Oracle® Communications Solution Test Automation Platform

User Operations Guide

Release 1.25.1.0.0
G23299-02
May 2025

ORACLE

Oracle Communications Solution Test Automation Platform User Operations Guide, Release 1.25.1.0.0

G23299-02

Contents

Preface

This document describes how to implement and use Oracle Communications Solution Testing Automation Platform.

Audience

This document is intended for anyone who installs, configures, administers, or customizes Solution Testing Automation Platform.

Documentation Accessibility

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

Access to Oracle Support

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

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.

Part I
Learning About STAP

Learn about concepts and terms used in Oracle Communications Solution Test Automation Platform (STAP).

1

About Solution Test Automation Platform

Learn about Oracle Communications Solution Test Automation Platform (STAP), its key features, benefits, and architecture.

Topics in this chapter:

Introduction to STAP

STAP is a powerful automation platform that allows users to automate their end-to-end business use cases without writing a single line of code. By providing a no-code automation solution, STAP enables users to automate their workflows easily with a built-in Behavior-Driven Development (BDD) language, without much technical expertise. This makes it an ideal automation platform for improving efficiency and productivity.

STAP's key feature is Virtual Tenant functionality. Virtual Tenant functionality enables you to simulate customer-like traffic to measure potential issues with a software application under a significant real-time volume of load for an extended period of time. This helps test customer workflows before deploying them in a live environment.

STAP is a highly extensible platform, and comes with several built-in plugins that allows you to interact with different types of application interfaces, such as REST and SOAP.

> **Note:**
>
> STAP can be used for testing in a lab environment and is licensed to be used only on test or lab platforms and environments.

Features of STAP

STAP offers a robust suite of features designed specifically for automating testing processes.

Table 1-1 describes the various features of STAP.

Table 1-1 Features of STAP

Feature	Description
Extensible plugins	Provides a comprehensive set of plugins and frameworks for automating the end-to-end validation of software applications. It supports various types of plugins, including web, mobile, and API testing.

Table 1-1 (Cont.) Features of STAP

Feature	Description
Customer Environment Simulation or Virtual Tenant	Enables the simulation of customer profiles to test software applications under real-world conditions. The Virtual Tenant represents a typical tenant, covering how they run their business and the various subscriptions and services offered.
Monitoring	Monitors application interfaces such as Web or REST endpoints in real-time and provides insights into the performance and behavior of the application, allowing users to identify potential issues and optimize performance.
No-code Automation	Allows users to automate tests without code. This feature makes it easy for all teams to use, including those with no or limited coding knowledge.
End-to-end Scenario Automation	Supports end-to-end scenario automation, which enables users to test complete workflows. This feature helps ensure that software is tested in a real-world scenario, providing accurate results.
Customer Environment Simulation	Allows users to simulate customer environments, making it easier to test software in different environments. This feature helps to identify any potential issues that may arise in different environments.
Integration with Other Testing Tools	Works seamlessly with other testing tools, enabling users to integrate it into their existing workflows. This feature makes it easier for teams to adopt STAP without disrupting their current processes.
Virtual Tenant	Simulates customer traffic to measure potential problems. This functionality is not available at the moment but may be supported in future releases.
Reduce Dependency with Stubs	Helps in designing end-to-end tests without access to a service, prototyping and creating a mock service for runtime.
Data-driven Testing	Supports data-driven testing. Data sets are mapped to the tests to run repeatedly against multiple data sets.
Seed Data Loaders	Loads seed data into target systems with configuration and without any code or scripts.
Swift Issue Detection	Helps detect failures swiftly. The screenshot and test execution video gives a visual replay of the test execution and help in identifying the error.
Error Handling and Logging	Robust error handling and detailed error logging.
Performance and Metrics	Logs performance information which can be used to generate metrics and comparisons with previous runs (builds or releases).
Reports	Generates standard reports and supports plugins to generate reports.
Core Functionality as Library	Integrates the core engine with any application Integrated Development Environment (IDE), and enables you to store data in the file system and include the execution in build systems.
Continuous Integration and Continuous Delivery or Deployment. (CI/CD)	The lightweight STAP core engine library enables you to run the scenarios in CI/CD with ease.
STAP Microservices	Robust automation platform which has a web interface and stores the data in a database.
STAP User Experience	Runtime web application enables the users to configure, run, monitor execution in real-time, and view the results in modern dashboards.
STAP Container	A valuable STAP tool for teams looking to streamline their testing processes and improve the quality and reliability of their software applications. It provides a flexible and scalable testing environment, enabling teams to achieve faster and more efficient testing results.

Benefits of STAP

The key benefits of STAP include:

- **Improved software quality**: STAP helps to improve software quality by automating the tests and identifying potential issues. It provides accurate results that help to ensure that software is functioning as expected.
- **Time-saving**: STAP automates testing, saving time and effort for testing teams. It enables teams to focus on other critical tasks, such as improving software functionality.
- **Scalability**: STAP is designed to handle high traffic and growing demands, making it an ideal automation solution for diverse testing requirements. It supports horizontal scaling, allowing you to add more servers to distribute the load efficiently.

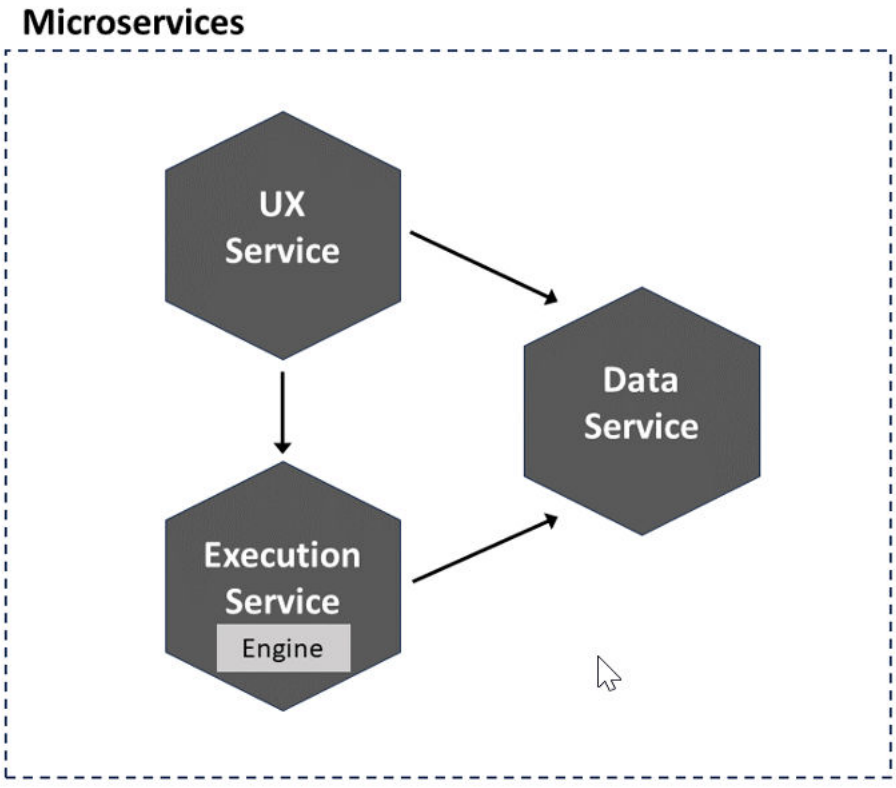
Microservice Architecture

In addition to its extensive automation capabilities, STAP is designed with a microservice architecture. Microservices allows the platform to be broken down into smaller, more manageable components that can work together to deliver the full functionality of the platform.

There are four sub-microservices that make up STAP: the Engine microservice, the execution microservice, the data microservice, and web application (user experience) microservice.

Figure 1-1 shows the STAP architecture.

Figure 1-1 Microservice Architecture



This figure has the following components:

STAP Engine

The STAP Engine microservice is the core of STAP and is responsible for the actual execution of tests and simulation functionality. It enables you to author and run the test cases which interact with the system being tested. It is a standalone library that can be used either independently or as a dependency which enables users to integrate STAP functionality into their existing testing frameworks.

The Engine microservice provides an automation engine that supports end-to-end scenario automation, allowing you to test software applications across multiple systems and components. It helps testing processes achieve faster and more reliable testing results. It is highly extensible, with a plugin architecture that enables users to customize the engine to support specific testing requirements.

The Engine microservice also includes advanced simulation functionality, enabling you to simulate real-world conditions and test applications under a variety of different scenarios. This includes the ability to simulate network latency, data throttling, and other performance factors.

Execution Service

The Execution Service microservice is responsible for running test cases in STAP. It manages the execution of test cases and ensures that all necessary resources are available for testing. The Execution Service can run test cases in parallel, allowing for faster testing and more efficient use of resources.

Data Service

The Data Service microservice is responsible for managing the data used in STAP. It stores test case data, test results, and other important information related to testing. The Data Service is designed to be highly scalable, allowing it to handle large amounts of data without impacting STAP performance.

STAP UI Service

The UI microservice provides a web-based interface allowing you to interact with the STAP application. It offers a user-friendly interface for creating environment details for applications being tested and running test jobs. The service features a dashboard that gives real-time insights into test execution. The history of executions can be tracked using the History dashboard, which provides detailed reports of each test scenario and case.

2

Introduction to STAP Behavior-Driven Development Language

Learn about the Oracle Communications Solution Test Automation Platform (STAP) Behavior-Driven Development (BDD) language and its keywords.

Topics in this chapter:

- Understanding STAP BDD

Understanding STAP BDD Language

STAP BDD is a proprietary language developed by Oracle. It uses a set of special keywords to structure and give meaning to executable business use-case specifications. This approach ensures that the use cases are both human-readable and executable by the testing framework. Each line in a STAP BDD document that is not a blank line has to start with a STAP BDD keyword. Some keywords are followed by text.

There are two types of keywords in the STAP BDD language.

- **Primary** keywords are alphabetic words and end with a colon (:).
- **Secondary** keywords are words and special characters.

> **Note:**
>
> Most lines in a STAP BDD document start with one of the primary or secondary keywords. Any line that is not a blank line must begin with a STAP BDD keyword.

Table 2-1 lists the primary keywords in the STAP BDD language.

Table 2-1 Primary Keywords

Primary Keywords	Description
Scenario	Indicates the beginning of a specific situation or use case and is followed by a name for the scenario.
Description	Describes the use case.
Tags	Defines elements and structure within a use case.
Case	Defines a specific use case.
Data	Refers to the information.
Validate	Indicates the beginning of the validation conditions for the data.
Save	Allows you to specify whether to store the entered or modified data.

Table 2-2 lists the secondary keywords in the STAP BDD language.

Table 2-2 Secondary Keywords

Secondary Keywords	Description
Given	Sets up the initial context or state.
When	Describes the action or event that triggers the behavior.
Then	Specifies the expected outcome or result.
And	Adds additional context or actions within Given, When, or Then steps.
|	Used as a separator.
#	When placed as the first character in a line, used anywhere in the file to denote a comment. Block comments are currently not supported.
' '	Used to indicate the bounds of a string value.
. (dot) , (comma) and ; (semi-colon) -	Step description separators.

The BDD language treats white space in the following ways::

- Indentation: Spaces can be used for indentation and they do not affect the contents.
- Blank Lines: There are no restrictions on using blank lines to separate contents in a BDD document.

BDD Use Case

This example details the process for verifying that discounted rates are applied to Friends and Family accounts through the Diameter Gateway. In the integrated ECE, BRM, and PDC environment, the objective is to ensure that calls between Friends and Family members are charged at a special discounted rate, while calls involving non-Friends and Family members are charged at standard rates.

Pricing Structure

- **Calls between non-Friends and Family members:** $0.05 per minute
- **Calls between Friends and Family members:** $0.01 per minute

Products Involved

- **BRM (Billing and Revenue Management)**
- **ECE (Elastic Charging Engine)**
- **PDC (Pricing Design Center)**

Use-Case Steps

1. **Load Pricing Configurations**: Set up pricing configurations, including discounts for Friends and Family groups.
2. **Create Non-Friends and Family Accounts**: Create accounts that are not associated with the Friends and Family group.
3. **Generate Usage and Validate Charges**: Generate 20 minutes (1200 seconds) of usage through the Diameter gateway for the standard accounts.

> **Note:**
>
> Ensure that the standard (non-discounted) charge of $0.05 per minute is applied, resulting in total charges of $1.00.

4. **Add Accounts to Friends and Family Group**: Add the previously created accounts to the Friends and Family group.
5. **Generate Usage and Validate Discounts**: Generate another 20 minutes (1200 seconds) of usage for these accounts.

> **Note:**
>
> Ensure the Friends and Family discounted rate of $0.01 per minute is applied, resulting in total charges of $0.20.

JSON Data Processing (Release 1.25.1.1.0 or later)

JSON Data Processing refers to manipulating and transforming JSON data using predefined actions. These functions help automate the creation, modification, extraction, and saving of JSON objects to streamline data handling.

The different JSON data processing functions are:

1. Creation and Modification
   - **CREATE_FROM_JSON**: Generates a new entity from JSON data.
   - findAndReplace: Replaces specific values within a JSON object.
2. Extraction and Transformation
   - addFromJsonArray: Extracts data from an array and creates a new JSON object.
   - addFromJson: Extracts specified values from JSON and creates a new JSON object.
   - appendFromJsonArray: Adds new data to an existing JSON structure.
3. Saving the Result
   - newJson: Stores the final processed JSON object for further use.

These functions provide structured ways to interact with JSON data dynamically, ensuring efficient data processing without manual intervention.

CREATE_FROM_JSON

The CREATE_FROM_JSON function creates a new entity based on the provided JSON data. It takes a JSON string as input, uses the JSON data to create a new entity, and performs necessary validation and processing to ensure successful creation. Use the $json action to pass the actual JSON string.

The following sample depicts the input syntax:

```
Data:
| $action | CREATE_FROM_JSON |
| $json | {"data":[{"name":"James Brown","id":"1"},{"name":"Rowan
Blake","id":"2"},{"name":"Nora Miller","id":"3"},{"name":"Lily
John","id":"4"}]} |
```

The following sample depicts the output from the data provided above:

```
| myjson | $JSON{todoJson} | {"data":[{"name":"James Brown","id":"1"},
{"name":"Rowan Blake","id":"2"},{"name":"Nora Miller","id":"3"},{"name":"Lily
John","id":"4"}]} |
```

findAndReplace

Replaces a specified value in the JSON data with a new value.

Syntax: | $findAndReplace | find_value | replace_value |

Description: Searches for occurrences of find_value in the JSON data and replaces them with replace_value

For example,

```
| $json | {"id":"2","name":"Emily Brown","description":"Residential
customer","status":"TODO", "Due Date":"INITIAL DATE", "str":{ "Due
Date":"INITIAL DATE2", "str2":{ "str3":{ "Due Date":"INITIAL DATE3", "str4":
{ "Due Date":"INITIAL DATE4" } } } } } |
| $findAndReplace | Due Date, New Date Value|
```

After update:

```
$json: {"id":"2","name":"Emily Brown","description":"Residential
customer","status":"TODO", "Due Date":"New Date Value", "str":{ "Due
Date":"New Date Value", "str2":{ "str3":{ "Due Date":"New Date Value", "str4":
{ "Due Date":"New Date Value", } } } } }
```

addFromJsonArray

Adds data from a JSON array to a new JSON object.

Syntax: `| array | $addFromJsonArray($json,selector, key1,key2,...) |`

Description: Extracts data from the specified source_array_path in the JSON data and adds it to a new JSON object. The extracted data is mapped to the corresponding keys (key1, key2, and so on.) in the new object.

For example,

This JSON array:

```
{"data":[{"name":"John","age":25},{"name":"Alice","age":30}]}
| array | $addFromJsonArray($json,[*],name) |
```

You will have the following result:

```
{"array":[{"name":"John"},{"name":"Alice"}]}
```

addFromJson

Adds data from a JSON object to a new JSON object.

Syntax: `$addFromJson($json,key1,key2,...)`

Description: Extracts data from the specified keys (key1, key2, and so on.) in the JSON data and adds it to a new JSON object. The extracted data is assigned to the corresponding keys in the new object.

For example,

```
JSON Object - {"name":"John","age":30,"occupation":"Developer"}

| data | $addFromJson($json,name,name,value,value) |

data Runtime Value - {"data":{"name":"John","value":"Developer"}}
```

appendFromJsonArray

Appends data from a JSON array to an existing JSON object.

Syntax: `$appendFromJsonArray($json,source_array_path,key1,key2,...)`

Description: Extracts data from the specified source_array_path in the JSON data and appends it to an existing JSON object. The extracted data is mapped to the corresponding keys (key1, key2, and so on.) in the existing object.

For example,

```
Step:
| array | $addFromJsonArray($json,[*],name,value) |

Runtime Value:

|  JsonArray  |  $newJson  | {"array":[{"description":"Purchase Fees (srvc)
(srvc): Supremo Broadband Installation
Service","remainingAmount.value":19.99}]} |

Step:
| array | $appendFromJsonArray($json,[*],description,remainingAmount.value) |

Runtime Value:

|  JsonArray  |  $newJson  |  {"array":[{"description":"Purchase Fees (srvc)
(srvc): Supremo Broadband Installation
Service","remainingAmount.value":19.99},{"description":"Cycle Forward Fees
(srvc): Supremo Basic Internet Service","remainingAmount.value":12.34}]}  |
```

newJson

Saves the newly created or updated JSON object.

Syntax: `$newJson`

Description: Saves the resulting JSON object to a variable named newJson.

```
Save:
| newJson | $newJson |
```

3

About BDD Operators

Learn about the different operators in Oracle Communications Solution Test Automation Platform (STAP).

An Operator is a function that takes arguments and returns the result of operation as Passed or Failed. Behavior-Driven Development (BDD) operators are used in Validation section of the Test Step.

Topics in this chapter:

String Operators

BDD String Operators use string text as an argument.

The following are the string operators used in BDD:

- STRING_EQUALS
- STARTS_WITH
- ENDS_WITH
- CONTAINS
- MATCHES

> **Note:**
>
> By default (without mentioning operator), BDD uses String Equals as the operator.

BDD Example:

The following example shows how to use a string operator in STAP BDD:

First, set up the variables:

```
Save:
| planType | Premium |
| emailID | JohnDoe@bills.com |
| errorLog | Connection Timeout |
| name | John Doe |
| connectionStatus | Active |
| smsContent | Your Bill Number is 1 |
| billEnd | John Doe Your Bill Number is 1 |
```

The following commands get the response, which contains various variables.

```
Data:
| id | getbill |
```

The following validation will be successful, given the values set above.

```
| $status | 200 |
| planType | Premium |
| errorLog | %STARTS_WITH(Connection) |
| name | %ENDS_WITH(Doe) |
| smsContent | %CONTAINS(Bill Number) |
| billEnd | %CONCAT(${name}, ${smsContent}) |
| emailID | %MATCHES((.*)@(.*)) |
```

Runtime BDD:

The following is the runtime BDD response for the string operator:

```
Then get mock response, validating bill details
Data:
#|  Property  |  Value    |  Runtime Value  |
 |  id        |  getbill  |  getbill        |
Validate:
#|  Property         |  Value                                          |
Property Value                  |  Runtime Value                  |  Result
|
 |  $status          |  200                                            |
200                             |  SUCCESS                        |  PASSED
|
 |  planType         |  Premium                                        |
Premium                         |  Premium                        |  PASSED
|
 |  errorLog         |  %STARTS_WITH(Connection)                       |
Connection Timeout              |  Connection Timeout             |  PASSED
|
 |  name             |  %ENDS_WITH(Doe)                                |  John
Doe                        |  John Doe                          |  PASSED  |
 |  smsContent       |  %CONTAINS(Bill Number)                         |  Your
Bill Number is 1           |  Your Bill Number is 1             |  PASSED  |
 |  billEnd          |  %CONCAT(${name}, ${smsContent})                |  John
Doe Your Bill Number is 1  |  John Doe Your Bill Number is 1    |  PASSED  |
 |  emailID          |  %MATCHES((.*)@(.*))                            |
JohnDoe@bills.com               |  JohnDoe@bills.com              |  PASSED
|
```

Numeric Operators

Numeric operators use numbers as arguments, such as integer, double, big integer, big double, or a saved variable representing these numbers.

Instead of spelled out numeric operators, you have the option to use symbol-based operators.

Table 3-1 lists the numeric operators.

Table 3-1 Operator Symbols

Symbol	Text	BDD Example	Numeric Example
==	%EQUALS()	==${amount} ==20.50	123==123 12.45==12.4 12 == 12.0
!=	%NOT_EQUAL()	!=${value} !=24	123 != 321 12.34 != 12.3456
>	%GREATER_THAN()	>123 >${value}	123>120 123.0 > 120 123 > 120.0
<	%LESS_THAN()	<123 < ${value}	120 < 123 120.0 < 123 120 < 123.0
>=	%GREATER_THAN_OR_EQUAL	>=123 >=${value}	123>=120 123.0 >=120 123 >=120.0
<=	%LESS_THAN_OR_EQUAL	<=123 <=${value}	120 <=123 120 <=123.0 120.0 <=123

BDD Example:

The following example shows how to use a numeric operator in STAP BDD:

First, set up variables:

```
Save:
| billAmount | 2000 |
| discount | 10 |
| transactionId | 5 |
| creditScore | 400 |
| subscriptionFee | 200 |
```

The following commands get the response, which contains various variables.

```
Data:
| id | getbill |

Validate:
| $status | 200 |
| billAmount | == 2000 |
| discount | %EQUAL(10) |
| transactionId | != 1 |
| subscriptionFee | %NOT_EQUAL(0) |
| creditScore | > 200 |
| billAmount | %GREATER_THAN(1500) |
| discount | < 12 |
| transactionId | %LESS_THAN(6) |
| creditScore | %GREATER_THAN_OR_EQUAL(${subscriptionFee}) |
```

```
| billAmount | >=2000 |
| discount | %LESS_THAN_OR_EQUAL(10) |
| subscriptionFee | <= ${creditScore} |
```

The following commands get the response, which validates the bill details:

```
Then get mock response, validating bill details
Data:
#|  Property  |  Value    |  Runtime Value  |
 |  id        |  getbill  |  getbill        |
Validate:
#|  Property         |  Value                                        |
Property Value                  |  Runtime Value                    |  Result
|
 |  $status          |  200                                          |
200                             |  SUCCESS                          |  PASSED
|
 |  billAmount       |  == 2000                                      |
2000                            |  2000                             |  PASSED
|
 |  discount         |  %EQUAL(10)                                   |
10                              |  10                               |  PASSED
|
 |  transactionId    |  != 1                                         |
5                               |  5                                |  PASSED
|
 |  subscriptionFee  |  %NOT_EQUAL(0)                                |
200                             |  200                              |  PASSED
|
 |  creditScore      |  > 200                                        |
400                             |  400                              |  PASSED
|
 |  billAmount       |  %GREATER_THAN(1500)                          |
2000                            |  2000                             |  PASSED
|
 |  discount         |  < 12                                         |
10                              |  10                               |  PASSED
|
 |  transactionId    |  %LESS_THAN(6)                                |
5                               |  5                                |  PASSED
|
 |  creditScore      |  %GREATER_THAN_OR_EQUAL(${subscriptionFee})   |
400                             |  400                              |  PASSED
|
 |  billAmount       |  >=2000                                       |
2000                            |  2000                             |  PASSED
|
 |  discount         |  %LESS_THAN_OR_EQUAL(10)                      |
10                              |  10                               |  PASSED
|
 |  subscriptionFee  |  <= ${creditScore}                            |
200                             |  200                              |  PASSED
|
```

Array Operators

Array operators are used to compare two arrays. The array operators are:

- General Array Operators
- Array Operators for Quoted Strings

General Array Operators

The following operators compare elements in two arrays. To match, the elements must both either be inside quotation marks or both be without them.

If you set the following data:

```
Save:
| $ARRAY{bills1} | 25.213 |
| $ARRAY{bills1} | 30.456 |
| $ARRAY{bills1} | "Bill is complete." |
```

And then you get the response (which contains an array variable called **bills**):

```
Data:
| id | getdata |
```

- The **ARRAY_COMPARE** operator can compare the **bills** array from the returned JSON data to the **bills1** array created above:
  ```
  Validate:

  | bills | %ARRAY_COMPARE($ARRAY{bills1}) |
  ```
  Validation will succeed only if the **bills** array contains the following values in the following order:
  ```
  "bills": [25.213, 30.456, "Bill is complete."]
  ```
- The **ARRAY_COMPARE_IGNORE_ORDER** operator can compare the **bills** array from the returned JSON data to the **bills1** array created above:
  ```
  Validate:

  | bills | %ARRAY_COMPARE_IGNORE_ORDER($ARRAY{bills1}) |
  ```
  Validation will succeed if the **bills** array contains the following values in any order. For example, the following array will pass validation:
  ```
  "bills": [30.456, 25.213, "Bill is complete."]
  ```
- The **ARRAY_IN** operator can compare the **bills** array from the returned JSON data to the **bills1** array created above:
  ```
  Validate:

  | bills | %ARRAY_IN($ARRAY{bills1}) |
  ```
  Validation will succeed if the **bills** array contains any selection of elements matching those in the **bills1** array, in any order. For example, the following array will pass validation:
  ```
  "bills": [30.456, 25.213]
  ```

Array Operators for Quoted Strings

If you set the following data:

```
Save:
| $ARRAY{products1} | "5G Lite Data Service" |
| $ARRAY{products1} | "5G Basic Data Service" |
| $ARRAY{products1} | "123456" |
| $ARRAY{products1} | "Wireless Bundle" |
```

And then you get the response (which contains an array variable called **products**):

```
Data:
| id | getdata |
```

- The **ARRAY_COMPARE_IGNORE_QUOTES** operator can compare the **products** array from the returned JSON data to the **products1** array created above:
- The **ARRAY_COMPARE_IGNORE_ORDER_QUOTES** operator can compare the **products** array from the returned JSON data to the **products1** array created above:
  `Validate:`

  ```
  | products | %ARRAY_COMPARE_IGNORE_ORDER_QUOTES($ARRAY{products1}) |
  ```

  Validation will succeed if the **products** array contains the following values in any order, even though some of the values are not enclosed in quotes. For example, the following array will pass validation:

  ```
  "products": [123456, "5G Basic Data Service", "5G Lite Data Service",
  "Wireless Bundle"]
  ```

- The **ARRAY_IN_IGNORE_QUOTES** operator can compare the **products** array from the returned JSON data to the **products1** array created above:
  Validate:

  ```
  | products | %ARRAY_IN_IGNORE_QUOTES($ARRAY{products1}) |
  ```

  Validation will succeed if the **products** array contains any selection of elements matching those in the **products1** array, in any order, even though some of the values are not enclosed in quotes. For example, the following array will pass validation:

  ```
  "products": [123456, "5G Lite Data Service"]
  ```

- (Release 1.25.1.1.0 or later) The **ARRAY_IN_IGNORE_ORDER_QUOTES** operator can compare the **products** array from the returned JSON data to the **products1** array created above:
  `Validate:`

  ```
  | products | %ARRAY_IN_IGNORE_ORDER_QUOTES($ARRAY{products1}) |
  ```

  Validation will succeed if the **products** array contains any selection of elements matching those in the **products1** array, in any order, even though some of the values are not enclosed in quotes, and disregarding empty strings. For example, the following array will pass validation:

  ```
  "products": [123456, "5G Basic Data Service", ""]
  ```

4

Using Variables

Get an overview of variables and their supported operations in Oracle Communications Solution Test Automation Platform (STAP) Behavior-Driven Development (BDD) language.

Topics in this chapter:

Overview

Variables refer to pieces of data that are stored and used during the execution of a scenario. These variables can hold different types of information, such as numbers, text, or other data types, which are essential for the scenario's logic and flow.

Context refers to the storage of variable values saved during the execution of steps in a scenario.

- A new context is created (or cleared) at the beginning of each scenario execution.
- If the load context option is enabled in **config.properties**, the context is loaded for the scenario. The load context feature is only used at design time and not during the execution of scenarios in a pipeline.

Variable Lifecycle

- Local variable: Local variables are available only for the duration of a scenario.
- Global variables are prefixed with _ and are available from the time they are created until the end of the job.

For example, all variables defined using the Save keyword are local variables unless they begin with an underscore (_).

In the example below, **projectId** and **projectName** are the variable values stored in the context.

```
Save:
#|  Property     |  Value  |
|  _projectId    |  id     |
|  projectName  |  name   |
```

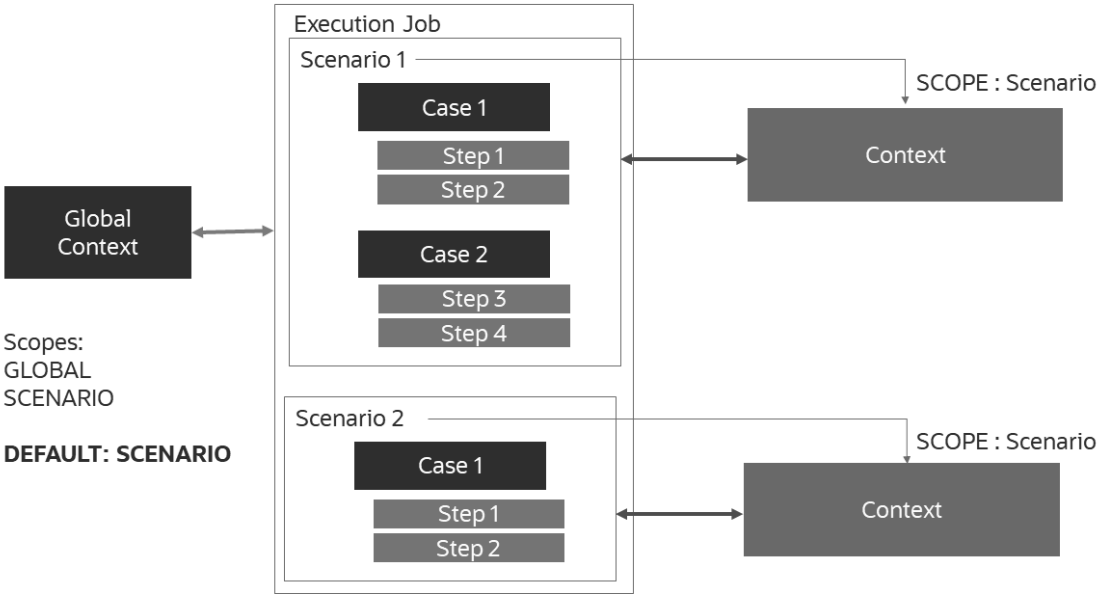
projectId which is prefixed with _, is designated as Global variable and the context stores this variable value from the definition until the job execution completes ie., **_projectId** can be used in any scenario/case/step after its definition.

