Oracle® Communications Solution Test Automation Platform

User Operations Guide

Release 1.26.1.0.0
G48702-01
April 2026

ORACLE

Oracle Communications Solution Test Automation Platform User Operations Guide, Release 1.26.1.0.0

G48702-01

Contents

5 BDD Functions

6 Using Control Structures in Steps

7 STAP Action Plug-ins

About This Content

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.

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 low-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 plug-ins 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 plug-ins	Provides a comprehensive set of plug-ins and frameworks for automating the end-to-end validation of software applications. It supports various types of plug-ins, 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.
Low-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 plug-ins 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.

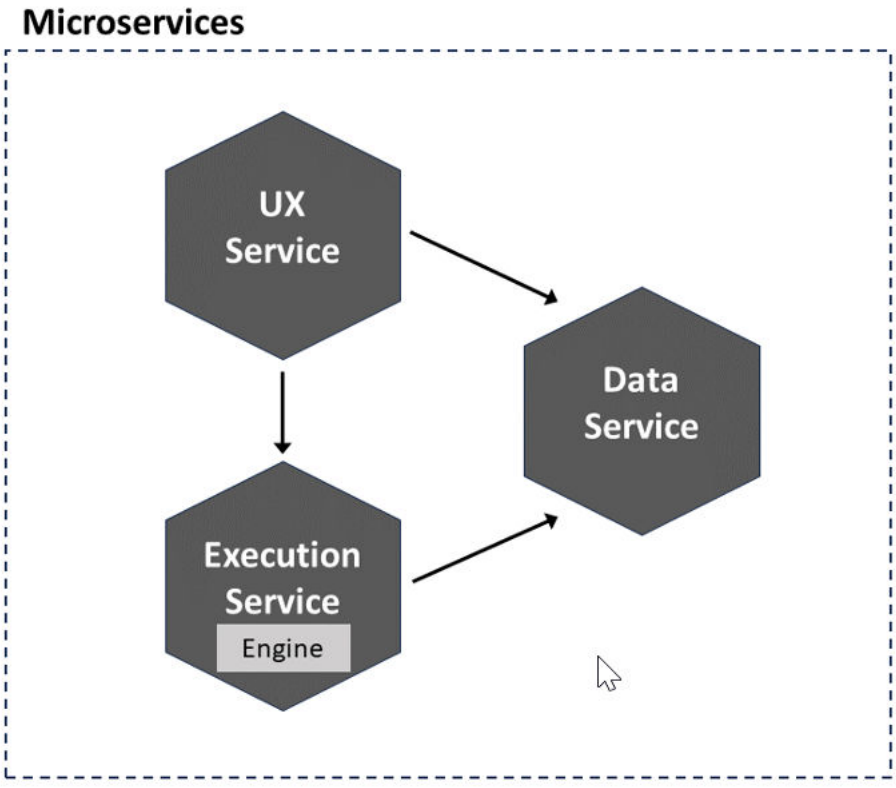
Microservice Architecture

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 plug-in 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 BDD validation operators used to compare the contents of two arrays. Typically, one array is returned by the system under test in a JSON or XML payload, while the other is stored in a STAP variable. Each array operator evaluates the comparison between the two arrays and returns a result of passed or failed. This allows you to validate whether the response data matches the expected array exactly, partially matches it, or contains specific expected elements.

The array operators are:

- General Array Operators
- Array Operators for Quoted Strings
- Array Utility Functions

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", ""]
```

Array Utility Functions

The **ARRAY_UNIQUE** can derive a new array that contains only unique values from an input array. Any duplicate values present in the source array are removed.

This function is useful when response data may contain duplicate entries but validation must be performed against a distinct set of elements.

If you set the following data:

```
Save:
| $ARRAY{products3} | "5G Basic Data Service" |
| $ARRAY{products3} | "5G Lite Data Service" |
| $ARRAY{productListWithDup} | "5G Basic Data Service" |
| $ARRAY{productListWithDup} | "5G Basic Data Service" |
| $ARRAY{productListWithDup} | "5G Lite Data Service" |
| productListUniq | %ARRAY_UNIQUE($ARRAY{productListWithDup}) |
```

In this example:

- `productListWithDup` contains duplicate product values.
- `productListUniq` stores the result of the **ARRAY_UNIQUE** function, which removes duplicate entries and retains only unique values.

When you run a mock action and read the task data:

```
| $request | $arraydata |
```

You can validate the response using the following validations:

```
Validate:
| services[0].products[*].name |
%ARRAY_COMPARE_IGNORE_ORDER_QUOTES($ARRAY{products3}) |
| services[0].products[*].name |
%ARRAY_COMPARE_IGNORE_ORDER_QUOTES($ARRAY{productListUniq}) |
```

Validation will succeed if the `services[0].products[*].name` array in the response contains the same unique product values as `productListUniq`, regardless of order and quotation differences.

This ensures that duplicate values in the response do not cause validation failures when only the presence of unique elements is required.

Logical Operators

Logical operators help you define and evaluate multiple conditions in BDD scenarios. Use these operators to create flexible, condition-based expressions.

Using %EVAL_CONDITION()

The `%EVAL_CONDITION()` function evaluates logical expressions using operators, such as AND, OR, and NOT. Use this function in the validation step to handle complex conditions.

Syntax

```
%EVAL_CONDITION((condition1) OPERATOR (condition2))
```

You can use the following operators:

- AND: Returns true if both conditions are true.
- OR: Returns true if at least one condition is true.
- NOT: Returns true if the condition is false.

Examples:

- AND Operator
  Validates if both conditions are satisfied.

  ```
  | age | %EVAL_CONDITION((>10) AND (<40)) |
  ```

- OR Operator
  Validates if either condition is satisfied.

  ```
  | category | %EVAL_CONDITION((Platinum) OR (Gold)) |
  ```

- NOT Operator
  Validates if the condition is not satisfied.

  ```
  | category | %EVAL_CONDITION(NOT(Regular)) |
  ```

- Combining Operators
  Validates using a combination of AND, OR, and NOT.

  When combining operators (AND, OR, NOT), it is important to understand their order of precedence:

  – NOT is evaluated first.
  – AND is evaluated next.
  – OR is evaluated last.

  Parentheses `()` can be used to group conditions and explicitly control the evaluation order.

  ```
  | category | %EVAL_CONDITION(((Platinum) OR (Gold)) AND (NOT(Regular))) |
  ```

Using %IF_ELSE()

Use `` `%IF_ELSE()` `` during the save operation. This function saves one of two values based on a specified condition.

Syntax:

```
`%IF_ELSE(condition, valueIfTrue, valueIfFalse)`
```

Example:

Save `_subscription.id` as `id` if `status` is `Active`; otherwise, save `error`.

```
| _subscription.id | %IF_ELSE(status==Active, id, error) |
```

Example Scenario:

The following sample demonstrates using these logical operators in a BDD workflow for subscription management.

