fragment variable receiving the reference doesn't need to be configured with the `writeback` property.

Example 1-63 Define the 'incidentId' interface variable in a fragment

In this example, a page uses the fragment below to provide a value for the variable via parameter.

```
"incidentId": {
  "type": "string",
  "description": "extensions can update the value",
  "input": "fromCaller",
```

```
  "writeback": true
}
```

When the fragment variable value changes, the value is automatically written back into the outer variable `selectedIncidentId`.

```
<oj-vb-fragment id="incs-list1" name="incidents-list">
  <oj-vb-fragment-param name="incidentId"
    value="{{ $page.variables.selectedIncidentId }}"></oj-vb-fragment-param>
</oj-vb-fragment>
```

Note:

The expression is wrapped in `{{ }}`. This is required for the web component framework to enable writeback.

As an alternative to the configuration above, the other recommended way for a page to be notified of updates to a fragment variable is for the fragment to fire a custom event (with the `propagationBehavior` property set to "container") that 'emits' the event to the page, which has a listener to handle the event. See Custom Fragment Events for details.

Deferred Rendering of a Fragment

The default behavior of a fragment is for it to load/run immediately when it's encountered by the page rendering it. By wrapping a fragment in the `<oj-defer>` component, you can control when a fragment loads and renders in a page. The fragment can be hidden until loaded by a trigger. The trigger to load a fragment can either be a configurable or it can be determined by the framework. Configurable triggers that can be used to load a fragment include button clicks, tab selection, dialog open, and oj-bind-if components. In this case, UI events or the application state drives the fragment that is loaded.

Deferring the rendering of a fragment can improve performance, so that, for example, an action chain for a hidden fragment is delayed until the fragment is actually loaded. For examples on using `oj-defer`, see Deferred Rendering in the *JET Developer Cookbook*.

For examples of using `<oj-defer>` with `<oj-vb-fragment>`, see Fragment Patterns below.

Fragment Events

Fragments support several lifecycle events defined by the system. In addition, fragments also support custom events that can be handled by listeners defined in the fragment, and further propagated to the listener bound on the fragment container.

Lifecycle Events

When the lifecycle event is raised, the framework calls the event listener with the name of the event. Fragments can fire these events when the fragment artifacts load, when the fragment state is activated, or when the fragment is disposed. Other lifecycle events are currently not supported by fragments.

Table 1-2 Fragment Event Parameters

Name	Description	Returns
vbEnter	Dispatched when the fragment state is activated. Variable scopes available: • $application: All application variables • $extension: All extension variables • $fragment: All local variables in the fragment	None
vbExit	Dispatched when the fragment is disposed (generally by navigating away from a page or the page is disposed).	None

Framework Events

vbNotificationEvent is an example of a framework event that raises a notification for further processing by a parent container and to display the notification message. This is a special event that is automatically bubbled up to the parent container(s) without any need for binding the event on the fragment component. Other specialized types of notification events, such as SDP vbDataProviderNotification events, also have the same behavior.

Component Events

The behavior and usage of component events in fragments is similar to that in other components. See Component Events.

Custom Events

Custom events can be declared in fragments under the "events" property. There are two types of custom events in fragments:

Event Type	Description
Events that can be handled by the same fragment and its extensions	This type of event is similar to other Visual Builder custom events, and is handled similarly. For details, see Custom Events.
Events that 'emit' a custom event to the fragment container	This type is expressly used for the purpose of propagating to the outer container component (the oj-vb-fragment component). For details, see below.

Event that 'emits' a custom event to the fragment container

By setting the "propagationBehavior" property of a custom event to "container", the event will 'emit to the container" when fired, allowing the fragment's parent container (`oj-vb-fragment`) to listen to the custom event.

For example, if you want to use a fragment event to call an action chain to perform some business logic, or to save data to a REST backend, you would fire a custom event that 'emits to the container' so that a listener on the parent can handle the event and trigger the action chain.

Property	Description
propagationBehavior	When this property is set to `container`, the fragment component (`oj-vb-fragment`) can listen to the fragment event, but fragment listeners cannot listen to the event. When this property is not set, the default value is "self", implying the event can only be handled by the fragment listeners. **Note:** This property is only supported by fragment events.

Example 1-64 A fragment event that is listenable by the parent container

The following code describes a "saveincident" event, where the `propagationBehavior` is set to `"container"`.

```
{
  "description": "An incident form fragment",
  "title": "Incidents Form Fragment",
  "events": {
    "saveincident": {
      "description": "fired when an incident has to be saved. The mutated 
incident data provided in payload",
      "propagationBehavior": "container",
      "payloadType": {
        "data": {
          "id": "string",
          "problem": "string",
          "priority": "string",
          "status": "string",
          "customer": {
            "id": "string"
          }
        }
      }
    }
  }
...
}
```

This allows the `oj-vb-fragment` component that loads the fragment to bind an event listener to the same event, as shown below:

```
<oj-vb-fragment name="incident-form" 
id="[[ $page.functions.fragmentUniqueId ]]" bridge="[[ vbBridge ]]"
    on-saveincident="[[ $page.listeners.saveIncident ]]">
```

```
   <oj-vb-fragment-param name="currentIncident"
    value="[[ $page.variables.currentIncident ]]"></oj-vb-fragment-param>
</oj-vb-fragment>
```

> **WARNING:**
>
> Note the 'on-saveincident' attribute. It is important that the event name be lowercase or camelCase with no hyphens as defined by Web Component DOM event naming conventions.

Auto-wire custom events from fragment to container

When a custom fragment event can be emitted to its parent container (with the `"propagationBehavior"` property set to `"container"`), the custom fragment event can be automatically wired to an event listener on its immediate parent container and/or its extension. To do this, a fragment author can add the `autoWire` property to the event that is to be propagated to the parent container.

For a custom fragment event to be auto-wired to a listener on the parent container, the `autoWire` property must be added:

- On the fragment event to specify the name of the event listener it expects the parent container to have (for example, `"autoWire": "performSaveIncident"`)
- On the parent event listener to enable auto-wiring and further clarify its behavior.

  The `autoWire` event listener property can be set to:

  – `none` (default): Disables automatic wiring of an event listener.

  – `full`: Setting `autoWire` to `full` on the fragment's base parent container event listener allows this event listener to be invoked when the auto-wired event is fired from a fragment that is part of the extended parent container. For example, if an auto-wired event listener is defined on a page, and the page is extended and in turn contains a fragment, when the fragment fires an auto-wired event, the base page's event listener is invoked. If both the fragment's base parent container and extended parent container have an auto-wired event listener defined, both are invoked.

  – `selfOnly`: Setting `autoWire` to `selfOnly` limits the event listener invocation to the one defined on the immediate parent alone.

Example 1-65 Auto-wiring a fragment event and eventListener

The following code shows how a fragment declares an auto-wired event:

```
{
  "description": "A incident form fragment",
  "title": "Incidents Form Fragment",
  "interface": {
     "events": {
       "saveincident": {
        "description": "fired when an incident has to be saved. The mutated
incident data provided in payload",
        "autoWire": "performSaveIncident",
        "propagationBehavior": "container",
        "payloadType": {
          "data": {
```

```
            "id": "string",
            "problem": "string",
            "priority": "string",
            "status": "string",
            "customer": {
              "id": "string"
            }
          }
        }
      }
    }
  },
  "chains": {
    "fireSaveIncidentChain": {
      "variables": {
        "incidentPayload": {
          "type": "fragment:incidentEventPayload",
          "description": "the payload of the incident data to send with
event",
          "input": "fromCaller",
          "required": true
        }
      },
      "root": "fireCustomSaveIncidentEvent",
      "actions": {
        "fireCustomSaveIncidentEvent": {
          "module": "vb/action/builtin/fireCustomEventAction",
          "parameters": {
            "name": "saveincident",
            "payload": {
              "data": "{{ $variables.incidentPayload }}"
            }
          }
        }
      }
    }
  },
  "eventListeners": {
    "fireSaveIncident": {
      "chains": [
        {
          "chainId": "fireSaveIncidentChain",
          "parameters": {
            "incidentPayload": {
              "id": "{{ $variables.currentIncident.id }}",
              "problem": "{{ $variables.currentIncident.problem }}",
              "priority": "{{ $variables.currentIncident.priority }}",
              "status": "{{ $variables.currentIncident.status }}",
              "customer": {
                "id": "{{ $variables.currentIncident.customer.id }}"
              }
            }
          }
        }
      ]
    }
```

```
  },
...
}
```

Here's the code for `page-x`, an extended parent container which uses the above fragment:

```
<oj-vb-fragment name="incident-
form" :id="[[ $page.functions.fragmentUniqueId ]]">
  <oj-vb-fragment-param name="currentIncident"
    value="[[ $page.variables.currentIncident ]]"></oj-vb-fragment-param>
</oj-vb-fragment>
```

Here's the code for `page`, the base parent container of the above fragment that defines the auto-wired listener:

```
...
"eventListeners": {
 "performSaveIncident": {
   "autoWire": "full",
   "chains": [
     {
       "chainId": "performSaveIncidentChain",
       "parameters": {...}
     }
   ]
 }
}
...
```

In this example, when the fragment fires the `saveincident` auto-wired event, the `performSaveIncident` auto-wired event listener from the base page container is invoked because its `autoWire` property is set to `full`.

Let's say the base page container defined the auto-wired listener with `autoWire` set to `selfOnly` as shown here:

```
...
"eventListeners": {
 "performSaveIncident": {
   "autoWire": "selfOnly",
   "chains": [
     {
       "chainId": "performSaveIncidentChain",
       "parameters": {...}
     }
   ]
 }
}
...
```

In this case, no auto-wired listener is invoked because the fragment's immediate parent is the extended page container that does not have an auto-wired listener defined, while the base

page container's `performSaveIncident` auto-wired event listener is set up to be invoked only if the base page container is an immediate parent of the fragment.

Referencing Fragments in Extensions

In an extension, you can reference fragments in the same extension as well as fragments defined in dependent extensions. When an extension references a fragment defined in a dependent extension, the dependent extension name is prepended to the fragment name.

To reference a fragment defined in a dependent extension, the fragment's JSON descriptor must include the `"referenceable": "extension"` property.

```
{
  "description": "A product list fragment",
  "title": "Product List Fragment",
  "referenceable": "extension",
...
}
```

In an extension, you can reference fragments:

- In a page in your extension's App UI,
- In a section template that extends a dynamic container,
- In a field or form template for your extension's dynamic tables and forms,
- In a field or form template that extends a dynamic form or table in a dependent extension.

Here's an example of a page in an extension referencing a fragment in a dependent extension (in this example, `extA` is the name of the dependent extension):

```
<oj-dialog id="newProductDialogDynamic" title="New Product" initial-
visibility="hide">
  <div slot="body" style="border:2px">
    <oj-defer>
      <oj-vb-fragment id="createProd1" name="extA:create-product">
      </oj-vb-fragment>
    </oj-defer>
  </div>
</oj-dialog>
```

Example 1-66 Reference a fragment in a page

In this example, the fragment `products-list` is defined in a dependent extension (`extA`). A page defined in an App UI of an extension can include the `products-list` fragment using a prefix before the fragment name: `extA:products-list`).

```
<oj-vb-fragment id="prod-list" name="extA:products-list"
bridge="[[ vbBridge ]]">
  <oj-vb-fragment-param name="catalog" value="[[ 'US' ]]"></oj-vb-fragment-
param>
</oj-vb-fragment>
```

To bind an event fired by the fragment onto a listener in the calling page of the extension, the event in the fragment must be part of the interface. It must also have the 'propagationBehavior' set to 'container'. For details, see Custom Fragment Events.

This example shows the 'saveproduct' event declared by the fragment 'products-list':

```
"interface": {
  "events": {
    "saveproduct": {
      "description": "fired when a product has been created. The mutated
product is fixed up in a local array and returned",
      "propagationBehavior": "container",
      "payloadType": {
        "data": [
          {
            "id": "string",
            "name": "string",
            "unitPrice": "number",
            "productCategory": "string"
          }
        ],
        "message": "string"
      }
    }
  }
}
```

A page in the extension that references the above fragment from the dependent extension can bind the event to a listener on the page using the on-<*eventname*> attribute:

```
<oj-vb-fragment id="prods" name="extA:products-list" bridge="[[ vbBridge ]]"
  on-saveproduct="[[ $page.listeners.onSaveProduct ]]">

</oj-vb-fragment>
```

> **WARNING:**
>
> It is important that the event name be lowercase or camelCase with no hyphens as defined by Web Component DOM event naming conventions.

Example 1-67 Reference a fragment in a dynamic container template

Generally, the only artifacts in a fragment that can be extended are its model and the JavaScript code. However, if a page in a dependent extension contains a dynamic container, an extension could override the dynamic container's section template(s) to then reference a fragment defined in the dependent extension.