> **Note:**
>
> If the same variable name is used in the Save section of multiple steps, its value gets replaced.

To save and load the context, use **config.properties** context configuration. For more information, see "Context Folder".

Figure 4-1 shows the variables available during an execution job.

Figure 4-1 Loading Context Configurations



Using Array Variables

An array variable is a type of variable that stores multiple values in a single instance, making it useful for handling lists of data. When working with JSON path, an array variable helps in extracting and storing multiple values from a JSON structure.

The supported operations for array variables include:

- Storing multiple values in a single variable.
- Iterating over the array elements.
- Accessing a single value from the array.

The examples below assume that you are starting with the following JSON:

```
{
  "subscriptions": [
    {
      "id": "1",
      "plan": "Premium",
      "status": "ACTIVE",
      "expiry": "2025-03-15"
```

```
    },
    {
      "id": "2",
      "plan": "Basic",
      "status": "EXPIRED",
      "expiry": "2024-01-01"
    }
  ]
}
```

Getting a single value from an array variable

To extract the **plan** value from the first element of the **subscriptions** array and append it to the **users.plan** array:

```
Save:
| $ARRAY{users.plan} | subscriptions[0].plan |
```

In this case, the **users.plans** array would have one element added to it: **Premium**.

Getting multiple values from an array variable

The following example shows how to get multiple values from an array variable:

```
Save:
| $ARRAY{users.plans} | subscriptions[*].plan |

#returns [Premium,Basic]
```

In this case, the **users.plans** array would have two elements added to it: **Premium** and **Basic**.

Adding a single value to an array variable

The following example shows how to add a single value to an array variable:

```
When add details, adding new subscription plan
Data:
| plan | Gold |
Validate:
| $status | 200 |
Save:
| $ARRAY{users.plan} | subscriptions[2].plan |

#returns Gold
```

Adding multiple values to array

The following example shows how to add multiple values to an array:

```
Then get mock response, read all values that are created above.
Validate:
| $status | 200 |
```

```
Save:
# Store a list of plans from the JSONPath *.plan into the array variable
users.plans
| $ARRAY{users.plans} | subscriptions[*].plan |
```

In the above example , **todos.id** is the array created to save ids of all the tasks read.

> **Note:**
>
> If the todos.id array is already existing, the *.id array values are replaced. When we add an array to existing array indicates creating new array.

Using Dynamic Array Variable

Use **${index}** to create dynamic array variable names. Only **${index}** is allowed as a context variable or ID in array names.

Dynamic Array Variable Name

To use the index of an array to set the name of a variable:

```
RepeatTimes:
| $times | 2 |
Data:
| index | ${nextValue} |
| $urlSuffix | /getarray |
Validate:
| $status | 200 |
Save:
| $ARRAY{dynamicVariable_${index}} | subscriptions[?
(@.status=='ACTIVE')].plan |
```

The following example shows how dynamic values are stored in the test context folder:

```
dynamicVariable_1=[Premium,Premium,Premium,Premium,Premium,Premium,Premium,Pre
mium,Premium,Premium]
dynamicVariable_0=[Premium,Premium,Premium,Premium,Premium,Premium,Premium,Pre
mium,Premium,Premium]
```

Using Array Variable Values

Arrays are used in controlled steps. Iteration happens for the number of times equivalent to length of the array.

To work with the indexes of an array variable,

- Access the array value with index keyword **${index}**. Index starts with **0**.
- **${nextValue}** Gives the next element in the array. **${nextValue}** Can be used in Data, Validate, or Save sections.

The following example shows how to add and read customer bill amounts using array variable values. First, create the array containing the variables:

```
Save:
| $ARRAY{bills1} | 25.213 |
| $ARRAY{bills1} | 20.378 |
| $ARRAY{bills1} | 21.643 |
| $ARRAY{bills1} | 24.211 |
| $ARRAY{bills1} | 22.113 |
```

Then set the code to iterate over the entire array:

```
RepeatTimes:
| $times | $ARRAY{bills1} |
Data:
| index | ${nextValue} |
Validate:
| $status | 200 |
| bills1[${index}] | $ARRAY{bills1[${index}]} |
```

For more information on array variables, see "Controlled actions".

5

BDD Functions

Learn about the different types of Behavior-Driven Development functions in Oracle Communications Solution Test Automation Platform (STAP).

Topics in this chapter:

Overview of BDD Functions

A BDD function is a pre-defined command set that performs an operation and returns a single value. These functions are useful while performing mathematical calculations, string Concatenations (Concat), sub-strings, JSON operations, and so on.

Allowing Commas in Function Data
Function arguments are separated by a comma (,). If the function arguments or variable values contain a comma, you can escape it using %{,}.

Escape only the values provided for function. If there is a comma in the context values or JSON property values escape is not required and done internally.

For example, if a comma is in the text for a variable value:

```
Save:
| subscriptions | Need to purchase 'premium%{,}active plan' from catalog on
Tuesday and 'basic%{,}active plan on Wednesday' |
```

Or if the argument to the pattern matching function contains a comma:

```
| secondPlan | %PATTERN_MATCHER(${subscriptions},'basic%{,}(.*?)',0) |
```

Using Response Properties and Variables in the Functions

Using Data from the Response

If you want to use a property from the response, you can access it by name if you are not using it inside a function. For example, you can assign the name property from the response to the firstName variable like this:

```
Save:
| firstName | name  |
```

However, when you are using that response property inside a function, you should use a dollar sign ($) before the name, like this:

```
Save:
| firstName | %LOWERCASE($name)|
```

Using Scenario Variables
To use saved scenario variables as function argument, use **${<variable>}**. For example,

```
Save:
| firstName | %LOWERCASE($name)|
| updatedFirstName | %UPPERCASE(${firstName})|
```

Using functions in Validate property
You can use functions in validating both properties and values.

For example,

```
Validate:
|  %ARRAY_VALUE(subscriptions[?(@.status=='ACTIVE')].plan)  |  Premium   |
|  %SUBSTRING(${notificationText},33) | test@example.com  |


Validate:
| plan | %SUBSTRING(${subscriptionPlan},0,7) |
| orderID | %PATTERN_MATCHER(${orderConfirmation},\d+,0) |
```

The different types of functions available in the STAP BDD language are:

- String Functions
- Numeric Functions
- JSON or Response Functions
- Data Type Functions
- Date Type Functions
- Format Number Functions

String Functions

String functions are used to manipulate and handle string data.

These functions take a string as an input argument and return a modified string:

- Substring
- Pattern matcher

- Replace
- Replace first
- Concat
- Uppercase and lowercase

SUBSTRING

The SUBSTRING function allows you to retrieve part of a string. You can either the part of a string that starts at a specified character number, or only a specified number of characters starting at a specified character. The format of the function is:

%SUBSTRING(string,beginIndex,noChars)

where:

- string is either a text string or a variable
- beginIndex is he number of the character from which to start reading the string. If noChars is not present, it will read to the end of the string. Set this to 0 to read from the beginning of the string.
- noChars is optional and specifies the exact number of characters to read.

For example, after the commands below, the **emailId** variable contains the string **test@example.com**.

```
Save:
| notificationText | Notification sent at 10:30 AM to test@example.com |

Validate:
| emailID | %SUBSTRING(${notificationText},33) |
```

After the commands below, the **plan** variable contains **Premium**.

```
Save:
| subscriptionPlan | Premium Subscription Activated Successfully |

Validate:
| plan | %SUBSTRING(${subscriptionPlan},0,7) |
```

PATTERN_MATCHER

A pattern matcher retrieves a substring using a regular expression. In STAP, the regular expression used by the pattern matcher contains characters that need to be escaped. If these characters are not escaped, the publish scenario scripts might fail.

The following functions are used to extract specific substrings from a given string:

%PATTERN_MATCHER(<string>,<reg.exp>)
Retrieves a substring which matches the given regular expression pattern.

For example,

```
When set variable, get the Customer information
Save:
| userMessage | Important Notice : 'Your subscription is expiring soon' |
Validate:
```

```
| extractedNotice | %PATTERN_MATCHER(${userMessage},'(.*?)',0) |

extractedNotice returns 'Your subscription is expiring soon'
```

%PATTERN_MATCHER(<string>,<reg.exp>,index)
Retrieves a sub string at the index from the set of matches for a regular expression pattern.

For example,

```
When set variable, get the Customer information
Save:
| orderConfirmation | Order #INV-12345 confirmed for your subscriptionPlan |
Validate:
| orderID | %PATTERN_MATCHER(${orderConfirmation},\d+,0) |

 orderID returns 12345
```

%PATTERN_MATCHER(<string>,<reg.exp>,index,groupIndex)

- index : index of the match
- groupIndex : Group Index of the match

For example,

```
When set variable, get the Customer information
Save:
| notificationText | Notification sent at 10:30 AM to test@example.com |
Validate:
| emailDomain | %PATTERN_MATCHER(${notificationText},@([\w-]+)\.com,0,1) |

emailDomain returns example
```

Replace

The following string manipulation function is used to replace text dynamically:

%REPLACE(<search string>,<replace string>)
Replaces all occurrences of the given search string with replace string.

For example,

```
When set variable, get the Customer information
Save:
| notificationText | Notification sent at 10:30 AM to test@example.com |

Validate:
| modifiedNotification | %REPLACE($
{notificationText},test@example.com,anonymous@example.com) |

modifiedNotification returns Notification sent at 10:30 AM to
anonymous@example.com
```

Replace First

The following string manipulation function is used to replace text dynamically:

%REPLACE_FIRST(<search string>,<replace string>)

Replaces the first occurrence of the given search string with replace string.

For example,

```
When set variable, get the Customer information
Save:
| orderConfirmation | Order #INV-12345 confirmed for your subscriptionPlan |

Validate:
| modifiedOrder | %REPLACE_FIRST(${orderConfirmation},O,BO) |

modifiedOrder returns Border #INV-12345 confirmed for your subscriptionPlan
```

Concat

The following string concatenation function is used to join multiple string arguments into a single string. It helps merge different pieces of text dynamically.

%CONCAT(<arg1>,<arg2>[,<arg3>...]) : Concatenate the given string arguments.

For example,

```
When set variable, getting Customer information
Save:
| subscriptionPlan | Premium Subscription Activated Successfully |
| billingDetails | Your next billing date is 15-03-2025 |

Validate:
| finalMessage | %CONCAT(${subscriptionPlan} ,${billingDetails}) |

finalMessage returns Premium Subscription Activated Successfully Your next
billing date is 15-03-2025
```

Uppercase and Lowercase

These functions are used to convert the string into Lowercase or Uppercase.

%LOWERCASE(<string>) :
Converts the given string into lowercase

%UPPERCASE(<string>) :
Converts the given string into uppercase

For example,

```
When set variable, getting Customer information
Save:
| subscriptionPlan | Premium Subscription Activated Successfully |
| billingDetails | Your next billing date is 15-03-2025 |

Validate:
| planName | %LOWERCASE(${subscriptionPlan}) |
| nextBilling | %UPPERCASE(${billingDetails}) |
```

```
planName returns premium subscription activated successfully
nextBilling returns YOUR NEXT BILLING DATE IS 15-03-2025
```

Numeric Functions

Numeric functions help perform operations on numbers in various sections, including Data, Save, and Validate. They assist in rounding numbers and generating random values dynamically. For supported arithmetic expression, see Numeric Function: Evaluate to Process Arithmetic Expressions.

Rounding Numbers (%ROUND(<arg1>))

This function rounds the given numeric input to the nearest whole number (long numeric value).

For example, %ROUND(3.6) - Returns 4.

Refer to the following BDD Example:

```
When set variables,
Save:
| chocolates | 3.6 |

When buy chocolates,
Data:
| number | %ROUND(${chocolates}) |
```

Generating Random Numbers (%RANDOM())

This function returns a pseudorandom double greater than or equal to 0.0 and less than 1.0

For example, %RANDOM() - Returns 0.753524282283047

Refer to the following BDD Example:

```
When buy chocolates,
Data:
| number | %RANDOM() |
```

Numeric Function: Evaluate to Process Arithmetic Expressions

STAP supports all standard arithmetic operations, such as +,-,*,/. Specify the expression in reverse polish notation or postfix notation.

STAP requires the postfix operation for its arithmetic operations for the following reasons:

- Postfix notations are easier to parse for compiler
- Rules out the need for left - right association and precedence
- Faster to evaluate (less time for parsing)
- Can be expressed without parenthesis
- No 3rd party library dependency required

Using Arithmetic Operations

You must use the following format to perform arithmetic operations:

```
%EVAL(<arithmetic_operations_written_in_reverse_polish_notation>)

# each operand and operator should be comma separated
# to pass in STAP variables use: ${<variable>}

Example:
# (2+1)*3
| name | %EVAL(2,1,+,3,*) |
# (arg3+arg5)
| name | %EVAL(${arg3},${arg5},+) |
```

The following example shows how to evaluate expressions using arithmetic operations:

```
Case: Evaluate Expressions

When set variable, saving various signal datas into variables
Save:
| arg1 | 10 |
| arg2 | 9  |
| arg3 | 4  |
| arg4 | 2  |
| arg5 | 14 |
| arg6 | 20 |

When set variable, evaluating various communication fields
Save:
#|  Property         |
Value                                                                  |  Runtime
Value  |
 |  signalQuality    |
%EVAL(2,1,+,3,*)                                                       |
9              |
 |  transmissionRate |  %EVAL(${arg3},$
{arg5},+)                                              |  18             |
 |  networkLatency   |  %EVAL(${arg1},${arg2},+,$
{arg3},*)                                  |  76             |
 |  packetDropRate   |  %EVAL(${arg1},${arg2},${arg3},*,${arg4},${arg5},-,/,$
{arg6},*,+)  |  -50            |
```

JSON and Response Functions

JSON functions perform operations on response JSON. These can be used in **Validate** or **Save** blocks. JSON functions include the following:

- Array value
- Array size
- Response header

Array Value:

This function retrieves elements from an array using JSON Path:

- **%ARRAY_VALUE(<JSON Path>)**: Returns the first element in the array resolved by the JSON Path.
- **%ARRAY_VALUE(<JSON Path>, <index>)**: Returns the index element in the array resolved by the JSON Path. Index starts from 0.

The following is the response body in JSON format:

```
{
   "user": "John Doe",
   "email": "john@billing.com",
   "subscriptions": [
   {"plan": "Premium", "status": "ACTIVE", "expiry": "2025-03-15"},
   {"plan": "Basic", "status": "EXPIRED", "expiry": "2024-01-01"}
   ]
}
```

The following are some examples of Array Value:

- Get the first email for the matched JSON Path
  %ARRAY_VALUE(emails[?(@.status == 'VERIFIED')].email**)** returns first@email
- Get the email at index 1 for the matched JSON Path
  %ARRAY_VALUE(emails[?(@.status == 'VERIFIED')].email,1**)** returns third@email
- Get the value at index 1 for the matched JSON Path
  %ARRAY_VALUE(emails[?(@.status == 'VERIFIED')].value,1**)** returns 30

The following is a BDD example for an Array Value:

```
Then get mock response, processing Customer subscribed date and subscription
details
Validate:
| firstPlan | %ARRAY_VALUE(subscriptions[?(@.status=='ACTIVE')].plan) |
| activePlanExpiry | %ARRAY_VALUE(subscriptions[?
(@.status=='ACTIVE')].expiry,0) |
```

The following is the runtime BDD response:

```
Validate:
#| Property | Value  | Property Value | Runtime Value | Result |
 | firstPlan | %ARRAY_VALUE(subscriptions[?(@.status=='ACTIVE')].plan)
Premium | Premium | PASSED |
 | activePlanExpiry | %ARRAY_VALUE(subscriptions[?
(@.status=='ACTIVE')].expiry,0)| 2025-03-15 | 2025-03-15 | PASSED |
 | subscriptionCount | %ARRAY_SIZE(subscriptions) | 2 | 2 | PASSED |
```

Array Size

This function returns the number of elements in an array.

%ARRAY_SIZE(<JSON Path>) : Returns the size of the array returned by the JSON path

For example,

```
Validate:
#|  Property          |  Value                      |  Property Value  |
```

```
Runtime Value | Result |
 |  subscriptionCount |  %ARRAY_SIZE(subscriptions) |  2               |
2              |  PASSED  |
```

Returns: 2 (since there are two subscription entries)

Response Header

This function returns the value for the given header key, if it is present in response headers.

For example,

```
{
...
"headers" : {
              "transfer-encoding" : "chunked",
              "connection" : "keep-alive",
              "Date" : "Wed, 25 Aug 2021 04:51:40 -0700",
              "Content-Type" : "application/json"
            }
...
}
```

Get the Date header from response.

%RESPONSE_HEADER(Date) returns "Wed, 25 Aug 2021 04:51:40 -0700"

The following is a BDD example for using %RESPONSE_HEADER() in Save block:

```
Then get mock response, processing Customer subscription details
Save:
|  Date | %RESPONSE_HEADER(Date)     |
| Connection | %RESPONSE_HEADER(connection) |
```

The following is the runtime BDD response for using %RESPONSE_HEADER() in Save block:

```
Then get mock response, processing Customer subscription details
 Save:
#|  Property   |  Value                       |  Runtime
Value                                 |
|  Date        | %RESPONSE_HEADER(Date)       |  Wed, 25 Aug 2021 04:51:36
-0700             |
| Connection   | %RESPONSE_HEADER(connection) |  keep-
alive                                |
```

The following is a BDD example for using %RESPONSE_HEADER() in Validate block:

```
Then get mock response, processing Customer subscription details
Validate:
| %RESPONSE_HEADER(connection)  | ${connection} |
```

The following is the runtime BDD response for using %RESPONSE_HEADER() in Validate block:

```
Then get mock response, processing Customer subscription details
Validate:
#|  Property                    |  Value            |  Property Value     |
Runtime Value          |  Result           |
| %RESPONSE_HEADER(connection) | ${connection}     |  keep-alive
|      keep-alive          |      PASSED       |
```

Data Type Functions

Data Type functions are used in Data block to represent the type of property value. By default, all data is treated as a string. To convert data to other types, use the appropriate data type functions.

Table 5-1 describes Data Type functions used in Data block to represent the type of property value.

Table 5-1 Data Type Functions

Function	Description	Example: Data
%INT(<int value>)	To represent integer values	%INT(200) -→ "value": 200
%DOUBLE(<double value>)	To represent floating/double values	%DOUBLE(35.75) -→ "billAmount": 35.75
%BOOLEAN(<boolean value>)	To represent boolean values	%BOOLEAN(true) →> "created" : true

Date Type Functions

These functions retrieve, modify, and transform dates in various formats and are useful for timestamping, scheduling, and handling date-based calculations.

Retrieve Current Date (%NOW())

Returns current date in **YYYY-MM-ddTHH:mm:ss.SSSZ** format. For example, `%NOW() → "2021-08-25T14:16:28.312Z"`

The following is a BDD example for retrieving current date:

When add todo task, for booking appointment

Data: Table 5-2 lists out the values in NOW format.

Table 5-2 NOW format

Property	Value	Runtime Value
description	%NOW()	2021-08-25T14:16:28.312Z

Retrieve Current Date in a Custom Format (%NOW(<format>))

Returns current date in specified format. For example, `%NOW(YYYY-MM-dd) → "2021-08-25"`

The following is a BDD example for retrieving current date in a custom format:

When add todo task, for booking appointment

Data: Table 5-3 lists out the values in NOW format.

Table 5-3 NOW format

Property	Value	Runtime Value
description	%NOW(YYYY-MM-dd)	2021-08-25

For more information on formatting the date, see Class SimpleDateFormat in *Oracle Java documentation*.

Add or Subtract Time (%NOWADD(<field>, <+/- value>))

Modifies the current date or time by adding or subtracting a specific amount from a time field.

Default format (YYYY-MM-dd'T'HH:mm:ss.SSS'Z')

For example, `| dateTime | %NOWADD(5,10) | # Adds 10 units to field 5 | dateTime | %NOWADD(5,-10) | # Subtracts 10 units from field 5`

Custom Format (%NOWADD(<field>, <+/- value>, <output format>))

For example, `| dateTime | %NOWADD(5,10,yyyy-MM-dd HH:mm:ss) |`

Output:

```
"2024-05-07 10:10:10"
```

Modify a Saved Date (%NOWADD(<field>, <+/- value><output format>))

Adds or Subtracts from a date field and returns date in specified format.

Add or Subtract from current time using Custom Format

| dateTime | %NOWADD(5,10,yyyy-MM-dd HH:mm:ss) |

%DATEADD(<field>, <+/- value>)

Add/Subtract from a date field and returns date in default format YYYY-MM-dd'T'HH:mm:ss.SSS'Z'.

For example, `| dateTime | %DATEADD(${datavar},5,5,yyyy-MM-dd HH:mm:ss) |`

Advanced example, (%DATEADD(<field>, <+/- value>, <input format>, <output format>):

```
| dateTime | %DATEADD(2024-05-07 10:10:10,5,-5,yyyy-MM-dd HH:mm:ss,dd-MM-YYYY) |
```

Transforms: `"2024-05-07 10:10:10" → "07-05-2024"`

Transform Date Formats (%TRANSFORM(<date1><inputFormat><outputFormat>))

Transforms given date in the input format to specified output format.

The following BDD example uses Transform function to trasnform date in the Save section to specified format:

```
When execute mock action, reading the task
Data:
| $request | $arraydata2 |
```

```
Save:
| dateTime | %TRANSFORM(2024-05-07 10:10:10,YYYY-MM-dd HH:mm:ss,dd-MM-YYY) |
```

Format Number Functions

The Format Number function formats a numeric value according to a specified pattern, applying different rounding modes as needed such as FLOOR, CEILING, and ROUND. It supports various separators, custom decimal places, and string interpolation within the formatted output.

Table 5-4 describes variants of Format Number Functions.

Table 5-4 Variants and Descriptions

Variant	Description
CEILING	Rounding mode to round towards positive infinity.
DOWN	Rounding mode to round towards zero.
FLOOR	Rounding mode to round towards negative infinity.
HALF_DOWN	Rounding mode to round towards nearest neighbor unless both neighbors are equidistant, in which case you round down instead.
HALF_EVEN	Rounding mode to round towards the nearest neighbor unless both neighbors are equidistant, in which case, round towards the even neighbor.
HALF_UP	Rounding mode to round towards nearest neighbor unless both neighbors are equidistant, in which case you round up instead.
UNNECESSARY	Rounding mode to assert that the requested operation has an exact result, hence no rounding is necessary.
UP	Rounding mode to round away from zero.

The following example shows the BDD code to format a number:

```
Case: Format Number
When set variable, customer bill value is taken as input
Save:
| price |  1234567.89 |
When set variable, to get formatted customer bill details
Save:
| formattedBill | %FORMAT_NUMBER(481.195) |
| decimalBill | %FORMAT_NUMBER(${price},0.0) |
| roundedBill | %FORMAT_NUMBER(${price},0) |
| roundedBill2 | %FORMAT_NUMBER(${price},#.##,CEILING) |
| roundedBill3 | %FORMAT_NUMBER(${price},#%{,}###.##,CEILING) |
| roundedBill4 | %FORMAT_NUMBER(${price},Amount to be payable is $#%{,}###.#
for this month,CEILING) |
| discountedBill | %FORMAT_NUMBER(${price},#,FLOOR) |
| discountedBill1 | %FORMAT_NUMBER(${price},#,HALF_EVEN) |
| discountedBill2 | %FORMAT_NUMBER(${price},#,HALF_UP) |
| discountedBill3 | %FORMAT_NUMBER(${price},#,HALF_DOWN) |
Output (Runtime BDD):
When set variable, to get formatted customer bill details
```

```
Save:
#|  Property         |
Value
       |  Runtime Value                                          |
 |  price            |
1234567.89
       |  1234567.89                                             |
 |  test             |
12.053548387096775
       |  12.053548387096775                                     |
 |  formattedBill    |
%FORMAT_NUMBER(481.195)
       |  481.20                                                 |
 |  decimalBill      |  %FORMAT_NUMBER($
{price},0.0)                                                              |
1234567.9                                             |
 |  roundedBill      |  %FORMAT_NUMBER($
{price},0)                                                                |
1234568                                               |
 |  roundedBill2     |  %FORMAT_NUMBER($
{price},#.##,CEILING)                                                     |
1234567.89                                            |
 |  roundedBill3     |  %FORMAT_NUMBER($
{price},#%{,}###.##,CEILING)                                              |
1,234,567.89                                          |
 |  roundedBill4     |  %FORMAT_NUMBER(${price},Amount to be payable
is $#%{,}###.# for this month,CEILING)  |  Amount to be payable
is $1,234,567.9 for this month  |
 |  discountedBill   |  %FORMAT_NUMBER($
{price},#,FLOOR)                                                          |
1234567                                               |
 |  discountedBill1  |  %FORMAT_NUMBER($
{price},#,HALF_EVEN)                                                      |
1234568                                               |
 |  discountedBill2  |  %FORMAT_NUMBER($
{price},#,HALF_UP)                                                        |
1234568                                               |
 |  discountedBill3  |  %FORMAT_NUMBER($
{price},#,HALF_DOWN)                                                      |
1234568                                               |
```

Format Patterns

DecimalFormat is a concrete subclass of NumberFormat that formats decimal numbers. It has a variety of features designed to parse and format numbers in any locale, including support for Western, Arabic, and Indic digits. It also supports different kinds of numbers, including integers (123), fixed-point numbers (123.4), scientific notation (1.23E4), percentages (12%), and currency amounts ($123). All of these can be localized.

To obtain a NumberFormat for a specific locale, including the default locale, use one of NumberFormat's factory methods, such as **getInstance()**. In general, avoid using the DecimalFormat constructors directly, since the NumberFormat factory methods may return subclasses other than DecimalFormat. A DecimalFormat comprises a pattern and a set of symbols. The pattern may be set directly using **applyPattern()**, or indirectly using the API methods. The symbols are stored in a DecimalFormatSymbols object. When using the

NumberFormat factory methods, the pattern and symbols are read from localized ResourceBundles. To customize format object, perform the following action:

A DecimalFormat comprises a pattern and a set of symbols. The pattern may be set directly using applyPattern(), or indirectly using the API methods. The symbols are stored in a DecimalFormatSymbols object. When using the NumberFormat factory methods, the pattern and symbols are read from localized ResourceBundles.

```
Patterns
DecimalFormat patterns have the following syntax:
 Pattern:
         PositivePattern
         PositivePattern ; NegativePattern
 PositivePattern:
         Prefixopt Number Suffixopt
 NegativePattern:
         Prefixopt Number Suffixopt
 Prefix:
         any Unicode characters except \uFFFE, \uFFFF, and special characters
 Suffix:
         any Unicode characters except \uFFFE, \uFFFF, and special characters
 Number:
         Integer Exponentopt
         Integer . Fraction Exponentopt
 Integer:
         MinimumInteger
         #
         # Integer
         # , Integer
 MinimumInteger:
         0
         0 MinimumInteger
         0 , MinimumInteger
 Fraction:
         MinimumFractionopt OptionalFractionopt
 MinimumFraction:
         0 MinimumFractionopt
 OptionalFraction:
         # OptionalFractionopt
 Exponent:
         E MinimumExponent
 MinimumExponent:
         0 MinimumExponentopt
```

Understanding DecimalFormat Patterns

DecimalFormat patterns help format numerical values for proper display. They define prefixes, numeric values, and suffixes while handling positive and negative subpatterns, separators, and formatting symbols.

The following are the key features of DecimalFormat Patterns:

- Contains positive and negative subpatterns (for example, `"#,##0.00;(#,##0.00)"`).
- If no negative subpattern is provided, the positive pattern is prefixed with a localized minus sign (`-` in most locales).
- Customizable prefixes and suffixes can be used for different formatting styles.

Here is the behavior of positive and negative subpatterns.

- `"0.00"` is equivalent to `"0.00;-0.00"` since the minus sign is automatically applied.
- If a negative subpattern is explicitly defined, only the prefix and suffix change while the numerical rules remain the same.
  For example, `"#,##0.0#;(#)"` behaves exactly the same as `"#,##0.0#;(#,##0.0#)"`.

Formatting Symbols and Separators

Symbols for infinity (`∞`), digits (`0-9`), thousand separators (`,`), and decimal points (`.`) are fully customizable. Care must be taken to avoid conflicts to ensure:

- Positive and negative prefixes or suffixes are distinct for accurate parsing.
- Decimal separator and thousand separator are unique to prevent errors.

Grouping Separators and their Behavior

Typically used for thousands, though some locales use them for ten-thousands. The grouping size determines the digit intervals.

For example, `3` for `"100,000,000"` or `4` for `"1,0000,0000"`.

If multiple grouping characters are provided, the last grouping separator before the integer end is used. For example, `"#,##,###,####"` == `"######,####"` == `"##,####,####"`.

6

Using Control Structures in Steps

Learn to use different control structures in steps in Oracle Communications Solution Test Automation Platform (STAP).