```
Case: Operators

When set variable, for default values
Save:
| subscriber.name | John Doe |
| subscriber.category    | Platinum    |
| subscriber.type    | Residential    |
| subscriber.age    | 30 |
| subscriber.address    | 2685 California Street |
| subscriber.state    | CA |
| subscriber.city    | Mountain View |
| subscriber.country    | USA    |
| subscriber.code    | 94040    |
| subscriber.emailId    | john.doe@domain.com |
| subscriber.mobile    | 9876543211 |
| plan.name | 5GBasic |
| payment.mode | Cash |
| bill.expectedTotalAmount | 3 |
| payment.mode | Cash |
| payment.amount | 3 |
| payment.paidAt | KIOSK |
| bill.totalTax | 0 |

When create a new subscription, in the billing system
Data:
| name | ${subscriber.name} |
|    category | ${subscriber.category}    |
|    type    | ${subscriber.type}    |
|    age    |    ${subscriber.age} |
|    address    | ${subscriber.address}    |
|    state    | ${subscriber.state}    |
|    city    | ${subscriber.city} |
|    country    | ${subscriber.country}    |
|    code    | ${subscriber.code}    |
|    emailId    | ${subscriber.emailId} |
|    mobile    | ${subscriber.mobile} |
Validate:
| $status | %EVAL_CONDITION((201) OR (200)) |
| category | %EVAL_CONDITION((Platinum) OR (Gold)) |
| age | %EVAL_CONDITION((>10) AND (<40)) |
| category | %EVAL_CONDITION(((Platinum) OR (Gold)) AND (NOT(Regular)) ) |

Save:
| _subscription.id   | %IF_ELSE(status==Active,id,error) |
| accStatus | status |

Then read subscription
RepeatUntil:
| ${accStatus} | %EVAL_CONDITION((Active) OR (Ready)) |
| $startAfter | 1 |
| $interval  | 1 |
```

```
| $endAfter | 5 |
Data:
| id | ${_subscription.id} |
Validate:
| $status | 200   |
Save:
| accStatus | status |
```

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 (Scenario Execution Storage)

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.
- If global context loading is enabled, variables can be added to the context by updating the **global.ctx** file located in **globalcontext.home** (in the persistent volume).

Variable Lifecycle

- **Local variable**: Available only for the duration of a scenario.
- **Global variables**: 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   |
```

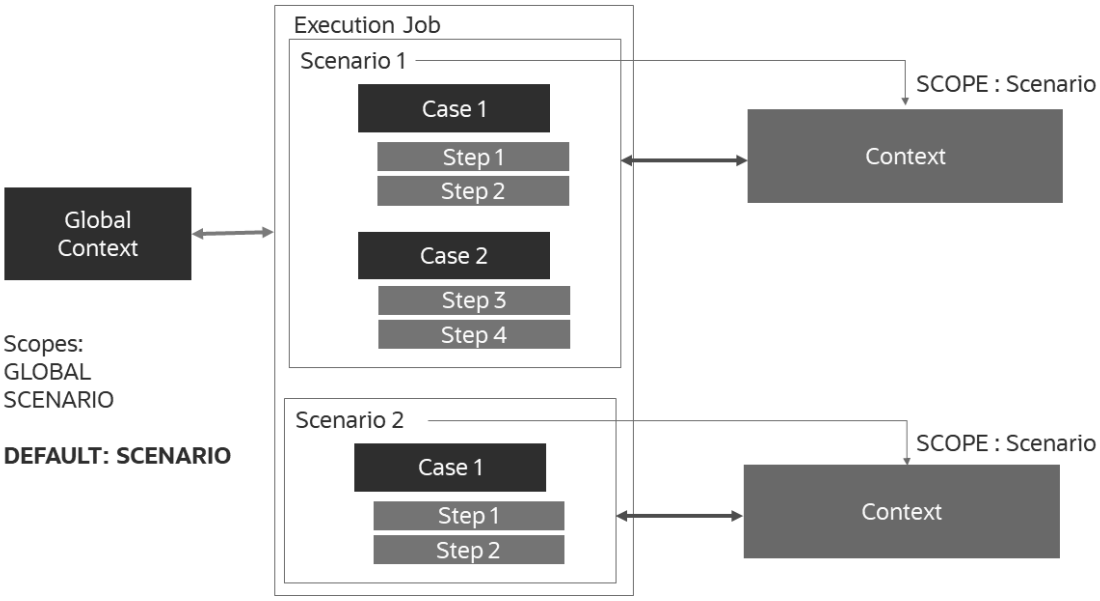
projectId which is prefixed with _, is designated as Global variable and the context stores this variable value from the definition until the job execution completes ie., **_projectId** can be used in any scenario/case/step after its definition.

> **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 "Setting Up Context".

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

Figure 4-1 Loading Context Configurations



Environment Variables (Externalized Configuration Inputs)

Environment variables are values defined in the environment file and injected into scenarios at runtime. They are typically used for data/configuration that should change by environment (for example, dev/test/prod) without changing scenario files.

- Define them in the environment file using: **variable.<name>=<value>**
- Access them in scenario steps using: **${ENV_<name>}**

Scenario File

```
Case: Create a customer profile and view details

# "add category" is bdd defined in action file - this block of code is used
to show post request working
When add category, for posting customer details
Data:
| name | John Doe |
```

```
| category | Platinum |
Validate:
| $status | 201 |
| category | ${ENV_category} |
Save:
| name | name |
| category | category |
```

Environment File

```
# Environment name
name=mockserver
type=REST
#REST Configuration
#Hostname
hostname=http://localhost:9999
#Base URL
url=http://localhost:9999

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

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

# Environment Variables
variable.category=Platinum

# Retry Enabled since 1.26.1.0.0
retryCount=3
```

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.plans** 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 an 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
Separate function arguments with a comma (,). If an argument itself contains a comma, or if a data variable value includes a comma, use the escape function **%{COMMA}** to escape it. Only escape commas within the values provided to the function—commas in context values or JSON property values are handled automatically and do not require manual escaping.

Other characters, such as backslash (/), single quote ('), and double quote (''), are escaped internally, so no additional action is needed for them.

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' |


# comma is present in the function arguments
Then get mock response, processing user subscription and notifications
Data:
| firstPlan | %PATTERN_MATCHER(${subscriptions},'(.*?)',0) |

# here, the second argument to pattern matcher function contains comma :
basic%{,}(.*?)
| secondPlan | %PATTERN_MATCHER(${subscriptions},'basic%{COMMA}(.*?)',0) |

Validate:
```

```
| $status | 200 |
| firstPlan |  'premium,active plan' |
| secondPlan | 'basic,active plan on wednesday' |
```

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 retrieve 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 the 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.

- Character: ,

- Description: Comma
- Escape: %{COMMA}

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 +,-,*,/allowing users to efficiently process calculations in reverse Polish (postfix) notation. This simplifies parsing, eliminates the need for parentheses or operator precedence rules, and ensures speedy evaluation without third-party library dependencies. Additionally, the Eval function is enabled in the Validate section, allowing users to leverage expression evaluation directly when performing validation tasks within STAP workflows.

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 an 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 an 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 transform 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) |
# with backslash format
| dateTime | %TRANSFORM(2024-05-07 10:10:10,YYYY-MM-dd HH:mm:ss,MM\/dd\/yyyy
HH:mm:ss) |
```

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 for Oracle Communications Solution Test Automation Platform (STAP).

Topics in this chapter:

Overview

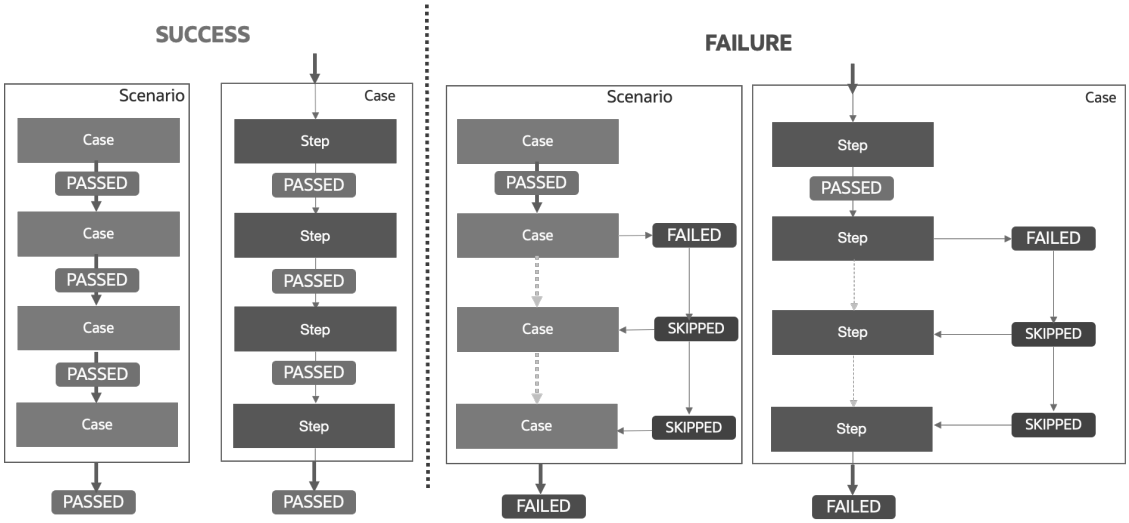
You can use control structures such as **if**, **for**, and **while** for steps in the Behavior-Driven Development (BDD) language. They are the building blocks within each test case and determine the flow of execution for each step based on specific conditions. Steps dictate the flow within test cases,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