In the following example, the section template references the fragment 'incidents-list' in a dependent extension (using the dependent extension's name as the prefix before the fragment name: `extA:incidents-list`).

```
<!-- dynamic container section template -->
<template id='tmplExtB'>
```

```
  <oj-vb-fragment id="incs-list" name="extA:incidents-list"
bridge="[[ vbBridge ]]">
    <oj-vb-fragment-param name="technicianId"
value="[[ $application.user.userId ]]"></oj-vb-fragment-param>
  </oj-vb-fragment>
</template>
```

It's important to note that the fragment 'incidents-list' must be marked as 'referenceable' so that a dependent extension can use it.

```
{
  "description": "A incidents list layout fragment",
  "title": "Incidents List Fragment",
  "referenceable": "extension",
...
}
```

To bind an event fired by a referenced fragment to a listener in the calling page template, the event in the fragment must be part of the interface. It must also have the '`propagationBehavior`' set to '`container`'. For details, see Custom Fragment Events.

```
"interface": {
  "events": {
      "updatedincidentmessage": {
        "description": "fired when an incident has been updated. The mutated
incident data is provided in payload",
        "propagationBehavior": "container",
        "payloadType": {
          "data": {
            "id": "string",
            "problem": "string",
            "priority": "string",
            "status": "string",
            "customer": {
              "id": "string"
            }
          }
        }
      }
    }
  }
}
```

The template in the extension (that references the fragment) can bind the event to a listener using the "on-<*eventname*>" attribute.

```
<oj-vb-fragment id="incs-list" name="extA:incidents-list"
bridge="[[ vbBridge ]]"
       on-
updatedincidentmessage="[[ $listeners.updateMessageBarWithUpdatedIncident ]]">
</oj-vb-fragment>
```

> **WARNING:**
>
> It is important that the event name be lowercase or camelCase with no hyphens as defined by Web Component DOM event naming conventions.

Example 1-68 Reference a fragment in a dynamic layout form template

A fragment can be referenced from field and form templates used in dynamic forms. When doing so, it's important to pass the context property setup by the layout component (`$dynamicLayoutContext`) to the fragment as a parameter. This context is an umbrella property that contains all other dynamic component-related context variables, such as `$value` and `$metadata`.

In the following example of a form template, the form is rendered using the fragment `dynamic-form-template-employee`:

```
<template id="formTemplateSimple">
  <oj-vb-fragment id="formTemplateSimple_EmpFrag" name="dynamic-form-template-
employee" bridge="[[ vbBridge ]]">
      <oj-vb-fragment-param name="$dynamicLayoutContext"
                            value="[[ $dynamicLayoutContext ]]"></oj-vb-
fragment-param>
      </oj-vb-fragment>
</template>
```

The fragment `dynamic-form-template-employee` stores the layout property in a variable defined in the fragment. Within the fragment markup (HTML), it can be used in an expression like `$variables.dynamicLayoutContext.fields...` (or whichever sub-properties you may need in your template markup) to access sub-properties of the layout context:

```
<oj-input-text :id="[[ $fragment.info.id + '-empname']]"
  label-hint="Employee Name"
  value="{{ $variables.dynamicLayoutContext.fields.firstName.value  }}"></oj-
input-text>
```

$dynamicLayoutContext

The `$dynamicLayoutContext` context property needs to be configured when a fragment is used in form or field templates in layout components. The `$dynamicLayoutContext` allows:

- To write back to 'dynamic layout managed' objects.
  Some fragments are intended to be used both within form templates and field templates.

  Particularly when used within a field template, it can be very desirable to be able to not only read a value provided by the dynamic component field, but also to write back to the same. The `$dynamicLayoutContext` property enables this without requiring you to configure a fragment event to notify the parent of the changed value.

  Using `$dynamicLayoutContext`, you can pass this context property provided by the layout component into the fragment as a reference (using an input parameter as shown in the example above). You can then bind the input component / 'value' property in the fragment to read and write to this variable. By doing this, any changes in the input value is automatically known by the parent layout.
- To consolidate context properties in one basket.

Before fragments were used in layout component templates, authors would have used any number of the context properties (like `$fields`, `$value` etc.) that the parent layout exposed, and bind those to the components they use in the templates. But after adding support for fragments inside templates, a new container boundary is introduced, so these context properties are now no longer available/bindable directly by the components inside the fragment. In order to expose these context properties to fragment components, this top-level context property was introduced.

Extending a Fragment

When extending a fragment, an extension can override the fragment's metadata (JSON) and JavaScript. For example, to extend the fragment `my-example-fragment`, the fragment artifacts in the extension would be *`myexample-fragment`*`-x.json` and *`my-example-fragment`*`-x.js`.

When you extend a fragment, the fragment overrides are picked up automatically.

For example, an extension `extA` might define a fragment `dynamic-form-employee` using the following HTML and model (omitting the JavaScript for this example):

dynamic-form-employee-fragment.html

```
<oj-dynamic-form :id="[[ $fragment.info.id + 'oj-dynamic-form-1' ]]"
  metadata="[[ $fragment.metadata.employeeByIdMetadata.provider ]]"
  layout="{{ $constants.layoutName }}"
  value="{{ $fragment.variables.getEmployeeById }}"
  value-loading="[[ $variables.getEmployeeByIdDetailFormLoadingStatus ]]"
  rendered-fields="{{ $variables.getEmployeeByIdDetailFormRenderedFields }}"
</oj-dynamic-form>
```

dynamic-form-employee-fragment.json

```
{
  "fragmentModelVersion": "22.01.0",
  "description": "Fragment that loads a dynamic form",
  "title": "Fragment Dynamic Form Employee",
  "referenceable": "extension",
  "types": {
    "getEmployeeByIdResponse": "object"
  },
  "interface": {
    "constants": {
      "layoutName": {
        "type": "string",
        "mode": "readWrite",
        "defaultValue": ""
      }
    }
  },
  "metadata": {
    "employeeByIdMetadata": {
      "type": "vb/DynamicLayoutMetadataProviderDescriptor",
      "defaultValue": {
        "endpoint": "employees/getEmployeeById"
      }
    }
  },
```

```
  "variables": {
    "getEmployeeById": {
      "type": "fragment:getEmployeeByIdResponse"
    },
    "getEmployeeByIdDetailFormLoadingStatus": {
      "type": "string",
      "defaultValue": "pending"
    },
    "getEmployeeByIdDetailFormRenderedFields": {
      "type": "any[]"
    }
  }
...
}
```

A downstream extension (`extB`) could extend the fragment in `extA` above, for example, by overriding the constant `layoutName` in order to load a different layout template from the extension layout. The fragment artifacts in `extB` might look like the following (in this example, the JavaScript code is not included because there are no meaningful changes). The `layoutName` constant in the extension redefines the layout to be one defined in its extended layout (`extB/formlayout_extended`).

dynamic-form-employee-fragment-x.json

```
{
  "fragmentModelVersion": "22.01.0",
  "title": "Dynamic form employee fragment extension",
  "description": "A fragment extension for dynamic-form-employee-fragment",
  "extensions": {
    "constants": {
      "layoutName": {
        "description": "layout name override; layout provider loaded in base
fragment",
        "defaultValue": "extB/formlayout_extended"
      }
    }
  },
  "variables": {},
  "chains": {},
  "eventListeners": {},
  "imports": {}
}
```

Fragment Patterns

Example 1-69 Tab Bar containing three tabs, and all tabs except the first one are hidden

In this example, when the page loads, only the 'list' tab item fragment is loaded and rendered. The 'map' and 'schedule' tab items are hidden, and the fragment and associated artifacts are not loaded, and the components inside those fragments are not rendered.

1. The `<oj-vb-fragment>` component is used to isolate the content of each tab item

2. In the switcher associated with the tab bar, the `<oj-defer>` slot is used to hide tabs. The fragments are loaded and rendered when their tabs become visible.

3. For details on configuring the component, see Deferred Rendering in the *JET Developer Cookbook*.

```
<oj-tab-bar selection="{{ $variables.incidentsLayout }}">
<ul>
  <li id="list">List</li>
  <li id="map">Map</li>
  <li id="schedule">Schedule/li>
</ul>
</oj-tab-bar>
<oj-switcher value="[[ $variables.incidentsLayout ]]">
  <div slot="list">
    <oj-vb-fragment id="incLL" name="incidentsListLayout"></oj-vb-fragment>
  </div>

  <oj-defer slot="map">
    <oj-vb-fragment id="incML" name="incidentsMapLayout"></oj-vb-fragment>
  </oj-defer>

  <oj-defer slot="schedule">
    <oj-vb-fragment id="incSL" name="incidentsScheduleLayout"></oj-vb-
fragment>
  </oj-defer>
</oj-switcher>
```

Example 1-70 Content inside a dialog is hidden initially, and loaded when the user opens the dialog

1. The `<oj-vb-fragment>` component is used to isolate the content of the dialog.

2. In the dialog 'body' slot, `<oj-defer>` is used to wrap the `oj-vb-fragment`. When the dialog is opened, the input parameters are passed to the fragment component, and the fragment is loaded and rendered.

3. If the fragment fires an event, binding the event to a listener on the page enables the page to listen to it. The "saveproduct" event has the `"propagationBehavior": "container"` property, so the fragment component on the page can listen to it, and then call the 'onSaveProduct' listener on the page.

4. For details on configuring the component see Deferred Rendering in the *JET Developer Cookbook*.

> **Note:**
>
> It's best to have all the content of the dialog within the fragment and the 'body' slot, rather than splitting it, for example, having the buttons in the footer and having the content within the `<oj-defer>`.

```
<oj-dialog id="newProductDialog" title="New Product" initial-
visibility="hide">
    <div slot="body">
```

```
      <oj-defer>
        <oj-vb-fragment id="createProd1" name="create-product"
                        on-saveproduct="[[ $page.listeners.onSaveProduct ]]"
                        on-
cancelproduct="[[ $page.listeners.onCancelProduct ]]">
          <oj-vb-fragment-param name="products"

value="[[ $page.variables.productList ]]"></oj-vb-fragment-param>
      </oj-vb-fragment>
    </oj-defer>
  </div>
</oj-dialog>
```

The event is declared in the fragment:

```
"saveproduct": {
  "description": "fired when a product has been created. The mutated product
is returned in
        payload",
  "behavior": "notify",
  "payloadType": {
    "data": [
      {
        "id": "string",
        "name": "string",
        "unitPrice": "number",
        "inventory": "number",
        "productCategory": "string"
      }
    ]
  },
  "propagationBehavior": "container"
}
```

Example 1-71 A single fragment used to display different content

It's possible to reuse a fragment in multiple places in the page. To use the same fragment in two different parts of the page, use a different unique id on each `oj-vb-fragment` component.

In the example below, the 'edit-product' fragment is used by two components, and each fragment has a unique id. The parameters and event configurations are also different.

```
<oj-bind-if test="[[ $page.variables.productIdDynamic ]]">
  <oj-vb-fragment id="editProd1" name="edit-product"
    on-saveproduct="[[ $page.listeners.onEditProductDynamic ]]">

    <oj-vb-fragment-param name="products"

value="[[ $page.variables.productListDynamic ]]"></oj-vb-fragment-param>
  </oj-vb-fragment>
<oj-bind-if>

<!-- fragment used for static case -->
<oj-bind-if test="[[ $page.variables.productIdStatic ]]">
  <oj-vb-fragment id="editProd2" name="edit-product"
```

```
    on-saveproduct="[[ $page.listeners.onEditProductStatic ]]">
     <oj-vb-fragment-param name="products"

value="[[ $page.variables.productListStatic ]]"></oj-vb-fragment-param>
  </oj-vb-fragment>
<oj-bind-if>
```

Components

Components are written as an HTML file, using standard HTML syntax.

HTML Source

Components are written as standard HTML files.

The HTML file for a page is located as a peer to the page model, as *name*`-page.html`. This HTML source can be edited as a normal JET page.

There are currently two kinds of expressions, write-back and no write-back. This can be seen in the component properties.

```
<oj-input-text maxlength='30' placeholder="[[$variables.searchText]]"
               value="{{$variables.searchVariable}}"</oj-input-text>
```

VB Switcher Component

The VB switcher web component that is used to display the content of one of many VB flows in a VB page, and to quickly switch which one is displayed.

An API is provided to select which flow to render and to add or remove flows from an array of available flows.

The following features are supported:

- the view and viewModel is persisted when switching flows
- navigation within a switcher element is allowed
- record the transition in the browser history

DOM and viewModel caching

In order to provide a quick switching between flows, support pages with iframe and to preserve the selection and scrolling position, the content of the flow is preserved when switching to an other flow. This is done by by showing and hiding the DOM nodes. The resources taken by a switcher element are only released when the element is removed from the ArrayDataProvider.

Note:

Memory usage

Be aware that having a large amount of flows open in the switcher can result in a large memory usage in the browser.

VB Switcher Navigation

Page navigation inside a switcher element or when switching elements does not update the URL but the change is recorded in the browser history. As a result, the bookmarked page will not restore the current state of the switcher.

Navigation within a switcher element

It is possible to navigate to a different page inside a switcher element. When navigation occur, the URL is not updated but the navigation is recorded in the browser history. Using the browser's back button restores the previous page of the current switcher element. Navigation should be done using the `navigateAction`.

A switcher element can navigate to a different flow in the current App UI, open a different App UI, or navigate to an App UI.

Switching between elements

When switching between elements, the transition is recorded in the browser history. Using the back button restores the previously displayed element. This behavior can be altered using `vbBeforePopState`.

When switching between elements, the page lifecycle events are not dispatched because the page does not enter or exit. `vbBeforeExit` and `vbExit` are dispatched only after an element of the switcher is deleted.

VB Switcher Usage and Properties

The VB switcher is a web component that can only be used in a Visual Builder page. Usage consists of specifying an array of switcher elements, and which one is current.

Limitations

- Only one switcher component can be present in a page.
- When the switcher component is present in a page, no other flow can be displayed in that page (no oj-vb-content component).
- Only flows marked with the `embeddable = "enabled"` property, or flows where the default page is marked with the `embeddable` property, can be embedded in a switcher.