Topics in this chapter:

Overview

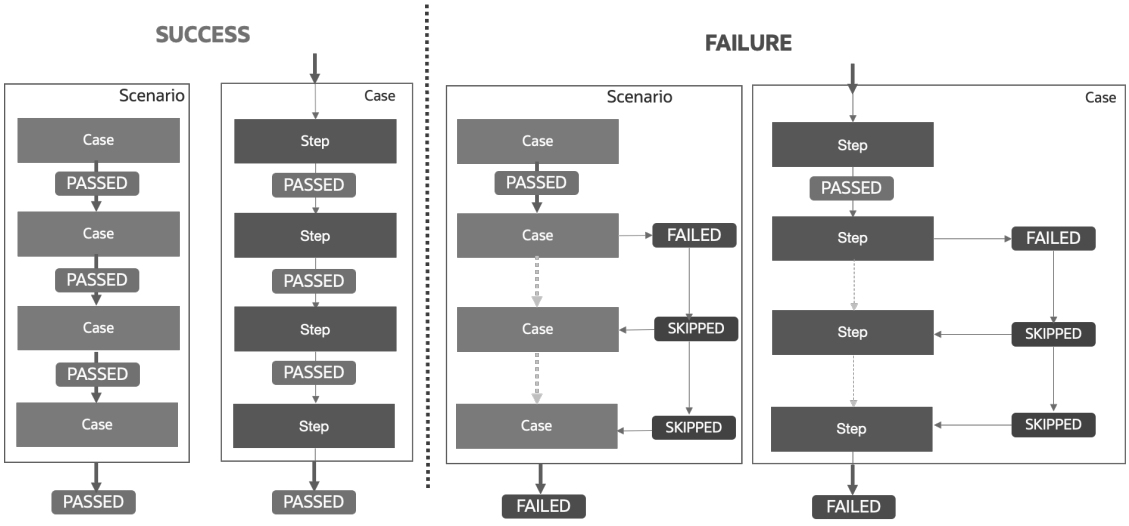
You can use control structures like **if**, **for**, and **while** for steps in the Behavior-Driven Development (BDD) language. They are the building blocks within each test case and determine the flow of execution for each step based on specific conditions. Steps dictate the flow within test cases, while scenario execution flow governs the execution of the entire test scenario.

Scenario Execution Flow

The Scenario Execution Flow relies on the outcomes of the steps in the scenario. If a step within a test case fails, it impacts the flow by skipping remaining steps and potentially other test cases within the scenario.

Figure 6-1 shows the detailed flow of a scenario execution.

Figure 6-1 Scenario Execution Flow



If the scenario execution is successful:

- **Test Scenario Execution:** If all the test cases within a scenario are run successfully, the entire scenario is considered passed.
- **Test Case Execution:** When all test steps within a test case are run without any errors, the test case is considered passed.

If the scenario execution fails:

- **Test Scenario Execution:** If a test case within a scenario fails, all subsequent test cases in that scenario are skipped, and the entire scenario is marked as failed.
- **Test Case Execution:** If any test step within a test case fails, the remaining test steps in that test case are skipped, and the test case is marked as failed.

This detailed flow ensures that the execution process is efficient and that any failures are quickly identified and addressed, preventing unnecessary execution of subsequent steps or cases.

Action Execution

There are two types of Action Executions:

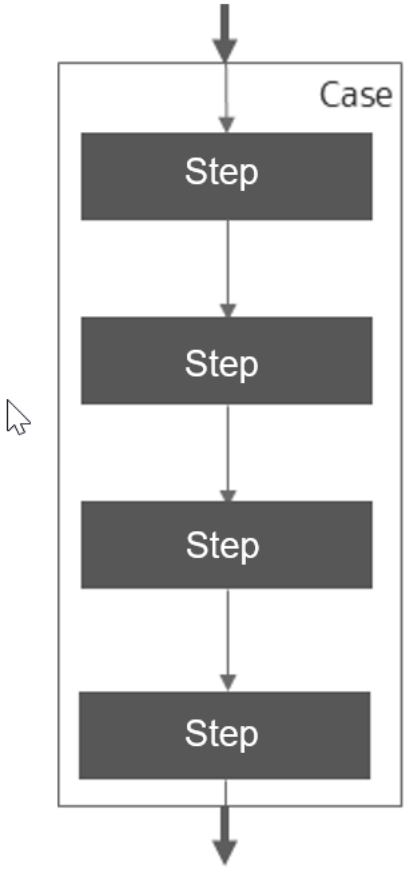
- Static Action (Default)
- Controlled Step

Static Action (Default)

Performs the action once (in sequence).

Figure 6-2 shows the detailed flow of a static action.

Figure 6-2 Static Step



Controlled Step: Dynamic Action

Controlled execution of Step

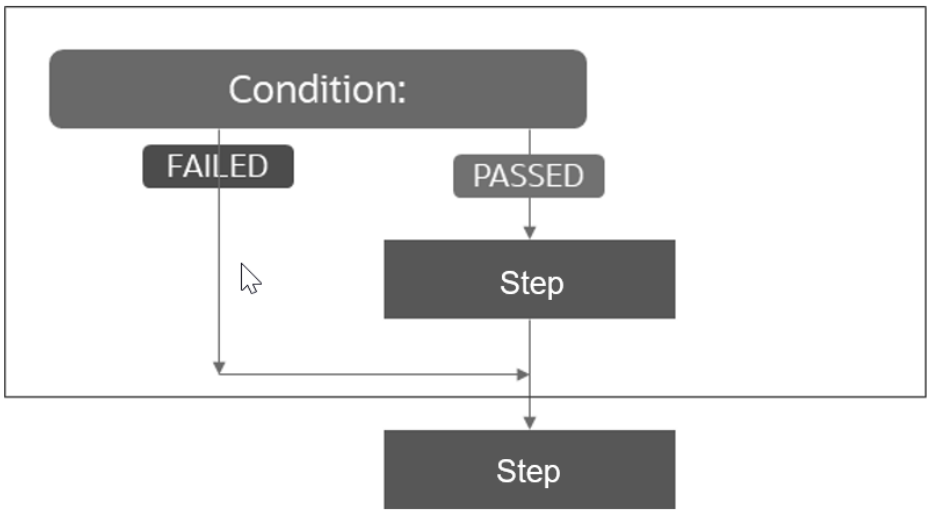
- Condition: Conditional Execution (IF)
  - Perform action when the condition is PASSED.
- repeatTimes (FOR)
  - Repeat number of times.
- repeatUntil (UNTIL)
  - Repeat until the condition is PASSED.
- repeatWhile (WHILE)
  - Repeat while the condition is true.

Conditional Execution

- Perform Action when the condition is successful.
- Support multiple conditions using 'Condition'.

Figure 6-3 shows the detailed flow of a conditional execution.

Figure 6-3 Conditional Execution



For example,

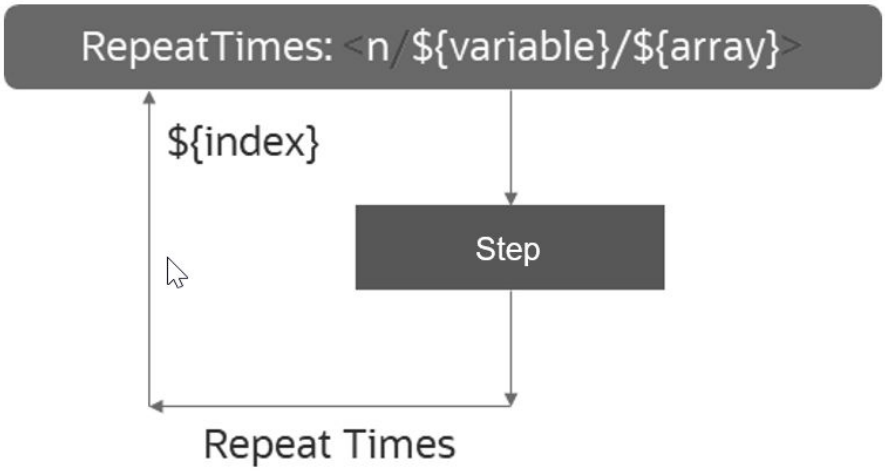
```
# This block of code checks if the category is Platinum, if yes, then it
changes to Gold through the action file request
When change category, for changing customer category
Condition:
| ${category} | Platinum |
Validate:
| $status | 201 |
| category | Gold |
Save:
| name | name |
| category | category |
RepeatTimes
```

Repeat Times

Repeatedly perform the action given number of times.

Figure 6-4 shows the flow of repeat times.

Figure 6-4 Repeat Times



The success of the step depends on each action. If any iteration fails – the step fails even then continues.

The following are the various ways through which you can specify the repeat number of times:

- **n : integer**: number of times
- **${variable}**: integer variable of times
- **${array}**: array of integers. Action is repeated for array length number of times
- **${index}**: index value of iteration. Values: 1-n
- **${nextValue}** gives next array value.
- **$breakOnFailure**: YES breaks the loop, Default: NO

Example

```
Case: RepeatTimesAction
When set variable, create bills list
Save:
| $ARRAY{bills} | 25.213 |
| $ARRAY{bills} | 30.456 |
Then get mock response, repeatedly to send payment reminders of bills
# executes this block for variable times - size of array
RepeatTimes:
| $times | $ARRAY{bills} |
Data:
| id | getdata |
| index | ${nextValue} |
Validate:
| $status | 200 |
| bills[${index}] | $ARRAY{bills[${index}]} |
```

```
Then get mock response, repeatedly to send payment reminders of bills
#executes this block of code for predefined number of times
RepeatTimes:
| $times | 2 |
Data:
| id | getdata |
| index | ${nextValue} |
Validate:
| $status | 200 |
| bills[${index}] | $ARRAY{bills[${index}]} |
RepeatUntil
```

Repeat Until

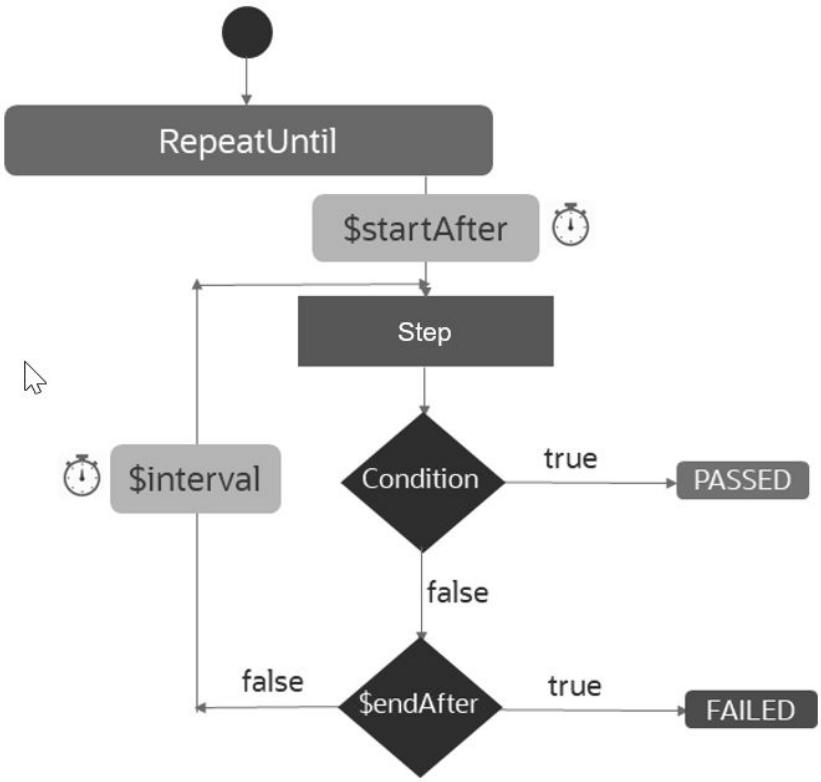
- Repeatedly perform the action until the given condition is true.
- At least one Condition is mandatory. (?)
- $breakOnFailure : YES breaks the loop on action validation failure. Default: NO.

```
When set variable, create bills list
Save:
| $ARRAY{bills} | 25.213 |
| $ARRAY{bills} | 30.456 |
| $ARRAY{bills} | 28.712 |
| $ARRAY{bills} | 26.389 |
| $ARRAY{bills} | 31.243 |
# executes this block of code until all the conditions are true
Then get mock response, sending notifications until customer bill equals a
value
RepeatUntil:
| $ARRAY{bills[${index}]} | 30.456 |
Data:
| id | getdata |
| index | ${nextValue} |
Validate:
| $status | 200 |
Repeat Until with time durations and frequency interval
```

Repeat Until with Time Durations and Frequency Interval

Figure 6-5 shows the flow of Repeat Until with Time Durations and Frequency Interval.

Figure 6-5 Repeat Until with Time Durations and Frequency Interval




```
$startAfter : Optional. Start executing action after this duration of time.
By default, starts immediately.The duration is in seconds.
$endAfter : Mandatory. Break after the completion of this time duration.
$interval: Optional. interval duration to run the action. By default,
executes continuously.
Specify duration in Seconds.
Breaks if the condition is true even before $endAfter.
$breakOnFailure : YES will breaks loop on action validation failure, Default:
NO.
```

Example scenario:
example

```
Case: RepeatUntilAction
When set variable, create bills list
Save:
| $ARRAY{bills} | 25.213 |
| $ARRAY{bills} | 30.456 |
| $ARRAY{bills} | 28.712 |
| $ARRAY{bills} | 26.389 |
| $ARRAY{bills} | 31.243 |
# executes this block of code until all the conditions are true
```

```
Then get mock response, sending notifications until customer bill equals a
value
RepeatUntil:
| $ARRAY{bills[${index}]} | 30.456 |
# start execution after 1 second
| $startAfter | 1 |
# 1 second interval for every execution
| $interval  | 1 |
# stop execution after 5 seconds
| $endAfter | 5 |
Data:
| id | getdata |
| index | ${nextValue} |
Validate:
| $status | 200 |
RepeatWhile


When set variable, setting customer bill Amount
Save:
| billAmount | 30 |
# All the conditions must hold true -> While executes the condition first
Then get mock response, sending notifications of pending bills while amount
is under a threshold
RepeatWhile:
# The variable used here must be defined already
| ${billAmount} | %GREATER_THAN(25) |
Data:
| id | getcust |
| index | ${nextValue} |
Validate:
| $status | 200 |
| extractedNotice | 'Your subscription is expiring soon' |
RepeatWhile with time durations and interval
```

Repeat While

- Repeatedly perform the action while the given condition is true.
- $breakOnFailure: YES will break loop on action validation failure, Default: NO.

Repeat While

- Repeatedly perform the action while the given condition is true.
- $breakOnFailure: YES will break loop on action validation failure, Default: NO.

Figure 6-6 shows the flow of 1st iteration.

Figure 6-6 Repeat While 1st Iteration

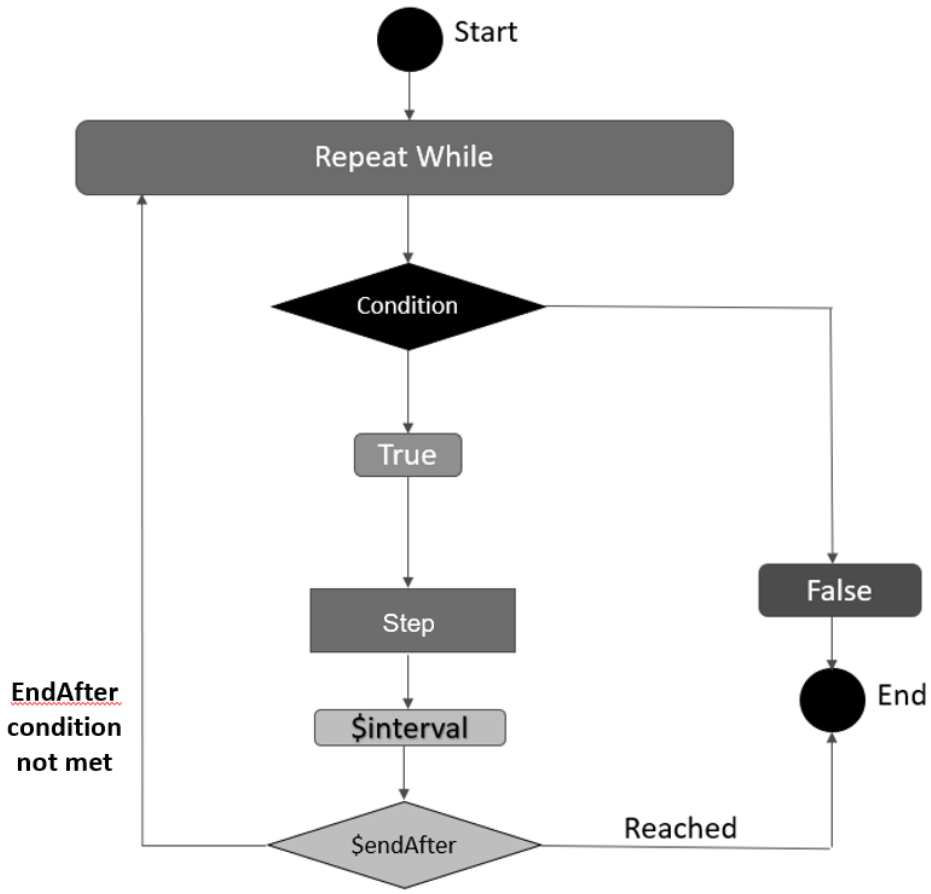
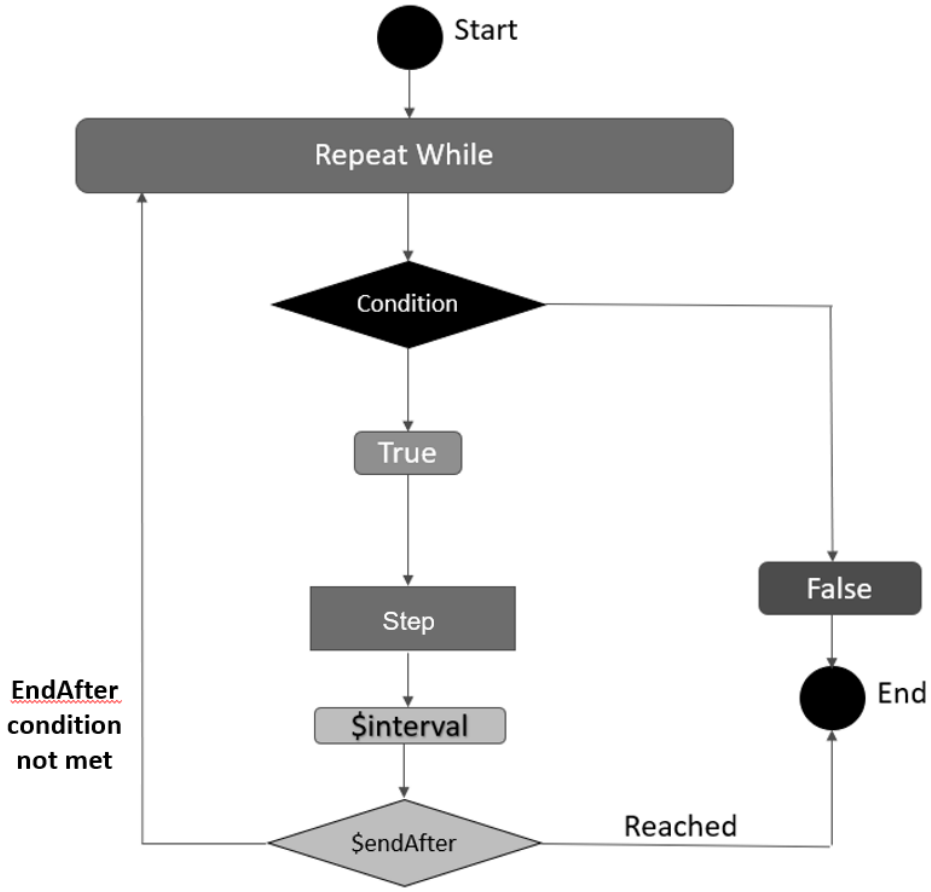


Figure 6-7 shows the flow of other iterations.

Figure 6-7 Repeat While Other Iterations




```
Case: RepeatWhileAction
When set variable, setting customer bill Amount
Save:
| billAmount | 30 |
# All the conditions must hold true -> While executes the condition first
Then get mock response, sending notifications of pending bills while amount
is under a threshold
RepeatWhile:
# The variable used here must be defined already
| ${billAmount} | %GREATER_THAN(25) |
# starts execution after 1 second
| $startAfter | 1 |
# 1 second interval for every execution
| $interval  | 1 |
# stop execution after 5 seconds
| $endAfter | 5 |
Data:
| id | getcust |
| index | ${nextValue} |
Validate:
```

```
| $status | 200 |
| extractedNotice | 'Your subscription is expiring soon' |
```

Repeat While: Examples of Time Durations and Interval

```
$startAfter : Optional. Start executing action after this duration of time.
By default, starts immediately.
$endAfter : Mandatory. Break after the completion of this time duration.
$interval: Optional. interval duration to run the action. By default,
executes continuously.
Specify duration in Seconds.
Breaks if the condition is true even before $endAfter.
$breakOnFailure : YES breaks a loop on action validation failure. Default: NO.


Case: RepeatWhileAction
When set variable, setting customer bill Amount
Save:
| billAmount | 30 |
# All the conditions must hold true -> While executes the condition first
Then get mock response, sending notifications of pending bills while amount
is under a threshold
RepeatWhile:
# The variable used here must be defined already
| ${billAmount} | %GREATER_THAN(25) |
# starts execution after 1 second
| $startAfter | 1 |
# 1 second interval for every execution
| $interval  | 1 |
# stop execution after 5 seconds
| $endAfter | 5 |
Data:
| id | getcust |
| index | ${nextValue} |
Validate:
| $status | 200 |
| extractedNotice | 'Your subscription is expiring soon' |
```

Using multiple test data files in control actions

```
Action using multiple data
When create product offering, with multiple data sets
repeatTimes: 2
Data:
| $request | $FILE(productOffering_${index}.json) |
(data/product Offering_0.json
data/productOffering_1.json)
| variable | I Value${index}-${UlD} |
Validate:
| statusCode | 201 |
Save :
| $ARRAY{productOfferingId} | id |
(data/productOffering_O.json
data/productOffering_1.json
productQtferingJd Array)
```

```
Action using multiple data sets
When Dummy, save some values
Save:
| $ARRAY{poNames} | VoicePO |
| $ARRAY{poNames} | SMSPO |
| $ARRAY{poNames} I VolPPO |
When create product offering, with multiple data sets
Repeat Times:
| $times | $ARRAY{poNames} I
Data:
| $request |$FILE(productOffering_${nextValue}.json |
(data/productOffering_VoicePO.json
data/productPffering_SMSPO.json
data/productOffering_VolPPO.json)
| variable | Values${nectValue}-${index}-${UID} |
Validate:
| statusCode | 201 |
Save :
| $Array{productOfferinfId} | id |
```

Using Conditional Cases

Cases to run are mentioned in the **scenario.config** file. With conditional case execution, you can specify a set of conditions, and only the cases which satisfy all specified conditions are run.

> **Note:**
>
> - You mention the conditions after the case name within curly brackets, separated by a comma. For example, **sampleCase {condition1, condition2}**.
> - If the condition value or condition variable is from a saved variable in any of the previous cases run, they are to be specified within **${ }**.
> - Only **=** or **Equals to** operation is supported for condition evaluation.

The following are the configurations to set to run conditional cases.

The syntax for **scenario.config** configuration file:

```
Header.info
Data.case
MockAction.case
MockAction.case{${executeMockAction}=${value}}
#MockAction.case{${executeMockAction}=true}
MockAction.case{${executeMockAction}=true,${day}=wednesday}
```

The following is the syntax for **data.case** file:

```
Case: Data creation for conditional execution

When dummy
Save:
| value | true |
| executeMockAction | true |
| day | wednesday |
```

The following is the syntax for **MockAction.case** file:

```
Case: Mock action test

When execute mock action, creating a task
Data:
| id | WeekdayTask-${UID} |
| name | WeekdayTask-${UID} |
Save:
| taskId | id |
| taskName | name |

When execute mock action, reading the task
Data:
| $requestString | {"id":"id"} |
| id | ${taskId} |
```

7

STAP Action Plugins

Learn about different Oracle Communications Solution Test Automation Platform (STAP) Action Plugins and their functions.

Topics in this chapter:

Introduction to STAP Action Plugins

STAP Action plugins enable automation to interact seamlessly with various product interfaces, such as REST and SOAP. These plugins enable developers and testers to automate tasks, ensure consistency, and improve efficiency in managing interactions with diverse systems. Automation plugins significantly enhance productivity by eliminating manual interventions.

Adding tools like Seagull process execution plugins further broadens the scope of automation, making it easier to manage diverse and complex workflows. Selecting the right plugin depends on factors such as the complexity of the task, integration requirements, and the technology stack in use.

The available automation plugins are:

- REST API
- SOAP API
- SSH/SFTP
- Process
- Kafka
- Seagull (Multi Protocol Traffic Generator)
- URL Validator

REST Plugin

Representational State Transfer (REST) is a widely used interface for web services due to its simplicity and scalability. The REST plugin facilitates tasks such as making requests, handling JSON requests/payloads, and validating status and response data.

The key features of the REST plugin are:

- **Payload Management:** Simplifies sending and receiving JSON or XML data.
- **Request Handling:** Includes constructing the payload along with the REST methods such as GET, POST, PUT, DELETE, and other HTTP methods.
- **Authentication Support:** Handles OAuth, API keys, and Basic Authentication.
- **Response Validation:** Supports for assertions on HTTP status codes, headers, and body content.

The Rest plugin is used to automate the execution of REST API endpoints and to validate the response.

REST Connection

To use the REST interface, you must first set up the connection environment. An environment is a setup where applications or integrated solutions operate. A connection serves as an interface to the application running in the environment, allowing communication with the application.

Environment configuration includes the settings for these connections. Each STAP plugin has its own environment connection configuration, and some plugins can have multiple environment configuration files for different products tested using various scenarios. For more information, see "Environments Folder"

You can combine REST and SOAP in a single environment, but other types of interfaces each need to have their own environment:

- **Multiple**: This includes REST, SOAP
- **Single**: This includes SSH, KAFKA, URL_VALIDATION, SEAGULL

REST supports two types of authentications:

- Basic
- OAuth

Basic Authentication
Basic Authentication is a straightforward authentication method where the client provides credentials (username and password).

Following is a sample of an **environment.properties** file for basic authentication.

```
# Environment name
name=todo
type=REST

hostname=hostname
url=url

#================================================================================
=
# Authorization Configuration
# Values = YES : Use authorization NO : No authorization required.
#================================================================================
=
authorization=YES
# oauth2/basic
authorization.type=basic

#- BASIC Authorization
```

```
basic.username=
basic.password=
```

OAuth2 Authentication

OAuth2 supports **client_credentials** and **password_credentials** grant types.

Following is a sample of an **environment.properties** file for a **client_credentials** grant using OAuth authentication.

```
#--------------------------------------------------------
# Environment name.
#--------------------------------------------------------
name=care
#--------------------------------------------------------
# Type of the connection.
#--------------------------------------------------------
type=REST

#--------------------------------------------------------
# REST Configuration
#--------------------------------------------------------
#Hostname
hostname=hostname
#--------------------------------------------------------
#Base URL
#--------------------------------------------------------
url=url
#--------------------------------------------------------
#=============================================================================
=
# Authorization Configuration
#=============================================================================
=
authorization=YES
#--------------------------------------------------------
# Authorization Type
# One of oauth2/basic
#--------------------------------------------------------
authorization.type=oauth2
#--------------------------------------------------------
# OAUTH2 - IDCS Configuration
#--------------------------------------------------------
oauth2.grantType=client_credentials
oauth2.clientId=****************************
oauth2.clientSecret=****************************
oauth2.tokenUrl=****************************
oauth2.scope=***************
```

Following is a sample of an environment.properties file for a password_credentials grant using OAuth authentication.

```
#--------------------------------------------------------
# Environment name.
#--------------------------------------------------------
```

```
name=care
#------------------------------------------------------
# Type of the connection.
# One of api.rest, api.soap or ssh
#------------------------------------------------------
type=REST

#------------------------------------------------------
# REST Configuration
#------------------------------------------------------
#Hostname
hostname=hostname
#------------------------------------------------------
#Base URL
#------------------------------------------------------
url=url
#------------------------------------------------------

#=============================================================================
=
# Authorization Configuration
#=============================================================================
=
authorization=YES
#------------------------------------------------------
# Authorization Type
# One of oauth2/basic
#------------------------------------------------------
authorization.type=oauth2
#------------------------------------------------------
# OAUTH2 - IDCS Configuration
#------------------------------------------------------
oauth2.grantType=password_credentials
oauth2.clientId=****************************
oauth2.clientSecret=*****************************
oauth2.tokenUrl=****************************
oauth2.scope=***************
oauth2.authorization=YES
oauth2.authorization.username=************
oauth2.authorization.password=************
```

Gateway types

The REST plugin supports two gateway types for constructing URLs dynamically:

default : Resource mentioned in the action file is added to the base URL to construct the final URL.

fabric : When the base url remains same but during execution, different resource endpoints need to be tested, connection URLs can be used.

Configuration key: connection.uri.resourceName

URL is constructed by joining the base url, value of the connection uri for the resource mentioned in action file, and the resource in the action file.

For example,

```
#------------------------------------------------------
# Environment name. Ref. Supported list above.
#------------------------------------------------------
name=care
#------------------------------------------------------
# Type of the connection.
# One of api.rest, api.soap or ssh
#------------------------------------------------------
type=REST
#------------------------------------------------------
# REST Configuration
#------------------------------------------------------
#Hostname
#------------------------------------------------------
#Hostname
hostname=hostname
#------------------------------------------------------
#Base URL
#------------------------------------------------------
url=url
#------------------------------------------------------
# Connection Type : Direct or through Fabric
# connectionType=fabric/default
#------------------------------------------------------
connection.type=fabric
connection.uri.customerBill=customerBillManagement/v4
connection.uri.customerBillOnDemand=customerBillManagement/v4
connection.uri.payment=payment/v4
connection.uri.paymentAllocation=payment/v4
connection.uri.adjustBalance=prepayBalanceManagement/v4
connection.uri.usage=usageManagement/v2
connection.uri.appliedCustomerBillingRate=customerBillManagement/v4
connection.uri.disputeBalance=prepayBalanceManagement/v4
#================================================================================
=
# Authorization Configuration
# Values = YES : Use authorization NO : No authorization required.
#================================================================================
=
authorization=NO
#------------------------------------------------------
```

Action Files in REST Plugin

Action files define how API requests are constructed and executed within the REST plugin.