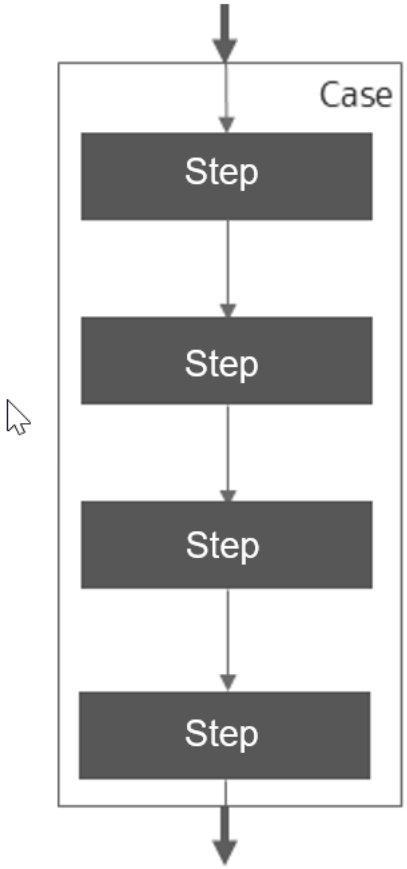
- Static Action (Default)
- Controlled Step

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

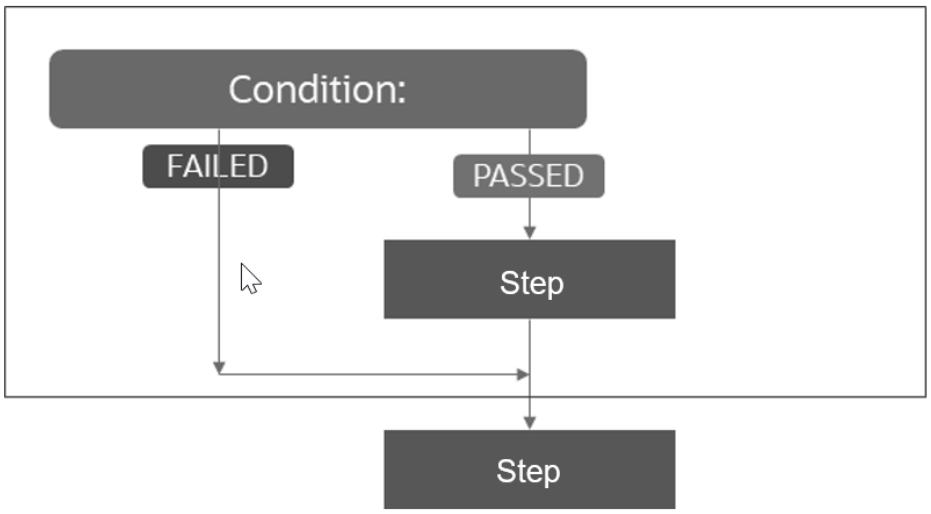
- Condition: Conditional Execution (IF)
  - Perform action when the condition is PASSED.
- repeatTimes (FOR)
  - Repeat number of times.
- repeatUntil (UNTIL)
  - Repeat until the condition is PASSED.
- repeatWhile (WHILE)
  - Repeat while the condition is true.

Conditional Execution

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

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

Figure 6-3 Conditional Execution



For example,

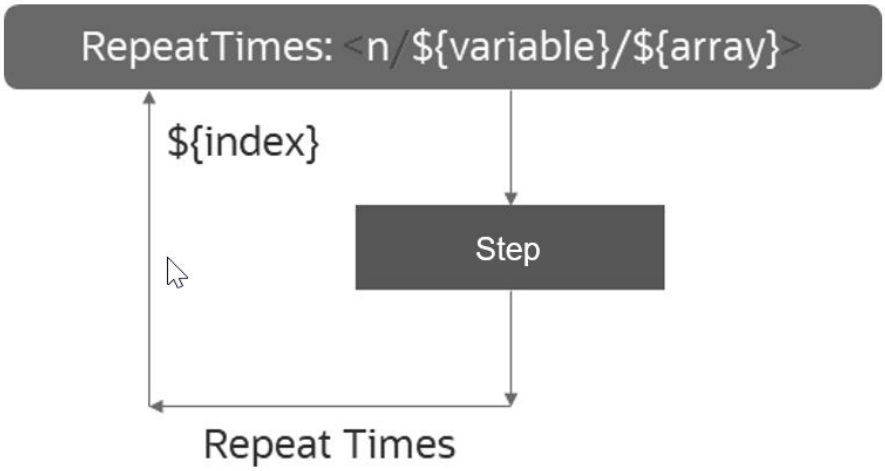
```
# This block of code checks if the category is Platinum, if yes, then it
changes to Gold through the action file request
When change category, for changing customer category
Condition:
| ${category} | Platinum |
Validate:
| $status | 201 |
| category | Gold |
Save:
| name | name |
| category | category |
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 the outcome of each action. If any iteration fails, the step is marked as failed but continues to run.

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

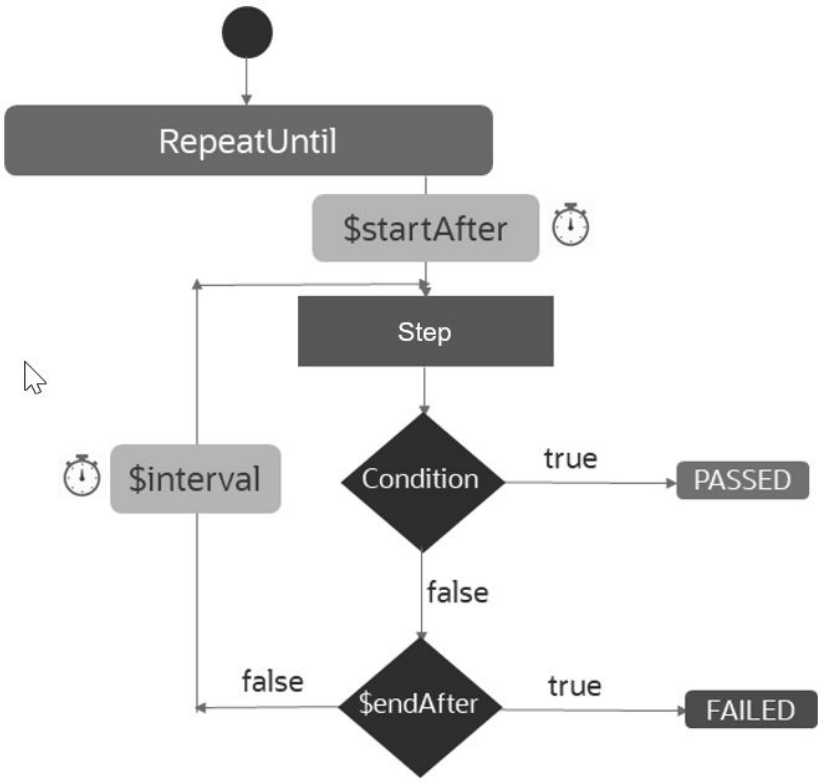
- Repeatedly perform the action until the given condition is true.
- At least one Condition is mandatory. (?)
- $breakOnFailure : YES breaks the loop on action validation failure. Default: NO.

```
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.