Properties

Name (Type)	Description
data (ArrayDataProvider)	An ArrayDataProvider, where each element of the element array is a switcher element. For adding elements dynamically, it should be an ArrayDataProvider that supports mutation, like the JET MutableArrayDataProvider or the VB ArrayDataProvider2. Either flow or application property is required for the element to be valid.
data.id (String)	The id of the switcher element (required)
data.application (String)	The id of the App UI. (optional, if not specified, the flow property is used with the current App UI)
data.flow (String)	The id of the flow
data.page (String)	The id of the page of the flow to display if different than the default page (optional)

Name (Type)	Description
data.params (String)	An object, where the properties are page or flow input variable names (optional)
currentItem	The id of the flow element to display. If the value is null, no switcher element is displayed. The value can be set to change which switcher element is displayed. If the id does not match an element of the data array, an error is thrown.
bridge (Object)	A reference to the internal property `vbBridge`, which is already available in the VB page model. The value is always "[[vbBridge]]"

VB Switcher Methods

closeItem

Dispatch the `vbBeforeExit` event to all the containers of a switcher element the same way it is displayed when navigating. Returns a promise with the result of the event. This allows a page to veto the closing of a switcher element, for example, when dirty data is detected.

Name (Type)	Description
id (String)	The item to close.
`<return>` (Promise)	A Promise that resolves to a boolean. If the result is false, the application should not remove switcher element from the array. It is recommended to invoke this function in the listener of the `ojBeforeRemove` event of the tabBar component so that the array of switcher elements is not affected.

navigate

Navigate the content of the current switcher element from a page containing the switcher component.

The parameters and the return value are the same as the navigate action.

Name (Type)	Description
options (Object)	The navigate options is an object with the same properties as the navigate action.
options.page (String)	The path to the destination page. The path can be absolute, starting at the application, or it can be relative to the current page. When used in combination with a flow or application, the path cannot be absolute, and it navigates to the page relative to the flow or App UI.
options.flow (String)	The id of the destination flow. Change the content of the flow displayed in the current page. When used in combination with a page, navigates to the page in that flow.
options.application (String)	The id of the destination App UI. Change which App UI is displayed in the host application. When used in combination with a page and flow, navigates to the page in that App UI.
options.target (String)	The target of the destination flow. The valid values are "parent" or "self" (default). This is used in combination with a flow to change the content of the parent flow instead of the nested flow.

Name (Type)	Description
options.history (String)	Defines the effect of the navigation on the browser history. The allowed values are "replace", "skip" and "push" (default). If the value is "replace", the current browser history entry is replaced, meaning that the browser's Back button will not go back to it. If the value is "skip", the URL is left unchanged.
options.params (String)	A key/value pair map that will be used to pass input parameters to a page (optional).
`<return>` (Promise)	A Promise that resolves to an object with the boolean property navigated, indicating if the navigation succeeded.

VB Switcher Events

vbBeforePopState

This event is dispatched when the browser history changes (Back and Forward buttons), and it is used for two purposes:

1. To be notified when a change to the switcher current item will be made due to a browser Back or Forward button.
2. Cancel the default handling by setting `preventDefault` to "true".

Properties

All of the event payloads listed below can be found under `event.detail`.

Name (Type)	Description
item (String)	The potential new current item.
previousItem (String)	The previous item.
pagePath (String)	The path of the potential page to be display.
previousPagePath (String)	The path of the previous page displayed.

VB Switcher Examples

Example 1-72 Switcher elements ADP declaration using JET ojmutablearraydataprovider

```
"switcherArray": {
  "type": "object[]",
  "defaultValue": [
    {
      "flow": "aaa",
      "name": "Flow aaa",
      "id": "a"
    }
  ]
},
"switcherMutableArrayDP": {
  "type": "ojs/ojmutablearraydataprovider",
  "constructorParams": [
    "{{ $variables.switcherArray }}",
    {
```

```
      "keyAttributes": "id"
    }
  ]
},
```

Example 1-73 Switcher elements ADP using vb/ArrayDataProvider2

```
"switcherArray": {
  "type": "object[]",
  "defaultValue": [
    {
      "flow": "aaa",
      "name": "Flow aaa",
      "id": "a"
    }
  ]
},
"switcherADP": {
  "type": "vb/ArrayDataProvider2",
  "defaultValue" : {
    "keyAttributes": "id",
    "data": "{{ $variables.switcherArray }}",
    "itemType": "object"
  }
}
```

Example 1-74 How to mark a page or a flow to be embeddable

```
{
  "title": "Start Page",
  "description": "Landing page of the flow",
  ...
  "navigation": {
    "embeddable": "enabled"
  }
}
```

Example 1-75 Usage in page HTML

```
<oj-vb-switcher
  data="[[ $variables.switcherADP ]]"
  current-item="{{ $variables.selectedItem }}"
  bridge="[[ vbBridge ]]"
  on-vb-before-pop-state="[[ $listeners.beforePopstate ]]">
</oj-vb-switcher>
```

Example 1-76 Entry in imports section of the page definition to load the component

```
"imports": {
  ...
  "components": {
    "oj-vb-switcher": {
      "path": "vb/components/oj-vb-switcher/loader"
```

```
      }
    }
  }
```

Imports

The sections below discuss how to import components, CSS, and modules.

Import Custom Components

JET Custom Components can be loaded using the "`imports`" section in a shell or page.

The "`components`" section contains a map of component IDs to objects which contain a (requireJS) path to the JET Custom Components loader javascript. The ID should match the component tag.

Example 1-77 Example:

```
"imports": {
  "components": {
    "demo-card": {
      "path": "resources/components/democard/loader"
    }
  }
}
```

Import Custom Modules

You can load custom modules inside the "`imports`" section in an application, flow, page, and other containers. The "`modules`" section contains a map of module objects, which in turn contain a 'path' to the module JavaScript loader.

The path property can be a requireJS path to the JavaScript, or it can be a path scheme that resolves to a requireJS path to the JavaScript module loader.

Import Modules Using requireJS Path Mapping

The example below shows how to import two modules in a `page.json` using requireJS path mapping.

```
{
  "imports": {
    "modules" : {
      "converterUtils": {
        "path": "ojs/ojconverterutils-i18n"
      },


      "arrayUtils": {
        "path": "faCommon/arrayUtils"
      }
    }
  }
}
```

- "`converterUtils`" specifies a path to a JET module using the implicit requireJS mapping ('ojs') that is set up for JET modules in VB.
- "`arrayUtils`", on the other hand, uses a requireJS path 'faCommon' that is a requireJS path mapping defined in the application metadata.

Each module defined in the section is available through an un-scoped "`$imports`" built-in variable.

The built-in "`$imports`" context property is un-scoped and limited to the current container to avoid performance issues and module conflicts at different context (for example, `$page`, `$flow`, `$application`).

```
<div>
  <oj-bind-text
    value="[['Last Updated on - '
+ $imports.converterUtils.IntlConverterUtils.dateToLocalIso(new Date())]]">

  </oj-bind-text>
</div>
```

In a page.json action chain, the `assignVariablesAction` uses the external module imported as "`arrayUtils`", to call a filter method, as shown here:

```
{
  "removeTab": {
    "module": "vb/action/builtin/assignVariablesAction",
    "parameters": {
      "$page.variables.switcherArray": {
        "module": "{{ $imports.arrayUtils }}",
        "functionName": "filter",
        "params": [
          {
            "array": "{{ $page.variables.switcherArray }}",
            "callback": "{{ function (p) { return p.id !
== $variables.itemToRemove } }}"
          }
        ]
      }
    }
  }
}
```

where the `arrayUtils` method 'filter' might look like this:

```
class ArrayUtils {
  /**
   * Returns a new array with all elements that pass the test implemented by
the provided function.
   * @param helper assignment helper
   * @param defaultValue the default value of the variable
   * @param {Object} params properties are
   * @param {Array} params.array - the array being filtered
   * @params {Function} params.callback - function used as callback to filter
values.
```

```
   * @see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/
Global_Objects/Array/filter
   * @return {Array} filtered array or the array1 if args invalid
   */
  static filter(helper, defaultValue, params) {
    const array1 = params.array1;
    const callback = params.callback;
    if (Array.isArray(array1) && callback && typeof callback === 'function') {
      return array1.filter(callback);
    }

    console.warn('invalid params provided for the filter method', params);
    return array1;
  }
}
```

Import Modules Using a Global Functions Resource Path

While a JavaScript resource defined within a Visual Builder application can be imported into a container using existing conventions, it is often the case that an extension may want to share it for reuse by downstream extensions.

A downstream extension that extends from an upstream extension can also define its own global functions. Global functions can be defined at each extension level.

For example, a top level extension might have a set of JS modules that are used by layout business rules associated to layout artifacts. Rather than copy the code for the utility resource JavaScript in every downstream extension that requires the same logic, using the 'global functions' feature, an extension can define the resource and declare the rules for its usage in a `functions.json` metadata, . This allows a dependent downstream extension to import the JS module and use it with minimal effort.

As stated, global functions for use within the extension (and its App UIs), or across extensions, must first be declared in a `functions.json` metadata, along with level of access afforded to downstream extensions to the methods the JS module provides.

Note:

The term 'global function' refers to its usage as a globally available module, and must not be confused with a context, such as `$global`.

Define Global Functions

Global functions JavaScript modules must be defined under `resources/functions` for an extension, and the metadata for each JavaScript resource to be declared in `resources/functions/functions.json`.

Note:

Global functions can only be defined for an extension. They cannot be defined under an App UI or other resources folders, including in a unified app.

For example, ext-layout, is an extension that defines two JavaScript files that are shared by all containers, like page, fragment or dynamic layout artifacts, in the current extension. The files - `dateUtils.js` and `standardUtils.js` are located under `ext-layout/ui/self/resources/functions`.

The JavaScript files are meant to be utilities style classes so that when the module is loaded (using requireJS) it returns a map of function names to function callbacks.

Example 1-78 Global function JS module

The `resources/functions` folder contains a `functions.json`, a configuration file that defines the list of JavaScript modules (in the 'files' section).

In this example, the `standardUtils.js`, defined in the location above, defines several static methods, and returns the methods that are allowed for general use:

```
'use strict';

define([], () => {
  class StandardFunctions {
    static join(arr = []) {
      if (arr.length === 0) {
        return '';
      }
      const newArr = (arr.slice(1, arr.length).map((o) => (${o})));
      return [arr[0]].concat(newArr).join(' ');
    }

    /**
     * Returns true if the field provided as parameter #1 contains the
characters provided as parameter #2, else
     * returns false.
     * @param {string} field
     * @param {string} characters
     * @returns {boolean}
     */
    static contains(field = '', characters) {
      return field.indexOf(characters) >= 0;
    }

    /**
     * Returns a string converted from a decimal.
     * @param {number} field
     * @returns {string}
     */
    static convertNumberToString(field) {
      return field.toString();
    }

    /**
     * Convert string to number
     * @param {string} field
     * @return {number}
     */
    static convertStringToNumber(field) {
      return parseInt(field, 10);
    }
```

```
    /**
     * Returns the number of characters in a string.
     * @param {string} field
     * @return {number}
     */
    static lengthOfString(field = '') {
      return field.length;
    }
  }

  return {
    contains: StandardFunctions.contains,
    convertNumberToString: StandardFunctions.convertNumberToString,
    convertStringToNumber: StandardFunctions.convertStringToNumber,
    join: StandardFunctions.join,
    lengthOfString: StandardFunctions.lengthOfString,
  };
});
```

Declaring global functions in functions.json metadata

In the sample above, the `standardUtils` JavaScript module exports an Object with five properties mapped to the function callback. These methods can be declared in the metadata, and then exposed to the current extension and downstream extensions using the `functions.json` metadata configuration.

```
{
  "files": {
    "utils": {
      "path": "standardUtils",
      "label": "Standard Utility Functions",

      "referenceable": "extension",

      "functions": {
        "contains": {
          "params": {
            "field": {
              "label": "field",
              "description": "",
              "type": "string"
            },
            "characters": {
              "label": "characters",
              "description": "",
              "type": "string"
            }
          },
          "return": "boolean"
        },
        "convertNumberToString": {
          "referenceable": "self"
        },
        "convertStringToNumber": {
          "referenceable": "self"
```

```
        },
        "join": {},
        "lengthOfString": {}
      }
    },
    "dateLocalUtils": {
      "path": "date/dateUtils",
      "label": "Date Utility Functions",

      "referenceable": "self",
      "functions": {
        "dateToIsoString": {
          "referenceable": "extension"
        }
      }
    }
  }
}
```

Note:

The metadata in the sample JSON above is edited to show the relevant details. It is not a complete configuration.

The "files" section includes one or more JavaScript files. "utils" is an alias to the JavaScript file "standardUtils.js", defined under the "path" property (the .js extension can be dropped because it is a requireJS module).

The "referenceable": "extensible" declares that the file is accessible to downstream dependent extensions. The file alias "dateLocalUtils", with the path "date/dateUtils", is set to "referenceable": "self", which means it is only accessible to artifacts in the current extension.

The "functions" section can be used to list functions that are available to callers. The function name can be used in expressions, if present (see below). Additionally, function metadata can define whether it can be referenced from the current extension or a dependent downstream extension. While a function can be less permissive about its access, it cannot supersede the access set on the file.

For example, the file "utils" allows access to all dependent extensions (it is set to "referenceable": "extension"), whereas the method "convertNumberToString" within "utils" only allows access to the current extension (it is set to "referenceable": "self") . This is allowed because a function can be less permissive. This means a dependent extension that imports this module, will not be able to call the "convertNumberToString" function (it will result in a log error).

Another example, is where the file "dateLocalUtils" that defines a function "dateToIsoString", which expands its access beyond what the file allows. This is not allowed and ignored. The function can only be called by artifacts in the current extension.

When a function does not define "referenceable", access is set on the file. The default access for a file is "self".

Use Global Functions in a Container

Global functions must be imported into a page (or any Visual Builder container) using the `imports` section of the container metadata (and its `modules` property) before the global functions can be used.