For example, in the following JSON file:

```
{
"path":"care/customerBill/read-customerBill/read-customerBill-by-id",
"name":"Read customer bill by id",
"bdd":"read customer bill by id",
"description":"Read customer bill by id",
"product":"care",
```

```
 "actionType":"REST",
"tags":["customer","bill"],
"resource":"customerBill",
"method":"GET",
"expectedStatusCode":200
}
```

The final URL for the example is constructed by combining the following elements:

resource : customerBill

Value of connection uri for the resource in action file: customerBillManagement/v4

The supported action types are:

- GET
- POST
- PUT
- PATCH
- DELETE

Method: GET

read-todo-task.action.json

```
{
"path":"/category/getcategory",
"name":"Read all categories",
"bdd":"read all categories",
"description":"Reading all categories of customer",
"product":"mockserver",
 "actionType":"REST",
"tags":["category","read","all"],
"resource":"getcategory",
"method":"GET",
 "expectedStatusCode":200
}
```

Method: POST

mockpost.action.json

```
{
    "path":"/category/postdetails/",
    "name":"add category",
    "bdd":"add category",
    "description":"Adding category",
    "product":"mockserver",
     "actionType":"REST",
    "tags":["add","category"],
    "resource":"mock/postcust",
    "method":"POST",
    "requestType":"FILE",
    "request":"mockpost.request.json",
```

```
 "expectedStatusCode":201
}
```

Request Json :

add-todo-task.request.json

```
{
    "name":"John Doe",
    "category":"Platinum",
}
```

Method: PUT

mockput.action.json

```
{
    "path":"/category/changedetails/",
    "name":"change details",
    "bdd":"change details",
    "description":"Changing customer details",
    "product":"mockserver",
     "actionType":"REST",
    "tags":["change","details"],
    "resource":"mock/patchcust",
    "method":"PATCH",
    "requestType":"FILE",
    "request":"mockpatch.request.json",
     "expectedStatusCode":200
    }
```

Request Json :

put-todo-task.request.json

```
{
    "name" : "John Doe",
    "category" : "Gold"
}
```

Method: PATCH

mockpatch.action.json

```
{
    "path":"/category/changedetails/",
    "name":"change details",
    "bdd":"change details",
    "description":"Changing customer details",
    "product":"mockserver",
     "actionType":"REST",
    "tags":["change","details"],
    "resource":"mock/patchcust",
    "method":"PATCH",
    "requestType":"FILE",
```

```
    "request":"mockpatch.request.json",
     "expectedStatusCode":200
    }
```

Request Json :

mockpatch.request.json

```
{
    "name":"Sam Curran",
    "category":"Platinum"
    }
```

Method: DELETE

mockdelete.action.json

```
{
"path":"category/deletecategory",
"name":"Delete category",
"bdd":"delete category",
"description":"Delete category of customer",
"product":"mockserver",
 "actionType":"REST",
"tags":["category","delete"],
"resource":"deletecategory",
"method":"DELETE",
 "expectedStatusCode":202
}
```

Dynamic Request JSON

Creating a dynamic request JSON file enhances flexibility in API automation by allowing dynamic data injection at runtime instead of relying on predefined request structures.

To use a dynamic request JSON file instead of the request JSON file mentioned in the action file:

1. Create a folder named 'data' under the folder for scenario.
2. Create a dynamic request JSON file with the name in the following format: actualName.dynamicName.**request.json**, where actualName is the actual name of the request file up to the first period, and dynamicName is a one-word name for the dynamic request. , followed by the one word name for dynamic request and ending with .request.json.
3. In the test step's data section, use **$request** for the variable name to access the information, and use dynamicName as the value.

Refer to the following example to see how to use a dynamic request json, replacing predefined request files for greater flexibility.
If the ordinary request file is named **update-one-todo-task.UpdateStatus.request.json**, and you name the dynamic file **update-one-todo-task.UpdateTodo.request.json**, you access the data this way:

```
Data:
| $request  | $UpdateTodo |
```

```
| id | ${id} |
| description | Arrange meeting for service updates |
Validate:
| $status | 202 |
```

Query parameters
Query parameters in REST are key-value pairs added to the URL after a **?** (question mark). They are used to filter, sort, or modify a request without changing the resource path.

Query parameters to the endpoint can be configured in the test step using **$query** for GET and POST methods.

The following BDD example provides query parameter **account.id** value in the url to read the payment details:

```
# Provide direct value in the query parameter value
Then read payment, Retrieve the Payment details
Data:
| $query | account.id=abcde |

#Using saved context variable in query parameter value
Then read payment, Retrieve the Payment details
Data:
| $query | account.id=${accountPoid} |

# multiple query parameters
Then read payment, Retrieve the Payment details
Data:
| $query | account.id=${accountPoid}&limit=1 |
```

> **Note:**
>
> For Patch method use **$urlSuffix** to send query parameters as part of url.

Using Variables in Query Parameters (Release 1.25.1.1.0 or later)

Query parameters in REST calls can include variables, which are dynamically substituted with runtime values. For example,

```
https://api.example.com/resource?searchspec=([Name]="${accountName}")
```

In this case, **${accountName}** will be replaced with its runtime value before the request is sent.

Refer to the following BDD example:

Scenario: Query Param processor for Variable substitution

Description: Automation for validating correct handling of query parameters containing multiple equals signs.

```
Tags: Test, E2E, QueryParamProcessing

Case: Process query params
```

```
Given set variable, to set name
Save:
| accountName | Marlan Brando  |

Then get query param response, to search for given name
Data:
| id | param |
| $query | searchspec=([Name]="${accountName}") |
Validate:
| $status | 200 |
Save:
| resp | $data |
```

Custom Headers

Custom header parameters can be passed in the test step.

- To provide a custom value to a request header parameters, prefix the header key with " $header_ ".
- Custom values for header parameter can be either a string or a variable saved in any of the previous steps.
- Passing Authorization header :
  - If other custom headers are present, but not an authorization header, then a new access token will be generated depending on the authorization type configured in the corresponding **environment.properties** file and will be passed in the authorization header while executing the step.
  - If there is an access token already available, to pass it in the step, use the custom value $header_Authorization for the access token to be passed with appropriate prefix (Example: Basic/ Bearer) depending on the authorization type being used.

For example,

```
When add category, for verifying customer details
Data:
| $header_Date | Wed, 17 April 2024 04:51:36 -0700 |
| name | John Doe |
| category | Platinum |

# Authorization header : Bearer token
When add category, for verifying customer details
Data:
| $header_Authorization | Bearer abcedeeeeeeeee |
| name | John Doe |
| category | Platinum |

# Authorization header : Basictoken
When add category, for verifying customer details
Data:
| $header_Authorization | Basic abcedeeeeeeeee |
| name | John Doe |
| category | Platinum |

# Using saved context variables in the header value
When add category, for verifying customer details
Data:
```

```
| $header_Date | ${Date} |
| $header_Authorization | %CONCAT(Bearer, ,${Token}) |
| name | John Doe |
| category | Platinum |
```

URL Suffix:

Suffixes to actual url can be added dynamically using $urlSuffix variable

For example,

```
Data:
| $urlSuffix | /purge |



# Using saved context variable in url suffix
Given set variable, dummy step
Save:
| param | /paramValue |

Given post step test, URL Suffix is a saved variable
Data:
| $urlSuffix | ${param}     |
Validate:
| $status | 404 |


#
Case: URL Params URL Suffix and URL Id test
Given put step test, test post
Data:
| $urlSuffix | /$urlId/checkin  |
| $urlId     | MyURLID400       |
Validate:
| $status | 404 |
```

Url Id with Url suffix
$urlId can be used to add an id value along with url suffix.

For example,

```
Given put step test, test post
Data:
| $urlSuffix | /$urlId/checkin  |
| $urlId     | MyURLID400       |
Validate:
| $status | 404 |

#using saved context variable in urlId
Given put step test, test post
Data:
| $urlSuffix | /$urlId/checkin  |
| $urlId     | ${accountId} |
```

```
Validate:
| $status | 404 |
```

Scenario Example :

TodoAppScenario.json

```
Scenario: RestAPI Scenarios
Description: Scenario for validating all the RestAPI plugin calls

Tags: RestAPI, Category, Customer

Case: Create a customer profile and view

When add category, for verifying customer details
Data:
| name | John Doe |
| category | Platinum |
Validate:
| $status | 200 |
Save:
| firstUser.id   | id |
| firstUser.name | name |
| firstUser.category | category |

Then read category, by id
Data:
| id | ${firstUser.id} |
Validate:
| $status | 200 |
| name | ${firstUser.name} |
| category | ${firstUser.category} |

When add category, for buying gold subscription
Data:
| name | John Doe |
| category | Gold |
Validate:
| $status | 200 |

Then read all todo tasks, that are created above.
Validate:
| [0].id | 1 |
| [0].name | John Doe |
| [0].category | Platinum |
| [1].id | 2 |
| [1].name | John Doe |
| [1].category | Gold |
Save:
| variable1 | %ARRAY_VALUE([?(@.category == 'Platinum')].name) |
```

SOAP Plugin

Simple Object Access Protocol (SOAP) plugin is used to automate the execution of SOAP API endpoints and to validate their responses. Automation plugins for SOAP focus on handling XML-based payloads and ensuring Web Services (WS-*) standard compliance.

The following are the key features of SOAP:

- **Message Customization:** Support for modifying SOAP body.
- **Security:** Handle WS-Security, SSL, and SAML token integration.
- **Assertions:** Validate SOAP responses against schemas and expected values.

SOAP Connection supports two types of authentications:

- Basic
- OAuth2

Refer to the following example for a Basic Authorization.

soap-environment.properties

```
#============================================================================
=
# BRM SOAP Environment Configuration
#============================================================================
=
name=brm
type=SOAP

#SOAP BASE URL
url=url

#============================================================================
=
# Authorization Configuration
# Values = YES : Use authorization NO : No authorization required.
#============================================================================
=
authorization=NO


#- BASIC Authorization
basic.username=
basic.password=

connection.uri.read_services_uri=BrmWebServices/BRMReadServices_v2?WSDL
connection.uri.cust_services_uri=BrmWebServices/BRMCustServices_v2?WSDL
connection.uri.payment_services.uri=BrmWebServices/BRMPymtServices_v2?WSDL
```

Refer to the following example for a Oauth2Authorization.

soap-environment.properties

```
#============================================================================
=
```

```
# BRM SOAP Environment Configuration
#================================================================================
=
name=brm
type=SOAP

#SOAP BASE URL
url=url

connection.uri.read_services_uri=BrmWebServices/BRMReadServices_v2?WSDL
connection.uri.cust_services_uri=BrmWebServices/BRMCustServices_v2?WSDL
connection.uri.payment_services.uri=BrmWebServices/BRMPymtServices_v2?WSDL

#================================================================================
=
# Authorization Configuration
# Values = YES : Use authorization NO : No authorization required.
#================================================================================
=
authorization=YES
authorization.type=oauth2

# OAUTH2 - IDCS Configuration
#oauth2.grantType= password_credentials OR client_credentials

oauth2.grantType=client_credentials
oauth2.clientId=
oauth2.clientSecret=
oauth2.tokenUrl=
oauth2.scope=
#username and password in case of password_credentials grant type
oauth2.authorization.username=
oauth2.authorization.password=
```

Action Configuration:

Action Configuration involves making SOAP API calls to perform operations such as creating a customer, updating information, or retrieving data.

Refer to the following example for creating a customer (create-customer.action.json).

```
{
 "path":"soap/brm/customer/create-customer",
 "name":"create customer",
 "description":"Create customer",
 "product":"brm",
 "actionType":"SOAP",
 "serviceURI":"${cust_services_uri}",
 "bdd":"create customer",
 "tags":["create","account"],
 "requestType":"FILE",
 "request":"create-customer.request.xml",
 "expectedStatusCode":200
}
```

Custom Headers

In Custom Headers, parameters can be passed in the test step.

> **Note:**
>
> 1. Prefix the header key with "$header_" to provide a custom value.
> 2. The custom value can be a string or a variable saved in previous steps.

Refer to the following example.

```
Then search plan,  Search the Plan Poid by Giving the plan name in BRM
Data:
| $header_Date | Wed, 17 April 2024 04:51:36 -0700 |
| planName | ${VistaOfferSalePOName} |
Validate:
| $status | 200     |
Save:
| timoDealPoid  | //DEALS/DEAL_OBJ/text()|
| timoPlanPoid  | //RESULTS/POID/text()  |
```

Scenario Example :

brm-soap.scenario

```
Scenario: BRM Scenario steps to create customer for E2E Scenario POC
Description: BRM Scenario steps to create customer for E2E Scenario POC

Case: Creating customer
Then search plan,  Search the Plan Poid by Giving the plan name in BRM
Data:
| planName | ${VistaOfferSalePOName} |
Validate:
| $status | 200     |
Save:
| timoDealPoid  | //DEALS/DEAL_OBJ/text()|
| timoPlanPoid  | //RESULTS/POID/text()  |

Then search deal,  Search the Deal Poid by Giving the Deal name in BRM
Data:
| dealName | ${VistaOfferSalePOName} |
Validate:
| $status | 200     |
Save:
| timoProductPoid  |  //PRODUCT_OBJ/text()  |

When create customer, Create a subscription account in BRM with the same
account no as Fusion
Data:
| productPoid | ${timoProductPoid} |
| dealPoid | ${timoDealPoid} |
| planPoid | ${timoPlanPoid} |
| serviceName | telco/gsm/telephony |
| accountNo | ${subscrAccountNumber} |
```

```
| qty | 1 |
| firstName | Tony |
| lastName | Stark |
| email | no-reply@oracle.com |
| address | 123 Main St |
| city | San Jose |
| state | CA |
| country | US |
| zip | 95110 |
| login | ts${UID} |
Validate:
| $status | 201 |
Save:
| accountPoid  |  //ACCOUNT_OBJ/text()      |
| billingInfoPoid  |  //BILLINFO_OBJ/text() |
```

XML API: Support for Sending Body in x-www-form-urlencoded

Any data sent in the case file needs to be appended with **key**_ to indicate that this is a key-value pair content that needs to be sent in the request body with type as `x-www-form-urlencoded`.

> **Note:**
>
> The 'Login to XML API' step is required to obtain the JSession ID from a successful login response. This ID must be included in the request headers of subsequent calls as a cookie to maintain the session.

The following are the contents of a case file that contains an XML API test:

```
Case: XML API Test with URL Encoding Content Type


Given login to XML API, using basic auth credentials

Validate:
| statusCode | 200 |
Save:
| JSESSIONID | %RESPONSE_HEADER(Set-Cookie)  |

Given external reference id for getting order id
Data:
| $header_Cookie | ${JSESSIONID} |
| $contentType | URL_ENCODED |
| key_xmlDoc | <Query.Request
xmlns="urn:com:metasolv:oms:xmlapi:1"><Reference>465-119337432</
Reference><OrderType>PO_OrderFulfillment</
OrderType><OrderSource>PO_OrderFulfillment</OrderSource><SingleRow>true</
SingleRow></Query.Request> |
Validate:
| statusCode | 200 |
```

```
Save:
| order_id | //Orderdata/_order_seq_id/text() |
```

The following are the contents of an action file that contains XML API:

login.action.json

```
{
 "path":"soap/xmlAPI/login",
 "name":"login",
 "description":"login",
 "product":"xmlAPI",
 "actionType":"API",
 "apiActionType":"SOAP",
 "serviceURI":"${xmlapi.login}",
 "bdd":"login to XML API",
 "tags":["login","XML API"],
 "expectedStatusCode":200
}
```

order.action.json

```
{
 "path":"soap/xmlAPI/xmlAPI",
 "name":"order",
 "description":"order",
 "product":"xmlAPI",
 "actionType":"API",
 "apiActionType":"SOAP",
 "serviceURI":"${xmlapi.order}",
 "bdd":"external reference id for getting order id",
 "tags":["order","reference"],
 "expectedStatusCode":200
}
```

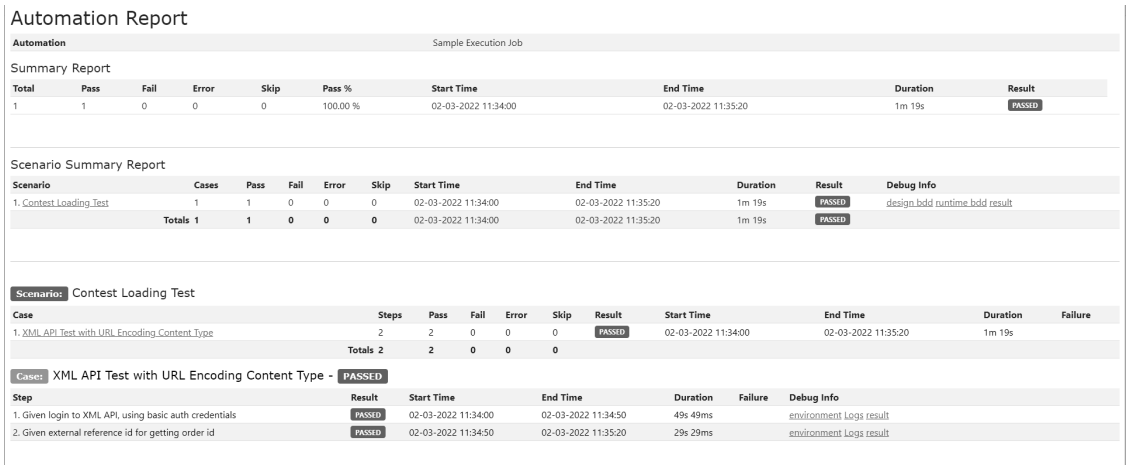
The following are the contents of properties file that contains XML API:

```
#================================================================================
=
# BRM SOAP Environment Configuration
#================================================================================
=
name=xmlAPI
type=api.soap
#*********************************************************************s*******
**
# Pre Defined Environment Properties
#********************************************************************************
*
\u200B
#SOAP BASE URL
#url= example.com
```

```
url= example.com
#================================================================================
=
# Authorization Configuration
# Values = YES : Use authorization NO : No authorization required.
#================================================================================
=
authorization=YES
authorization.type=BASIC
\u200B
#- BASIC Authorization
basic.username=omsadmin
basic.password=Osmpass1
\u200B
#********************************************************************************
*
# Custom Environment Properties
#********************************************************************************
*
#custom.read_services_uri=BrmWebServices/BRMWSReadServices_V2.wsdl
\u200B
connection.uri.xmlapi.login=login
connection.uri.xmlapi.order=XMLAPI
```

Figure 7-1 shows a sample of the automation report.

Figure 7-1 Automation Report Sample



SSH SFTP Plugin

Secure Shell (SSH) Plugin is used to run shell commands and SFTP is used to transfer files. They automate interactions with remote servers, making them invaluable for configuration management, server monitoring, and deploying applications.

The following are the key features of SSH SFTP Plugin:

- **Command Execution:** Automate execution of shell commands on remote servers.

- **File Transfers:** Transfer files securely using SCP or SFTP protocols.
- **Session Management:** Handle multiple sessions with session reusability.

Environment Connection Configuration

SSH SFTP supports two types of authentications:

- Basic
- Key (Public/Private)

Basic Authorization

Basic Authentication supports a straightforward authentication method where the client provides credentials (username and password).

Refer to the following example for basic authorization.

```
# Environment name
name=tasstest-ssh
type=SSH

#Configuration
hostname=hostname.oracle.com
port=22
#-------------------------------------------------------
# Authorization
#-------------------------------------------------------
authorization=YES
authorization.type=basic
username=
password=
```

Private key Authorization

Supports only RSA private key.

> **Note:**
>
> The key.file has to be present in the user's local system from where the scenario is performed.

```
#-------------------------------------------------------------------------------
-------
# SSH Command Sample Environment Connection Configuration
# Using Authentication KEY
#-------------------------------------------------------------------------------
-------
name=dx4c-ssh
type=SSH

#Configuration
hostname=123.456.78.9
port=22
#-------------------------------------------------------
# Authorization
```

```
#-------------------------------------------------------
authorization=YES
authorization.type=KEY
key.file=C:/Users/MSHAIK/.ssh/id_rsa
key.user=opc
```

Action Configuration:

The following are the contents of an action file that contains SSH commands:

```
{
"path":"SSHCommand/run-ssh-command",
"name":"run SSH command",
"bdd":"run SSH command",
"description":"run SSH command",
"tags":["ssh"],
"product":"ssh-test",
 "actionType":"SSH",
 "subType":"SSHCommandAction",
 "expectedStatusCode":0
}
```

TestStep

Step: run SSH command

Data parameter: SSH command, environment name

Validation parameters:

- SSH Command exit code using **$status**
- Response string : Using validation variable : **$data**
- Error response: Using validation variable : **$error**

Save parameters:

- Use save variable with value **'$data'** to save the command response.
- If the command is known to return an error, use **$error** to save the error response.

Scenario Example :

```
Then run SSH command, to check the current directory
Data:
| $command | pwd |
| $environment | tasstest-ssh |
Validate:
| $status | 0 |
| $data | %CONTAINS(tenant1) |
Save:
| currentDir | $data |
| homeDir | %SUBSTRING(${currentDir},0,5) |

Then run SSH command, to check the current directory and to check the user
Data:
| $command | pwd;whoami |
| $environment | tasstest-ssh |
```

```
Validate:
| $status | 0 |
| $data | %CONTAINS(tenant1) |

#command that generates both response and error
Then run SSH command, command generating both response and error
Data:
| $command | pwd;ls -lrt dummy.txt |
| $environment | ssh-test |
Validate:
| $status | 2 |
| $error | %CONTAINS(No such file or directory) |
Save:
| response | $data |
| errorResponse | $error |
```

Replacing Special Characters

If SSH Command has any of the following special characters, they should be replaced with keywords, otherwise publish scenario scripts might fail.

Table 7-1 Replacing Special Characters

Character	Description	Replace with
'	Single Quote	%{SQUOTE}
"	Double Quote	%{DQUOTE}
\	Backslash	%{BACKSLASH}
,	Comma	%{COMMA}

For example,

```
Then run SSH command, update the subscriberIdentifier in the
scenario_params_tmp.csv file
 Data:
 | $command | cd $HOME/enablement/seagull ; awk 'NR==2 {$2="\"${login_details}
\""} 1' FS=";" OFS=";" scenario_params_tmp.csv > temp && mv temp
#scenario_params_tmp.csv |
 | $environment | pdc-ssh |
 Validate:
 | $status | 0 |
```

SSH command in the example above should be provided as follows.

```
Then run SSH command, update the subscriberIdentifier in the
scenario_params_tmp.csv file
 Data:
 | $command | cd $HOME/enablement/seagull ; awk %{SQUOTE}NR==2 {$2=%{DQUOTE}%
{DQUOTE}${login_details}%{DQUOTE}%{DQUOTE}} 1%{SQUOTE} FS=%{DQUOTE};%{DQUOTE}
OFS=%{DQUOTE};%{DQUOTE} scenario_params_tmp.csv > temp && mv temp
scenario_params_tmp.csv |
 | $environment | pdc-ssh |
 Validate:
 | $status | 0 |
```

ExitCondition

Commands that do not exit on their own or take a long time to complete can be assigned as an exit condition.

$exitCondition: A predefined response from the SSH command can be used as an exit condition. If the SSH command freezes during execution or fails to return control, the response is checked for this exit condition. If it is detected, the SSH channel is closed by STAP.

$endAfter: When exit condition is present, it is mandatory to provide the end after time, to avoid indefinite wait time. While checking for the exit condition in the SSH response, if it is not found even after the end after duration elapses, STAP forcefully closes the SSH channel. **$endAfter** is mentioned in **seconds**.

> **Note:**
>
> The exit status of the SSH command in the above case is set to **-1** to indicate forceful termination.

For example,

```
#command that does not exit by itself
Then run SSH command, echo command, usage of expected response
Data:
| $command | sleep 5;echo done;sleep 20 |
| $exitCondition | %CONTAINS(done) |
| $endAfter | 15 |
| $environment | ssh-test |
Validate:
| $status | -1 |
```

Note:

- Only the SSH command can be passed as a data parameter to "run SSH command" step.
- More than one command can be passed in a single step, by separating the commands using semicolon(;).
- Supported validations are:
  - Exit code of the command using validation property $status.
  - %CONTAINS check for any string that may be a part of the command response or error.
- In response validation, single string can be passed to the **%CONTAINS** operator.
- Save variable with value '$data' should be used to save the command response. If the command generates any errors, it can be saved in $error. Functions can be operated on these saved variables.
- Both $data and $error can be used in single step. For instance, it is possible that a command generates some response but there is also an error in response, in which case both $data and $error can be used to validate and save the response accordingly.
- Each SSH Step opens a new ssh session with the remote server and hence any prerequisites needed for the command such as environment variables should also be set in the command.

Some exit codes and their definitions

- Exitcode 0: Command successfully performed
- Exitcode 1: Catchall for general errors
- Exitcode 99: Problem in the context of the specific program
- Exitcode 126: If a command is found but is not executable
- Exitcode 127: Command not found

SFTP Commands

SSH File Transfer Protocol commands for uploading and downloading files are supported as shown below.

For example,

```
Then run SSH command, upload file
 Data:
 | $command | $sftp:UPLOAD_FILE |
 | $environment | brm-ssh |
 | $source | $FILE(usageFile.csv) |
 | $target | /scratch/ri-user-1/dummy/sample.csv |
 Validate:
 | $status | 0 |


Then run SSH command, download file
```

```
Data:
| $command | $sftp:DOWNLOAD_FILE |
| $environment | brm-ssh |
| $source | /scratch/ri-user-1/dummy/sample.csv |
| $target | $FILE(usageFile1.csv) |
Validate:
| $status | 0 |
```

Step: run SSH command

Data parameters: SFTP command, environment name, source and target paths for file transfer.

Validation parameters: SFTP Command exit code.

Commands:

- **$sftp:UPLOAD_FILE**: Used to transfer file from local system to remote server.
  Parameters:
  - Source: Name of the local file to be transferred to remote server, where the file name should be specified as $FILE(filename) and it should be present inside "data" folder.
  - Target: The absolute path of the file destination on remote server.
- $**sftp:DOWNLOAD_FILE**: Used to transfer file from remote server to local system.
  Parameters:
  - Source: The absolute path of the source file on remote server.
  - Target: Name of the local file to which the remote file should be copied, where the file name should be specified as $FILE(filename).

> **Note:**
>
> - Both the source and target paths are mandatory for file transfer.
> - File names should be specified with extension.

SSH Private Key

STAP SSH Command supports only RSA private key.

If you see this error in STAP.

```
**********------------------------------------------------------------------
----------------
Running...SSH Command Action
Server : ssh
Action : run SSH command
Error : Failed to run command. Error : invalid privatekey: [B@222a59e6
**********------------------------------------------------------------------
----------------
```

If your private key appears similar to the example below when viewed in a text editor, you should convert it to an RSA private key.

```
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC...
...
...
...MAECAwQF
-----END OPENSSH PRIVATE KEY-----
```

Use ssh-keygen to convert your private key to RSA private key

```
ssh-keygen -p -f ~/.ssh/id_rsa -m pem
```

> **Note:**
>
> Replace the location of private key ~/.ssh/id_rsa

```
-----BEGIN RSA PRIVATE KEY-----
MIIG4wIBAAK...
...
...
...E428GBDI4
-----END RSA PRIVATE KEY-----
```

Troubleshooting

If the command is a script execution, ensure any prerequisites needed for it are also set in the command.

For example,

```
Then run SSH command and the script for modifying the account's profile (it
calls PCM_OP_CUST_MODIFY_PROFILE internally)
Data:
| $command | sh associateFFmember.sh ${profileObj} |
| $environment | pdc-ssh |
Validate:
| $status | 0 |
```

Generates an error:

```
testnap: error while loading shared libraries: libportal.so: cannot open shared
object file: No such file or directory
```

Here command contains execution of a script associateFFmember.sh that internally runs a command that needs the proper path set on $LD_LIBRARY_PATH. Since each STAP ssh step opens a new ssh connection, it is important to make sure path is set properly.

Resolution:

```
Then run SSH command, run the script for modifying the account's profile (it
calls PCM_OP_CUST_MODIFY_PROFILE internally)
```

```
Data:
| $command | export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/scratch/ri-user-1/opt/
portal/BRM/lib64:/scratch/ri-user-1/opt/portal/BRM/
lib;echo $LD_LIBRARY_PATH;sh associateFFmember.sh ${profileObj} |
| $environment | pdc-ssh |
Validate:
| $status | 0 |
```

Process Plugin

STAP process plugin is used to run the shell commands locally using java.lang.process.

Action

Command to be run using process plugin is mentioned in the action.json's field '**command**'.

Supported Types of commands :

1. Simple shell command
2. Command with variables
3. Command with parameters

1. Simple command:

Example: To run a shell command to fetch current directory :

run-pwd.action.json

```
{
"path":"process/run-command",
"name":"run pwd command",
"bdd":"run pwd command",
"description":"run pwd command",
"product":"process",
 "actionType":"PROCESS",
"tags":["custom","process"],
 "expectedExitCode":0,
"command":"sh,-c,pwd"
}
```

Example: To launch Notepad.exe

launch-notepad.action.json example

```
{
"path":"process/run-command",
"name":"launch notepad",
"bdd":"launch notepad",
"description":"launch notepad",
"product":"process",
"actionType":"PROCESS",
"tags":["custom","process"],
 "expectedExitCode":0,
"command":"notepad.exe"
}
```

2.Command with variables

Command can contain the variables whose value is updated from the context during runtime.

Syntax : ${ VariableName }

> **Note:**
>
> The variable name should have been saved in any of the steps that are performed before the step (action) which has that variable name in the action's command.

For example, in the following action.json, command has a variable : ${messageScript} that indicates the location of the script file to be run.

process-action.json example

```
{
"path":"process/run-command",
"name":"run message script",
"bdd":"run message script",
"description":"run message script",
"product":"process",
 "actionType":"PROCESS",
"tags":["custom","process"],
 "expectedExitCode":0,
"command":"sh,-c,sh ${messageScript}"
}
```

In the following scenario, the value for variable messageScript is saved in the step: '**set variable**' before the step '**run message script**'

So that updated command during execution will be : **"sh,-c,sh ProcessPlugin/Message.sh"**

message.scenario

```
#saving scripts paths
When set variable,
Save:
| messageScript | ProcessPlugin/Message.sh |

When run message script,
Validate:
| $status | 0 |
```

3.Command with parameters

Parameters/arguments in the command can be mentioned in format : %{ ParameterName : ParameterValue }

'ParameterValue' is the default value to be used. ParameterName is used just to check if value for it is passed from the Test Step's '**Data**' section.