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

Figure 6-6 Repeat While 1st Iteration

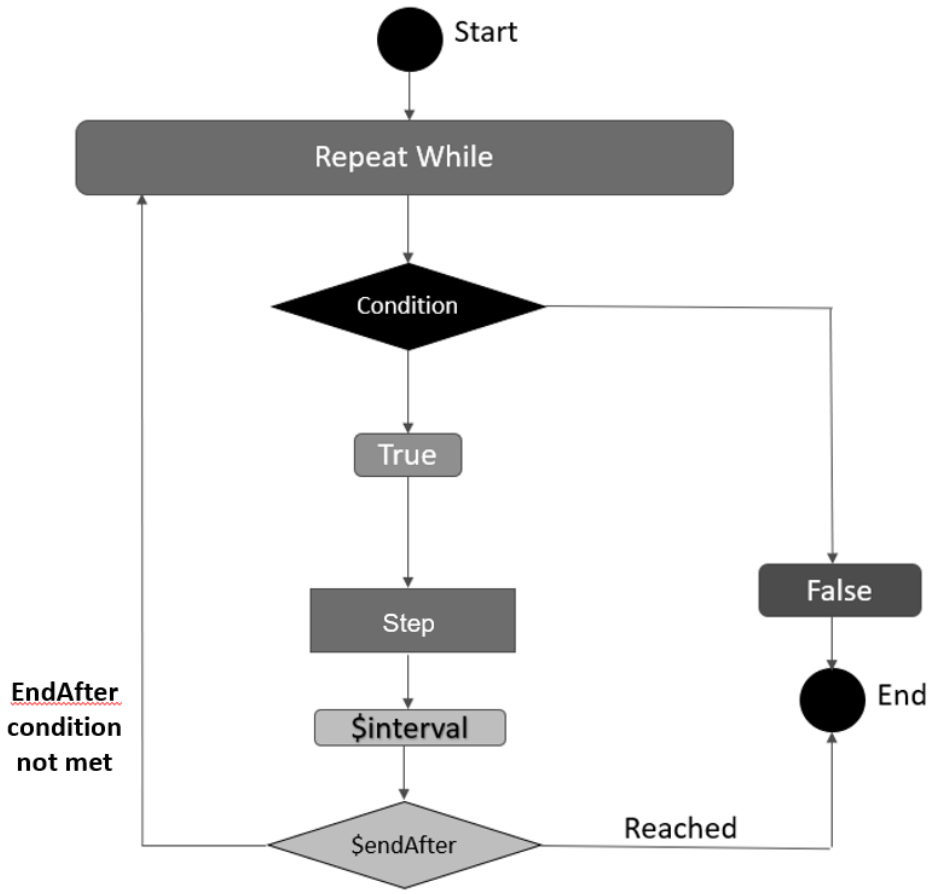
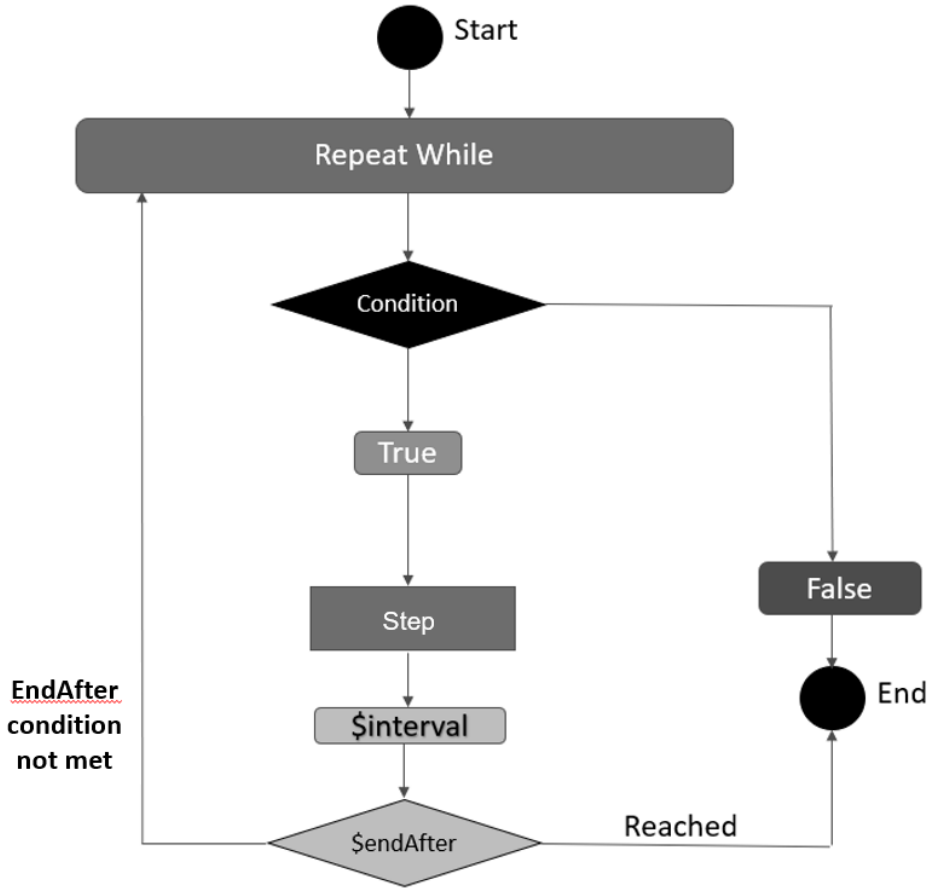


Figure 6-7 shows the flow of other iterations.

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' |
```

Repeat Case

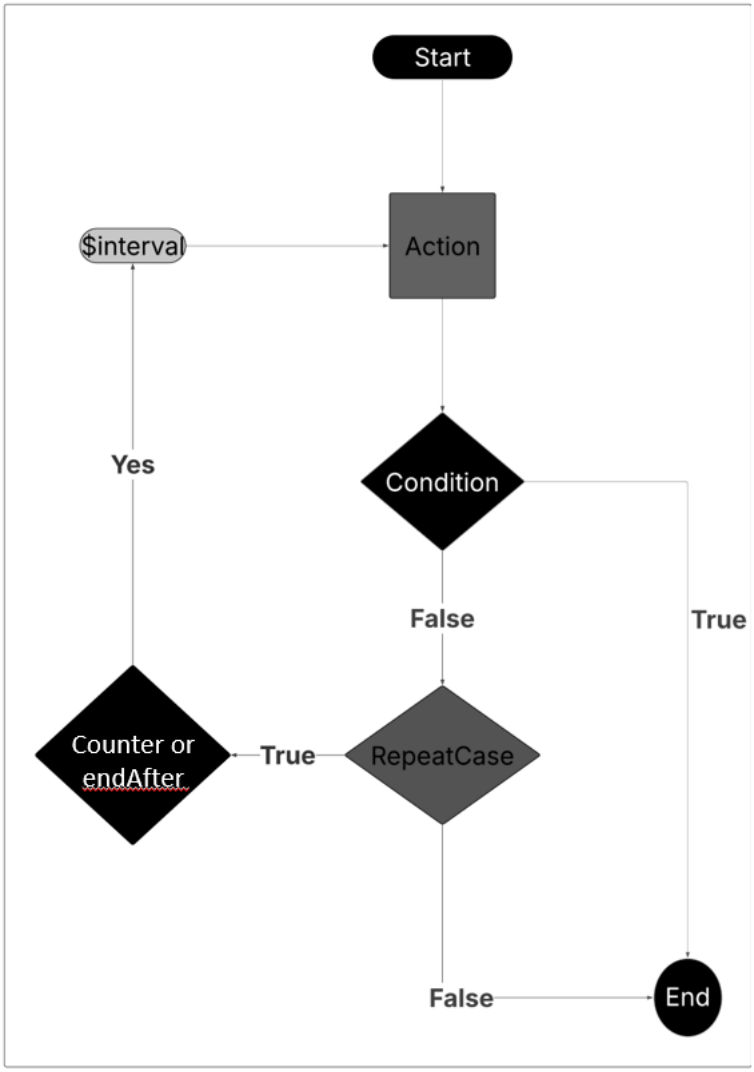
- Repeatedly run the case until the validation passes
- Ensure that at least one condition is met.

For example,