For example, a `layout.json`, defined in the same extension (ext-layout) where the functions metadata is defined, can specify the modules in its imports like this:

```
{
  "imports": {
    "modules": {
      "utils": {
        "path": "self:$functions/utils"
      },
      "dateUtils": {
        "path": "self:$functions/dateLocalUtils"
      },
      "commonUtils": {
        "path": "ext-common:$functions/utils"
      }
    }
  }
}
```

The `path` property uses a scheme for locating the JavaScript resource, particularly functions, using a convention with three elements:

```
{extId}:{resourceType}/{resourceAlias}
```

The path resolves to the actual require path to the JavaScript module loader.

Element	Description
{extId}	Refers to the extension the resource belongs to. `'self'` means the current extension. Any other extension will be identified by its id. • `'self:` - a reserved word, is required to refer to extension level resources. – Example: `"self:$functions/dateLocalUtils"` refers to the current extension functions. The namespace `'self:'` is required to refer to extension resources. • `'ext-common'` - refers to the name of an upstream extension that the current extension depends on. `'utils'` is the resource alias defined there
{resourceType}	Uses a special keyword for the `/functions` resources (for example, `"$functions"`). • `self:$functions` resolves to an actual resource path (`'ext-layout/ui/self/resources/functions'`) • `ext-common:$functions` resolves to an actual resource path (`'ext-common/ui/self/resources/functions'`)

Element	Description
{resourceAlias}	For global functions, `resourceAlias` is the alias of the JavaScript defined in `functions.json`. • `self:$functions/dateLocalUtils'` resolves to `ext-layout/ui/self/resources/functions/date/dateUtils.js'`, where '`dateLocalUtils`' is the alias defined in `functions.json`. • `ext-common:$functions/commonUtils` resolves to `ext-common/ui/self/resources/functions/utils/common.js`, where '`commonUtils`' is the alias for the resource located under '`utils/common.js`', that is defined in `functions.json`.

Here is an example of how `functions.json`, defined under `ext-common/ui/self/resources/functions`, might look:

```
{
  "commonUtils": {
    "path": "utils/common",

    "label": "Common Utility Functions",
    "referenceable": "extension",
    "functions": {
      "titleCase": {}
    }
  }
}
```

Note:

For "ext-layout" to import global functions from "ext-common", it must have a dependency on the "ext-common" (see below).

Reference Global Functions in an Expression

For the layout that defines the above modules, the following expressions can be used to call the methods exposed by a particular global functions module:

- `$layout.modules.commonUtils.titleCase()`
- `$layout.modules.dateLocalUtils.dateToIsoString()`

Where `titleCase` and `dateToIsoString` are the function names defined in `functions.json` in the extension.

Note:

`$modules` is the shortened form for use in the current container.

> **Note:**
>
> Any container that imports modules can expect `$modules` to be available (for example, `$page.modules`, `$fragment. modules`).

An action chain JavaScript can provide `$modules` as part of the context:

```
class MyChain extends ActionChain {

  async run(context, params) {
    context.$layout.modules.dateUtils.today();

    const result = context.$application.modules.appUtils.update(); // can
also call modules in parent scopes

    await Actions.assignVariable(context,
      {
        variable: '$layout.variables.someValue',
        value: result / 2,
      });
  }
}
```

Examples of Accessing Global Functions in Extensions

The following examples describe the level of access available to global functions from extension artifacts.

Example 1-79 extA is a top level extension that defines a functions JavaScript extAUtils.js

extA defines its own functions files, under `ui/self/resources`. Additionally, it also defines functions under an App UI (appUi-A) and under `appUi-A/pages`.

- extA has no dependencies on other extensions
- extA defines a public `extAUtils.js` that can be accessed by downstream extensions.
  - Additionally, UI artifacts such as `some-page.json` can access the extAUtils.

```
extA/
  ui/
    self/
      resources/
        functions/
          extAUtils.js
          functions.json // { extAUtils: { referenceable: "extension" } }


      applications/
        appUIA/
          .../
            some-page.json
```

```
  manifest.json // { dependencies: [] }
```

In the example above, `some-page.json` can have an `imports` section like this:

```
{
  "imports": {
    "modules": {
      "AUtils": {
        "path": "self:$functions/extAUtils"
      }
    }
  }
}
```

The imported module in the example above can be accessed using the expression (`{{ $page.modules.AUtils.someMethod() }}`).

Example 1-80   extA1 depends on extA and defines functions (both private and public)

- extA1 has a dependency on extA
- extA1 defines a private `a1PrivateUtils.js` that can be imported only by extA1 containers.
  - For example, `a1-form-fragment.json` can import the private module.
- It also defines a public `extA1Utils.js` that can be imported by containers in the current and downstream extensions.

```
extA1/
  ui/
    self/
      resources/
        functions/
          private/
            a1PrivateUtils.js
          public/
            extA1Utils.js
          functions.json


      fragments/
        a1-form/
          a1-form-fragment.json
          a1-form-fragment.js
          a1-form-fragment.html


  manifest.json // { dependencies: [ extA ] }
```

In the example above, `a1-form-fragment.json` can import the private module and have an `imports` section. If a module is not declared here, it's not automatically available. For example, 'extA1Utils' is not defined here, so it cannot be used.

```
{
  "imports": {
```

```
    "modules": {
      "A1PrivateUtils": {
        "path": "self:$functions/a1PrivateUtils"
      },
      "extAUtils": {
        "path": "extA:$functions/extAUtils"
      }
    }
  }
}
```

Some fragment markup can call a method exposed on the private utils:

```
<oj-bind-text
text="{{ $modules.A1PrivateUtils.titleCase($variables.name)) }}"/>

<oj-bind-text text="{{ $modules.extAUtils.region($variables.code) }}"/>
```

Because extA1 depends on extA, it can also refer to public functions from extA.

Example 1-81 extB depends on extA and extA1 and defines a functions JS extBUtils.js

- extB defines a layout 'buttons'
- extB defines its own global functions (extBUtils)
- extB also extends the fragment 'a1-form-fragment' from extA1

```
extB/
  dynamicLayouts/
    self/
      buttons/
        layout.json

  ui/
    self/
      resources/
        functions/
          extBUtils.js
          functions.json
        js/


    ext-A1/

      fragments/
        a1-form/
          a1-form-fragment-x.json
          a1-form-fragment-x.js


  manifest.json // dependencies: [ extA, extA1 ]
```

`buttons/layout.json` references the fragment from extA1. It can call the public extAUtils and extA1Utils, in addition to the global functions extBUtils. This could be defined in `layout.json`:

```
{
  "imports": {
    "modules": {
      "extA1Utils": {
        "path": "extA1:$functions/extA1Utils"
      },
      "BUtils": {
        "path": "self:$functions/extBUtils"
      }
    }
  }
}
```

The `layout.html` could define a template that uses a fragment from extA1. The fragment, because it's defined by extA1, can internally call a 'functions' module that is private to extA1.

```
<template id="formTemplateSimple">

    <oj-vb-fragment name="extA1:a1-form"
                    bridge="[[vbBridge ]]">
      <oj-vb-fragment-param name="dynamicLayoutContext"
                            value="[[ $dynamicLayoutContext ]]"></oj-vb-
fragment-param>

      <oj-vb-fragment-param name="foo"
value="[[ $modules.BUtils.titleCase($layout.variables.lucy) ]]">
      </oj-vb-fragment-param>

    </oj-vb-fragment>

</template>
```

extB also extends the fragment from extA1. This means the fragment-x will need to explicitly define all imports it needs from the current extension and any of its upstream extensions. The use of `$base` is not recommended for accessing imports of the base container, because the exact list of imports that are accessible by the extended artifact cannot be easily determined, nor can it be complete. For example, the `a1-form-fragment.json` from extA1 did not import extA1Utils. Likewise, it imports local resources that should be hidden for the current extension.

It is recommended that authors explicitly import the modules they need. A sample imports on `a1-form-fragment-x.json` might look like this:

```
{
  "imports": {
    "modules": {
      "extA1Utils": {
        "path": "extA1:$functions/extA1Utils"
      }
    }
  }
}
```

Example 1-82 extZ extends layout from extB and also depends on extA1

- extZ extends the 'buttons' layout from extB and also defines its own 'zippers' layout.
- extZ also overrides some pages from the unified app.
- extZ also defines its own functions, in addition to having a 'functions' folder at the App UI level (`resources/functions`) folder under appUiZ (see the example below).
  - While there appears to be a `functions.json` defined at the App UI resources level, these cannot be imported into the App UI using the `$functions` scheme. These files can be imported into the App UI pages using the current schemes for importing such files.

```
extZ/
  dynamicLayouts/
    extB/
      buttons/
        layout-x.json

    self/
      zippers/
        layout.json

  ui/
    self/
      applications/
        appUiZ/
          app.json
          pages/
            shell-page.html
            shell-page.json

          resources/
            functions/
              appUiZtils.js
              functions.json

      resources/
        functions/
          extZUtils.js
          functions.json

    base/
      pages/
        root/
          first-page-x.json
      app-flow-x.json

  manifest.json // dependencies: [ extB, extA1 ]
```

The following sections detail the level of access for functions and the `$modules` usage in the various Visual Builder containers.

Functions Access and $modules Usage in extZ/dynamicLayouts/extB/buttons/layout-x

When layout-x is an extension of a layout from extB, it should be possible to import extB's functions, as well as the dependency extension (extA1).

extA is not specified in the dependencies list, so its functions cannot be imported.

Note:

An extension must explicitly import the resources it needs, and **not** use `$base` to access the imports set up by the artifact it extends.

extZ defines its own functions, so layout-x can import any of these.

```
{
  "imports": {
    "modules": {
      "extBUtils": {
        "path": "extB:$functions/extBUtils"
      },
      "extA1Utils": {
        "path": "extA1:$functions/extA1Utils"
      },
      "ZUtils": {
        "path": "self:$functions/extZUtils"
      }
    }
  }
}
```

Functions Access and $modules Usage in extZ/dynamicLayouts/self/zippers/layout-x

The files allowed in the previous section are also allowed here.

Functions Access and $modules Usage in extZ/ui/self/applications/appUiZ/pages/shell-page

The files allowed in the previous section are also allowed here.

Functions Access and $modules Usage in extZ/ui/base/pages/app-flow-x.json

The files allowed in the previous section are also allowed here.

Import Custom CSS

You can load custom CSS through require-css('css![module-path]') inside the `imports` section in an application, flow, page, and other containers.

The `"css"` section contains an array for strings for each CSS import. Each string is a (requireJS) 'path' to the CSS to be loaded. The requireJS path can be an absolute path with respect the application, a relative path with respective to the current context (application, flow or page) or an external URL (for example, a CDN path) that can be accessed by the application. Using custom CSS loading through metadata gives the flexibility to load CSS through require-css vs hardcoding it in the HTML markup.

The CSS resources are typically defined at the extension level and App UI level, and can be imported using the conventions mentioned below.

In the following application structure, ext-A depends on ext-B, and extends a layout "incidents" defined in ext-B. extA defines its own CSS files, under `ui/self/resources`. Additionally, it also defines an App UI (appUi-A) with its own CSS resource.

```
ext-A/
  dynamicLayouts/
    self/
      orders/
        layout.json
    ext-B/
      incidents/
        layout-x.json

  ui/
    self/
      applications/
        appUi-A/
          app.json
          pages/
            shell-page.json
            resources/
              css/
                shell-2.css

          resources/
            css/
              app.css
              shell.css

      resources/
        css/
          ext.css

    base/
      pages/
        root/
          first-page-x.json
      app-flow-x.json
```

Example 1-83   Import CSS in shell-page.json

```
{
  "imports": {
    "css" : [
      "self:/resources/css/ext.css",   // starts from the extension
      "/resources/css/shell.css",      // starts from the App UI
      "resources/css/shell-2.css",     // not supported, will throw an error
      "https://static.oracle.com/cdn/fnd/gallery/2007.0.0/some.css" // same
    ]
  }
}
```

In the example above:

- If the path starts with `self:/`, the path starts at the root of the current extension (for example, `ext-A/ui/self/resources`).
- If the path is absolute, the path starts at the root of the current App UI (appUi-A), equivalent to the path starting with `extA/ui/self/applications/appUi-A/resources`.
- If the path is relative, throw an error because a relative path is not supported.
- If the path is a URL, use that URL.

Example 1-84 Import CSS in app.json

```
{
  "imports": {
    "css" : [
      "self:/resources/css/ext.css",
      "/resources/css/app.css"
    ]
  }
}
```

The `app.json` has access to both the extension level resources (`self:/`) and the App UI ones (starting with `/resources`).

Example 1-85 Import CSS in layout-x, app-flow-x, flow-x, page-x

```
{
  "imports": {
    "css" : [
      "self:/resources/css/ext.css"
    ]
  }
}
```

Extension artifacts can only access resources defined at the current extension level.

Security

The **security** entry provides certain access limits.

The **security** entry provides a way to limit access to UI level artifacts, such as pages, flows, or applications. These artifacts can require either a specific role or a specific permission in order to enter and display the resource. If the user does not have the correct role or permission, the runtime will refuse entry into that UI artifact. Currently the application, flows, and individual pages can be protected in this manner.

Security Configuration

The security configuration is managed in several resources.

The configuration for security resides in the model for each of these resources: `app-flow.json`, *name*-`flow.json`, *name*-`page.json`. If *requiresAuthentication* is false, specifying roles or permissions results in an error. By default an artifact inherits the *requiresAuthentication* property from its parent. If this is not present in the application configuration, it defaults to true. This means that if no security section is defined in any of the artifacts, the application will require authentication when starting.

The configuration follows the format seen in this example:

```
"security": {
  "access": {
    "requiresAuthentication": true/false,
    "roles": ["role1", "role2"],
    "permissions": ["perm1", "perm2"]
  }
}
```

When an anonymous user navigates to an artifact (page, flow or application) and the artifact is secure, the user is prompted to login, and is then redirected to the artifact. This functionality is provided by the default implementation of the Security Provider.