If yes, then data variable's value overrides the default 'ParameterValue' . Final value of the parameter replaces %{ ParameterName : ParameterValue } in the command.

For example, in the following action.json, command has two parameters : %{FirstName:John} and %{SecondName:Tribbiani}.

If custom value for parameters **FirstName** and **SecondName** are specified from the test steps's Data section, then those values override the default values **John** and **Tribbiani** respectively.

process-action.json example

```
{
"path":"process/run-command",
"name":"run test script",
"bdd":"run test script",
"description":"run test script",
"product":"process",
 "actionType":"PROCESS",
"tags":["custom","process"],
"expectedExitCode":0,
"command":"sh,-c,sh ${testScript} %{FirstName:John} %{SecondName:Tribbiani}"
}
```

In the following scenario, custom value is provided for parameter '**FirstName**' only. Parameter '**SecondName**' takes the default value.

So that updated command during execution will be **: "sh,-c,sh ProcessPlugin/test.sh Joey Tribbiani"**

test.scenario

```
When set variable,
Save:
| testScript | ProcessPlugin/test.sh |

When run test script
Data:
#passing custom value for the parameter 'FirstName'
| FirstName | Joey |
Validate:
| $status | 0 |
```

Test Step:

```
Data:
 a) Parameters/Arguments for the command to be run.
 b) waitAfter : By default stap process plugin waits for 2 seconds for the
command to finish execution. If a command is known to take more than 2
seconds, then user must specify custom wait time in the Test Step using data
variable  'waitAfter'

Validation:
 a) $status : Expected exit code for the process executing the command.
Multiple comma separated exit codes can be specified.
 b) $data : String to be validated against the entire Response of the process
executing the command.
```

```
Save:
a) $data :  Entire Response of the process executing the command
```

Validation:

1. If Validation for exit code is not explicitly given in the Test Step (that is $status), then expectedExitCode mentioned in the action.json is used to validate if the execution is successful or not.
2. Only Validation properties supported in Process plugin are $status and $data. Functions and operators are supported on the $data as shown in below example.

   example

```
When run test script
Data:
| UserName | Joey |
Validate:
| $status | 0 |
| $data | %CONTAINS(Joey) |
```

Save:

Only Save property supported in Process plugin is $data. Once $data is saved in a variable, Functions and operators are supported on that variable as shown in below example.

example

```
When run test script
Data:
| UserName | Joey |
Save:
| scriptResponse | $data |
| scriptResponse2 | %UPPERCASE(${scriptResponse}) |
| scriptResponse3 | %SUBSTRING(${scriptResponse},0,4) |
```

Scenario Example:

process.scenario

```
Scenario: Process Plugin Automation Scenario
Description: Process Plugin Automation Scenario

Tags: Test, Process

Case: Process action test

When launch notepad
Validate:
| $status | 1 |

When run pwd command
Validate:
| $status | 0 |
```

```
#Multiple exit codes in validation
When run pwd command, multiple validation codes
Validate:
| $status | 0,1,2 |

#saving scripts paths
When set variable,
Save:
| messageScript | ProcessPlugin/Message.sh |
| testScript | ProcessPlugin/processPluginTest.sh |

#variables to be updated in action file's command
When run test script, sending variables to be updated in action file's command
Data:
| UserName | Joey |
| FullName | Joey_Tribbiani |
| Age | 30 |
Validate:
| $status | 0 |
| $data | %CONTAINS(Joey) |
#Saving response and operations on response and validation
Save:
| scriptResponse | $data |
| scriptResponse2 | %UPPERCASE(${scriptResponse}) |

#specifying waitAfter time
When run message script,
Data:
| message | Hello_Good_morning |
| waitAfter | 2 |
Validate:
| $status | 0 |
```

Seagull

Seagull is an open-source tool for testing and simulating network protocols. STAP Seagull plugin is used to run the seagull test scenarios. It can be used to generate the diameter traffic, provided the scenario and the required configuration files are present.

Key Features:

- **Protocol Simulation:** Simulate protocols like SIP, Diameter, and HTTP.
- **Traffic Generation:** Generate high volumes of traffic for stress testing.
- **Custom Scenarios:** Define custom test scenarios with dynamic parameters.
- **Performance Analysis:** Measure response times and system behavior under load.

Seagull Connection:

seagull-environment.properties

```
#=============================================================================
=
# Seagull Connection Configuration
#=============================================================================
=
```

```
#Fixed fields of seagull connection,do not modify.
name=seagull
type=SEAGULL

# User modifiable fields of seagull connection.
#Absolute path of seagull installation directory.
seagull.installationDirectory =
#seagull supported log levels
seagull.logLevel = ETMA
#Absolute path to store seagull execution logs.
seagull.logDirectory =
```

Action:

Supported action types:

- Creating seagull instance (Fixed action)
- Running seagull scenario

Create seagull instance

Following action.json is used to create seagull instance. The field '**instanceName**' is the default name used to create the instance. This is the fixed action to create seagull instance and should not be modified. Multiple seagull instances (that is having different config file and dictionary file) can be created by reusing this same action and saving the instance with different name using **$name** save variable in the test step.

create-seagull-instance.action.json

```
{
"path":"CustomAction/seagull-action",
"name":"Create seagull instance",
"bdd":"create seagull instance",
"description":"create seagull instance",
"product":"seagull",
 "actionType":"SEAGULL",
 "subType":"CREATE_INSTANCE",
"tags":["custom","process"],
"instanceName":"seagull"
}
```

Running seagull scenario

Depending on the scenario to run, any number of action.json can be created.

Name of the scenario to be performed is specified using the field 'scenario'.

client-scenario-sar.action.json

```
{
"path":"CustomAction/seagull-action",
"name":"Run client scenario",
"bdd":"run client scenario sar",
"description":"run client scenario",
"product":"seagull",
 "actionType":"SEAGULL",
 "subType":"RUN_SCENARIO",
```

```
"tags":["custom","process"],
"scenario":"sar-saa.client.xml"
}
```

Test Step:

Creating seagull instance:

```
Data:
a) $configFile : Name of the config file to be used for creating seagull
instance.
b) $dictionaryFile : Name of the dictionary file to be used for creating
seagull instance.

Save :
a)$name : Custom name for the seagull instance. This name overrides the
instanceName given in action.json.
```

For example,

create-seagull.case

```
When create seagull instance,
Data:
| $configFile | conf.client.xml |
| $dictionaryFile | base_cx.xml |
Save:
#
| seagull1 | $name |
```

Running seagull scenario

```
Data:

a) $name : Name of the seagull instance to be used for running the scenario.
An instance of this name should have been created before using 'create
seagull instance' step, otherwise execution will result in failure.
b) $externalDataFile : Name of the external data file ( CSV format ). This
data file is used to change content of the message in seagull scenario before
sending.
c) $params : To send the dynamic values for one or more fields, using these
values, the external data file is updated.

   Syntax : Data types of the field separated by comma ;Values of the fields
separated by comma.
   Example:
    | $params | number;16 |
```

For example,

create-seagull.case

```
When run client scenario sar,
Data:
| $name | seagull1 |
| $externalDataFile | external_client_data.csv |
| $params | number;16 |
```

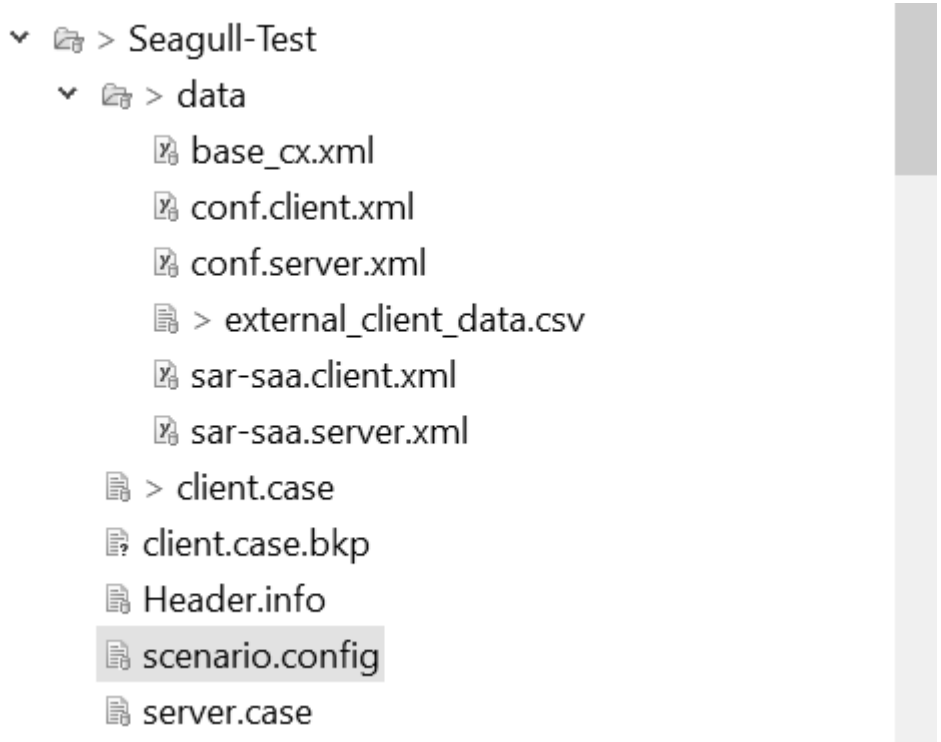
> **Note:**
>
> If the **$externalDataFile** is specified and **$params** is not specified, then the external data file is used as it is during scenario execution. If **$params** is present, then contents of external data file is overridden with the value of **$params.**
>
> You must carefully supply data types and values depending on the seagull scenario to be run.

Test Step Data:

You should create a folder named '**data**' under the same folder where STAP scenario to run seagull is created. The data files for creating seagull instance such as config.xml and dictionary.xml , Seagull scenario file scenario.xml and the external data file should be copied to this 'data' folder.

Figure 7-2 displays the Seagull folder structure:

Figure 7-2 Seagull Folder Structure



> **Note:**
>
> - In STAP, Seagull is launched in the background mode because otherwise it expects keyboard input.
> - If there are any errors found in the seagull log file, then error is thrown and STAP execution fails. User needs to have the knowledge of the seagull configurations (config.xml, dictionary.xml) and the seagull scenarios and should put these appropriate files under the 'data' folder in order to ensure successful execution of the STAP scenario.

Scenario Example:

seagullServer.case

```
Case: Seagull test-Server instance

#instance creation using default name
When create seagull instance,
Data:
| $configFile | conf.server.xml |
| $dictionaryFile | base_cx.xml |


When run server scenario sar,
Data:
| $name | seagull |
```

seagullClient.case

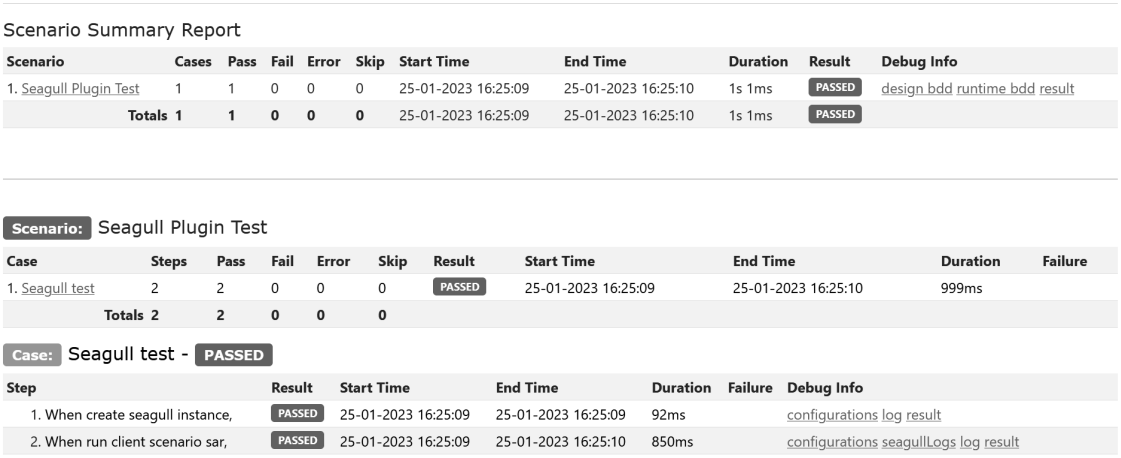
```
Case: Seagull client test

When create seagull instance,
Data:
| $configFile | conf.client.xml |
| $dictionaryFile | base_cx.xml |
Save:
| seagull1 | $name |


#scenario execution with external data file
When run client scenario sar,
Data:
| $name | seagull1 |
| $externalDataFile | external_client_data.csv |
| $params | number;16 |
```

Report

- **configurations** hyperlink in the report shows the seagull instance created and used for the scenario execution.
- **seagullLogs** hyperlink shows the logs generated by the seagull scenario execution.

Figure 7-3 displays an example Seagull Plugin Test yScenario Summary Report:

Figure 7-3 Seagull Plugin Test yScenario Summary Report



Scenario Summary Report

Scenario	Cases	Pass	Fail	Error	Skip	Start Time	End Time	Duration	Result	Debug Info
1. Seagull Plugin Test	1	1	0	0	0	25-01-2023 16:25:09	25-01-2023 16:25:10	1s 1ms	PASSED	design bdd runtime bdd result
Totals	**1**	**1**	**0**	**0**	**0**	25-01-2023 16:25:09	25-01-2023 16:25:10	1s 1ms	PASSED	

Scenario: Seagull Plugin Test

Case	Steps	Pass	Fail	Error	Skip	Result	Start Time	End Time	Duration	Failure
1. Seagull test	2	2	0	0	0	PASSED	25-01-2023 16:25:09	25-01-2023 16:25:10	999ms	
Totals	**2**	**2**	**0**	**0**	**0**					

Case: Seagull test - **PASSED**

Step	Result	Start Time	End Time	Duration	Failure	Debug Info
1. When create seagull instance,	PASSED	25-01-2023 16:25:09	25-01-2023 16:25:09	92ms		configurations log result
2. When run client scenario sar,	PASSED	25-01-2023 16:25:09	25-01-2023 16:25:10	850ms		configurations seagullLogs log result

Kafka

STAP Kafka is a component used within the Kafka Connect framework to integrate Apache Kafka with various data systems.

Message Queue Interface for Kafka

Automation plugins for message queues enable efficient testing and monitoring of message-driven systems.

Key Features:

- **Message Publishing:** Automate sending messages to queues.
- **Consumption:** Automate message retrieval and processing.
- **Serialization Support:** Handle Text, JSON and XML formats.

Kafka Connection

```
#--------------------------------------------------------------------------
----------
# Environment name
#--------------------------------------------------------------------------
----------
name=test
type=Kafka

#--------------------------------------------------------------------------
----------
# Bootstrap Servers
# List of comma separated bootstrap servers
#--------------------------------------------------------------------------
----------
servers=servername

#--------------------------------------------------------------------------
```

```
----------
# Authorization
# -- Not used in this version --
#-------------------------------------------------------------------------------
----------
authorization=NO
```

Action

The following table lists the action properties:

Table 7-2 Action Properties

Property	Mandatory	Description	Default Value	Allowed Values
actionType	Yes	Kafka Plugin Type	Kafka	Kafka
subType	Yes	Kafka action sub types		GET_TOPIC_LATEST_MESSAGE, PING_SERVER, SEND_TOPIC_MESSAGE, GET_MESSAGE_COUNT, DELETE_TOPIC_MESSAGES
topic	Yes	Topic name		
commit	No	Commit message read	false	true, false

Supported Action Types:

- Get Topic Last Message
- Ping Server
- Send Topic Message
- Get Message Count

Get Topic Last Message

```
{
"path":"Kafka",
"name":"Get Topic Last Message",
"bdd":"get topic last message",
"description":"get topic last message",
"tags":["Kafka,get,topic,message"],
"product":"test",
 "actionType":"Kafka",
 "subType":"GET_TOPIC_LATEST_MESSAGE",
 "topic":"test-topic",
 "commit": false
}
```

Ping Server

```
{
"path":"Kafka",
```

```
"name":"Ping Server",
"bdd":"ping server",
"description":"ping server",
"tags":["Kafka,ping,server,test"],
"product":"test",
 "actionType":"Kafka",
 "subType":"PING_SERVER",
 "topic":"test-topic",
 "commit": false
}
```

Send Topic Message

```
{
"path":"Kafka",
"name":"Send Topic Message",
"bdd":"send topic message",
"description":"Send Topic Message",
"tags":["Kafka,send,message"],
"product":"test",
 "actionType":"Kafka",
 "subType":"SEND_TOPIC_MESSAGE",
 "topic":"test-topic",
 "commit": false
}
```

Get Message Count

```
{
"path":"Kafka",
"name":"Get Message Count",
"bdd":"get message count",
"description":"get number of messages",
"tags":["Kafka,get,message,count"],
"product":"test",
 "actionType":"Kafka",
 "subType":"GET_MESSAGE_COUNT",
 "topic":"test-topic",
 "commit": false
}
```

Scenario Examples

Read last JSON message

```
When set variable,
Save:
| name | USER |

When get topic last message, for validating account creation message
Data:
| $messageType | JSON |
Validate:
| $status | SUCCESS    |
```

```
| name | stap user |
| %SUBSTRING($name,5)  | user |
| %SUBSTRING($name,5)  | %LOWERCASE(${name}) |
| address.residenceNo | 100001 |
Save:
| id | id |
| name | %SUBSTRING($name,5) |
| pin | address.pin |
```

Runtime Scenario

```
# Auto-generated by stap-BDD Formatter Version 1.0
Scenario: Kafka Automation Scenarios
Description: Kafka Automation Scenarios
#Tags:


#Persona:
Case: Kafka test

When set variable,
Save:
#|  Property  |  Value  |  Runtime Value  |
 |  name      |  USER   |  USER           |

When get topic last message, for validating account creation message
Data:
#|  Property      |  Value  |  Runtime Value  |
 |  $messageType  |  JSON   |  null           |
Validate:
#|  Property             |  Value               |  Property Value  |
Runtime Value       |  Result  |
 |  $status              |  SUCCESS             |  SUCCESS         |
SUCCESS             |  PASSED  |
 |  name                 |  stap user           |  stap user       |  stap
user           |  PASSED  |
 |  %SUBSTRING($name,5)  |  user                |  user            |
user                |  PASSED  |
 |  %SUBSTRING($name,5)  |  %LOWERCASE(${name})  |  user           |
CONDITION: SUCCESS  |  PASSED  |
 |  address.residenceNo  |  100001              |  100001          |
100001              |  PASSED  |
Save:
#|  Property  |  Value               |  Runtime Value          |
 |  id        |  id                  |  53245723485723487959 4  |
 |  name      |  %SUBSTRING($name,5)  |  user                   |
 |  pin       |  address.pin         |  560001                 |
```

Read Last XML Message

```
When set variable,
Save:
| name | USER |

When get topic last message, for validating account creation message
```

```
Data:
| $messageType | XML |
Validate:
| $status | SUCCESS      |
| //name | stap user |
| //address/city | Bangalore |
| %SUBSTRING($//name,5)  | user |
| %SUBSTRING($//name,5)  | %LOWERCASE(${name}) |
| %SUBSTRING(${name},1)  | SER |
Save:
| id | //id |
| name | %SUBSTRING($//name,5) |
| pin | //address/pin |
```

Runtime Scenario

```
# Auto-generated by stap-BDD Formatter Version 1.0
Scenario: Kafka Automation Scenarios
Description: Kafka Automation Scenarios
#Tags:


#Persona:
Case: Kafka test

When set variable,
Save:
#|  Property  |  Value  |  Runtime Value  |
 |  name      |  USER   |  USER           |

When get topic last message, for validating account creation message
Data:
#|  Property      |  Value  |  Runtime Value  |
 |  $messageType  |  XML    |  null           |
Validate:
#|  Property               |  Value                |  Property Value  |
Runtime Value       |  Result  |
 |  $status                |  SUCCESS              |  SUCCESS         |
SUCCESS             |  PASSED  |
 |  //name                 |  stap user            |  stap user       |  stap
user           |  PASSED  |
 |  //address/city         |  Bangalore            |  Bangalore       |
Bangalore           |  PASSED  |
 |  %SUBSTRING($//name,5)  |  user                 |  user            |
user                |  PASSED  |
 |  %SUBSTRING($//name,5)  |  %LOWERCASE(${name})  |  user            |
CONDITION: SUCCESS  |  PASSED  |
 |  %SUBSTRING(${name},1)  |  SER                  |  SER             |
SER                 |  PASSED  |
Save:
#|  Property  |  Value                  |  Runtime Value          |
 |  id        |  //id                   |  53245723485723487959 4  |
 |  name      |  %SUBSTRING($//name,5)  |  user                   |
 |  pin       |  //address/pin          |  560001                 |
```

Runtime Scenario with all cases:

```
Scenario: Kafka Automation Scenarios
Description: Kafka Automation Scenarios

Case: Kafka test

When set variable,
Save:
#|  Property  |  Value  |  Runtime Value  |
 |  name      |  USER   |  USER           |

When ping server, checking if kafka is available
Validate:
| $status | SUCCESS |

When send topic message, sending message for a topic
Validate:
| $status | SUCCESS |

When get message count, getting number of messages
Validate:
| $status | SUCCESS |

When get topic last message,
Data:
| $messageType | JSON |
Validate:
| $status | SUCCESS    |
| name | stap user |
| %SUBSTRING($name,5)  | user |
| address.residenceNo | 100001 |
Save:
| id | id |
| name | %SUBSTRING($name,5) |
| pin | address.pin |

When get message count, getting number of messages
Validate:
| $status | SUCCESS |
```

URL Access Validation

Accessibility of URLs can be verified from automation using URL Validation actions.

Environment connection:

URLs are specified with prefix "url." and request headers are specified with prefix "header." in the environment.properties file.

The value given for step's data variable: "url" should match with one of the url names mentioned in environment.properties file.

ui-environment.properties

```
name=test-ui
type=URL_VALIDATION

#UI Urls
url.launch=https://example.oracle.com/
url.care = https://example.oracle.com/
url.billingcare=http://example.oraclecloud.com/
url.pdc=http://example.oraclecloud.com/
url.osm_task=http://example.osm.org/
url.osm_orchestration=http://example.osm.org/
url.siebel=https://example.oracle.com/enu
url.siebel2=https://example.oracle.com/

#Request header configurations
header.Host = example.oraclecloud.com
header.Accept = text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,*/*;q=0.8
header.Accept-Encoding = gzip, deflate
header.Accept-Language = en-US,en;q=0.5
header.Upgrade-Insecure-Requests = 1
#header.User-Agent = Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:91.0) Gecko/
20100101 Firefox/91.0
```

Action:

Action file structure

```
{
"path":"CustomAction/url-action",
"name":"Validate URL",
"bdd":"validate URL",
"description":"run URL validation",
"tags":["custom","URL"],
"product":"test-ui",
 "actionType":"URL_VALIDATION",
 "expectedStatusCode":200
}
```

Request json

```
{
  "url":"url"
}
```

Scenario Example :

Case file

```
Case: Check accessibility of the DX4C UI Urls

Given validate URL, Launch UI
Data:
| url | launch |
```

```
Validate:
| $status | 200 |

Given validate URL, Care UI
Data:
| url | care |
Validate:
| $status | 200 |

Given validate URL, Billing care UI
Data:
| url | billingcare |
Validate:
| $status | 200 |

Given validate URL, PDC UI
Data:
| url | pdc |
Validate:
| $status | 200 |

Given validate URL, osm_task UI
Data:
| url | osm_task |
Validate:
| $status | 200 |

Given validate URL, osm_orchestration UI
Data:
| url | osm_orchestration |
Validate:
| $status | 200 |

Given validate URL, Siebel UI
Data:
| url | siebel |
Validate:
| $status | 200 |

Given validate URL, Siebel UI
Data:
| url | siebel2 |
Validate:
| $status | 200 |
```

Report:

> **Note:**
>
> The **Response** section in step result shows the static web page of the URL specified, if the URL returns HTML content.

Custom Actions

Following custom actions can be used to generate pass, validation error and general error cases from the scenarios.

Action

Action file structure

```
{
"path":"CustomAction/run-custom-action",
"name":"run custom action",
"bdd":"run custom action",
"description":"run custom action",
"tags":"custom",
"product":"custom",
 "actionType":"CUSTOM",
 "customActionType":"CustomTestAction",
 "expectedStatusCode":0
}
```

Test Step

```
Data parameters
a) type : Custom action type ( PASS / THROW_ERROR / THROW_VALIDATION_ERROR )
b) duration : Duration in milliseconds for which the execution should be paused.
c) error_message : Meaningful error message  in case the type passed is THROW_ERROR /
THROW_VALIDATION_ERROR
```

Scenario Example

Examples

```
When run custom action, pass case
Data:
| type | PASS |
| duration | 2000 |

When run custom action, validation error case
Data:
| type | THROW_ERROR |
| duration | 2000 |
| error_message | Error occurred, please try again |

When run custom action, validation error case
Data:
| type | THROW_VALIDATION_ERROR |
| duration | 2000 |
| error_message | Validation error occurred |
```

Mock Custom Action

Mock actions are the custom actions mainly used for testing. Test steps using mock actions , update the request with dynamic values and context values if present, and return it as response.

Action

Action file structure

```
{
"path":"CustomAction/mock-action",
"name":"run mock action",
"bdd":"run mock action",
"description":"run mock action",
"product":"custom",
 "actionType":"CUSTOM",
 "subType":"MockTestAction",
"tags":["custom","mock"],
"requestType":"FILE",
"request":"run-mock-action.request.json",
 "expectedStatusCode":200
}
```

Request json:

mock-action.request.json

```
{
    "id": "1",
    "name": "Buy 2L Milk",
    "description": "Buy 2L milk from nandini booth",
    "status": "CREATED"
}
```

data/tasks/mock-action.request.json

```
{
"id":"$ReferenceTask[0]",
"description":"$ReferenceTask[0]"
}
```

Scenario Example

Case file

```
Case: Mock action test

When run mock action, creating a task
Data:
| id | WeekdayTask-${UID} |
| name | WeekdayTask-${UID} |
Save:
| taskId | id |
| taskName | name |

#Updating request field id with saved taskId
When run mock action, reading the task
Data:
| $requestString | {"id":"id"} |
| id | ${taskId} |

#Saving data in reference object
```

```
When run mock action, creating a task
Data:
| id | WeekEndTask-${UID} |
| name | WeekEndTask-${UID} |
| description | Take a walk in the park |
Save:
| $REFERENCE{ReferenceTask} | id |
| $REFERENCE{ReferenceTask} | name |
| $REFERENCE{ReferenceTask} | description |

When run mock action, creating a task
Data:
| id | WeekEndTask2-${UID} |
| name | WeekEndTask2-${UID} |
| description | Do yoga and meditation |
Save:
| $REFERENCE{ReferenceTask} | id |
| $REFERENCE{ReferenceTask} | name |
| $REFERENCE{ReferenceTask} | description |

#Using control structure on mock action
When run mock action, reading the task
RepeatTimes:
| $times | 2 |
Data:
| $requestString | {"id":"id","description":"description"} |
#| id | %CONCAT(${taskId},"-",tuesday) |
| id | $REFERENCE{ReferenceTask:WeekEndTask} |
| description | $REFERENCE{ReferenceTask:WeekEndTask} |

#Reference data passed in both request json and Data section of the step.
When run mock action, reading the task
Reference:
| $referenceData | tasks |
| ReferenceTask | WeekEndTask |
Data:
| id | $REFERENCE{ReferenceTask:WeekEndTask2} |
#| description | $REFERENCE{ReferenceTask:WeekEndTask2} |
```

Runtime Scenario

run-mock-action.runtime.scenario

```
# Auto-generated by stap-BDD Formatter Version 1.0
Scenario: Contest Loading Test
Description:  Test to validate the context loading
#Tags:


#Persona:
Case: Mock action test

When run mock action, creating a task
Data:
#|  Property  |  Value               |  Runtime Value               |
 |  id        |  WeekdayTask-${UID}  |  WeekdayTask-mphAhXsyVrnjuA  |
```

```
 |  name       |  WeekdayTask-${UID}  |  WeekdayTask-mphAhXsyVrnjuA  |
Save:
#|  Property  |  Value  |  Runtime Value                |
 |  taskId    |  id     |  WeekdayTask-mphAhXsyVrnjuA  |
 |  taskName  |  name   |  WeekdayTask-mphAhXsyVrnjuA  |

When run mock action, reading the task
Data:
#|  Property        |  Value        |  Runtime Value                |
 |  $requestString  |  {"id":"id"}  |  {"id":"id"}                  |
 |  id              |  ${taskId}    |  WeekdayTask-mphAhXsyVrnjuA  |

When run mock action, creating a task
Data:
#|  Property     |  Value                     |  Runtime Value                |
 |  id           |  WeekEndTask-${UID}        |  WeekEndTask-mphAhXsyVrnjuA  |
 |  name         |  WeekEndTask-${UID}        |  WeekEndTask-mphAhXsyVrnjuA  |
 |  description  |  Take a walk in the park  |  Take a walk in the park     |
Save:
#|  Property                   |  Value        |  Runtime Value
|
 |  $REFERENCE{ReferenceTask}  |  id           |  WeekEndTask-mphAhXsyVrnjuA
|
 |  $REFERENCE{ReferenceTask}  |  name         |  WeekEndTask-mphAhXsyVrnjuA
|
 |  $REFERENCE{ReferenceTask}  |  description  |  Take a walk in the park
|

When run mock action, creating a task
Data:
#|  Property     |  Value                     |  Runtime Value                 |
 |  id           |  WeekEndTask2-${UID}       |  WeekEndTask2-mphAhXsyVrnjuA  |
 |  name         |  WeekEndTask2-${UID}       |  WeekEndTask2-mphAhXsyVrnjuA  |
 |  description  |  Do yoga and meditation  |  Do yoga and meditation       |
Save:
#|  Property                   |  Value        |  Runtime
Value                |
 |  $REFERENCE{ReferenceTask}  |  id           |  WeekEndTask2-
mphAhXsyVrnjuA  |
 |  $REFERENCE{ReferenceTask}  |  name         |  WeekEndTask2-
mphAhXsyVrnjuA  |
 |  $REFERENCE{ReferenceTask}  |  description  |  Do yoga and
meditation       |

When run mock action, reading the task
Data:
#|  Property        |  Value                                    |  Runtime
Value                          |
 |  $requestString  |  {"id":"id","description":"description"}  |
{"id":"id","description":"description"}  |
 |  id              |  $REFERENCE{ReferenceTask:WeekEndTask}    |
WeekEndTask-mphAhXsyVrnjuA               |
 |  description     |  $REFERENCE{ReferenceTask:WeekEndTask}    |  Take a
walk in the park                  |

When run mock action, reading the task
```

```
Data:
#|  Property        |  Value                        |  Runtime Value
|
 |  id              |  $REFERENCE{ReferenceTask:WeekEndTask2}   |
WeekEndTask2-mphAhXsyVrnjuA    |
 |  $requestString  |
{"id":"$ReferenceTask[0]","description":"$ReferenceTask[0]"} |
{"id":"$ReferenceTask[0]","description":"$ReferenceTask[0]"}  |
```

8
Synthetic Data

Learn about Oracle Communications Solution Test Automation Platform (STAP) Synthetic Data generation.