```
$endAfter : Optional. Break after the completion of this time duration.
$interval: Optional. interval duration to run the Step. By default, executes
continuously.
Specify interval duration in Seconds.
$times: Optional. Number of times the case can be repeated.
```

> **Note**
>
> If both **$times** and **$endAfter** are provided, the case will repeat up to **$times** within the specified duration.

Figure 6-8 Repeat Case




```
Case: Import and Publish Mobile Product Model
Given create an import job

Data:
| $formData.name | primaryFile |
| $formData.file | $FILE(DBE_PI2_Mobile_Model_PSP.json) |

Validate:
| $status | 201 |
| status | NOT_STARTED |
```

```
Save:
| importJobId | id |


Then verify the import job status by id
Data:
| id | ${importJobId}
Validate:
| ${jobstatus} | SUCCEEDED |
RepeatCase:
| $interval | 5 |
| $times | 2 |
| $endAfter | 6 |
```

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} |
```

Reusable Artifacts

STAP uses artifacts to organize and structure automation assets for reuse and scalability. Artifacts include cases, steps, and files. You define these artifacts independently, so they can be shared across different scenarios. STAP separates each core artifact definition from its usage. This approach increases reusability, simplifies maintenance, and improves test execution.

Types of Artifact Definitions

- **Cases**: Defined in case.casref files
- **Steps**: Defined in step.stepref files
- **Files**: Placed in common folder

Assign a unique reference ID to each case and step within these files. These reference IDs can then be used in scenarios (`.scenario file`) and cases (`.case file`) to build and test flows. This way, you reference the predefined artifacts instead of duplicating logic.

Benefits of Artifact-Based Design

- Promotes consistent behaviour across different workspaces
- Minimizes duplication of test logic
- Enables centralized updates and easier maintenance
- Optimizes execution by reusing tested artifacts

This artifact-driven structure enables modular, extensible, and efficient automation processes in STAP.

Using Reference Cases

If you want to test the same case across different scenarios, you can define the case once as a reference case and reuse the case file across scenarios.

Reference cases let you define a case once and reference it in other locations, instead of defining the same case file again. Updates to the original case appear everywhere it is used.

Reference cases are present under the **referenceCaseLibrary** folder with the file extension **.caseref**. To create the reference case library, run the following command in **config.properties**:

```
referenceCaseHome
referenceCases.home=${WORKSPACE}/referenceCaseLibrary
```

where:

- *referenceCaseHome* is the title of the folder where you want to store reference cases.
- *workspace* is your STAP workspace directory.

To run a scenario using the **.caseref** file, create and define it under the **referenceCaseLibrary**.

The following is an example for a reference case **.caseref** file:

```
Case: case title
Description: case description
tag: tag1, tag2
ReferenceCaseId: caseTitle

When...,to...
Data:

Then..., in the ...
Data:

Validate:
| $status | status code |

Save:
| _subscription.id   | id |
```

Then, create a **.case** file under the scenario folder that you want to run, and refer to the reference case ID in the **.case** file. The following is the syntax for the **.case** file:

```
Case: caseName
ReferenceCaseId: referenceCaseID
```

When you run a test case using this file as the case, it automatically fills in details from the **.caseref** file. The following example uses the **.caseref** to create a new subscriber in a billing system:

```
Case: Test Case
Description: Test description
tag: test

ReferenceCaseId: SetVariable

When set variable, for default values
Save:
| subscriber.name | John Doe |
| subscriber.category   | PLATINUM  |
| subscriber.type   | RESIDENTIAL   |
| subscriber.age    | 30 |
| subscriber.address    | 123 California Street |
| subscriber.state  | CA |
| subscriber.city   | Mountain View |
| subscriber.country    | USA   |
| subscriber.code   | 12345 |
| subscriber.emailId    | john.doe@oracle.com |
| subscriber.mobile | 1234567890 |

Then create a new subscription, in the billing system
Data:
| name | ${subscriber.name} |
| category | ${subscriber.category} |
| type | ${subscriber.type} |
| age | ${subscriber.age} |
| address | ${subscriber.address} |
| state | ${subscriber.state} |
| city | ${subscriber.city} |
```

```
| country | ${subscriber.country} |
| code | ${subscriber.code} |
| emailId | ${subscriber.emailId} |
| mobile | ${subscriber.mobile} |
Validate:
| $status | 201 |

Save:
| _subscription.id | id |
```

Using Reference Step

Reference steps enable you to define a step once and reuse it across multiple locations, promoting modular design and simplifying updates. Use reference steps when the same step logic is needed in different cases or scenarios.

Configuration

- Open your `config.properties` file.
- Add or update the following property to define the home directory for reference steps: `referenceSteps.home=${WORKSPACE}/referenceStepLibrary`
  - `referenceSteps.home`: Directory path for storing reference step definitions.
  - `${WORKSPACE}`: Path to your STAP workspace. Use directly in commands and configurations rather than hard coding the path.

Defining Reference Steps

- Define each reference step in a `.stepref file` under the directory set by `referenceSteps.home`.
- Assign a unique `ReferenceStepId` for each step.

```
When read subscription
ReferenceStepId: readSubscription
Data:
| id | ${_subscription.id} |
Validate:
| $status | 200 |
| name | ${subscriber.name} |
|    category | ${subscriber.category}    |
|    type    | ${subscriber.type}    |
|    age    |    ${subscriber.age} |
|    address    | ${subscriber.address}    |
|    state    | ${subscriber.state}    |
|    city    | ${subscriber.city} |
|    country    | ${subscriber.country}    |
|    code    | ${subscriber.code}    |
|    emailId    | ${subscriber.emailId} |
|    mobile    | ${subscriber.mobile} |
Save:
| name | name |
```

Usage

- In your `.case file`, refer to a predefined reference step by its `ReferenceStepId`.
- Prefix the usage with the relevant keyword (`Then, When, or And`) based on the case flow.

Example `.case file` usage:

```
Case: Create a new subscription in the billing system and validate the
creation
When create a new subscription, in the billing system
Data:
| name | ${subscriber.name} |
|    category | ${subscriber.category}    |
|    type    | ${subscriber.type}    |
|    age    |    ${subscriber.age} |
|    address    | ${subscriber.address}    |
|    state    | ${subscriber.state}    |
|    city    | ${subscriber.city} |
|    country    | ${subscriber.country}    |
|    code    | ${subscriber.code}    |
|    emailId    | ${subscriber.emailId} |
|    mobile    | ${subscriber.mobile} |
Validate:
| $status | 201 |

Save:
| _subscription.id   | id |


#Keyword(Then, When, And)
Then referenceStepId: readSubscription
Data:
| overrideParam | overrideValue |
| additionalParam | additionalValue |

Validate:
| overrideCondtion | overrideValue |
| additionalCondition | additionalValue |

Save:
| overrideSave | overrideValue |
| additionalSave | additionalValue |
```

Run-Time Output

When the test executes, the reference step is expanded. The system injects the referenced step's content at the appropriate keyword and merges any overridden or additional parameters. For example:

```
Case: Create a new subscription in the billing system and validate the
creation
When create a new subscription, in the billing system
Data:
| name | ${subscriber.name} |
|    category | ${subscriber.category}    |
|    type    | ${subscriber.type}    |
|    age    |    ${subscriber.age} |
|    address    | ${subscriber.address}    |
|    state    | ${subscriber.state}    |
|    city    | ${subscriber.city} |
|    country    | ${subscriber.country}    |
```

```
|    code    | ${subscriber.code}    |
|    emailId    | ${subscriber.emailId} |
|    mobile    | ${subscriber.mobile} |
Validate:
| $status | 201 |

Save:
| _subscription.id   | id |



#the key word will be picked from the usage step, not from the referenced one
i.e 'Then' keyword is picked up instead of 'When'
Then read subscription
ReferenceStepId: readSubscription
Data:
| id | ${_subscription.id} |
| overrideParam | overrideValue |
| additionalParam | additionalValue |
Validate:
| $status | 200 |
| name | ${subscriber.name} |
|    category | ${subscriber.category}    |
|    type    | ${subscriber.type}    |
|    age    |    ${subscriber.age} |
|    address    | ${subscriber.address}    |
|    state    | ${subscriber.state}    |
|    city    | ${subscriber.city} |
|    country    | ${subscriber.country}    |
|    code    | ${subscriber.code}    |
|    emailId    | ${subscriber.emailId} |
|    mobile    | ${subscriber.mobile} |
| overrideCondtion | overrideValue |
| additionalCondition | additionalValue |

Save:
| name | name |
| overrideSave | overrideValue |
| additionalSave | additionalValue |
```

Using ForEach in a Test Case

ForEach enables data-driven execution of a Case by running the same set of test steps multiple times, once per row of input data. This is useful when you want to validate the same workflow with different users, roles, categories, products, or any other test data variations without duplicating test logic.