Security Provider

Security for an application is enabled using a pluggable mechanism called **Security Providers**.

In the application model, `app-flow.json`, you can specify a "userConfig" element. The userConfig element selects which Security Provider to use and how to configure it:

Example of an entry in app-flow.json to specify the Security Provider

```
"userConfig": {
  "type": "vb/DefaultSecurityProvider",
  "configuration": {
    "url": "url to some security api"
  }
}
```

A Security Provider takes a configuration object with a url. The url property should point to a REST API. It must be possible to retrieve the current Security Provider configuration via this REST API. The configuration contains user information and configuration information such as loginUrl and logoutUrl.

A Security Provider performs the following functions.

Function	Description
fetchCurrentUser(config)	Fetch the configuration from the url and initialize the userInfo property as well as the loginUrl and logoutUrl properties.
static getUserInfoType()	Return an object describing the type of the user info.
isAccessAllowed(type, path, accessInfo	Check if the current user can access a resource with the given access info. If the user is not authenticated, this method returns *false*. Otherwise, if the user role is one of the roles in accessInfo, or if the user permission is one of the permissions in accessInfo, then the method returns *true*.
handleLoadError(error, returnPath)	This function is called by the client when an error occurs while loading a page. It attempts to handle the load error for a Visual Builder artifact, and returns *true* if it does.

Function	Description
handleLogin(returnPath)	Handle the user login process. Redirects to the login page using the login URL given by the security provider configuration. If defined, the returnPath is added to the login URL using the query parameter name. This is defined in the 'returnPathQueryParam' property of the SecurityProvider class.
handleLogout(logoutUrl)	Handle the user logout process. The default implementation navigates to the URL defined by the logoutUrl argument. If the logoutUrl argument is not defined, it uses the logoutUrl of the SecurityProvider configuration.

User Information

The **userInfo** contains the user information fetched by the Security Provider.

For the default implementation, the **userInfo** has the following type:

```
{
  "userId": "string",
  "fullName": "string",
  "email": "string",
  "roles": "string[]",
  "permissions": "string[]",
  "isAuthenticated": "boolean"
}
```

The **userInfo** is made available to the application with the help of the `$application.user` built in variable. This allows content in the page to be rendered conditionally.

Example 1-86 Example of conditional content rendering

```
<!-- Render 'I am a manager' if manager is a role of the current user -->
<oj-bind-if test='[[!$application.user.roles.manager]]'>
    I am a manager
</oj-bind-if>


<!-- Render the 'Sign In' button if the current user is not authenticated -->
<oj-bind-if test='[[!$application.user.isAuthenticated]]'>
  <oj-button id='signIn' on-oj-action='[[$listeners.onSignIn]]'Sign In</oj-button>
</oj-bind-if>
```

Error Handling

Support for unauthorized error handling is provided by several functions.

When loading an artifact returns an error, the function **handleLoadError** is called with an error object that has a **statusCode** property. If the artifact is secure and the roles and permissions of the current user do not match the ones required by the artifact, the error statusCode is 403. The default implementation of the **handleLoadError** will check if the user is authenticated, and if not, will call the **handleLogin** function. This redirects to the loginUrl provided by the Security Provider configuration.

The default implementation of the Security Provider handles status 401 and 403 errors. Other security schemes will need to implement their own security provider and specify it in the UserConfig section of the application descriptor. To implement your own security provider:

1. Create your own class extending **vb/types/securityProvider** and override any method necessary.
2. If the user information is different, make sure to match the content of the userInfo property and the type information returned by getUserInfoType(), since this determines what information is exposed in the `$application.user` variable.
3. Enter your new type in the "type" section of the userConfig in app-flow.json as well as the URL to retrieve the Security Provider configuration.

Example 1-87 Example of a custom Security Provider

```
define(['vb/types/securityProvider'],
(SecurityProvider) => {
  class TestSecurityProvider extends SecurityProvider {
    handleLogin(returnPath) {
      // implement your own login mechanism here
    }
  }

  return TestSecurityProvider;
});
```

Helper Utilities

The run time provides public JavaScript helpers to help with implementing some features in JavaScript when a lower level of control is desired or needed.

These can be imported in your Javascript module functions.

REST Helper

The REST helper utility allows calling REST endpoints, which are defined in the service definitions.

The Visual Builder runtime uses this helper internally.

The REST helper looks at the *content-type* header, if available, to try to determine how to read and parse the response. If no *content-type* is available, text is assumed.

Table 1-3 REST helper content types

Content type	Response method
contains "json"	Response.json()
starts with "image/"	Response.blob()
application/octet-stream	Response.blob()

This behavior can be overridden using the *responseBodyFormat()* method.

Here's an example of how to use of the REST helper:

```
define(['vb/helpers/rest'], (Rest) => {
...
async callRestViaHelper() {
    const myparameters = { qparam1 : "value1", qparam2 : "value2" } ;
    // parameter info will be appened to endpoint URL using this format: ?
qparam1=value1&qparam2=value2
```

```
    const rest = Rest.get('myservice/myendpoint').parameters(myparameters);
    const result = await rest.fetch();
    return result.body;
}
```

Here's a snippet showing how to use the REST helper with an extension, with the second parameter defining the scope:

```
define(['vb/helpers/rest'], (Rest) => {
...
async callRestViaHelper () {
    const rest = Rest.get('serviceExtensionId:serviceId/endpointId',
{extensionId:myExtensionId});
    const result = await rest.fetch();
}
```

In this example, a header is passed to the REST helper:

```
define(['vb/helpers/rest'], (Rest) => {
...
async callRestViaHelper () {
    const initConfig = { headers : { someHeader : "abc"  } } ;
    // add a header called "someHeader" with value "abc"
    var const rest = Rest.get('myservice/
myendpoint').initConfiguration(initConfig);
    var const result =await  rest.fetch();
}
```

If the header has hyphens, create the `initConfig` object like this:

```
// add a header called "x-dynamic-header" with value "def"
const initConfig = { headers : { ["x-dynamic-header"] : "def" } };
```

Table 1-4 REST helper methods

Method	Parameters	Return Value	Description
static get(endpointId)	endpointId: *serverID/ operationID*, same as RestAction, ServiceDataProvider	Instance of REST object	Factory method
initConfiguration(initConfig)	initConfig: the initConfig of the fetch() Request object	REST helper, to allow chaining of method calls	See the Request Web API
parameters(parametersMap)	parametersMap: object of key/value pairs, same as RestAction 'uriParams'	REST helper	Set the parameter for the call. Parameters defined as path parameters for the endpoint will be inserted in the URL as appropriate; the rest will be appended as query parameters.

Table 1-4 (Cont.) REST helper methods

Method	Parameters	Return Value	Description
requestTransformationFunctions (transformationFunctionMap)	transformationFunctionMap: map of functions.	REST helper	See Call REST Action
requestTransformationOptions (transformationOptionMap)	transformationOptionMap: map of request transform parameters	REST helper	See Call REST Action
responseTransformationFunctions (transformationFunctionMap)	transformationFunctionMap: map of functions.	REST helper	See Call REST Action
body(body)	body: actual payload to send with the request	REST helper	-
hookHandler(handler)	handler: should extend RestHookHandler, and may override the following: `handlePreFetchHook(rest)` `handleRequestHook(request)` - `returns request` `handleResponseHook(response)` - `returns response` `handlePostFetchHook(result)` `handlePostFetchErrorHook(result)`	REST helper	Allows installation of callbacks for various phases of the REST call, which may configure the REST helpers, modify the request and response, or do special processing based on the result or result error. `define(['vb/helpers/rest', 'vb/helpers/rest'], (Rest, RestHookHandler) => { class MyHandler extends RestHookHandler {`
responseBodyFormat(format)	format: one of: *text*, *json*, *blob*, *arrayBuffer*, *base64*, or *base64Url*. The response body type is the same as the corresponding method for Response (except *base64*, which returns just the encoded portion of the *base64* URL).	REST helper	Overrides the default behavior, which looks at the "content-type" header to determine how to read (and parse) the response.
fetch()	-	Promise	Performs the configured fetch() call
toUrl() toRelativeUrl()	-	Promise	Utility methods for building requests and responses that require the endpoint path. Resolves with the full (or relative) path of the endpoint, or empty string if the endpoint is not found.

The REST helper **fetch()** call returns a Promise that resolves with an object that contains the following properties:

Table 1-5 fetch() call return value

Property	Description
response	The Response object from the native fetch() call, or the return from a HookHandler's handleResponseHook, if one is being used.
body	The body of the response object; the helper will attempt to call the appropriate Response method (json(), blob(), arrayBuffer(), etc) based on responseBodyFormat() and Content-Type.

Module Function Event Builder

Within the context of module functions including `main-page.js` and `app-flow.js`, there is an event helper available to allow raising custom events, similar to the Fire Custom Event Action.

The helper is made available to the module function through a context passed to the Module classes constructor, and has two methods available.

Table 1-6 Module function event helper methods

Method	Description
fireCustomEvent(name, payload)	See Fire Custom Event Action.
fireNotificationEvent(options)	See Fire Notification Event Action.

Example 1-88 Usage in a module function

```
'use strict';

define(function () {
  function MainPageModule(context) {
    this.eventHelper = context.getEventHelper();
  }

  MainPageModule.prototype.fireCustom = function (name, payload) {
    return this.eventHelper.fireCustomEvent(name, payload);
  }

  MainPageModule.prototype.fireNotification = function (subject, message) {
    return this.eventHelper.fireNotificationEvent({ subject, message, type:
'info' });
  }

  return MainPageModule;
});
```

Security Helper

The `SecurityHelper` utility provides methods to retrieve security-related data.

getServiceAccessToken() Method

The `SecurityHelper.getServiceAccessToken()` method returns the JWT token for a configured service connection that uses a JWT token based authentication method and that doesn't use the VB Proxy. The method returns an error if the connection type is proxy based (for example, if the connection type is "Always use proxy, irrespective of CORS support" or "Dynamic, the service does not support CORS"). This method is supported for these authentication types:

- Oracle Cloud Account
- OAuth 2.0 User Assertion
- OAuth 2.0 Client Credentials
- OAuth 2.0 Resource Owner Password Credentials

This table provides further details about this method:

Parameter	`servicename`: The name of the Service Connection for which the JWT token is to be retrieved.
Constructor	`public SecurityHelper()`

Here's an example of how this method is used:

```
define(['vb/helpers/securityHelpers'], (helper) => {
…
    async getToken(servicename) {
     let tokenOP = await helper.getServiceAccessToken(servicename);
     return tokenOP;
    }
```

Events

There are several types of events, all of which the application can react to, using the event listener syntax.

There are several types of events in the runtime: page events, flow events, system events, custom or developer-defined system events, component (DOM) events, and variable events. Event types are all handled by executing action chains.

The application reacts to events through event listeners, which declaratively specify action chains to execute when the event occurs.

Event Listener Syntax

An event listener is an object with the following properties:

- "chains": an array of action chains to execute; includes "chainId" and optional "parameters".

- "stopPropagation": optional, used only by custom and component events. An expression that is evaluated immediately; if true, the event will not propagate to the current hander's container's parent.
- "preventDefault": optional, used only by component events. Like "stopPropagation", it is evaluated immediately. If true, The default (DOM) handling is not executed.

The "chainId" refers to an action chain to trigger when this variable changes. Optional parameters can be sent to the action chain in response to the event (see the next section for more details). To gain access to the old or new values, these are exposed in the $event implicit object, where $event.value is the new value and $event.oldValue is the old value.

The following example defines three event listeners; one for the vbNotification built-in event, a custom event listener, and a component listener. The syntax for all three is the same, though how they are invoked is different:

- The built-in vbNotification event is called when that event is fired by the system. No explicit wiring of the listener is required. The name identifies which action should invoke this listener.
- The custom myCustomEventOne, is called when the application explicitly fires that event. As with vbNotification, no explicit wiring of the listener is required.
- onButtonClicked is a component event, and is explicitly bound to a component action.

```
"eventListeners": {
  "vbNotification":
    "chains": [
        {
          "chainId": "application:logEventPayloadChain",
          "parameters": {
            "message": "{{ $event.message }}"
            "type": "{{ $event.type }}"
          }
        }
    ]
  },
  "myCustomEventOne": {
    "stopPropagation": "{{ $event.type === 'error' }}",
    "chains": [
      {
        "chainId": "application:fireEventChain",
        "parameters": {
          "name": "customEventOne",
          "payload": {
            "value1": "some value",
            "value2": 3
          }
        }
      }
    ]
  },
  "onButtonClicked": {
    "chains": [
        {
          "chainId": "application:logEventPayloadChain",
          "parameters": {
            "eventPayload": "{{ $event }}"
          }
```

```
        }
      ],
   }
```

The following HTML example shows explicit component event binding:

```
<oj-button href="#" id='myButton'
           disabled="[[true]]"
           chroming='half'
           on-click='[[$listeners.onButtonClicked]]'>My Button!!!</oj-button>
```

Event Prefix

An event prefix is a way for event listeners to define which custom event they are listening to. Two aspects of the event listener are represented in the reference: the extension where the event is defined, and the scope. The syntax for the scope is the same in the base and in the extension. The extension reference is placed before the scope, and is separated with a slash (/).

Inside base (Local)

The syntax is `myScope:myEventName` where `myScope` can be omitted if it refers to an event defined in this object.

Reference	Description
page:eventName	Refer to an event defined in current page
flow:eventName	Refer to an event defined in the flow containing this page
application:eventName	Refer to an event defined in the App UI
global:eventName	Refer to an event defined in the Unified App

Inside extension (Export)

The syntax is `extension/myScope:myEventName` where extension can be omitted if it refers to an event defined in the base object.