Topics in this chapter:

STAP Synthetic Data Generation

Overview

Synthetic Data Generator is a critical component of a test automation platform, designed to produce diverse, scalable, and high-quality data for testing applications. It eliminates the reliance on real-world data by generating customizable data sets that emulate production-like conditions, ensuring comprehensive test coverage and improving testing efficiency.

STAP offers two types of plugins for synthetic data generation: Internal and External.

- Internal plugins handle various data types, including numeric, alphanumeric, and text.
- External plugins connect with third-party providers, with the currently supported plugin being the Data Faker plugin, which integrates with Data Faker.

For more information, see https://www.datafaker.net/.

Configuration

Synthetic Data Generation plugins are assigned or configured with attribute data configuration which are used in STAP BDD automation. To configure and utilize synthetic data generation plugins within the STAP BDD automation framework, perform the following steps:

1. Configure the attribute home location in **config.properties**.
   Add property in ${WORKSPACE}/config/config.properties file. For more details see, Configuration Folder.

   attributeData.home=${WORKSPACE}/config/attributeData
2. Add attribute data configuration properties files in ${WORKSPACE}/**config/attributeData** directory. Each configuration file name should end with "-attributeData.properties".
3. In BDD, use the attribute values in Table 8-1 to retrieve next and current value:

Table 8-1 Synthetic Data Syntax

Value	Description	Syntax	Example
get Next Value	Computes the next value based on configuration and generates a new value	@{<attributeName>} or `@{<attributeName>.nextValue}`	`@{mobileNumber}` or `@{mobileNumber.nextValue}`
get Current Value	Retrieves the current generated value.	@{<attributeName>.currentValue}	`@{mobileNumber.currentValue}`

Plugin with External Generators

For more information on External Generators, refer to Fake Data Plugin.

Plugin with Internal Generators

Number Generation

Table 8-2 describes Unique Number Generation type, its properties, and runtime BDD:

Table 8-2 Unique Number Generation Table

Type	Description	Properties	Runtime BDD
NUMBER_UNIQUE_BOUND	number is bound in range of [startValue, endValue)	mobileNumber1-attributeData.properties # Attribute Name name=customerMobile # Short description description=10 digit mobile number #Plugin associated with the attribute plugin=NumberDataPlugin type=NUMBER_UNIQUE_BOUND # Persist data to be used in multiple executions # Persist YES/NO #persist=NO # Plugin Properties for generating data minValue=9999900000 maxValue=9999990009 increment=1	number_unique_bound.scenario Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: UniqueNumberGeneration When set variable, generating unique customer mobile numbers Save: #| Property | Value | Runtime Value | | name | @{customerMobile.currentValue} | 9999900000 | | name | @{customerMobile} | 9999900001 | | name | @{customerMobile.nextValue} | 9999900002 | | name | @{customerMobile.currentValue} | 9999900002 | | name | @{customerMobile} | 9999900003 |

Table 8-2 (Cont.) Unique Number Generation Table

Type	Description	Properties	Runtime BDD
NUMBER_UNIQUE_INFINITE	number has startValue and no endValue. Infinite values are generated	mobileNumber2-attributeData.properties # Attribute Name name=serviceMobile # Short description description=10 digit mobile number #Plugin associated with the attribute plugin=NumberDataPlugin type=NUMBER_UNIQUE_INFINITE # Plugin Properties for generating data minValue=999990004 increment=1	number_unique_infinite.scenario Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: UniqueNumberGeneration When set variable, generating unique service mobile numbers Save: #| Property | Value | Runtime Value | | name | @{serviceMobile.currentValue} | 999990004 | | name | @{serviceMobile} | 999990005 | | name | @{serviceMobile.nextValue} | 999990006 | | name | @{serviceMobile.currentValue} | 999990006 | | name | @{serviceMobile} | 999990007 |

Table 8-2 (Cont.) Unique Number Generation Table

Type	Description	Properties	Runtime BDD
NUMBER_UNIQUE_DIGITS	number has startValue and no endValue. Number of digits in the value is specified.	mobileNumber3-attributeData.properties # Attribute Name name=agentMobile # Short description description=10 digit mobile number #Plugin associated with the attribute plugin=NumberDataPlugin type=NUMBER_UNIQUE_DIGITS # Plugin Properties for generating data minValue=999990009 increment=1 numOfDigits=10	number_unique_digits.scenario Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: UniqueNumberGeneration When set variable, generating unique agent mobile numbers Save: #| Property | Value | Runtime Value | | name | @{agentMobile.currentValue} | 999990009 | | name | @{agentMobile} | 999990010 | | name | @{agentMobile.nextValue} | 999990011 | | name | @{agentMobile.currentValue} | 999990011 | | name | @{agentMobile} | 999990012 |

Table 8-2 (Cont.) Unique Number Generation Table

Type	Description	Properties	Runtime BDD
NUMBER_UNIQUE_VALUES	number has startValue and no endValue. Number of values generated is specified.	mobileNumber4-attributeData.properties # Attribute Name name=transactionMobile # Short description description=10 digit mobile number #Plugin associated with the attribute plugin=NumberDataPlugin type=NUMBER_UNIQUE_VALUES # Plugin Properties for generating data minValue=9999900014 increment=1 numOfValues=5	number_unique_values.scenario Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: UniqueNumberGeneration When set variable, generating unique transaction mobile numbers Save: #| Property | Value | Runtime Value | | name | @{transactionMobile.currentValue} | 9999900014 | | name | @{transactionMobile} | 9999900015 | | name | @{transactionMobile.nextValue} | 9999900016 | | name | @{transactionMobile.currentValue} | 9999900016 | | name | @{transactionMobile} | 9999900017 |

Random Number Generation

Table 8-3 decribes Randome Number Generation types, its properties, and runtime BDD:

Table 8-3 Random Number Generation

Type	Description	Properties	Runtime BDD
NUMBER_RANDOM_VALUES	random number; arguments passed are minimum_value and maximum_value; number is bound in range of [minimum_value, maximum_value)	randomNumber1-attributeData.properties # Attribute Name name=subscriptionID # Short description description=random number #Plugin associated with the attribute plugin=NumberDataPlugin type=NUMBER_RANDOM_VALUES # Plugin Properties for generating data minValue=999990000 maxValue=999999000	Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: RandomNumberGeneration When set variable, for generating random subscription IDs Save: #| Property | Value | Runtime Value | | name | @{subscriptionID.currentValue} | 9999900000 | | name | @{subscriptionID} | 9999943495 | | name | @{subscriptionID.nextValue} | 9999932406 | | name | @{subscriptionID.currentValue} | 9999932406

Table 8-3 (Cont.) Random Number Generation

Type	Description	Properties	Runtime BDD
			| | name | @{subscriptionID} | 9999980535 |

Table 8-3 (Cont.) Random Number Generation

Type	Description	Properties	Runtime BDD
NUMBER_RANDOM_DIGITS	random number; arguments passed are minimum_digits and maximum_digits	randomNumber2-attributeData.properties # Attribute Name name=phoneNumber # Short description description=random number #Plugin associated with the attribute plugin=NumberData Plugin type=NUMBER_RANDOM_DIGITS # Plugin Properties for generating data minDigits=5 maxDigits=10	Scenario: 3.AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: RandomNumberGeneration When set variable, for generating random phone numbers Save: #| Property | Value | Runtime Value | | name | @{phoneNumber.currentValue} | 4713264118 | | name | @{phoneNumber} | 9633152371 | | name | @{phoneNumber.nextValue} | 8724706855 | | name | @{phoneNumber.currentValue} | 8724706855 | | name | @{phoneNumber}

Table 8-3 (Cont.) Random Number Generation

Type	Description	Properties	Runtime BDD
			| 6736490057 |

Text Generation

Table 8-4 describes Text Generation types, its properties, and runtime BDD:

Table 8-4 Text Generation Table

Type	Description	Properties	Runtime bdd
TEXT_INIT_UPPER	Initial letter is uppercase, remaining letters are lower case	text1-attributeData.properties # Attribute Name name=MessageHeader # Short description description=random text of certain/variable length which starts with capital letter #Plugin associated with the attribute plugin=TextDataPlugin type=TEXT_INIT_UPPER # Plugin Properties for generating data minLength=4 maxLength=4	text_init_upper.scenario Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: TextDataGeneration When set variable, generating random Message headers Save: #| Property | Value | Runtime Value | | name | @{MessageHeader.currentValue} | Vzhn | | name | @{MessageHeader} | Cebx | | name | @{MessageHeader.nextValue} | Pyjc | | name | @{MessageHeader.currentValue} | Pyjc | | name | @{MessageHeader}

Table 8-4 (Cont.) Text Generation Table

Type	Description	Properties	Runtime bdd
			| Vqwl |

Table 8-4 (Cont.) Text Generation Table

Type	Description	Properties	Runtime bdd
TEXT_LOWER	All letters of the string are in lowercase	text2-attributeData.properties # Attribute Name name=channelId # Short description description=random text of certain/variable length which has all letters in lower case #Plugin associated with the attribute plugin=TextDataPlugin type=TEXT_LOWER # Plugin Properties for generating data minLength=5 maxLength=10	text_lower.scenario Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: TextDataGeneration When set variable, generating random channel IDs Save: #| Property | Value | Runtime Value | | name | @{channelId.currentValue} | wplqxfftdw | | name | @{channelId} | xnqnjnl | | name | @{channelId.nextValue} | ouedleyk | | name | @{channelId.currentValue} | ouedleyk | | name | @{channelId}

Table 8-4 (Cont.) Text Generation Table

Type	Description	Properties	Runtime bdd
			| hxbhksr |

Table 8-4 (Cont.) Text Generation Table

Type	Description	Properties	Runtime bdd
TEXT_UPPER	All letters of the string are in uppercase	ttext3-attributeData.properties # Attribute Name name=Transmission Code # Short description description=random text of certain/variable length which all letters are capital letters #Plugin associated with the attribute plugin=TextDataPlugin type=TEXT_UPPER # Plugin Properties for generating data minLength=7 maxLength=7	text_upper.scenario Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: TextDataGeneration When set variable, generating random Transmission Codes Save: #| Property | Value | Runtime Value | | name | @{TransmissionCode.currentValue} | QGJPQZM | | name | @{TransmissionCode} | GNCDAYG | | name | @{TransmissionCode.nextValue} | IIHHWYF | | name | @{TransmissionCode.currentValue} |

Table 8-4 (Cont.) Text Generation Table

Type	Description	Properties	Runtime bdd
			IIHHWYF | | name | @{TransmissionCod e} | JVCJIUA |

Table 8-4 (Cont.) Text Generation Table

Type	Description	Properties	Runtime bdd
TEXT_ALPHANUMERIC	Initial character of the string is a letter, remaining are alphanumeric characters	text4-attributeData.properties # Attribute Name name=sessionID # Short description description=random text of certain/variable length; Initial character of the string is a letter, remaining are alphanumeric characters #Plugin associated with the attribute plugin=TextDataPlugin type=TEXT_ALPHANUMERIC # Plugin Properties for generating data minLength=5 maxLength=15	text_alphanumeric.scenario Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: TextDataGeneration When set variable, generating random session IDs Save: #| Property | Value | Runtime Value | | name | @{sessionID.currentValue} | E3GcSGp | | name | @{sessionID} | llDCNvLmW7C | | name | @{sessionID.nextValue} | DUTyLGj40su | | name | @{sessionID.currentValue} | DUTyLGj40su | | name | @{sessionID}

Table 8-4 (Cont.) Text Generation Table

Type	Description	Properties	Runtime bdd
			| v4qqu70 |

Table 8-4 (Cont.) Text Generation Table

Type	Description	Properties	Runtime bdd
TEXT_ALPHANUMERIC_SPECIAL	Initial character of the string is a letter, remaining are alphanumeric and special characters	text5-attributeData.properties # Attribute Name name=accessKey # Short description description=random text of certain/variable length; Initial character of the string is a letter, remaining are alphanumeric and special characters #Plugin associated with the attribute plugin=TextDataPlugin type=TEXT_ALPHANUMERIC_SPECIAL # Plugin Properties for generating data minLength=10 maxLength=10	text_alphanumeric_special.scenario Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: TextDataGeneration When set variable, generating random access keys Save: #| Property | Value | Runtime Value | | name | @{accessKey.currentValue} | RU-6t60gH! | | name | @{accessKey} | LP02z8~Uoj | | name | @{accessKey.nextValue} | r$:K6UW[9Z \| \| name \| @{accessKey.currentValue} \| r$:K6UW[9Z | | name | @{accessKey}

Table 8-4 (Cont.) Text Generation Table

Type	Description	Properties	Runtime bdd
			| AJK/xg-/|I |

Unique ID Generation

Table 8-5 describes Unique ID Generation type, its properties, and runtime BDD:

Table 8-5 Unique ID Generation table

Type	Description	Properties	Runtime BDD
UNIQUE_ALPHABETIC	All characters are letters	uniqueID1-attributeData.properties # Attribute Name name=communicationToken # Short description description=Unique alphabetic value #Plugin associated with the attribute plugin=UniqueDataPlugin type=UNIQUE_ALPHABETIC # Plugin Properties for generating data length=8	unique_alphabetic.scenario Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: UniqueDataGeneration When set variable, for generating random communication tokens Save: #| Property | Value | Runtime Value | | name | @{communicationToken.currentValue} | poAAeAKL | | name | @{communicationToken} | LUoeAoAM | | name | @{communicationToken.nextValue} | peeUoKKN | | name | @{communicationTo

Table 8-5 (Cont.) Unique ID Generation table

Type	Description	Properties	Runtime BDD
			ken.currentValue} | peeUoKKN | | name | @{communicationTo ken} | fUoAKUKE |

Table 8-5 (Cont.) Unique ID Generation table

Type	Description	Properties	Runtime BDD
UNIQUE_ALPHANUMERIC	Random no. of letters and digits in the text; Initial character is a letter	uniqueID2-attributeData.properties # Attribute Name name=DeviceID # Short description description=Unique alphanumeric value; first character is a letter #Plugin associated with the attribute plugin=UniqueDataPlugin type=UNIQUE_ALPHANUMERIC # Plugin Properties for generating data length=18	Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: UniqueDataGeneration When set variable, for generating random device IDs Save: #| Property | Value | Runtime Value | | name | @{DeviceID.currentValue} | p73JD58DW79kjfA0e0 | | name | @{DeviceID} | L73d3F832HdQ6f0AU0 | | name | @{DeviceID.nextValue} | V7N9h5832vTkvBK000 | | name | @{DeviceID.currentValue} | V7N9h5832vTkvBK000 | | name | @{DeviceID} |

Table 8-5 (Cont.) Unique ID Generation table

Type	Description	Properties	Runtime BDD
			fv39N583279k810AUA |

Table 8-5 (Cont.) Unique ID Generation table

Type	Description	Properties	Runtime BDD
UNIQUE_ALPHANUMERIC_SPECIAL	Random no. of letters, digits and, special characters in the text; Initial character is a letter	uniqueID3-attributeData.properties # Attribute Name name=ProductKey # Short description description=Unique alphanumeric value including special characters; first character is a letter #Plugin associated with the attribute plugin=UniqueDataPlugin type=UNIQUE_ALPHANUMERIC_SPECIAL # Plugin Properties for generating data length=12	unique_alphanumeric_special.scenario Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: UniqueDataGeneration When set variable, for generating random product keys Save: #| Property | Value | Runtime Value | | name | @{ProductKey.currentValue} | pU!Uooo!!!0V | | name | @{ProductKey} | L0A00oeK!!0W | | name | @{ProductKey.nextValue} | L!Ke0eo!A!0r | | name | @{ProductKey.currentValue} | L!Ke0eo!A!0r | | name | @{ProductKey}

Table 8-5 (Cont.) Unique ID Generation table

Type	Description	Properties	Runtime BDD
			| L!!!!0!0!!Us |

Table 8-5 (Cont.) Unique ID Generation table

Type	Description	Properties	Runtime BDD
UNIQUE_FIRST_DIGITS	First x-characters are digits, rest are letters	uniqueID4-attributeData.properties # Attribute Name name=SerialNo # Short description description=Unique alphanumeric value; first x-characters are digits, rest are letters #Plugin associated with the attribute plugin=UniqueData Plugin type=UNIQUE_FIRST_DIGITS # Plugin Properties for generating data length=12 numOfDigits=4	unique_first_digits.scenario Scenario: AttributeData Description: Attribute Data Scenario for data generation Tags: [attribute, data] #Persona: Case: UniqueDataGeneration When set variable, for generating random device IDs Save: #| Property | Value | Runtime Value | | name | @{SerialNo.currentValue} | 1000eeAKAoop | | name | @{SerialNo} | 1000UoAoUoeW | | name | @{SerialNo.nextValue} | 1000AKUUUAAr | | name | @{SerialNo.currentValue} | 1000AKUUUAAr | | name | @{SerialNo}

Table 8-5 (Cont.) Unique ID Generation table

Type	Description	Properties	Runtime BDD
			| 1000KKeAeAUi |

Fake Data Generation

Datafaker is a library for Java and Kotlin to generate fake data. This is helpful when generating test data to fill a database, to generate data for a stress test, or anonymize data from production services.

STAP leverages data faker 2.4.2 (current or latest) and creates a plugin to use it to generate fake data for automation scenarios. It also supports the output in multiple languages.

For more information on Fake Data Plugin, see Data Faker Resource and Data Faker Github.

Table 8-6 lists the Supported Generator or methods:

Table 8-6 Supported Generator or Methods

Providers	Attributes
name	fullName, firstName, lastName, femaleFirstName, malefirstName, nameWithMiddle, prefix, suffix, title,username
internet	emailAddress, domainName, username, getIpV6Address
address	city, streetName, zipCode, buildingNumber, cityPrefix, citySuffix, country, countryCode, countyByZipCode, fullAddress, latitude, longitude, postcode, secondaryAddress, state, stateAbbr, zipCode, timeZone
number	randomNumber, digits, randomDouble, numberBetween, negative, positive, digit, randomDigitNotZero
timeAndDate	future, past,birthday
phoneNumber	phoneNumber, cellPhone, phoneNumberNational, subscriberNumber, extension
word	noun, preposition, conjunction, adverb, adjective,interjection, verb
text	text, lowercaseCharacter, uppercaseCharacter
barcode	gtin14
currency	name
subscription	paymentMethods, paymentTerms, statuses,subscriptionTerms
unique	fetchFromYaml
idNumber	idNumber

Configuration

To generate fake data, select the provider and corresponding attribute from the Data Faker Library Documentations mentioned in Table 8-7:

Table 8-7 Data Faker Library Documentations

property key	value (eg)	description
name	fakeData	Name of the attribute Data should be **fakeData**.
description	Fake data generator	Any short description.
plugin	DataFakerPlugin	Plugin name should be **DataFakerPlugin**.
type	collection	Should be the same value for pluginManager to recognize.
list	email,Double,Number,future	Enter comma separated custom named list of all the keys to be used in the case file for the scenarios.
<List> [n1] email [n2] firstName. . . . [n n] Double	internet.emailAddress firstName=name.fullName	Enter each of the keys entered in the list and in values the corresponding provider and its attribute to be used to generate random data. Format: `<key_name_provided_in_list> = <data_faker_provider>.<data_faker_attribute>(comma_separated_params/ custom_values_to_be_passed_in_attribute)` For example, list=Double Double=number.randomDouble(2,500,700) (the configuration is intended to generate a double upto two decimal places between 500 to 700)
locale	in ,ar	The language the output is expected in.

Ensure the *src/main/java/com/oracle/***cagbu/stap/data/plugins/datafaker/validMethods.properties** file supports the entry in attributeData.properties configuration. For more information, see Data Faker Resource .

The following is an example attributeData.properties File:

```
# Attribute Name
name=fakeData
# Short description
description=Fake data generator
#Plugin associated with the attribute
```

```
plugin=DataFakerPlugin
type=collection
# enter the list of methods to be used
list=emailAddress,ip,phoneNumber,fullName,discount,billcharge,dataPlan,future,
past,accessKey,networkName,barcode,currency
# enter the key as the method for each of the keys from the above list and
corresponding provider and attribute name as per data faker documentation
emailAddress=internet.emailAddress
ip=internet.getIpV6Address
phoneNumber=phoneNumber.phoneNumber
fullName=name.fullName
discount=number.numberBetween(1,5)
billcharge=number.randomDouble(2,500,700)
dataPlan=subscription.subscriptionTerms
future=timeAndDate.future
past=timeAndDate.past
accessKey=text.text(10,26,true,true,true)
networkName=word.noun
barcode=barcode.gtin14
currency=currency.name
# language to be used to generate data
locale=in
```

Fake Data Usage

Refer to the following format to invoke and use data faker plugin in a scenario case files:

```
| variable | @{$<key_mentioned_in_attributeData.properties_file>.<METHOD>} |


example:

Data:
| name | @{$firstName.currentValue} |
| name | @{$firstName.nextValue} |
| name | @{$firstName} |
```

Table 8-8 lists the methods supported.

Table 8-8 Methods Supported

METHOD	EXPECTED OUTPUT
`currentValue`	outputs the current value if there is no previously generated value, calls nextValue
nextValue	output is a newly generated value
<empty>	defaults to nextValue

Fake Data Generation Example

Example 1:

The following example shows how to generate and store random data using a variable-based approach:

```
Case: DataFaker

When set variable, generating random email addresses
Save:
| emailID1 | @{$emailAddress.currentValue} |
| emailID2 | @{$emailAddress.nextValue} |
| emailID3 | @{$emailAddress} |

When set variable, generating random ip addresses
Save:
| ipAddress1 | @{$ip.nextValue} |
| ipAddress2 | @{$ip} |

When set variable, generating random phone numbers
Save:
| mobile1 | @{$phoneNumber.nextValue} |
| mobile2 | @{$phoneNumber} |

When set variable, generating random service agent names
Save:
| agentname1 | @{$fullName.nextValue} |
| agentname2 | @{$fullName} |

When set variable, generating random discount percentages
Save:
| discount1 | @{$discount.nextValue} |
| discount2 | @{$discount} |

When set variable, generating random billing charges
Save:
| billing1 | @{$billcharge.nextValue} |
| billing2 | @{$billcharge} |

When set variable, generating random data plans
Save:
| dataplan1 | @{$dataPlan.nextValue} |
| dataplan2 | @{$dataPlan} |

When set variable, generating random expiry dates
Save:
| date1 | @{$future.nextValue} |
| date2 | @{$future} |

When set variable, generating random previous expiry dates
Save:
| expdate1 | @{$past.nextValue} |
| expdate2 | @{$past} |

When set variable, generating random access keys
Save:
| access1 | @{$accessKey.nextValue} |
| access2 | @{$accessKey} |
```

```
When set variable, generating random network names
Save:
| network1 | @{$networkName.nextValue} |
| network2 | @{$networkName} |

When set variable, generating random bar codes
Save:
| barcode1 | @{$barcode.nextValue} |
| barcode2 | @{$barcode} |

When set variable, generating random currencies
Save:
| currency1 | @{$currency.nextValue} |
| currency2 | @{$currency} |
```

Saving Synthetic Data into a Variable(Release 1.25.1.1.0 or later)

You can save the synthetic data generated using a data faker into a variable for further use. For instance, when generating IP addresses dynamically, the next generated IP value can be stored in a predefined variable for easy reference and reuse.

```
ipAddress1 = $ip.nextValue
```

Here, `$ip.nextValue` represents the next generated IP address, which is then stored in the variable `ipAddress1`. This allows the saved value to be used in subsequent operations or references within the application.

The following example shows how to validate if an account name already exists in Siebel:

```
And set variable, assign account name value to a variable
Save: | uniqueAccountName | ${accountName} |

And validate account name exists, in Siebel regardless of whether the status
code is 200 or 404
Data: | $query | searchspec=([Name] = "${accountName}") |
Validate: | $status |
$IGNORE_STATUS_VALIDATION |

And validate account name exists, in Siebel and execute the loop until the
status is 200 and
generate account name using Data Faker
RepeatWhile: | ${response.status} | 200 |
Data: | $query | searchspec=([Name] = "${accountName}")
| Validate: | $status | $IGNORE_STATUS_VALIDATION |
Save: | uniqueAccountName | ${accountName} |
| firstName | @{$firstName} | | lastName | @{$lastName} | | accountName |
%CONCAT(${firstName},
,${lastName}) |

And set variable, to save the account name which will be used to create
account in Siebel
Save: | accountName | ${uniqueAccountName} |

Save:
| uniqueAccountName | ${accountName} |
| firstName | @{$firstName} |
```

```
| lastName | @{$lastName} |
| accountName | %CONCAT(${firstName}, ,${lastName}) |
```

Part II
Getting Started with STAP UI

Learn how to use the Oracle Communications Solution Test Automation Platform (STAP) UI.

9

About STAP UI

Learn about Oracle Communications Solution Test Automation Platform (STAP) UI.

Icons in the STAP UI

Table 9-1 lists all icons present in the STAP UI.

Table 9-1 STAP UI Icons

Icon	Description
	View
	Edit
	Delete
	Add
	Run
	Restart
	Left Navigation Pane
	Expand Row
	Collapse Row
	More Actions (Only visible when the screen is zoomed at 90% or higher)

Using Keyboard Shortcuts

You can use keyboard shortcuts for many actions in Solution Test Automation Platform.

Table 9-2 lists the keyboard shortcuts in STAP user interface.

Table 9-2 Keyboard Shorcuts

Shortcut	Function
F2	Enters or exits Actionable Mode. Enables keyboard interaction with focusable elements inside an item.
Esc	Exits Actionable Mode.
Tab	In Actionable Mode: moves to the next focusable element within the item (loops to the first after the last). Outside Actionable Mode: moves to the next focusable element on page.
Shift + Tab	In Actionable Mode: moves to the previous focusable element within the item (loops to the last after the first) Outside Actionable Mode: moves to the previous focusable element on page.
Arrow Keys	Moves focus to the item in the respective direction (Up, Down, Left, Right).
Enter	Selects the current item. Does not deselect.
Space	Selects the current item or deselects any previously selected items.
Ctrl + Space	Toggles selection of the current item while preserving selection of other items.

Delete User Pop-up (Admin Dashboard):

- **Focus on the row, Press F2** then **Press Enter** for the first entry.
- To delete additional users, **Search user, Press F2** then **Press Enter**.

10

STAP UI Login Methods

Learn about how to get started with Oracle Communications Solution Test Automation Platform (STAP) UI.

STAP UI is highly extensible and comes with numerous built-in plugins that enable interaction with various application interfaces, such as REST.

Topics in this chapter:

Guidelines for Using STAP UI

The STAP user interface is accessible using any modern web browser. When using the UI:

- Do not use browser commands, such as Back, Forward, and Refresh, to avoid losing data. If you accidentally use a browser command, navigate to the primary link and, if required, sign in to STAP again.
- Do not open multiple instances of STAP in different browser windows or tabs of the same browser.
- Ensure that cookies are enabled in your browser.

About Authorization Modes

There are two modes of authentication available:

- **Basic Authentication** supports a straightforward authentication method where the client provides credentials (username and password).
- **Open Authorization (OAuth)** is an open standard authorization framework that enables users to grant third-party applications access to their data without exposing their usernames and passwords. Instead of sharing credentials directly, OAuth issues access tokens to authorize specific resource access.

About Login Page

The login page serves as the primary gateway to access STAP UI. It provides the necessary fields for authentication, where you enter your username and password.

1. Enter the **Username** and **Password**.
2. Click **Login**.

The system validates credentials and grants access if they match stored information, securely logging in the user.

Resetting Your Password

If you forget your password, follow these steps:

1. Click **Forgot Password**.
   This opens the **Reset Password** page.
2. Enter the user name and email address associated with the UI.
3. Click **Reset**.
   You will receive an email containing instructions for resetting your password.

Using IDCS Credentials for OAuth

For logging in using theOracle Identity Cloud Service (IDCS) environment:

1. Enter your username and password on the login page.
2. Click Sign In.
   If successful, you are redirected to the main **Dashboard**.

About STAP Dashboard

You can monitor real-time job execution details and track automation tasks in the main **Dashboard**. Table 10-1 shows the different components of main dashboard.

Table 10-1 STAP Dashboard

Field	Description
Scheduled	Total number of jobs.
Jobs	Total number of jobs scheduled to run at that point in time.
Completed	Total number of jobs running at that point in time.
Running	Total number of scenarios running.
Scenarios	Total number of scenarios.
Active	Total number of scheduled jobs that are running.

Monitoring Realtime Jobs

You can select the real-time jobs from the list to monitor. This section displays the fields listed in Table 10-2.