With ForEach, you define the Case once, and provide a CSV file that contains multiple data sets. The framework iterates through the CSV rows and runs the Case repeatedly using the values from each row. If the data has a Comma (,) in the CSV file, then the data should be wrapped in double quotes.

For example, `CA, NewYork -> "CA, NewYork"`

When you add **ForEach: $filename.csv** to a Case:

- The system loads the CSV file from the scenario data folder.
- The header row defines the parameter or variable names.
- Each subsequent row is treated as a single iteration (one data set).
- The Case is executed once per row (n times, where n = number of data rows).

If the CSV contains two rows:

- Iteration 1 uses: **(user1, Platinum)**
- Iteration 2 uses: **(user2, Silver)**

Rules and Requirements

CSV location

The CSV file referenced in **ForEach** must be present under the scenario data folder.

Header names must match variables in the Case

The variable names used inside the Case must match the CSV headers exactly.

For example, if the CSV header is:

```
username,category
```

Then the Case should reference those parameters as:

- `${&username}`
- `${&category}`

This ensures correct mapping of values during each iteration.

One row = one execution

Each row in the CSV represents one complete execution of the Case, running all steps in the Case using that row's values.

Syntax

Case file (example: **customer.case**)

```
Case: Create a customer profile and view details
ForEach: $filename.csv  # expected to be present under scenario data folder

When add category, for posting customer details
Data:
| name     | ${&username} |
| category | ${&category} |
Validate:
| $status  | 201 |
Save:
| name     | name |
| category | category |

Then read user category, for posting customer details
Data:
| $urlSuffix | ${&username} |
Validate:
```

```
| $status    | 201 |
| category   | ${&category} |
```

CSV file format (example: **test.csv**)

```
username,category
user1,Platinum
user2,Silver
```

> **Note**
>
> - The first row is the **header** (parameter names).
> - Each following row is a **data set** used for one iteration.

Common Data Files

Common data files provide a centralized method to store and reuse data files (such as payloads, scripts, or configuration files) across different STAP scenarios. This reduces file duplication and simplifies file maintenance.

Configuration

- Open your config.properties file.
- Add or update the following property:

  ```
  referenceFiles.home=${WORKSPACE}/files/
  ```

- This configuration property defines the **base directory** where all common reference files are stored.
- Any file referenced using `$FILEREF()` will be **resolved relative to this directory**.
- This allows a single file to be shared and reused across multiple scenarios.

Syntax

- Reference a common data file using the following syntax:

  ```
  $FILEREF(<filename>)
  ```

  `<filename>`: The relative path to the file under `referenceFiles.home`.
- At runtime, `$FILEREF(<filename>)` is replaced with the original file content by the automation engine.

Usage Examples:

Upload a file in a test step:

```
Then execute SSH command, upload file
Data:
| $command | $sftp:UPLOAD_FILE |
| $environment | ssh-test |
| $source | $FILEREF(test/test-file.txt) |
```

```
 | $target | /home/ops/test-stap |
Validate:
 | $status | 0 |
```

Use a file as a request payload:

```
Data:
| $request | $FILEREF(request.json) |
```

This approach allows the same file to be reused in multiple scenarios, improving efficiency and consistency.

7

STAP Action Plug-ins

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

Topics in this chapter:

Introduction to STAP Action Plug-ins

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

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

The available automation plug-ins are:

- REST API
- SOAP API
- SSH/SFTP
- Process
- Seagull
- JMX
- Kafka
- UI Automation Plug-in
- URL Validator

REST Plug-in

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

The key features of the REST plug-in 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 assertions on HTTP status codes, headers, and body content.

The Rest plug-in 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 plug-in has its own environment connection configuration, and some plug-ins can have multiple environment configuration files for different products tested using various scenarios. For more information, see Setting Up The STAP Environment

You can combine REST and SOAP in a single environment, but other types of interfaces 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=************
```

Retry Count

The **retryCount** property determines the number of times the REST Plug-in automatically attempts to retry a failed REST API call when the failure results from a **server-side error (HTTP 5xx)**.

Example:

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

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

#Retry Count
retryCount=
```

Gateway types

The REST plug-in 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 the same but different resource endpoints need to be tested during execution, 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 the REST Plug-in

Action files define how API requests are constructed and executed within the REST plug-in.

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 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 parameter can be passed in the test step.

- To provide a custom value to a request header parameter, 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 an actual url can be added dynamically using $urlSuffix variable.

Actual url : http://localhost/todoApp/todo

For example,

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

URL used during execution will be http://localhost/todoApp/todo/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 the URL suffix.

- Actual url: http://localhost/todoApp/todo
- Required url: http://localhost/todoApp/todo/{{id}}/checkin
- Final url: http://localhost/todoApp/todo/MyURLID400/checkin

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 |
```

URL ID with URL suffix incase of multiple dynamic parameters

- Endpoint: http://localhost/todoApp/apilayer/v1/
- Required url: http://localhost/todoApp/apilayer/v1/subscriptions/{{identifier}}/bundles/{{basebundle}}
- Final url: http://localhost/todoApp/apilayer/v1/subscriptions/2025092405/bundles/NOMT-123

For example,

```
When set variable, to concat url
Save:| suffix | %CONCAT(${serviceIdentifier},/bundles/,${baseBundle} ) |
#serviceIdentifier and baseBundle are dynamic



When Get bundle API, Get bundle via API
Data:
| $urlSuffix | /$urlId/bundles/$urlAdd  |
| $urlId     | ${serviceIdentifier}  |
| $urlAdd     | ${baseBundle}  |
| $header_countryCode | NO  |
| $header_orderId | 123456  |
```

> **Note**
>
> In case you do not have set variable action, use the following action.

```
{
"path":"set/setVariable",
"name":"set variable",
"bdd":"set variable",
"description":"set variable values in Save",
"product":"system",
 "actionType":"REST",
"tags":["set","variabhle"],
"resource":"NO_RESOURCE",
"method":"GET",
 "expectedStatusCode":0
}
```

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 Plug-in

The Simple Object Access Protocol (SOAP) plug-in is used to automate the execution of SOAP API endpoints and to validate their responses. Automation plug-ins 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() |
```

Query Parameters

Query parameters in the SOAP plugin can be defined using $query variable in Data. Only one $query should be defined and it is sent as part of URL with `?$query` after processing any variable in the value.