Reference	Alias/Shortcut	Description
base/page:eventName	/page:eventName	Refer to an event defined in the interface section of the base page
base/flow:eventName	/flow:eventName	Refer to an event defined in the interface section of the base flow
base/ application:eventName	/application:eventName	Refer to an event defined in the interface section of the base App UI
base/layout:eventName	/layout:eventName	Refer to an event defined in the interface section of the base layout
base/global:eventName	/global:eventName	Refer to an event defined in the interface section of the base Unified App

Declared Events

Declared events are events that are explicitly defined in the application model, to define a specific contract and type for the event.

Events can be declared at the Application, Flow, or Page level. References to events use prefixes, just like variables and chains.

Events may also be declared in Layouts; when used **within the Layout**, they behave like other Visual Builder events. But to be able to listen to a Layout event **outside of the Layout**, you **must** use the the "dynamicComponent" behavior (below).

Events have a "payloadType" which declares the type of the event payload. This type is limited to simple scalar types, or objects and arrays composed of scalar types; you **cannot** define a "payloadType" that references other type definitions.

Example 1-89 Declaration

```
"events": {
  "myPageEvent": {
    "payloadType": {
      "message": "string",
      "code": "number"
    }
  }
},
```

Example 1-90 Event Listener

The "page:" prefix is required only when listening outside the page, but is always recommended for clarity).

```
"eventListeners": {
  "page:myPageEvent": {
    "chains": [
      {
        "chainId": "handleEvent",
        "parameters": {
          "payload": "{{ $event }}"
        }
      }
    ]
  },
```

Lifecycle (Page and Flow) Events

Lifecycle events are defined by the system to indicate to a container (page, flow, or application) a change in its lifecycle. Event listeners are defined in a page or flow descriptor. When an event is raised, the framework calls the event listener with the name of the event defined in the descriptor.

Event listeners are defined in the page module under the “eventListeners“ property of the container model. Like all event types, a single event can have multiple event listeners. Event listeners call action chains and can pass parameters and return a payload.

The order of execution during navigation from page source to page target is:

1. **vbBeforeExit** is dispatched to the source page.
2. **vbBeforeEnter** is dispatched to the target page.
3. **vbExit** is dispatched to the source page.

4. **vbEnter** is dispatched to the target page.

Table 1-7 Lifecycle Events

Name	Container	Description	Return
vbBeforeEnter	Page	Dispatched to a page before navigating to it. At the point the event is dispatched, the previous page state still exists. Since the target page is not yet initialized, page variables are not available, but input parameters can be accessed using `$parameters`. Navigation to the page can be canceled by returning an object with the property `cancelled` set to `true`. This is useful for redirecting to another page.	{cancelled: boolean}
vbEnter	Page, Flow, Application	Dispatched after all page-scoped variables have been added and initialized to their default values, values from URL, or persisted values. This is a point where additional initialization work for the page (for example, data fetches) can be done. This event is "non-stopping" for a page, but can be stopped for other containers like application or flow. In other words, for application or flow, the processing of the web application will only continue after the chains called by the event ends.	None
vbBeforeExit	Page	Dispatched to a page before exiting it. Navigation away from a page can be canceled by returning an object with the property `cancelled` set to `true`. This is useful when the page has dirty data and leaving the page should not be allowed before saving. This event is dispatched to all pages in the current container hierarchy, starting with the leaf page (deepest nested page) and ending with the shell page (top level). When navigation is triggered by browser history (forward or back button), the payload is an object with the following properties: • `origin`: (String) Specify what triggered the vbBeforeExit event. The only valid value is `popState` • `direction`: (String) Specify if vbBeforeExit was triggered by navigating `backward` or `forward` in the browser history • `steps`: (Number) Specify how many steps navigation goes backward or forward in the history stack • `canBeCanceled`: (Boolean) Whether navigation in the browser can be canceled by returning the object `{ cancelled: true }` to the vbBeforeExit event.	{cancelled: boolean}
vbExit	Page, Flow	Dispatched when exiting the container (page or flow). This event can be used to clean up resources before leaving the page.	None

Table 1-7 (Cont.) Lifecycle Events

<table>
<tr><th>Name</th><th>Container</th><th>Description</th><th>Return</th></tr>
<tr><td>vbBeforeAppInstallPrompt</td><td>Page, Flow, Application</td><td>Dispatched when a PWA receives a BeforeInstallPromptEvent from the browser. The event will be dispatched after vbBeforeEnter, but there is no guarantee that it will be dispatched after vbEnter. The vbBeforeAppInstallPrompt event can be used to display a native application install prompt by calling <code>event.getInstallPromptEvent().prompt()</code>. Currently, this is only supported in Chrome. For PWAs, the event will be handled automatically by the root page.</td><td>{ getInstallPromptEvent() }</td></tr>
<tr><td>vbAfterNavigate</td><td>Page</td><td>Dispatched from the current page after navigation to this page is complete. The payload is an object with these properties:
<code>currentPage</code>: the path of the current page
<code>previousPage</code>: the path of the previous page</td><td>None</td></tr>
<tr><td>vbDataProviderNotification</td><td></td><td>Dispatched when a Data Provider's implicit fetch fails with an error. The event has the following payload:<pre>{
 severity: 'string', // severity level
 detail: 'any', // details of the error,
this could have the Rest failure details
 capability: 'object', // object with
the capabilities configured on the SDP
 fetchParameters: 'object', // object
with the parameters passed to the fetch
 context: 'object', // object
representing the state of the SDP at the
time fetch was initiated
 id: 'string', // uniqueId of the SDP
instance
 key: 'string', // since the event can
be fired multiple times, this identifies
the event instance
},</pre></td><td>None</td></tr>
<tr><td>vbResourceChanged</td><td></td><td>Dispatched when an application has been updated. This event allows the application to notify the user that they need to refresh to view the updated application. A default handler resourceChangedHandler is added in the application template.<pre>{
   error: {
      detail: 'string',
   },
}</pre></td><td>None</td></tr>
</table>

Table 1-7 (Cont.) Lifecycle Events

Name	Container	Description	Return
vbNewContentAvailable		Dispatched when an updated Web PWA service worker has been activated. The event will be dispatched after vbEnter. A typical example of how an application can respond to a vbNewContentAvailable event is to open a dialog prompting the user to reload the page.	None

Component Events

Component events (also known as DOM events) are similar to page events, except that they are fired by components on a page (or other container).

A component event listener can have any name, and is generally associated to a component event property via the binding expression on the component markup. Component event listeners are defined in the Page (or container) module under the `eventListeners` property, much like other Visual Builder events. For example, an event listener for the `selectionChange` event for the <oj-tab-bar> component can be defined within the `eventListeners` section as:

```
"eventListeners": {
  "onSelectionChange": {
    "chains": [
      {
        "chainId": "respondToChange",
        "parameters": {
          "text": "{{ $event.detail.value }}"
        }
      }
    ]
  }
}
```

Component event listeners are called in the same way as page lifecycle event listeners. There can be more than one listener. When there is more than one, they run in parallel.

To reference an event listener from a component, you can use the `$listeners.`*eventListenerName* implicit object. For example:

```
<oj-select-single ... on-selection-change="[[$listeners.onSelectionChange]]"
```

Component Event Objects

Within the context of component event listeners, there are three implicit objects.

- `$event`: The event payload sent by the component.
- `$current`: This represents the second parameter passed to the handler, if any. For JET, this can be either the `$current` binding variable, or the `$data` variable if `$current` does not exist in the component context.
- `$bindingContext`: represents the third parameter passed, if any. For JET, this is the (Knockout) view model, and it will therefore contain the `$current` or `$data` variable as a property.

These variables do not exist outside the listener context. In other words, you can reference these in the listener declaration, but you cannot reference them in the called action chain; any values needed in these variables must be passed explicitly to the action chain as arguments (chain variables).

These three variables represent the arguments passed to the listener, and are not directly tied to specific JET values. Their meaning could be different depending on the context.

For example, if using an event listener within an `<oj-list-item>` item, the value of `$current` could be different whether you are using the `item.renderer` attribute or the `itemTemplate` slot to display the item.

- Within an `item.renderer` script, JET does not define $current, so instead passes $data as the second argument, so the Visual Builder $current is JET/Knockout $data. In some JET contexts, like an`item.renderer` script, you will also need to prefix Visual Builder listeners with (Knockout) $parent in the HTML.
- Within an `itemTemplate` slot, JET defines `$current`, and passes that, so Visual Builder $current is JET $current.

To determine whether JET `$current` exists for your use case., refer to the JET documentation for the component to which you are adding a listener.

Additionally, the developer could decide to pass their own custom object for the parameters. In the example below, the listener is wrapped, so Visual Builder `$current` is "some string", and Visual Builder `$bindingContext` is undefined.

```
<oj-button on-click="{{ function(event, current, bindingContext)
{ $page.listeners.someListener(event, "some string") } }}">
  Click Me!
</oj-button>
```

Component Event Listener "preventDefault" Property

Component event listeners have an additional `preventDefault` property, which can be used to prevent the normal DOM event handling from being executed.

This example uses an expression to check the payload of the event to stop propagation:

```
"eventListeners": {
  "customEventTwo": {
      "chains": [
        {
          "actionsId": "handleEventInMod2PageChain",
          "parameters": {
            "eventPayload": "{{ $event }}"
          },
        }
      ],
      "preventDefault": "{{ $event.type === 'info' }}"
  }
```

Component Event Listener "asyncBehavior" Property

Some components such as the JET table support events that accept async event listeners, where the event accepts a Promise. This allows the component that fired the event to cancel it

asynchronously, if needed. The Promise provided by Visual Builder event listeners can also be resolved or rejected within Visual Builder based on the action chain's behavior.

To opt in to the async behavior for a component event, the `eventListeners` property `asyncBehavior` must be set to `"enabled"`. The default value for this property is `"disabled"`. Before implementing action chain logic, refer to the component docs to make sure you understand the implications of enabling async behavior.

Here's an example of enabling async behavior for a table component's `ojBeforeRowEditEnd` event, with the `asyncBehavior` property set to `"enabled"` within the `eventListeners` property:

```
{
  "eventListeners": {
    "tableBeforeRowEdit": {
      "asyncBehavior": "enabled",
      "chains": [
        {
          "chainId": "beforeRowEditChain",
          "parameters": {
            "rowIndex": "{{$event.detail.rowContext.status.rowIndex}}"
          }
        }
      ]
    }
  }
}
```

The table component bound to the `ojBeforeRowEditEnd` event in the preceding example can be configured as:

```
<oj-table scroll-policy="loadMoreOnScroll"
          id="oj-table-1"
          class="oj-flex-item oj-sm-12 oj-md-12"
          edit-mode="rowEdit"
          selection-mode='{"row": "single"}'
          data="{{ $page.variables.productsADP }}"
          scroll-policy-options.fetch-size="3"
          columns="{{ $page.functions.columnsArray }}"

          on-oj-before-row-edit="[[$listeners.table1BeforeRowEdit]]">
  ...
</oj-table>
```

Fragment Events

See Fragment Events.

Custom Events

Custom events are similar to page events, except that they are not limited to lifecycles. Their event listeners can be defined in a page, flow, or application.

An event name is defined by the user, and is explicitly fired by the application, using the event Actions provided, in the context of a page.

Custom event listeners are defined in the page or flow under the eventListeners property.

One difference between custom events and page events is that they 'bubble' up the containment hierarchy. Any event listeners in a given flow or page for the event are executed before looking for listeners in the container's parent. The order of container processing is:

- The page from where the event is fired.
- The flow containing the page.
- The page containing the flow.
- Recursively up the containment, ending with the application.

Custom and system event behavior can be modified using the stopPropagation property, which prevents the event from bubbling to this event listener's container's parents.

Example 1-91 stopPropagation Example

```
"eventListeners": {
  "customEventTwo": {
    "stopPropagation": "{{ $event.type === 'info' }}"
    "chains": [
      {
        "actionsId": "handleEventInMod2PageChain",
        "parameters": {
          "eventPayload": "{{ $event }}"
        }
      }
    ],
  }...
```

vbNotification Events

The vbNotification event is a built-in custom event, rather than a page, flow, or application event, as it is an event only explicitly fired by the application using the action 'vb/action/builtin/fireNotificationEventAction' (see Fire Notification Event Action)

The payload is an object with these properties:

- "summary": a short summary, subject, or title
- "message": any text meaningful to the application
- "displayMode": "persist" or "transient"
- "type": "error", "warning", "info", or "confirmation"
- "key": an optional GUID, which may be useful for the UI. If not provided, one is generated and provided in the payload.

System Events

System events are identical to custom and page events, except that the framework defines the event.

An event name is defined by the user, and is explicitly fired by the application, using the event Actions provided, in the context of a page.

System event listeners are defined in the page, shell, or flow under the eventListeners property.

System events also propagate or bubble up the page's container hierarchy, executing any listeners. Event bubbling can be stopped.

One difference between system events and page events is that they 'bubble' up the containment hierarchy. Any event listeners in a given flow or page for the event are executed before looking for listeners in the container's parent. The order of container processing is:

- The page from where the event is fired.
- The flow containing the page.
- The page containing the flow.
- Recursively up the containment, ending with the application.

Custom and system event behavior can be modified using the stopPropagation property, which prevents the event from bubbling to this event listener's container's parents.

Example 1-92 stopPropagation Example

```
"eventListeners": {
  "customEventTwo": {
    "stopPropagation": "{{ $event.type === 'info' }}"
    "chains": [
      {
        "actionsId": "handleEventInMod2PageChain",
        "parameters": {
          "eventPayload": "{{ $event }}"
        }
      }
    ],
  }...
```

Event Behavior

Event behavior refers to how the listeners are called in relation to each other, whether the result for the listeners is available, and what form the result would take.