Table 10-2 Monitoring Realtime Jobs

Field	Description
Job Details	The Job number, name, environment, build number, and release.
Progress	The percentage of scenarios completed.
Duration	Time taken to complete the job.
Result	The percentage of passed and failed scenarios.

Table 10-2 (Cont.) Monitoring Realtime Jobs

Field	Description
Failure Analysis	The number of passed and failed scenarios in a pie chart format.

Viewing the list of Running Jobs

You can view the list of jobs that are in progress/being run in real time. The fields related to the jobs running are displayed in Table 10-3:

Table 10-3 Fields in Running Jobs

Field	Description
Job #	Job number (a unique number generated automatically by the system).
Name	Name of the job.
Scenarios	Number of scenarios.
Environment	The environment in which the jobs are being run.
Start Time	The date and time that the job was started.
Progress	Indicates the percentage of job execution completion status.
Actions	Displays icons to edit ( ) or delete ( ) the job.

11

STAP System and Administrator Console

Learn about user profiles, creating new users, and managing existing users in Oracle Communications Solution Test Automation Platform (STAP) UI.

Topics in this chapter:

About the User Profile Page

The profile page allows users to view and update their profile data. In an OAuth environment, you can only view profile details; you cannot edit or change your password. Admin users have additional privileges and indicators.

About Viewing and Editing Profiles

You can view key information about a user profile with the following details:

- User Name
- First Name
- Middle Name
- Last Name
- Display Name
- Email Address
- Admin Batch Indicator
  - Visual cue indicating admin privileges.
- Admin Dashboard Button
  - Visible only to admin users.
  - Navigates to the Admin Console page.
- Change Password Button
  - Update the current password with a new one.
- To edit profile details, click on the edit ( ) icon.
- To save your changes, click **Update**.

- To discard the changes done in the current transaction, click **Cancel**.

Changing Passwords

You can change your password by clicking the **Change Password** button. This opens a password change form with the following fields.

Table 11-1 lists the fields and the descriptions on the password change form.

Table 11-1 Change Password

Field	Description
Current password	Enter the current password.
New password	Enter the new password. Note: The password must be between 6 and 12 characters long and contain only alphanumeric characters.
Re-enter new password	Re-enter the new password.

Actions:

- To save your changes, click **Update**.
- To discard them, click **Cancel**.

Viewing OAuth Environment Profiles

You are restricted to viewing profile details only. You cannot edit profile data or change passwords. Additionally, there is no admin batch indicator, and you do not have access to the admin dashboard or profile editing features.

Managing Administrator Environment

Administrator environment provides a comprehensive list of all user profiles and facilitates user management tasks, including viewing, deleting, and creating users. Upon clicking the **Admin Dashboard** button, an admin user can view the user profiles in a tabular format with the following columns:

- User Name
- First Name
- Middle Name
- Last Name
- Display Name
- Email Address
- Actions
  - Includes a delete ( ) icon to delete a user after confirmation.

> **Note:**
>
> This feature is accessible only to admin users.

Table 11-2 lists the additional fields on the Admin Dashboard page.

Table 11-2 Additional Fields in the Admin Dashboard

Field	Description
Filter	Allows searching users based on metadata such as first name or last name.
Create New User	Opens a drawer or modal form with fields to create a new user.
Cancel	Reverts to the admin user profile page and cancels the operation.

Creating a New User

As a admin user, you can create a new user by clicking the **Create New User** button on the **Admin Dashboard** which opens a drawer or a modal form with the following fields:

- First Name
- Middle Name
- Last Name
- Display Name
- Email Address

Click **Create.**

An email with a temporary password is sent to the new user. The user can use this password for initial login.

Role-based Access

STAP categorizes its users into two types:

- Admin Users:
  - Have full control over user management, including viewing, editing, deleting, and creating users.
  - Granted access to the admin dashboard for administrative tasks.
- OAuth Users:
  - Limited to read-only access for profile viewing.
  - Restricted from accessing management features.

12

STAP UI Environment Management

Learn about environments, creating a new environment, adding connections, and managing existing environments in Oracle Communications Solution Test Automation Platform (STAP) UI.

Topics in this chapter:

About the Environment Page

To access the environments, navigate to the **Menu** using the navigation panel. Select the **Environments** option to view and manage execution environments.

The Environments page displays a list of all configured environments. Each row represents a unique environment. Table 12-1 lists the columns for the selected unique environment.

Table 12-1 Environment Details

Field	Description
Name	Environment name.
Connections	Number of connections mapped to the environment.
Release	Release number.
Build	Build number.
Actions	Actions to edit or delete the environment.

Creating a New Environment

You can create new environments with connection mapping for specific testing scenarios. This is a critical part of STAP automation platform, where you can manage execution environments used across different jobs. Each environment can have zero or more connections based on the scope of the test. These environments serve as a base to limit or direct job execution within the STAP.

Perform the following steps to create a new environment:

1. On the Environments page, click **Create Environment.**The **Create environment** page is displayed.
2. Enter the **Name, Release, Build Number, and Description.** (Release, Build Number, and Description are optional parameters)
3. To add connections to the environment, click **Add Connection.**The **Create Connection** pop-up window is displayed.

4. Enter the connection Name, Description, Product (product or application name), and Type (for example, REST, SOAP, and Json).
5. Click the add (+) icon to add one or more **Properties.** Enter the **Property Name** and **Value.** Click **Add Connection.**The new connection is created and you are reverted to the **Create Environment** page.
6. In the **Search connection** text box under **Connection** section, specify the just created connection ensuring that you have a connection tagged to this environment.
7. Click **Create** button at the top right corner of the screen to create a new environment with the required connection details.
8. To attach the connections to the environment, Click **Add Connections.**You are reverted to the **Create Environment** page.
9. Click Create button to create the new environment.

The newly created environment is displayed at the top of the list on the **Environments** page.

Updating an Existing Environment

To edit an existing environment, perform the following steps:

1. Navigate to the **Environments** page, you have two ways to open the **Edit Environment** page:
   a. Click the edit (✎) icon under **Actions** column on the row corresponding to the environment you want to modify.
   b. Click anywhere in the row of the environment you want to modify, and the **Edit Environment** page will open.

This action opens the **Edit Environments** page, which consists of two sections: **Overview** and **Connections**.

1. Click the edit (✎) icon on each section to edit.
2. In the **Overview** section, edit the **Name, Description, Release**, and **Build Number** as needed.
3. In the **Connections** section, View or Delete the existing connection by entering the connection name in the search box.
4. Click the add (+) icon to add new connections.
5. Click **Update** to save the edits made.
6. Click **Cancel** to abort the changes made.

Deleting an Existing Environment

To delete an existing environment, perform the following steps:

1. Navigate to **Environments** page, click the delete (🗑) icon under **Actions** column on the row corresponding to the environment you want to delete.
   A confirmation dialog box titled **Delete Connection** appears.
2. Click **Delete.**

13

STAP Jobs Management

Learn about jobs, creating a new job, updating an existing job, and running a job in Oracle Communications Solution Test Automation Platform (STAP) UI.

Topics in this chapter:

About Jobs Page

The term 'job' represents a package dedication unit that combines one or more scenarios (sets of test steps) to be run against a specific environment. A job is a test suite that is configured and ready to be run as a single entity.

To access Jobs, navigate to the **Menu** using the navigation panel. Select the **Jobs** option to view and manage jobs. The Jobs page allows you to view and manage all the previously created jobs in a single tabular format. Each row represents a unique job.

Table 13-1 lists the columns for each unique jobs.

Table 13-1 Job Details

Field	Description
Name	Name of the job.
Description	Brief information about the job.
Scenarios	Number of scenarios running within the job.
Environment	Linked environment.
Actions	Action icons to edit, run, or delete the job.

Creating a New Job

To create a new job, navigate to the **Jobs** page:

1. Click the **Create New Job** button on the top right corner of the page. This allows you to define and configure a new job from scratch. The fields include:
   - Name
   - Tags (optional)
   - Environment
   - Description (optional)

- Scenarios (select one or more from the available list by checking the box against each scenario)

2. Click **Create Job** to add the new entry to the jobs table.

Updating an Existing Job

To update an existing job:

1. Click the edit ( ) icon under **Actions** column on the row corresponding to the job you want to modify.
   a. This opens the **Edit Job** page with all pre-filled data, allowing you to modify the following job details:
   - Name
   - Tags
   - Environment
   - Description
   - Scenarios (to add or delete scenarios, select the check boxes against each scenario)
2. Click **Update** to save the changes and update the jobs table accordingly.
3. Click **Cancel** to abort the changes made.

Running a Job

You can run a job in two modes:

- Background
- Run and visit Dashboard (For monitoring realtime execution of the job)

Deleting a Job

To delete an existing job:

1. Navigate to the **Jobs** page, click the delete ( ) icon under the **Actions** column on the row corresponding to the job you want to delete.
   A confirmation dialog box appears.
2. Click **Delete.**

14

Accessing Previously Run Jobs

Learn about accessing the status and details of jobs run in the Oracle Communications Solution Test Automation Platform (STAP) UI.

Topics in this chapter:

- Viewing Job History
- Viewing Scenario Details

Viewing Job History

To manage previously run jobs, navigate to **Menu** using the navigation panel from the **Dashboard**. Select **History** to view and manage jobs.

All jobs run are listed in the order of their run date, with the most recent job at the top of the page. The row for each respective job displays the job name, job start date and time in the deployment server time zone, and the job status: passed or failed. You can also perform actions within the job row. To view more details about the job run, click on the view icon ( ). See "Viewing Scenario Details". To re-run the job, click the restart icon ( ).

> **Note:**
>
> If your screen is zoomed in at 90% or higher, click on the more actions icon ( ) to view or restart the job.

You can access the details of a specific job by clicking its respective row. The row expands and you can view the details listed in Table 14-1:

Table 14-1 Viewing Job Details

Field	Description
Name	Name of the job run.
Type	Type of the job run. The default job type is **Instant Job**.
Environment	The environment in which the job was run. For example, test or production.
Start Time	The date and time that the job was commenced.
Duration	The duration of time for which the job was run, in seconds and milliseconds.
Result	The status of the job: passed or failed.

Viewing Scenario Details

You can view detailed information about the scenarios run in a previously run job by clicking the view icon ( ) at the end of the row of the respective job in **History**.

You can view the details listed in Table 14-2:

Table 14-2 Scenario Status

Field	Description
Result	Total scenarios run in percentage.
Percentage	Scenarios passed in percentage.
Passed	Total number of scenarios passed.
Failed	Total number of scenarios failed.
Skipped	Total number of scenarios skipped.

You can also view an overview of each scenario of the job in visual graphs:

- **Job Results** shows the number of passed, failed, and skipped scenarios of the job in a pie chart format.
- **Failure Analysis** shows the reason for failure in a pie chart format. For example, the number of scenarios failed due to validation error, the number of scenarios failed due to configuration error.
- **Results by Duration** shows the time taken to run each individual scenario of the job in a graph format.
  For more information, see "Viewing the Results of Each Scenario Under a Job".

To restart the job, click on the **Restart Job** button on the top-right corner.

Viewing the Results of Each Scenario Under a Job

You can view the results of each individual scenario under the selected job under **Scenarios Result**.

If you have multiple scenarios, you can search for the title of the scenario that you want to view details for in the **Filter** search bar.

> **Note:**
>
> This search bar does not support filter tags.

All scenarios run under the job are listed with the details described in the Table 14-3:

Table 14-3 Scenario Results

Field	Description
Name	Name of the scenario.
Duration	The duration of time for which the scenario was run, in seconds and milliseconds.

Table 14-3 (Cont.) Scenario Results

Field	Description
Start Time	The date and time that the scenario was commenced.
Result	The status of the total number of tasks in the scenario: number of tasks passed, number of tasks failed, number of tasks skipped, and number of tasks containing errors.
Status	The final status of the scenario: passed or failed.

To view a detailed report of the scenarios run, click **View Detailed Report**.

Viewing the Detailed Report of Scenarios

Upon clicking **View Detailed Report**, you can view each scenario in detail including the tasks that were passed, failed, or skipped.

The pane on the left lists all the scenarios run. To view details of a particular scenario, click on its row.

You can view details of each task of the scenario run on the right pane:

1. To expand each task, click on the respective task row. You can also filter tasks by enabling or disabling the **Pass, Fail,** and **Skip** filter tabs. By default, all filter tabs are enabled.
2. Upon clicking on a particular task, you can view a list of all the steps run to perform the selected task in the scenario.
3. Click on the row of each respective step to view a detailed report of each step run. This opens a pop-up detailing information about the task run.

Table 14-4 lists the following details about the task run:

Table 14-4 Details

Field	Description
Name	Name of the task.
Action	Action performed in the task.
Type	The type of task performed.
Start Time	The start date and time of the task, in the time zone set in your UI.
End Time	The end date and time of the task, in the time zone set in your UI.
Duration	The time duration for which the task was run, in milli seconds.

Data

Displays data configured in the step's BDD. If no data is configured, this section is blank.

Validation

Displays validations created under the step in BDD.

Save

Displays saved variables and values present in the step.

Log

Displays a detailed report of each action performed listed in Table 14-5.

Table 14-5 Log Details

Field	Description
Level	The level of the action. You can switch between the log levels **INFO, DEBUG, ERROR,** and **WARNING** to view detailed logs.
Timestamp	The day, date, and time at which the action was performed.
Message	Details of the action performed.
Error	Details of an error when the action was performed. If there is no error, the column is blank.

15

Viewing Scenarios

Learn about viewing details of scenarios run in Oracle Communications Solution Test Automation Platform (STAP) UI.

To view the different scenarios run, navigate to **Menu** using the navigation panel from **Dashboard**. Select **Scenarios** to view scenario library.

All scenarios are listed on the left of the page, with various cases and tasks present in the selected scenario on the right panel. To search for a specific scenario, use the Search bar. You can also search using previously set Tags.

To expand details for each case and tasks, click the respective row of the scenario you want to expand. To expand all cases, click **Expand Cases** on the top-right. To expand all cases and tasks, click **Expand All**. To collapse all cases, click **Collapse Cases**. To collapse all cases and tasks, click **Collapse All**.

Within each task, you can view the following sections:

- **Data**: Each property and its value.
- **Validate**: Each property and its validation.
- **Save**: Each variable and its corresponding value saved.

16

Viewing Actions

Learn about viewing the details of all actions present in the action library for Oracle Communications Solution Test Automation Platform (STAP) UI.

Viewing Action Details

To view details of each action, navigate to **Menu** using the navigation panel from **Dashboard**. Select **Actions**.

The left pane displays a list of all actions present in the action library.

To filter actions by product, select it in the drop-down list under **Products**. To filter actions by type, select the action type in the drop-down list under **ActionTypes**, for example, **REST, SOAP**. To view details of an action, select it in the left pane.

Details

You can view the following under the **Details** section:

- **BDD**: The behavioral driven development for this action.
- **Type**: The type of action. For example, REST.
- **Method:** The action method. For example, GET, PUT, POST, DELETE, PATCH.
- **Path**: The path to the file containing the request for the action type.
- **Request Type:** Refers to the type of request.
- **Request:** The source of the request file. Only applies to PUT, PATCH, and POST requests.

> **Note:**
>
> These properties vary by plug-in type. If an action does not contain a particular field, it doesn't show under **Details**.

Request

You can view the request body of the action under **Request**. The following is an example of a request body:

```
            {
    "type": "DEFAULT",
    "name": "subscriber name",
    "region": "default region",
    "category": "default category",
    "offer": "default offer",
    "paymentType": "default payment type"
}
```

Request Data

Displays the request input in JSON format.

Validation

Shows the expected status code in the response body if the action is successful. For example, 201.

Part III
Automating Using STAP

Learn about automating scenarios using Oracle Communications Solution Test Automation Platform (STAP).

17

Automating Without Code

This chapter provides an overview of how you can create automation scenarios in Oracle Communications Solution Test Automation Platform.

Topics in this chapter:

Overview

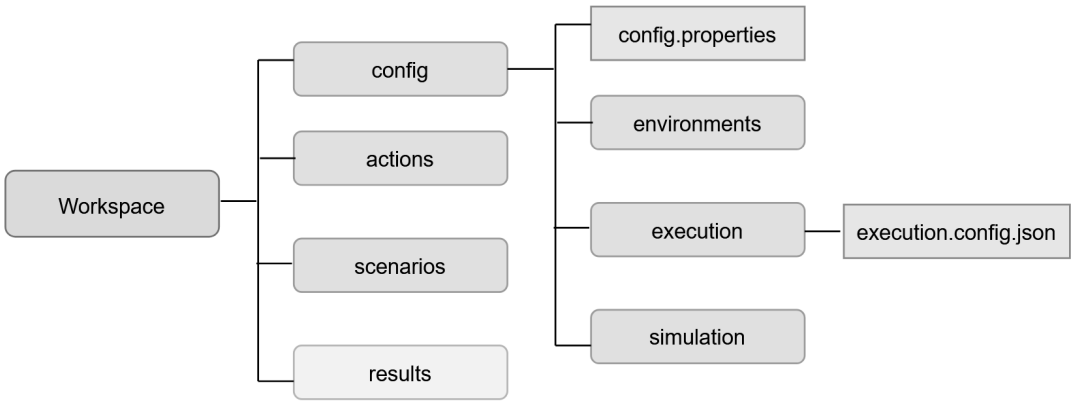
STAP enables users to automate workflows without writing complex lines of code. By leveraging the Behavior-Driven Development (BDD) language, STAP allows users to define automation scenarios in a clear, concise, and human-readable format. This approach simplifies the automation process, making it accessible to technical and non-technical users.

Automation Components

A well-organized project structure is essential for managing automation assets effectively within STAP.

Figure 17-1 shows a breakdown of the key project directories in STAP and their purposes.

Figure 17-1 Automation Workspace Hierarchy



- **Action Folder**
  - Contains action files required for running automation scenarios. For more information, see "Action".

> **Tip:**
>
> Maintain a clear hierarchy by creating subfolders for each product, with all related actions grouped within their respective product folders.

- **Config Folder**
  - Contains the **config.properties** file, which stores various configuration settings for your automation project. For more information, see "Configuration Folder".
    - **Environment:** Contains environment-specific configurations for your automation tests. For more information, see "Environment".
    - **Simulation:** Contains configurations related to test data simulation.
    - **Execution:** Contains execution configurations that define how automation scenarios are run.
- **Scenarios Folder:** Organizes your automation scenarios into a logical folder structure for improved maintainance and navigation.
- **Results Folder:** STAP automatically publishes all test execution results to this folder.
- **Context Folder:** Stores automation context data used to avoid redundant execution of steps during scenario automation. Refer to Context Folder for a deeper understanding.

Action

The Action component provides all necessary input to the respective plug-in. This input specifies how and with what data the plug-in should run the action.

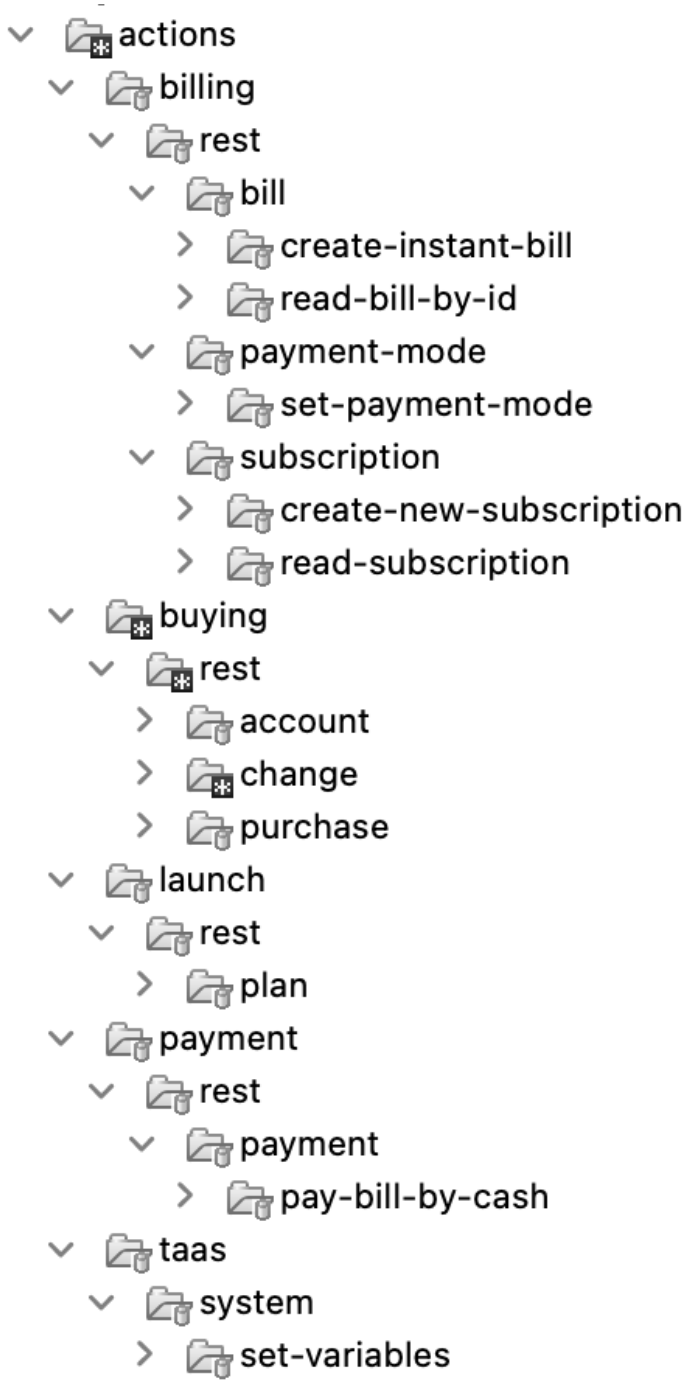
Action files contain common information, such as Name, BDD, Type, and Product. For more information on creating an action for a specific plug-in, see "Action Execution".

The structure of the Action folder is in the following hierarchy:

- Product Folder
- Plug-In Folder
- Path Folder
- Action Folder

Figure 17-2 displays a sample action file structure.

Figure 17-2 Sample Action File Structure



Guidelines and Best Practices

Follow these guidelines when creating the Action folder:

- Use lowercase letters with hyphens to separate words in action file names. For example, **create-new-subscription.action.json**. File names should be self-descriptive and end with the **.action.json** extension.
- Organize actions into sub directories based on logical groupings for better clarity.

- Optionally, share actions across automation projects as libraries by storing and publishing them as JAR files. For instructions on using action library JARs instead of folders, see "Configuration Folder".
- Structure your actions for easy searching and discovery by users.
- Provide descriptive names and descriptions for your actions.
- Use predefined tags to categorize actions and facilitate searching. Avoid duplicate tags.
- Write a concise BDD statement to describe the action's purpose.
- Always provide a default request/data for the action.
- Include a sample response in the documentation to inform users about the expected output. For example, for request **create-new-subscription.request.json**, the response file may be titled **create-new-subscription.response.json**.

The following example shows how to create an action using the REST plug-in:

```
{
"path":"subscription/create-new-subscription",
"name":"Create a new subscription",
"bdd":"create a new subscription",
"description":"Create a new subscription in the billing system",
"product":"billing",
"actionType":"REST",
"tags":["billing","subscription","create","new"],
"resource":"subscription",
"method":"POST",
"requestType":"FILE",
"request":"create-new-subscription.request.json",
 "expectedStatusCode":201
}
```

Scenario

A scenario outlines the conditions and expected outcomes of a test, focusing on the overall flow and user interactions. The file extension for a scenario in Solution Test Automation Platform is **.scenario**.

You must create a **README.md** file in each scenario folder. This file should include the following details:

- Author
- Supported product versions
- Revision history
- Exceptions (cases where the scenario may fail)
- FAQ for troubleshooting failures
- Other relevant notes

For more information about scenarios in STAP and how to create them, see "Creating Scenarios".

Guidelines and Best Practices

Follow these guidelines when creating Scenarios:

- Organize scenarios in a logical folder structure for improved design, execution, and maintainability.
- Create a hierarchical folder structure to group related scenarios together, making it easier to navigate, manage, and understand the overall test suite.
- Ensure unique scenario names. Consider adding the use case ID for clarity.
- Write detailed scenario descriptions. Describe the use case or end-to-end scenario comprehensively.
- Use tags to categorize scenarios for easy identification. For more information, see "Using Tags to Filter Components".

Case

A case represents a logical grouping of steps within a scenario. Cases allow you to modularize your automation scripts, improving readability, maintainability, and re-usability. Ideally, each case should focus on a single product or functionality within a broader scenario.

The file extension for a case is **.case**. You can break down your scenario into multiple case files under the scenario folder, ensuring easy distinction between functionalities and their test results.

Guidelines and Best Practices:

- Break down complex scenarios into smaller, more manageable cases based on the product or functionality they interact with. You can create single-step cases to enhance organization.
- Provide a detailed explanation of the case's functionality, including the steps it performs and the expected outcome.
- Utilize tags to categorize cases for easy identification and filtering based on various contexts like use case, feature, or functionality. Plan your tagging strategy before starting automation. For more information, see "Using Tags to Filter Components".

Using Default Cases

You can create a dedicated setup case to define the initial data and global variables required for the scenario. This improves clarity by centralizing data setup and highlighting the scenario's dependencies. You use multiple steps within the setup case to logically group variable assignments.

> **Note:**
>
> If any required global variable is missing, the setup case will fail.

Step

A step is the fundamental building block of a case within the STAP automation framework. Each step represents a single action or verification within the overall case flow.

Guidelines and Best Practices

- **BDD Syntax:** Utilize the Given-When-Then structure to clearly define the step's behavior within the context of the use case.
  - **Given:** Defines the initial state or preconditions.

  - **When:** Describes the action being performed.
  - **Then:** Specifies the expected outcome or verification.

  Complete the sentence after each keyword (Given, When, Then) with appropriate text following the comma, period, or semicolon.

- **Description:** Provide a concise and informative description of the step's purpose and functionality.
- **Tags:**
  - Inherit tags from the associated action.
  - Add additional step-specific tags to further categorize and filter steps. For example, product, feature, variation.
- **Data Usage:**
  - Avoid hard-coding data within step definitions.
  - Utilize variables defined in the setup case to ensure data consistency and re-usability across the scenario.
- **Validation:**
  - Implement robust validation checks within the step to verify the expected outcome.
  - Use clear and concise validation logic to easily identify and debug issues.
- **Data Saving:**
  - Save relevant data from a step's response or result for use in subsequent steps.
  - Employ a consistent naming convention for saved variables to prevent conflicts.

Environment

In STAP, environment configuration involves defining and managing the settings and parameters needed to run automation tests across different environments, such as development, testing, and production. This ensures that tests run correctly and produce accurate results across various target systems.

Each environment has it's own **environment.properties** file. You can have a single test environment, thereby having just one **environment.properties** file, or multiple environments, with multiple **environment.properties** files.

Guidelines and Best Practices

- Organize environment configurations into distinct folders within the **environment_configurations** sub directory. Use descriptive folder names that clearly identify the environment. For example, dev, qa, prod.
- Name environment files using a standardized format, for example, **<product>-<plugin>-environment.properties**. This enhances clarity and helps prevent naming conflicts.
- Designate a single individual as the owner of environment file updates for collaborative projects. This promotes consistency and reduces the risk of errors.
- The environment file owner should actively communicate any changes made to other team members.
- Utilize version control systems to track changes to environment files and facilitate collaboration.
- Conduct periodic reviews of environment configurations to ensure accuracy, completeness, and alignment with current system settings.

Project

The project structure plays a crucial role in organizing and managing automation assets within the STAP framework. A well-defined structure enhances code readability, maintainability, and collaboration among team members.

Guidelines and Best Practices

- Adhere to a Consistent Folder Structure:
- Utilize a standardized folder structure within your project to organize automation components effectively.
- Utilize the STAP reference project as a starting point for your own projects. Analyze the project structure and coding patterns to gain valuable insights into best practices.
- Regularly review and clean up the project structure to remove unused files and folders. Keep the project well-organized to facilitate easy navigation and maintainability.
- Employ a version control system to track changes, collaborate effectively, and revert to previous versions if necessary.

Naming Automation Components

This section outlines the best practices for naming automation components in STAP.

Files

- Use lowercase letters with hyphens to separate words. For example, **create-new-subscription.request.json**.
- Avoid special characters except hyphens.

Scenarios

There are two ways we can create Scenario:

- Single File: Simple scenario with less cases/steps. Created in a single **.scenario** file. For more information, see Single File Scenario
- Multi-Case Files Scenario (For Big/Complex scenarios): Complex multi product or end-to-end scenarios. Split the big scenario into multiple .case files and configure them in **scenario.config**. For more information, see Multi-Case Files Scenario

Include the use case ID for clarity. Optionally, add the product name. For example, **DBE1001-New-Subscription**.

Setup Cases

- Add **.setup.case** to the filename. For example, **1.set-default-data.setup.case**.
- Only create one **.setup.case** file per scenario, and run it first. This case can be skipped when using external data configurations.

Cases

- Use a concise and descriptive name that reflects the case's purpose.
- Include a unique identifier derived from the end-to-end use case or product functionality. This ensures uniqueness within the case library.
- Follow the format **<case-name>.product.case** to specify the product the case interacts with. For example, **2.create-new-subscription.billing.case**.

Variables

- Use lowercase letters with periods to separate words. For example, **subscription.id**.
- Add **Array** or **List** to identify array variables. For example, **orderItemArray**, **subscriberList**.
- Keep variable names concise and descriptive.
- Use periods (.) and hyphens (-) as separators.
- Avoid underscores (_) at the beginning, as these denote global variables.

Using Tags to Filter Components

Tags provide a mechanism for organizing, categorizing, and managing all automation components within STAP, including Scenarios, Cases, Steps, and Actions. You can plan and define a consistent set of tags before starting automation development.

You can filter Scenarios for execution based on specific tags. You can also select and run Cases within a Scenario using tags as criteria. Furthermore, you can generate automation execution configurations by filtering components based on tag criteria.