Refer to this example where the url is appended with '`?version=1`' as the query parameter.

```
When soap mock action with query param,
Data:
| $query | version=1 |
# | $query | version=1&name= |  #for multiple query params
Validate:
| $status | 200 |
```

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 an 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
}
```

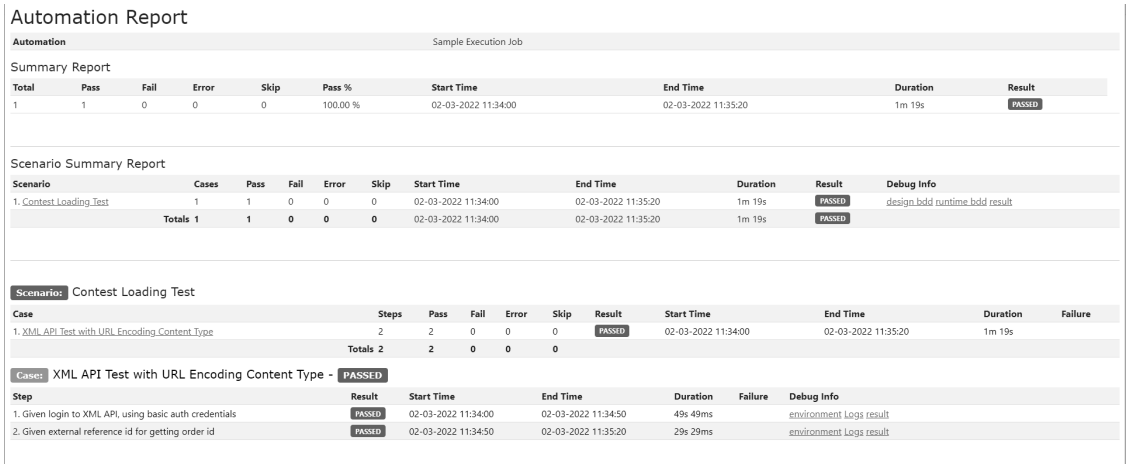
The following are the contents of properties file that contains an XML API:

```
#=============================================================================
=
# 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=********
#********************************************************************************
*
# 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 Plug-in

The Secure Shell (SSH) Plug-in 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 the SSH SFTP Plug-in:

- **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 the 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 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.

$endAfter: When an exit condition is present, it is mandatory to provide the end after time, to avoid an indefinite wait time. While checking for the exit condition in the SSH response, if it is not found even after the end after duration elapses, STAP forcefully closes the SSH channel. **$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 "the run SSH command" step.
> - More than one command can be passed in a single step, by separating the commands using semicolons(;).
> - Supported validations are:
>   - Exit code of the command using the validation property $status.
>   - %CONTAINS checks for any string that may be a part of the command response or error.
> - In response validation, a single string can be passed to the **%CONTAINS** operator.
> - the save variable with value '$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: 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 Plug-in

The STAP process plug-in is used to run the shell commands locally using java.lang.process.

Action

The command to be run using the process plug-in 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 th 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

A 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 | $FILE(Message.sh) |

When execute 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 the data variable's value overrides the default 'ParameterValue' . The final value of the parameter replaces %{ ParameterName : ParameterValue } in the command.

For example, in the following action.json, the 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, a 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 the exit code is not explicitly given in the Test Step (that is $status), then the expectedExitCode mentioned in the action.json is used to validate if the execution is successful or not.
2. The only Validation properties supported in Process plug-in 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:

The only Save property supported in Process plug-in 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 execute pwd command
```

```
Validate:
| $status | 0 |

#Multiple exit codes in validation
When execute pwd command, multple validation codes
Validate:
| $status | 0,1,2 |

#saving scripts paths
When set variable,
Save:
| messageScript | $FILE(Message.sh) |
| testScript | $FILE(processPluginTest.sh) |

#variables to be updated in action file's command
When execute 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 execute 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. The STAP Seagull plug-in 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

The 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 the seagull instance and should not be modified. Multiple seagull instances (that is, having different config files and dictionary files) can be created by reusing this same action and saving the instance with a different name using the **$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.jsons can be created.

The 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 a 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 the 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 an 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

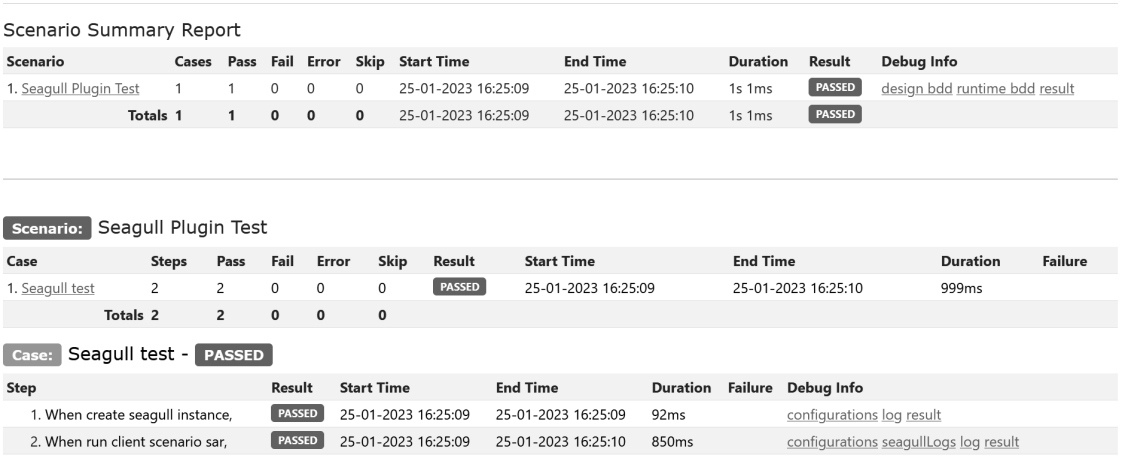
```
Case: Seagull client test

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 Plug-in Test Scenario Summary Report:

Figure 7-3 Seagull Plug-in Test Scenario 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

JMX

JMX plugins are used for monitoring and managing Java applications and their resources.

JMX Connection:

Supported Authorization types:

- Basic

- No Authorization

Basic Authorization

Example:

ece-jmx-environment.properties

```
name=Test-JMX
type=JMX

hostname=hostname
port=1234

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

authorization.type=basic

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

No Authorization

Example:

ece-jmx-environment.properties

```
name=Test-JMX
type=JMX

hostname=hostname
port=1234

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

#authorization.type=basic

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

Supported Actions:

- Get Attribute
- Set Attribute
- Set Attributes
- Get Bean Info
- Get Bean Config Info
- Invoke Operation Get
- Invoke Operation Set

Get Attribute

To fetch the value of the attribute of an Mbean on the JMX server provided in the Scenario file.

The beanName and the attributeName should be provided in the scenario file.

Action.json

get_attribute_value.action.json

```
{
"path":"CustomAction/brm-action",
"name":"get attribute",
"bdd":"get attribute",
"description":"get attribute",
"product":"ECE",
 "actionType":"JMX",
 "subType":"GET_ATTRIBUTE",
"tags":["custom","jmx"]
 }
```

Data needed:

Data

```
Data:
$beanName - beanName of the attribute
$attributeName - name of the attribute

Validate:
$data : fetched value to be validated against the expected value
```

Case:

Example

```
When get attribute, display the value of the attribute

Data:
| $beanName | Users:type=UserDatabase,database=UserDatabase |
| $attributeName | pathname |
Validate:
| $status | SUCCESS    |
| $data | %CONTAINS(testing)|
```

Set Attribute

To update value of the attribute of a mBean provided in the scenario file.

The beanName, attributeName, attributeValue and the attribute datatype should be provided in the scenario file.

The supported values for attributeType are: string, long, integer, and boolean

set_attribute_value.action.json

```
{
"path":"CustomAction/brm-action",
"name":"set attribute",
"bdd":"set attribute",
"description":"set attribute",
"product":"ECE",
 "actionType":"JMX",
 "subType":"SET_ATTRIBUTE",
"tags":["custom","jmx"]

 }
```

Data

```
Data:
$beanName - beanName of the attribute
$attributeName - name of the attribute
$attributeValue - the new value to be updated
$attributeType - the datatype of the attribute. (string | long | integer |
boolean)