Event behavior is meaningful for base applications, as well as extensions, and is not specific to events defined in the "interface".

Event Behavior Types

The event behavior does not define the *order* in which the listener chains are called; event behaviors define whether they are called *serially* or *in parallel*, whether the Action that raised the event waits for listener resolution, and what the "result" of the listener invocation looks like.

For event behavior, "serially" means:

- All event listener chains for a single event listener (in a container) are called *sequentially*, in declared order. This means that a listener action chain is not called until any previous actions in the chain have finished.
- The event listeners for the *next* container's listeners are not called until the listener action chains for any previous container's event listeners have finished.

The following table describes the event behavior types.

Event Behavior	Description
Notify	*Parallel* - The event is triggered but the application does not wait for the extension to process it. Chain results are **not** available to the Action (or helper) that fired the event (because the listeners are called without waiting). This is the **default** behavior.

<table>
<tr><th>Event Behavior</th><th>Description</th></tr>
<tr><td>NotifyAndWait</td><td> Serial - Each action chain listener must complete (and resolve any returned Promise, if any) before another event listener action chain is called. Chain results are not available to the Action (or helper) that fired the event. </td></tr>
<tr><td>CheckForCancel</td><td> Serial - Each action chain listener must complete (and resolve any returned Promise, if any) before another event listener action chain is called. If any of the listener Chains returns a "success" with a payload of <code>{ "stopPropagation": true }</code>, the application will stop calling event listeners. When calling listeners defined in both extensions and the base application, the listeners in the "closest" extension are called first. In other words, extensions of extensions are called before extensions of the base. This allows higher-priority extensions to cancel listening before the lower-priority extensions (or base) receive the event. Chain results are not available to the Action (or helper) that fired the event. </td></tr>
<tr><td>Transform</td><td>Deprecated. Replaced by TransformPayload.</td></tr>
<tr><td>TransformPayload</td><td> Serial - Each action chain listener must complete (and resolve any returned Promise, if any) before another event listener action chain is called. The "eventListener" will have access to a new context variable, <code>$previous</code>, which is a peer of <code>\$event</code>. This will be the result of the previous listener invocation's chain result, or undefined for the first invocation. The "eventListener" for a "transform" event can also have a "returnType" declaration, analogous to the "payloadType", but corresponding to the <code>\$previous</code> value. If the event declaration has a "returnType", <code>$previous</code> should match the type, otherwise, it will be coerced to the type. When calling listeners defined in both extensions and the base application, the listeners in the "base" are called first. In other words, the base fires an event with a value, and the extensions may optionally modify that value. (This convention is the opposite order of the "cancel" behavior). The final result, "returnType", when all the listeners have been called, will be returned as the result of the fireCustomEventAction that initially raised the event. If the application wishes to use the "transform" mode, the convention it should follow is: For any listeners for an event with a "transform" behavior that references more than one Chain, the Chains are called in array order.
All listeners for an event with a "transform" behavior should pass the <code>$previous</code> as an argument to their Chain (and this will typically be wired by the design time).
All listeners for an event with a "transform" behavior should define a "returnType", which matches the "returnType" for the event declaration.
The Chain variable argument that corresponds to <code>$previous</code> should have an argument that matches the "returnType" of the event, and the Chain should also have a "returnType" that matches the listener's "returnType" (note that " payloadType" and "returnType" cannot currently reference defined Types).
The Chain should have a "returnType" defined, that matches the "returnType" of the event. The design time can add parameters for the listener, and the inputs for the Chain, to provide <code>\$previous</code>, in the same way it currently provides the <code>$event</code>. </td></tr>
</table>

Variable ‘onValueChanged’ Events

Specific to variables, the 'onValueChanged' event is raised by the framework when a variable’s value changes.

To add an event listener to an event, specify it in the 'onValueChanged' property of the variable. Event listeners can only be added to the root variable, not to any sub-objects of the variable structure. It uses the same syntax as other event listeners.

```
"variables" : {
  "incidentId": {
    "type": "string",
    "input": "fromCaller",
    "required": true,
    "onValueChanged": {
      "chains": [
        {
          "chainId": "fetchIncidentChain",
          "parameters": {
            "incidentId": "{{ $event.value }}"
          }
        }
      ]
    }
  }
},
```

Old and new variable values are available in the $event implicit object.

- *$event.oldValue* provides the variable's old value.
- *$event.value* provides the variable's new value.
- *$event.diff* can be used for complex types, where it is necessary to know the properties within the variable that changed.

See the Variables section for details on variables.

Optional parameters can be sent to the action chain in response to the event. See the JSON Action Chains section for more information.

Multiple event listeners can be added for the same event (note that 'chains' is an array property). In this case, the event listeners will be run in parallel with respect to each other.

2 Related Topics

Declarative RequireJS Path Mapping

The application model supports declarative requireJS path mapping, using the "requirejs" property.

String values and expressions are supported. Expressions use the normal 'double-brace' convention to indicate it should be evaluated. Expressions cannot make references to application artifacts because evaluation happens before the application is created, but *can use* Declarative Initialization Parameters.

The "map","paths" and "bundles" sections of the requireJS.config object definition are currently supported:

```
{
  "applicationModelVersion": "19.3.1",
  "id": "myApp",
  "description": "Big Box FixitFast Technician App",
  "defaultPage": "shell",
  "requirejs": {
    "paths": {
      "myPathPrefix": "some/other/path/prefix",
      "expPrefix": "{{ $initParams.myPrefix + '/somepath' }}"
    }
  },
```

For more details, see:

- http://requirejs.org/docs/api.html#config-paths
- http://requirejs.org/docs/api.html#config-map

Service Resolution

This section provides an overview of how the Visual Builder runtime (RT) resolves a service name into the actual service definition that describes how to make a REST request. The behavior is applicable to applications and extensions/app UIs.

Note:

In this section, the term "module" refers to both the base app (base) and to extensions.

An extension depends on one or more modules

- All extensions depend on base.
- An extension may depend on one or more extensions.
- Starting from the extension that is farthest from base, the dependency relationship must be a "straight line from that extension and base".
  - Dependency cycles are not allowed
  - The following dependency relationships are allowed:
    * *ext2* depends on *ext1* depends on *base*
    * *ext3* depends on *base*
    * *ext4* depends on *ext3* depends on *base*

      ext4 depends on *ext2* depends on *ext1* depends on *base*
    * *base*
  - The following dependency relationships are not allowed:
    * *ext1* depends on *ext1*
    * *ext2* depends on *ext1* depends on *ext2*
    * *ext2* depends on *ext1* depends on *ext3* depends on *ext2*
    * *base* depends on *ext1*

Services and Backends can be defined in base and in extensions

- A backend is an object defined in the *catalog.json* artifact of the module. A backend describes how to access a server, including the server URL, required headers, and authentication details.
- The path for the *catalog.json* artifact in base is <app>/services/catalog.json.
- The path for the *catalog.json* artifact in an extension is <extension>/services/self/catalog.json.

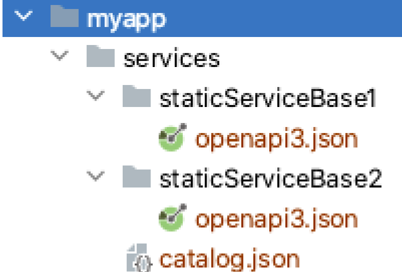
A service can be either dynamic or static

A dynamic service is contributed via the *catalog.json* artifact of a module.

- The entry on the catalog is not the "metadata" of the service but instead the way to fetch such metadata, hence the term "dynamic".
  - The metadata of a service is a document with the OpenAPI definition that describes the endpoints and the types of the service.
- In the *catalog.json* artifact, a service is defined as an entry of the services object:
  - The name of the service is used as the key.
  - The value is an OpenAPI definition with a single endpoint. The response yielded by fetching this endpoint should be the metadata for the dynamic service.
- Example of service entry in *catalog.json*:

```
{
  "backends": {...},
  "services": {
    "dynamicServiceBase1": {
      "openapi": "3.0",
      "info": {
        "x-vb": {
          "extensionAccess": true
        }
      },
      "servers": [{
        "url": "vb-catalog://backends/simplefiles"
      }],
      "paths": {
        "/baseDynamicServiceBase1.json": {
          "get": {
          }
        }
      }
    },
    "dynamicServiceBase2": {"openapi": "3.0"...}
  }
}
```

- A static service is contributed by providing the actual metadata for the service as an artifact of the application.
  - The default path for a static service in the base is `<app>/services/<service name>/openapi3.json`.
  - The default path for a static service in an extension is `<extension>/services/self/<service name>/openapi3.json`.
  - Example of static service:



A service may or may not be registered

- Users can choose to register a service in an artifact, such as *app-flow.json*.

  - The name of the service is used as the registration entry key.
  - The registration value indicates how to locate the service and is typically the path to an artifact or the URI that identifies a dynamic service (for example, `vb-catalog://services/extA:myservice`).
- It is no longer a requirement that all services are registered. See Endpoint ID below for a description of how the RT tries to resolve service names, regardless of whether it's registered or not.
- An advanced usage of the registration is to create new service names for existing services.
- Example:

```
app-flow.json
29  },
30  "services": {
31    "salesService": "services/sales-service.json",
32    "fixitfast": {
33      "path": "services/ifixitfast/openapi3.json"
34    },
35    "Accounts": {
36      "path": "vb-catalog://services/crm"
37    }
38  },
39  "types": {
```

A service is expected to be uniquely identified by its name

- **Not recommended:** if a module uses the same "not registered" name for both a dynamic service and a static service, RT uses the dynamic service.
- A service name should be unique across different modules.
  - DT will warn the user if that's not the case.
  - **Not recommended:** RT handles the scenario in which the same service name appears in multiple modules, as explained later on in this document. However this may lead to "app level" programming errors that are hard to debug.

A module may be able to use services and backends from another module

- A service and a backend can be only used by a different module if that "object" is marked as accessible and if the module using the "object" depends on the module defining the "object".
  - An accessible service or backend defined in *base* can be used by any module.
    - Conversely, *base* cannot use services and backends defined by other modules.
  - An accessible service defined in extension *extB* can only be used by *extB* itself and by extensions that depend on *extB* (directly or indirectly).
    - The same applies to a backend
- Services that are not marked as accessible are private and can only be used by the module that defines them.

- Example of an accessible static service:

```
{
  "openapi": "3.0.0",
  "servers": [...],
  "info": {
    "title": "static service on base",
    "version": "1.0.0",
    "x-vb": {
      "extensionAccess": true
    }
  },
```

- Example of an accessible dynamic service:

```
"services": {
  "dynamicServiceExtA1": {
    "openapi": "3.0",
    "info": {
      "x-vb": {
        "extensionAccess": true
      }
    },
    "servers": [{
      "url": "vb-catalog://backends/base:simplefiles"
    }],
    "paths": {
      "/extADynamicServiceExtA1.json": {
        "get": {
        }
      }
    }
  },
```

- Example of an accessible backend:

```
"backends": {
  "echoid": {      You, 5 months ago • VBS-9361 VBS-
    "extensionAccess": true,
    "servers": [{
      "url": "http://{host}:{port}/service/echoid",
      "variables": {
        "host": {
          "default": "localhost"
        },
        "port": {
          "default": "{@port}"
        }
      }
    }]
  },
```

Endpoint Id

- An endpoint id is a string that identifies a service and an operation within that service.
  - For example, `petstore/getAllPets` refers to the operation from the service `petstore`, and which is identified by the id `getAllPets`.
- With the introduction of extensions, an endpoint id is expected to be "namespaced" to precisely indicate the module that **contains** the service.
  - Examples: `base:petstore/getAllPets`, `extA:storage/createItem`
    - The DT warns the user if the endpoint id is not fully qualified (if it does not have the extension id).
    - Adding a namespace to an endpoint id does not circumvent the visibility rules for services and backends (see Using services and backends from another module above).
      - In other words, using the endpoint id `extB:myservice/getAll` on a module that depends on the extension *extB* fails to resolve if *extB* is not exposing a service named `myservice`.

> **Note:**
>
> The RT can handle endpoint ids without namespaces, however, this is **not recommended** because it may lead to "app level" programming errors that are hard to debug.

- An endpoint id is typically used as an argument to a *RestAction*, and to a code making a REST request via the RT's REST Helper.
  - They are also used on *ServiceDataProvider* and related APIs.

Service Transforms

In order to fetch data required by the application (from a backend service), callers can use the VB RestHelper utility directly or, use a Call Rest action or ServiceDataProvider.

Regardless of the mechanism used, for the request to happen the identifier of the endpoint along with the values of the endpoint "parameters" may need to be processed and transformed into a form that is expected by the target service / endpoint scheme. Likewise, the response received may need to be processed into to a form that is expected by the caller. In order to facilitate this every service configuration must register a transforms module that implements metadata, request and response transform functions.

Type of Transforms API	Description
Request Transforms API	Defines the various request transforms functions that a service can support. These are specific to the capabilities afforded by the service endpoints. For example, a Business Objects based service with a getAll operation (getAll /employees) automatically supports capabilities that include paginate, query, filter, sort. These can be used to define the transforms functions that allow authors to transform input parameters for a request - such as server variables, path and query parameters, header parameters, and other parameters (such as filter criterion, sort criteria) that are provided by the caller initiating the request. For details on request transform function, see Request Transformation Functions.
Response Transforms API	Defines the various response transforms functions that a service can support. Again these are specific to the capabilities afforded by the service endpoints. For example, a Business Objects based service endpoint with a getAll operation (getAll / employees) can return information that can be processed in response transforms functions, such as `paginate` and `body`. For details on reponse transform function, see Response Transformation Functions.
Metadata Transforms API	Defines a standard way for a service to provide a Map by a 'capabilities' that it supports. This allows the caller to be aware of level of support that the service has. Example of capabilities include filter, sort, fetchByKeys etc.