The following are some common examples that you can use when setting up Tags:

- Product Name
- Feature Name
- Use Case ID/Name
- Release
- Type (for example, Functional, Regression, Performance)
- Priority (for example, High, Medium, Low)
- Customer
- Topology/Setup/Environment
- Group/Category

Guidelines and Best Practices

- **Scenarios:** Use high-level tags like release, use case, type, and priority.
- **Cases:** Utilize product, feature, and priority tags.
- **Steps:** Apply product, feature, and variation tags.
- **Actions:** Tag with product, feature, and interface details.

18

Using the STAP Design Experience Package

This chapter details how you can use the Oracle Communications Solution Test Automation Platform Design Experience Package to simplify the end-to-end automation process.

Topics in this document:

Automating Using STAP Design Experience

The STAP Design Experience (DE) package simplifies the automation of end-to-end scenarios by offering a user-friendly Behavior-Driven Development (BDD) environment for creating, testing, and deploying automation. It includes streamlined scripts for compiling, running, and publishing automation, along with a sample workspace featuring diverse examples across various plugins. Additionally, the package provides ready-to-use environment templates tailored for specific plugins and environments to accelerate the automation process.

Before using the STAP Design Package, ensure you have set it up on your system. For more information, see "Setting Up The STAP Design Experience" in *Deployment Guide*.

The following is an end-to-end process of how to set up and run automation using the STAP Design Experience Package.

> **Caution:**
>
> Ensure you have securely stored your automation project in a third-party version control that includes initializing a repository, tracking changes, and collaborating efficiently.

1. **Create an Automation Workspace:** Create a dedicated folder within your project to serve as the automation workspace. STAP offers two ways to configure folder paths:
   - **Configuration Folder:** Create a **config** folder within the workspace. This folder contains the primary configuration file **config.properties**, which STAP run time uses to load other configurations. For more information, see "Configuration Folder". Create subfolders within the **config** to organize other configurations.
   - **(Optional) Environment Configurations:** Create an environments subfolder within the **config**. If you have multiple environments, inside each environment folder, create separate property files for each product API. If you only have one environment, create all environment property files directly under the environments folder. Update the **config.properties** file with the environment configuration location. For more information, see "Environments Folder".
2. **Results Folder:** STAP stores execution results in the results folder. The path can be relative to the workspace or an external location. Execution results are stored in timestamped folders under <*workspace*>**/results/**. You open **report.html** within each result folder to view the execution report. Configure the results storage location in **config.properties**. For more information, see "Results Folder".

3. **Context Folder:** The context folder stores test context data used during scenario development. Context helps visualize variables and their values used in each step. It allows executing specific steps while simulating previously run ones using the context. Configure the context storage location in **config.properties**. For more information, see "Context Folder".
4. **Scenarios Folder:** Define the location of the scenarios folder in **config.properties**. Each scenario is stored in a separate folder within this directory. For more information, see "Creating Scenarios".
5. **Compile and Run Automation:** Use the Command Line Interface to compile and run automation. For more information, see "Publishing Data using Command Line Interface".
6. **View Reports:** You can view the reports of the scenarios run in the Results folder. For more information, see "Results Folder".
7. **Publish Scenarios:** Once the automation is complete, you can publish scenarios. For more information, see "Publishing Data using Command Line Interface".

19

Creating an Automation Workspace Folder

This chapter describes the various components of the Automation Workspace folder in Oracle Communications Solution Test Automation Platform.

Topics in this chapter:

Configuration Folder

The configuration folder contains the primary configuration file titled **config.properties**. This file contains all the configuration required to run STAP.

> **Note:**
>
> Configurations not related to **config.properties** should be created under the configuration folder in their respective folders.

The following is the setup for the configuration folder:

```
#------------------------------------------------------------------------
--
# STAP Environment Configuration
# Version 1.2.0
#------------------------------------------------------------------------
--
# Scenarios location
scenarios.home=${WORKSPACE}/scenarios

#------------------------------------------------------------------------
--
# Environment configurations location
environments.home=${WORKSPACE}/config/environments

#------------------------------------------------------------------------
--
# Execution configurations location
execution.Config.file=${WORKSPACE}/config/execution/execution.config.json
#------------------------------------------------------------------------
--
# Actions location
```

```
actions.home=LOAD_FROM_LIBRARY
#
#actions.home=${WORKSPACE}/actions

#-------------------------------------------------------------------------
--
# Results storage location
#results=results
results.home=${WORKSPACE}/results
results.publish=NO
#results.publish.file=C:\software\Servers\apache-
Server-1\webapps\STAPReports\reports\SE2EReports\results.js
#-------------------------------------------------------------------------
--
# Context Configuration
#-------------------------------------------------------------------------
--
# Context Storage Location
context.home=${WORKSPACE}/context
# Scenario Context
# Load Context for the test case
# Default NO
context.load=NO
context.save=NO
# Global Context
context.global.load=NO
context.global.save=NO
#-------------------------------------------------------------------------
--
engine.configuration=${WORKSPACE}/config/engine.config.properties
#-------------------------------------------------------------------------
--



#-------------------------------------------------------------------------
--
 # JMeter Configuration

#-------------------------------------------------------------------------
--
 # JMeter threads
 tools.jmeter.thread=4000
 # JMeter rampup(seconds)
 tools.jmeter.rampup=150
 # JMeter result location
 tools.jmeter.results.home=${WORKSPACE}/results/tools/jmeter


#-------------------------------------------------------------------------
--
 # Plugin Configuration : INTERNAL
 # List of Supported Plugins : REST,SOAP,SSH,Kafka

#-------------------------------------------------------------------------
--
```

```
 plugin.internal=REST,SOAP,SSH,Kafka

#-------------------------------------------------------------------------
--
 # Plugin Configuration : CUSTOM
 # Provide plugin configuration in config/plugin folder

#-------------------------------------------------------------------------
--
 #plugin.custom=


#-------------------------------------------------------------------------
--
 # Attribute Home
 # Provide location to load attribute data

#-------------------------------------------------------------------------
--
 attributeData.home=${WORKSPACE}/config/attributeData
```

Environments Folder

The environments folder contains the various testing environments in STAP. This folder is a subfolder under the configuration folder. You create the environments folder under the configuration folder and create folders for each separate environment. Under each environment folder, create individual files for each product API.

The following is the configuration for adding the environment details in the **config.properties** folder:

```
#-------------------------------------------------------------------------
--
# Environment configurations location
#-------------------------------------------------------------------------
--
environments.home=${WORKSPACE}/config/environments
```

Results Folder

Results of the test run are stored in the **results** folder. The path to this folder can either be relative to your workspace or a direct path to store the results outside your workspace.

The results of each test run are created under this folder with its relative timestamp. The format of this timestamp is **$results/<timestamp>**. To view the execution report, you open the **report.html** file.

The following is the configuration for adding the result details in the **config.properties** folder:

```
#-------------------------------------------------------------------------
--
# Results storage location
#-------------------------------------------------------------------------
```

```
--
results.home=${WORKSPACE}/results
```

Context Folder

The context folder stores the data of previous steps, enabling the simulation of scenarios where only the current step needs execution. This eliminates the need to repeatedly run prior steps, as the context provides the necessary values for the current step.

You configure the location of the context folder in **config.properties**.

The following is the configuration for adding the context details in the **config.properties** folder:

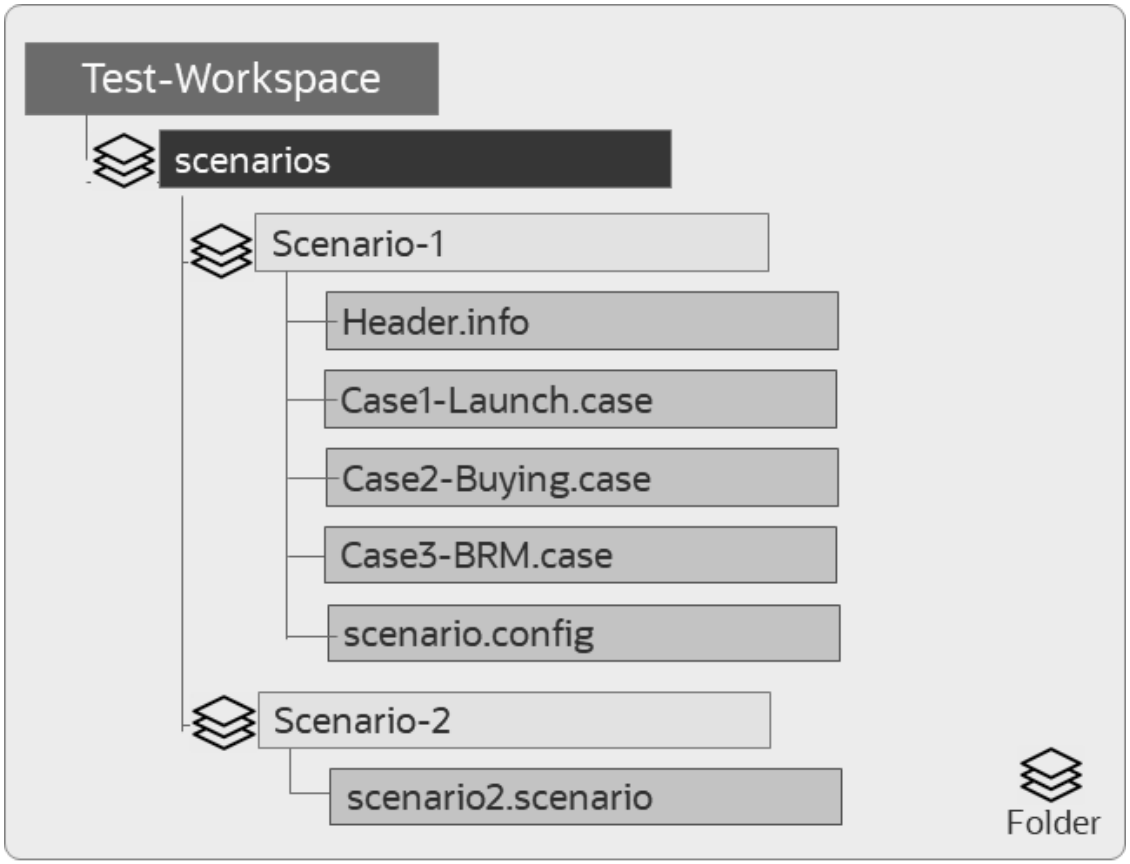
```
#--------------------------------------------------------------------------
--
# Context Configuration
#--------------------------------------------------------------------------
--
# Context Storage Location
context.home=${WORKSPACE}/context

#- Local Context. Load context while running. values : YES/NO
context.load=NO
#- Local Context. Save context for a scenario for debug. values : YES/NO
context.save=NO

# Global Context. Sharing data across scenarios.
context.global.load=NO
context.global.save=NO
```

Scenarios Folder

The scenarios folder contains the different scenarios to be run. Each scenario is stored in a separate folder within this folder.

Figure 19-1 displays the scenarios folder structure.

Figure 19-1 Scenarios Folder Structure



The following is the configuration for adding the scenario details in the **config.properties** folder:

```
#------------------------------------------------------------------------
--
# Scenarios location
#------------------------------------------------------------------------
--
scenarios.home=${WORKSPACE}/scenarios
```

For more information, see Creating Scenarios.

20

Creating Scenarios

This chapter details how to create scenarios to be tested and automated in Oracle Communications Solution Test Automation Platform.

Topics in this chapter:

There are two ways to create a scenario:

1. **Single File:** Created for a simple scenario containing a smaller set of cases and steps. It is stored in a single **.scenario** file.
2. **Multi-Case File:** Created for bigger scenarios containing complex multi-product or end-to-end scenarios. The scenario is split into multiple **.case** files and configured in the **scenario.config** configuration file.
   For more information on the scenario folder, see "Scenario".

Single File Scenario

The following format shows how to create a single file scenario:

```
Scenario:<<space>><Name of the E2E Scenario>

Description:<<space>><Description of the E2E Scenario>
<Description can be of multiple lines>
<<line space>>
<Description can have empty lines in between.>
<Entire text after Description: keyword till next keyword is considered as
description>
Tags:<<space>><List of tags separated by ','>

Case:<<space>><Case Name>    ====> Can have multiple cases in a scenario
Description:<<space>><<Case Description>>
Tags:<<space>><<List of tags separated by ','>
Given/When/Then/And<<space>><Step description>[<.|,|;><more step
description>]        ====> can have multiple steps in a case

Data:<<no text after this>>
| name  | value |
| name  | value |

Validate:<<no text after this>>
| name  | value |
| name  | value |

Save:<<no text after this>>
| Path | Variable |
```

Multi-Case Files Scenario

The following format shows how to create a multi-case file scenario:

1. **Header.info**: Contains the scenario details in the following format:

```
Scenario:<<space>><Name of the E2E Scenario>

Description:<<space>><Description of the E2E Scenario>
<Description can be of multiple lines>
<<line space>>
<Description can have empty lines in between.>
<Entire text after Description: keyword till next keyword is considered as
description>
Tags:<<space>><List of tags separated by ','>
```

2. **Case files**: Each **.case** file covers a specific logical step in a scenario. This logical criteria may be for any product. Use the following format when creating a case file:

```
Case:<<space>><Case Name>    ====> Can have only one case definition
Description:<<space>><<Case Description>>
Tags:<<space>><<List of tags separated by ','>
Given/When/Then/And<<space>><Step description>[<.|,|;><more step
description>]        ====> can have multiple steps in a case

Data:<<no text after this>>
| name  | value |
| name  | value |

Validate:<<no text after this>>
| name  | value |
| name  | value |

Save:<<no text after this>>
| Path | Variable |
```

3. **scenario.config** : Contains the list of files to be merged to create the **.scenario** file at run time. Use the following format when creating a scenario configuration file:

```
#====================================================================
# Scenario Configuration File
#====================================================================
# Merges the following scenario files in the specified order to run the
scenario
Header.info
1.Launch.case
2.Buying.case
3.fusionCDM.case
4.BRM.case
5.Care.case
```

Using Multiple Scenarios

Scenarios can be grouped to compile and run based on specific needs. This allows independent execution of groups, so the failure of one group does not halt the execution of others. This reduces the dependency of execution failures between independent scenarios.

Each group can be assigned a unique name and configured with a separate execution mode (serial or parallel). The execution mode dictates how the scenarios within a group are run.

To initiate a run, define at least one group entry. If multiple groups are not required, all scenarios can be listed under a single "group" keyword. You must define an **execution.config.json** file within the **Scenarios** folder to group scenarios.

The following is the syntax for the **execution.config.json** file:

```
{
    name keyword : parameter   //optional
    description : parameter    //optional
    release : parameter        //optional
    milestone : parameter      //optional
    build : parameter          //optional
    level : parameter          //optional
    reportTitle : parameter          //optional
    execution keyword : parameter       //optional

    group keyword : [
        {
            <<group 1>>
            name keyword : parameter            //optional
            execution keyword : parameter       //optional
            scenarios keyword : [
                <<folder name>>,
                <<folder name>>
            ]
        },
        {
            << group 2>>
            name keyword : parameter            //optional
            execution keyword : parameter       //optional
            group keyword : [
                {
                    <<subgroup 1>>
                    name keyword : parameter        //optional
                    execution keyword : parameter   //optional
                    scenarios keyword : [
                        <<folder name>>,
                        <<folder name>>
                    ]
                },
                {
                    <<subgroup 2>>
                    name keyword : parameter        //optional
                    execution keyword : parameter   //optional
                    scenarios keyword : [
                        <<folder name>>,
```

```
                        <<folder name>>
                    ]
                }
            ]
        }
    ]
}
```

Where:

- **Execution (Optional):** Execution can be configured as serial or parallel for each group or subgroup. This parameter is optional, and if not defined, groups will default to serial execution.
- **Group:** Groups are defined under the "group" keyword, and each group can contain subgroups or scenario folder entries. Each group has a unique name and its own execution mode. At least one "group" entry is required to define the scenario list.
- **Name:** Groups can be identified by user-entered names, and if no name is provided, a unique group ID is assigned.
- **Scenarios:**
  - If multiple **.scenario** files exist in a single folder, only the first .scenario file is run. Each group contains a scenarios list, which specifies the parent folder names where **.scenario** or **.scenario.config** files reside.
  - If **.scenario** files are located in nested folders, the parent folder names relative to the scenarios folder should be specified. Each group contains a scenarios list, which specifies the parent folder names where **.scenario** or **.scenario.config** files reside.
  - If **.scenario** files are located in nested folders, the parent folder names relative to the scenarios folder should be specified. Optional fields include a description, release version, milestone, build, level, and report title. The default report title is **Automation Report**.

The following is an example input for of the **execution.config.json** file:

```
{
    "execution" : "parallel",
    "group" : [
        {
            "name" : "groupOne",
            "execution" : "serial",
            "scenarios" : [
                "ToDo-E2E-Automation",
                "ToDo-FunctionsAndOperators"
            ]
        },
        {
            "name" : "groupTwo",
            "execution" : "parallel",

            "group" : [
                {
                    "name" : "subGroupOne",
                    "execution" : "serial",
                    "scenarios" : [
                        "ToDo-E2E-Automation",
```

```
                        "ToDo-FunctionsAndOperators"
                    ]
                },
                {
                    "name" : "subGroupTwo",
                    "execution" : "serial",
                    "scenarios" : [
                        "ToDo-E2E-Automation"
                    ]
                }
            ]
        }
    ]
}
```

21

Using the Command-Line Interface

This chapter describes how to use Command-Line Interface (CLI) to perform actions in Oracle Communications Solution Test Automation Platform.

Topics in this chapter:

Publishing Data using Command Line Interface

Solution Test Automation Platform utilizes the command-line interface to perform various actions. The **help** command provides a comprehensive list of all commands within STAP. Running the **help** displays each command's name alongside a brief description of its function

To retrieve information on how to run actions in STAP, run the help command:

```
$ ./stap --help
```

The following is the syntax of the output received after running the help command:

```
$ ./stap --help
============================================================
Solution Test Automation Platform CLI
Version : 1.25.0
============================================================
Usage: stap --<service> -<command> [<parameters>]

Global Options:

--version                                          Shows the STAP CLI version

--help                                             Shows the STAP CLI command
documentation
                        [<service> [<command>]]    Print help for module or
command in module

--automation                                       automation client
operations

        -compile                                   Compiles the automation
scenarios
                        workspace                  STAP workspace location
                                                   Valid folder path
                                                   Default Value: Current
Directory
                        scenarios                  one or more scenarios to
compile
                                                   List of values
                                                   Default Value:
```

```
                        generate                Generate the result files
from compile
                                                One of the values : [NO,
YES, MERGE]
                        config                  compile configuration
                                                Valid file path

        -run                                    Run the automation
scenarios
                        workspace               STAP workspace location
                                                Valid folder path
                                                Default Value: Current
Directory
                        scenarios               one or more scenarios to
run
                                                List of values
                                                Default Value: Selects
scenarios as per configuration or tags
                                                Optional Group : Scenario
Selection
                        tags                    Select scenarios matching
the tags
                                                List of values
                                                Optional Group : Scenario
Selection
                        caseTags                Select cases matching the
tags
                                                List of values
                                                Depends on : tags
                        config                  compile configuration
                                                Valid file path
                                                Optional Group : Scenario
Selection
                        mode                    Execution mode
                                                One of the values :
[trail, execute]

--publish                                       publish action

        -action                                 publish
                        workspace               STAP workspace location
                                                Valid folder path
                                                Default Value: Current
Directory

        -environment                            publish
                        workspace               STAP workspace location
                                                Valid folder path
                                                Default Value: Current
Directory

        -scenario                               publish
                        workspace               STAP workspace location
                                                Valid folder path
                                                Default Value: Current
Directory
```

```
--simulation                                    Run simulation

        -run                                    publish
                        workspace               STAP workspace location
                                                Valid folder path
                                                Default Value: Current
Directory

        -compile                                publish
                        workspace               STAP workspace location
                                                Valid folder path
                                                Default Value: Current
Directory

--secure                                        environment simulation

        -environment                            publish
                        filepath                path to the JCEKS file
                                                Valid folder path
                                                Mandatory: Yes
                        keyfilepath             Provide the path to
your .properties file containing the data to be encrypted.
                                                Valid folder path
                                                Mandatory: Yes
                        keystorepass            keystore password.
                                                Valid folder path
                                                Mandatory: Yes
                        aliasname               alias name identifying the
secret key
                                                Valid folder path
                                                Mandatory: Yes
```

The help command provides all the information required to perform various actions in STAP. For example, to retrieve information about your current STAP version, run the following command:

```
$ ./stap --version
============================================================
Solution Test Automation Platform CLI
Version : 1.25.0
```

Run the following command to run scenarios:

```
./stap --automation -run "workspace=<path>"
```

Alternatively, you can also run the **help** command to specifically search for command lines for a particular type of action. For example, to retrieve all commands related to running STAP run the following command:

```
$ ./stap --help secure
```

The following is the example output upon running this command: `$ ./stap --help secure`:

```
Solution Test Automation Platform CLI
Version : 1.25.0.0
============================================================
config/cli/secure.service.properties
Usage: stap --<service> -<command> [<parameters>]

Global Options:

--secure                                environment simulation

        -environment                    publish
                        filepath        path to the JCEKS file
                                        Valid folder path
                                        Mandatory: Yes
                        keyfilepath     Provide the path to your .properties
file containing the data to be encrypted.
                                        Valid folder path
                                        Mandatory: Yes
                        keystorepass    keystore password.
                                        Valid folder path
                                        Mandatory: Yes
                        aliasname       alias name identifying the secret key
                                        Valid folder path
                                        Mandatory: Yes
```

You can use this syntax to run the secure command in your STAP environment. On the basis of the above response, secure your STAP environment using the following command:

```
./stap --secure -environment filepath=<path> keyfilepath=<path>
keystorepass=<keystorepass> aliasname=<name>
```

Upon running this command, you will receive a response similar to the following:

```
./stap --secure -environment filepath=<path> keyfilepath=<path>
keystorepass=<keystorepass> aliasnam=<name>
[hostname STAP]$ ./stap --secure -environment keyfilepath=encrypt/env.jceks
filepath=sampleWorkSpace/config/environments/tdaasEnvironment.properties
keystorepass=Welcome@1 aliasname=password
============================================================
STAP Automation Platform CLI
Version : 1.25.1.0.0
============================================================
sampleWorkSpace/config/environments/tdaasEnvironment.properties
basic.password=${SECURE_PWD}
Enter new password for "basic.password":
password
Password updated successfully.
[hostname STAP]$
```

To publish automation reports to third-party web servers, see "Publishing Reports Using Third-Party Web Servers".

22

Publishing Reports Using Third-Party Web Servers

This chapter provides information about publishing automation reports to third-party web servers.

After automating scenarios using the command-line interface, you may want to share automation reports with other users, or access these reports from a web server for easy access. You can publish your reports to third-party web servers to perform these actions.

Topics in this document:

Viewing Automation Reports Using Tomcat

To view automation reports using Tomcat, follow these steps:

1. Install Tomcat. For more information, see the Tomcat website: https://tomcat.apache.org/

   Verify that your Tomcat server is running successfully by running the following in the URL of the Tomcat server:

   ```
   https://<tomcat-host>:<tomcat-server-port>
   ```

2. Navigate to the **config.properties** folder in your automation workspace, and configure the path of the results to publish by running the following command:

   ```
   results.home=${WORKSPACE}/results/reports
   results.publish=YES
   results.publish.file=${WORKSPACE}/results/results.js
   ```

3. Navigate to the Tomcat server using the path *${TOMCAT_HOME}*/**conf/server.xml**.

   a. Run the following command to configure the STAP-DE automation execution reports:

   ```
   <Connector port="8080" protocol="HTTP/1.1"
              connectionTimeout="20000"
              redirectPort="8443"
              maxParameterCount="1000"
              />

    <Connector port="8099" protocol="HTTP/1.1"
              redirectPort="8443" />
   ```

b. Navigate to the `<Content>` tag under the `<Host>` tag and insert the path to where you run the automation reports:

```
<Context docBase="${STAP_HOME}/sampleWorkSpace/results/" path="/stap-reports"
    />
```

This creates an endpoint titled **/stap-reports** which stores the automation reports.

4. Restart the Tomcat server.

Automation reports for all scenarios run are now published in the path `${host}:{port}/stap-reports`. To access individual automation execution results, click on the respective link of the job.

Viewing Automation Reports Using NGINX

To view automation reports using NGINX, follow these steps:

1. Install NGINX. For more information, see the NGINX website: https://nginx.org/
2. As an administrator, navigate to the command prompt in your system, and start the NGINX server by running the following command:

```
start nginx
```

3. Navigate to the **config.properties** folder in your automation workspace, and configure the path of the results to publish by running the following command:

```
results.home=${WORKSPACE}/results/reports
results.publish=YES
results.publish.file=${WORKSPACE}/results/results.js
```

4. Configure the path for the automation reports in the **nginx.conf** file by running the following in the NGINX server:

```
server {
        listen 80;
        server_name localhost;

        root ${STAP_HOME}/sampleWorkSpace/results;
        index index.html index.htm;

        location / {
            autoindex on;
            try_files $uri $uri/ /index.html;
        }
    }
```

5. Restart the NGINX server.

Automation reports for all scenarios run are now published in the path `${nginx-host}:{nginx-server}` in your web browser. To access individual automation execution results, click on the respective link of the job.

Viewing Automation Reports Using Apache HTTP Server

To publish automation reports using Tomcat, follow these steps:

1. Install and configure the Apache HTTP server. For more information, see the Apache website:
   https://httpd.apache.org/
2. Start the Apache HTTP server. Verify the successful installation by navigating to the port.
3. Navigate to the **config.properties** folder in your automation workspace, and configure the path of the results to publish by running the following command:

   ```
   results.home=${WORKSPACE}/results/reports
   results.publish=YES
   results.publish.file=${WORKSPACE}/results/results.js
   ```

4. Configure the path for the automation reports in the **httpd.conf** file by running the following:

   ```
   DocumentRoot "${STAP_HOME}/sampleWorkSpace/results/"
   <Directory "${STAP_HOME}/sampleWorkSpace/results/">
   ```

5. Restart the Apache HTTP server.

Automation reports for all scenarios run are now published in the path `${host}:{server}`. To access individual automation execution results, click on the respective link of the job.

A
Appendix

Action

An Action's functionality is defined by its constituent action files, enabling the execution of automated processes.

Action Plug-In

Enables automation to interact seamlessly with various product interfaces, including REST, SOAP, SSH, SFTP, and so on.

Array

A data structure that can hold multiple values, typically of the same type, in a single variable. Arrays are useful for organizing and managing collections of data.

Behavioral Development Data (BDD)

A proprietary language developed by Oracle. It uses a set of special keywords to structure and give meaning to executable business use case specifications.

Case

Represents a logical grouping of steps within a scenario.

Concat

A string function used to combine multiple strings into one. It takes two or more strings as input and returns a single concatenated string.

Controlled Steps

The way of executing test steps using different control structures such as if, for, and while.

Data-Driven Testing

A software testing methodology where test scripts are performed repeatedly using different sets of input data.

Environment

An environment refers to the combination of hardware, software, configurations, and settings required to execute automation tests.

JSON

A lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. JSON is commonly used to transmit data between a server and a web application, as well as to store configuration settings and other structured data.

Kafka

A distributed event streaming platform used for building real-time data pipelines and streaming applications. Kafka is widely used as a message queue to facilitate asynchronous communication between producers (who send messages) and consumers (who receive messages).

Microservice

A software development technique where an application is structured as a collection of loosely coupled, independently deployable services. Each microservice focuses on a specific business function and communicates with other services through well-defined APIs. This approach enhances scalability, flexibility, and maintainability, allowing teams to develop, deploy, and scale parts of an application independently.

Nested Function

A nested function is a function that is defined inside another function. It allows for code organization, data hiding, and, in some languages, the creation of closures.

Operator

An operator is a function that performs an operation on given arguments and returns a result as Passed or Failed. BDD operators are used in the Validation section of the Test Step.

Project

A STAP project organizes and manages automation assets within the STAP framework.

Pattern Matcher

A pattern matcher retrieves a substring using a regular expression. In STAP, the regular expression used by the pattern matcher contains characters that need to be escaped. If these characters are not escaped, the publish scenario scripts might fail.

REST

A widely used interface for web services due to its simplicity and scalability. Automation plugins for REST typically facilitate tasks such as making HTTP requests, handling JSON/XML payloads, and validating responses.

Scenario

Outlines the conditions and expected outcomes of a test, focusing on the overall flow and user interactions.

Seagull

An open-source tool for testing and simulating network protocols.

Seed Data

An initial set of data that is loaded into a system or database to set it up for use. This data is typically used to populate the database with essential information that the application needs to function correctly.

SFTP

Widely used for secure file transfers between systems. Automation plugins streamline tasks such as uploading, downloading, and verifying files.

SOAP

A widely used interface for web services due to its simplicity and scalability. Automation plugins for REST typically facilitate tasks such as making HTTP requests, handling JSON/XML payloads, and validating responses.

Solution Test Automation Platform (STAP)

Allows users to automate their end-to-end business use cases without writing a single line of code.

SSH

SSH plugins automate interactions with remote servers, making them invaluable for configuration management, server monitoring, and application deployment.

STAP Data Service

The Data Service microservice is responsible for managing the data used in STAP. It stores test case data, test results, and other important information related to testing. The Data Service is designed to be highly scalable, allowing it to handle large amounts of data without impacting performance.

Step

Represents a single action or verification within the overall case flow.

String

A sequence of characters used to represent text. Strings can include letters, numbers, symbols, and spaces.

Substring

Substring refers to a function used to extract a part of a string based on specified indexes. It takes a string and a starting index as input and returns the portion of the string from the starting index to the end of the string. The starting index is inclusive, meaning the character at the starting index is included in the resulting substring.

Synthetic Data

The Synthetic Data Generator is a critical component of a test automation platform, designed to produce diverse, scalable, and high-quality data for testing applications. It eliminates the reliance on real-world data by generating customizable datasets that emulate production-like conditions, ensuring comprehensive test coverage and improving testing efficiency.

Tags

Tags provide a mechanism for organizing, categorizing, and managing all automation components within STAP, including Scenarios, Cases, Steps, and Actions.

URL Validator

A tool or function used to ensure that the URLs specified in the configuration file (for example, **environment.properties**) are valid and correctly formatted.