Collectively, all three APIs are referred to as the Transforms API. The specific implementation of the Transforms API for a particular service is referred to by its name (for example, Business Objects Transforms). Refer to the JSDocs for serviceTransforms.js for details.

Standard Transforms File

A sample custom transforms implementation must return an object with three properties (metadata, request and response), each containing methods with specific signatures. Refer to the documentation for each type to understand how to implement them.

```
'use strict';

define([], () => {

  /**
   * Request class implements all the request tranforms functions needed to
transform the input parameters to the
   * Rest call. Each transforms function takes the parameters passed to it
```

```
and transforms it to a form that the
   * target service endpoint expects. Often this involves updating to the
configuration.url
   */
  class Request {
    /**
     * filter builds filter expression query parameter using filterCriterion
object set on the options.
     * @param {Object} configuration
     * @param {Object} options
     * @param {Object} transformsContext a transforms context object that can
be used by authors of transform
     * functions to store contextual information for the duration of the
request.
     * @returns {Object} configuration object, the url looks like ?filter=foo
eq bar
     */
    static filter(configuration, options, transformsContext) {
      // process filter options and fix up configuration.url
      return configuration;
    }

    static body(configuration, options, transformsContext) {}

    static fetchByKeys(configuration, options, transformsContext) {}

    static query(configuration, options, transformsContext) {}

    static paginate(configuration, options, transformsContext) {}

    static select(configuration, options, transformsContext) {}

    static sort(configuration, options, transformsContext) {}

    static vbPrepare(configuration, options) {}
  }

  /**
   * Response class implements all the response tranforms functions needed to
transform the response from the Rest
   * call. Each transforms function takes the parameters passed to it and
transforms the result to a form that the
   * caller expects. Often this involves updating configuration.body, which
   */
  class Response {
    static paginateResponse(configuration, transformsContext) {}

    static bodyResponse(configuration, transformsContext) {}

    someOtherMethod(config) {}
    somePrivateMethod() {}
  }

  class Metadata {
    static capabilities(configuration) {}
  }
```

```
  // Note: If the above classes implement other instance methods its best to
return
  // just the methods a transforms user will need. For example, the response
property
  // below returns just the exported methods
  return {
    metadata: Metadata,
    request: Request,
    response: { paginate: Response.paginateResponse, body:
Response.bodyResponse }
  };
});
```

Usage

Generally, all implementations for the various request, response and metadata transforms functions pertaining to a particular service, are included in a transforms file that is then associated to the service configuration.

In the example below, `vb/BusinessObjectsTransforms` is the pre-defined transforms implementation used with Business Objects based services.

In a `catalog.json` for a Business Objects based service, the transforms file is set in the *backends* (or *services*) section (see the example below). Refer to the Service Connection docs for exact details on how this is configured, and where it is defined.

This file contains the default implementations for the most common transforms functions, that are applied for the majority of service endpoints. However, specific endpoints can override the defaults, and/or add custom transforms implementations.

```
{
    "backends": {
        "crmBO": {
            "headers": {},
            "servers": [
                {
                    "variables": {
                        "faVersionVar": {
                            "default": "11.13.18.05"
                        }
                    },
                    "url": "vb-catalog://backends/fa/crmRestApi/resources/
{faVersionVar}"
                }
            ],
            "transforms": {
                "path": "vb/BusinessObjectsTransforms"
            }
        }
    },
    "services": {
      "journeys": {
        "info": {
          "x-vb": {
            "transforms": {
              "path": "./finderOperationTransform.js"
```

```
                }
              }
            },
            "servers": [
              {
                "x-vb": {
                  "headers": {
                    "Accept": "application/vnd.oracle.openapi3+json"
                  }
                },
                "url": "vb-catalog://backends/hcmBO/journeys/describe"
              }
            ]
          }
        }
}
```

Additionally, the default service transforms can also be overridden at the Service Data Provider, Call Rest action and the Rest Helper levels, using specific properties on each. Refer to the docs for the same for details.

> **Note:**
>
> In VB applications, services are generally BusinessObjects. VB Runtime provides a default implementation for a Business Objects based service.
>
> If your application uses a third party service, you may need to implement a custom transforms module that includes the appropriate metadata, request, response transforms functions specific to the capabilities afforded by the service.

Metadata Transforms

Service authors can implement the Metadata Transforms API in their transforms code, which returns a Map of capabilities (with keys such as filter, sort, fetchByKeys) supported by a particular service endpoint.

The default transforms implementations for Business Objects services processes the endpoint metadata to determine the level of support and returns a Map of capabilities for that endpoint, that can then be understood by Data Provider implementations. The capabilities can be retrieved using the Metadata.capabilities().

The following is a sample structure of the capabilities for a Business Objects based service. The list of capabilities generally applies to all endpoints that the service includes, but authors can provide custom capabilities for different endpoints.

```
{
    "fetchByKeys": {
        "implementation": "lookup",
        "multiKeyLookup": "yes"
    },
    "fetchFirst": {
        "implementation": "iteration"
    },
```

```
    "fetchByOffset": {
        "implementation": "randomAccess"
    },
    "sort": {
        "attributes": "single"
    },
    "filter": {
        "operators": [
            "$eq",
            "$ne",
            "$co",
            "..."
        ],
        "textFilter": true
    }
}
```

The Service Data Provider determines its own capabilities based on the information above.

Signature

The metadata transforms *capabilities* function has the following signature:

```
function capabilities(configuration) {
    const caps = {};

    // process the endpoint configuration and build the capabilities for this
endpoint
    return caps;
}
```

The parameters to this function are

- **configuration**: An object that has the following properties:
  - **endpointDefinition**: The metadata pertaining to the endpoint
  - **initConfig**: The configuration for the current Rest call. The 'initConfig' exactly matches the 'init' parameter of the request, as described in Request.
  - **parameters**: Server variables, path and query parameters
  - **url**: The full URL of the request.

The return value is a Map of capabilities whose JSON structure resembles the example in the previous section. For the full list of properties in the capabilities, refer to the Visual Builder Runtime public `serviceTransforms.js` API, and the JET Data Provider docs for the right set of values for each capability. Business Objects service transforms implementations only support a subset of the capabilities defined by JET Data Provider.

Example

A default implementation for the capabilities supported on an endpoint for a Business Objects based service is already provided as part of the default Business Objects transforms. It returns an object like the example shown below. The capabilities for an endpoint are cached once it's determined that the capabilities are not being modified. It's uncommon for page authors to

want to override the default capabilities, but it can be set on the Service Data Provider variable, if needed.

```
define([], function () {
  /**
   * transforms pertaining to a service or endpoint and not tied to a request.
   *
   * @type {{capabilities: (function(*): {})}}
   */
  class MetadataTransforms {
    /**
     * Returns the capabilities as defined by a DataProvider
     * @param configuration
     * @return {Object}
     * @private
     */
    // eslint-disable-next-line no-underscore-dangle
    static getCapabilities(configuration) {
      const caps = {};
      const c = configuration;
      const epDef = c.endpointDefinition;
      const paramsDef = epDef && epDef.parameters;
      if (paramsDef) {
        const queryParamsDef = paramsDef.query || {};

        if (queryParamsDef) {
          // using the endpoint definition build the capabilities
        }
      }

      return caps;
    }
  }

  class Response {};
  class Request {};

  // Note: as an example, the metadata object is expanded to include just the
sort property
  return {
    metadata: { capabilities: MetadataTransforms.getCapabilities },
    request: Request,
    response: Response
  };
});
```

Translations

The Translations API makes it possible to get localized strings using `$container.translations`.

Translation bundles may now be defined declaratively in Application, Flow, or Page containers. The properties of the "translations" object are the names of the bundle, and the value must contain a "path" property that is the path to the bundle.

When you declare a bundle at the Application level, an optional "merge" property allows you to specify an existing bundle path, which this bundle should merge with and override. This allows overriding existing bundles in JET, or JET CCs, with Application-level bundles. Expressions for "merge" are supported, but they cannot reference Application artifacts, as this evaluation happens before the creation of the Application.

The following paths are supported for "path":

- *container relative*: a path fragment prefixed by "`./`" (dot-slash) will be appended to the declaring container's (flow, page) path. Note that flows and pages are not allowed to reach outside of its container (the path cannot reference parent folders). This means that "`../`" is not allowed anywhere in the path. See the note about Using "merge" below.
- *application relative*: a path fragment without a "`./`" prefix will be relative to the application root. This is discouraged for Flows or Pages, except where a RequireJS path mapping is being used.
- *absolute*: paths that contain a host are used as-is.

The bundle must be in a folder named `nls` : the path can be any depth, but the last folder in the path must be `nls`, such that the root bundle is in the `nls/` folder.

Translation bundles have the standard JET bundle format. String resolution uses the JET oj.Config.getLocale() to get the current locale for the context.

Caution:

Using "merge"
When using "merge", take care to use requireJS mapped references consistently. A common failure is when the "merge" property does not use a requireJS mapping, but the defining path to the bundle does use a mapping. For example,when a CCA is loaded using a requireJS path ("mapped/foo/loader") and it references the bundle using a relative path ("./resources/nls/strings"), the app flow MUST also use the mapping: ("merge": "mapped/foo/resources/nls/strings").

When a dot (".") is used as a prefix in the bundle paths, be aware that "merge" **will not work**. Internally, Visual Builder 'normalizes' bundle paths, so the actual paths used to define the bundle do not have a "dot" prefix.

For example, the declaration below defines a bundle, and then overrides it; note the use of the "dot" prefix everywhere except the *"merge".* If "merge" is used in a a declaration in `app-flow.json`, which is typical, the "dot" prefix on the "path" properties are optional.

```
"translations": {
  "translations" : {
    "app" : {
      "path" : "./resources/strings/app/nls/app-strings"
    },
    "appoverride" : {
      "merge": "resources/strings/app/nls/app-strings",
      "path" : "./resources/strings/override/nls/override-strings"
    }
  },
```

Example 2-1 Bundles

Two bundles, `translations.js` and `moreTranslations.js`, are defined in a Page model JSON, named "app" and "anotherBundle":

```
"translations": {
  "app": {
    "path": "./resources/nls/translations"
  },
  "anotherBundle": {
    "path": "./resources/nls/moreTranslations"
  }
},
```

The corresponding expression syntax would be as follows, with one expression per bundle:

```
<h4><oj-bind-text value="[[$page.translations.anotherBundle.description]]"</oj-bind-
text></h4>
<span>
  <oj-bind-text value="[[$page.translations.format('app', 'info.instructions',
{ pageName: 'index.html' }) ]]"</oj-bind-text>
</span>
<br/>
```

Example 2-2 Overriding both JET strings and a component's strings

```
{
  "id": "demoCardDemo",
  "description": "Custom Component, Demo Card, with methods",
  "defaultPage": "shell",
  "translations": {
    "main": {
      "path": "resources/nls/translations",
      "merge": "ojtranslations/nls/ojtranslations"
    },
    "dcoverride": {
      "path": "resources/nls/demo-card-overrides",
      "merge": "resources/components/democard/resources/nls/demo-card-translations"
    }
  },
```

Expression Language

Similar to variable references and other references, the objectcan be prefixed with the container (for example, `application` in the example below), or you can omit the container, in which case the current container is assumed.

```
<oj-bind-text
  value="[[$translations.format('myPageBundle', 'info.instructions',
{ pageName: 'index.html' })
        ]]">
</oj-bind-text>
<!-- or -->
<oj-bind-text
  value="[[$application.translations.format('myPageBundle',
'info.instructions', { pageName:
        'index.html' }) ]]">
</oj-bind-text>
```

In the example above, the **`format()`** function allows both named and positional replacement.

```
<oj-bind-text
  value="[[ $page.translations.shell.shell_header_title ]]">
</oj-bind-text>
```

Strings can be referenced directly, using `$translations.<bundle>.<string id>`.

Existing Applications That Use Translations

Applications that used translations prior to 18.2.3 **must** manually migrate their translations. Translations previously used the JET configuration, and therefore had one bundle for the entire app. You have several options:

- Declare the bundle. You can choose to break the bundle up logically, but the simplest migration would be to use the exact example above in `app-flow.json`, which uses the path for the existing bundle provided for new apps.
- Change the expression syntax to the new syntax. Assuming you declared your single bundle in the same manner as the Bundles example, named "app":
  - For just the translated string, change `$application.translations.get(`*`key`*`)` to `$application.translations.app.`*`key`*
  - For Strings that require replacement, change `$application.translations.get(`*`key`*`,` *`arguments`*`)` to `$application.translations.format('app',` *`key`*`,` *`arguments`*`)`

Specifying the Locale

By default, VB defers to JET to determine the current locale for the client. This is typically done by first looking at the `<html>` tag 'lang' attribute, and then falling back to some browser settings.

There is a "localization" declaration section in the Application model (`app-flow.json`) that contains a "locale" property, which allows the developer to specify an alternate locale. This configures the JET ojL10n plugin to use this locale.

Expressions may be used, but the application is not created at this point, and therefore no application functions or variables are available. Instead, the developer must provide the necessary JavaScript. The developer should also set the 'lang' attribute on the `<html>` tag, so that JET, and anything that uses JET, will also use this locale.

Example 2-3 Locale Example

```
{
  "id": "demoCardDemo",
  "description": "Custom Component, Demo Card, with methods",
  "defaultPage": "shell",
  "services": {},
  "translations": {
    "main": {
      "path": "resources/nls/translations",
    },
  },
  "localization": {
    "locale": "{{ determineLocale() }}"
  },
  "types": {}
}
```