Validate:
$status: successful update of the attribute value
```

Example

```
When set attribute, set the value of the attribute

Data:
| $beanName | Users:type=UserDatabase,database=UserDatabase |
| $attributeName | pathname |
| $attributeValue | testing |
| $attributeType | string |
Validate:
| $status | SUCCESS    |
```

Set Attributes

To update multiple attributes under a single mBean.

The attributes and the values are specified in a separate json file.

set_multiple_attributes_value.action.json

```
{
"path":"JMX",
"name":"set attributes",
"bdd":"set attributes",
"description":"set attributes",
"product":"ECE",
 "actionType":"JMX",
 "subType":"SET_ATTRIBUTES",
"tags":["custom","jmx"],
"attributesType": "FILE"

 }
```

Data

```
Data:
$beanName - beanName of the attribute
$request - filename with the attributes data JSON
```

For example,

```
When set attributes, set the value of the attributes

Data:

| $beanName | Users:type=UserDatabase,database=UserDatabase |
| $request | $dynamic |
Validate:
| $status | SUCCESS    |
```

Attributes data JSON:

JMX.dynamic.request.json

```
{
"attributes": [
        {
            "name": "pathname",
            "value": "testing",
            "attributeType":"string"
        },
        {
            "name": "pathname",
            "value": "testing",
            "attributeType":"string"
        },
        {
            "name": "pathname",
            "value": "testing",
            "attributeType":"string"
```

```
        }
    ]
}
```

Get Bean Info

To display the mBeanInfo for the mBean name mentioned in the scenario file

The beanName should be provided in the scenario file.

displayMbean.action.json

```
{
"path":"CustomAction/brm-action",
"name":"display mBean Info",
"bdd":"display mBean Info",
"description":"display mBean Info",
"product":"ECE",
 "actionType":"JMX",
 "subType":"GET_BEAN_INFO",
"tags":["custom","jmx"]

 }
```

Data

```
Data:
$beanName - beanName of the attribute

Validate:
$status: successful display of the mBeanInfo

Save:
beanInfo:  Bean Info of the mBean
```

For example,

Step

```
When display mBean Info, display the bean info
Data:
| $beanName | Users:type=UserDatabase,database=UserDatabase |
Validate:
| $status | SUCCESS    |
Save:
| beanInfo | $data|
```

Get Bean Config Info

To display the Config info for the mBean name mentioned in the scenario file

The beanName should be provided in the scenario file.

get_config.action.json

```
{
"path":"CustomAction/brm-action",
"name":"get config",
"bdd":"get config",
"description":"get configuration",
"product":"ECE",
 "actionType":"JMX",
 "subType":"GET_CONFIG",
"tags":["custom","jmx"]
 }
```

For example,

```
When get config, get the value of configuration

Data:

| $beanName | ${beanName} |

Validate:
| name | Users:type=UserDatabase,database=UserDatabase |
| children[0].name | UserDatabase,database=UserDatabase |
Save:
| beanName | name |
| child | children[0].name |
| descriptor | children[0].info[0] |
| mBeanInfo | children[0].info[1] |
| attribute | children[0].attributes[0].name |
| attributeInfo | children[0].attributes[0].info[0] |
```

Invoke Operation Get

To invoke JMX operations that return data from the JMX server.

The returned data can be saved and validated.

This case requires a bean name, an operation name, and a JSON containing the parameters.

invoke_get_operation.action.json

```
{
"path":"JMX",
"name":"invoke get operation",
"bdd":"invoke get operation",
"description":"invoke get operation",
"product":"ECE",
 "actionType":"JMX",
 "subType":"INVOKE_OPERATION_GET",
"tags":["custom","jmx"]

 }
```

For example,

```
Example
When invoke get operation, invokes the JMX operation

Data:
| $beanName | ${beanName} |
| $operationName | findGroup |
| $params | $dynamicGetParams |

Validate:
| $status | SUCCESS    |
| $data |
%CONTAINS(Users:type=Group,groupname="johndoe",database=UserDatabase)|

Save:
| groupName | data |
```

Parameter JSON:

The order of parameters should be as mentioned in the JMX API documentation for the operation.

JMX.dynamicGetParams.request.json

```
{

"params": [
        {
            "name": "groupname",
            "value": "johndoe",
            "attributeType":"string"
        }
    ]

}
```

Invoke Operation Set

To invoke JMX operations that sets attributes or performs operations on the JMX server. There is no data returned from the JMX server when this operation is invoked.

This case requires a bean name, an operation name, and a JSON containing the parameters.

invoke_set_operation.action.json

```
{
"path":"JMX",
"name":"invoke set operation",
"bdd":"invoke set operation",
"description":"invoke set operation",
"product":"ECE",
 "actionType":"JMX",
 "subType":"INVOKE_OPERATION_SET",
"tags":["custom","jmx"],
"paramType":"FILE",
```

```
"paramFile":"invoke_set_operation.param.json"

 }
```

Param.json

The order of parameters should be as mentioned in the JMX API documentation for the operation.

invoke_set_operation.param.json

```
{

"params": [
        {
            "name": "groupname",
            "value": "johndoe",
            "attributeType":"string"
        },
        {
            "name": "description",
            "value": "johndoe group",
            "attributeType":"string"
        }
    ]

}
```

For example,

```
When invoke set operation, invokes the JMX operation

Data:
| $beanName | ${beanName} |
| $operationName | createGroup |

Validate:
| $status | SUCCESS    |
```

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 plug-ins 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
```

Kafka Connection supports two types of authentications:

- Basic
- OAuth2

Refer to the following example for a Basic Authorization.

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

#------------------------------------------------------------------------
----------
# Bootstrap Servers
# List of comma separated boot strap servers
#------------------------------------------------------------------------
----------
servers=<servername>

#------------------------------------------------------------------------
----------
# Authorization
# Supported Basic Auth (Since 1.26.1.0.0)
#------------------------------------------------------------------------
```

```
----------
authorization = YES
authorization.type = basic

basic.username = <user-name>
basic.password = <password>
```

Refer to the following example for a Oauth2Authorization.

```
#========================================================================
=
# KAFKA Environment Configuration with IDCS OAuth2
# This follows the same pattern as brm-rest-environment.properties
#========================================================================
=
# Environment name
name=test
type=Kafka

#========================================================================
=
# Bootstrap Servers (SSL port)
# List of comma separated bootstrap servers
#========================================================================
=
servers=<server1, server2, ...>

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

#========================================================================
=
# authorization.type can be: oauth2 or basic
#========================================================================
=
authorization.type=oauth2

#========================================================================
=
# OAuth2 - IDCS Configuration (same properties as REST plugin)
#========================================================================
=
oauth2.grantType=client_credentials
oauth2.clientId=<clientid>
oauth2.clientSecret=<client_secret>
oauth2.tokenUrl=https://idcs-<tenantid>.identity.oraclecloud.com/oauth2/v1/
token
oauth2.scope=http://localhost:9092/dbe

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

```
=
# SSL Configuration
# ssl=YES will use SASL_SSL, ssl=NO will use SASL_PLAINTEXT
#================================================================================
=
ssl=YES
ssl.truststore.location=C:/<ssltruststore_path>
ssl.truststore.password=*******
ssl.endpoint.identification.algorithm=

#================================================================================
=
# Optional: For password_credentials grant type (uncomment if needed)
#================================================================================
=
#oauth2.grantType=password_credentials
#oauth2.authorization.username=your-username
#oauth2.authorization.password=your-password

#================================================================================
=
# Optional: Connection Timeouts
#================================================================================
=
maxIdle=10000
requestTimeout=5000
```

Action

The following table lists the action properties:

Table 7-2 Action Properties

Property	Mandatory	Description	Default Value	Allowed Values
actionType	Yes	Kafka Plug-in Type	Kafka	Kafka
subType	Yes	Kafka action sub types	N/A	GET_TOPIC_LATEST_MESSAGE, PING_SERVER, SEND_TOPIC_MESSAGE, GET_MESSAGE_COUNT, DELETE_TOPIC_MESSAGES
topic	Yes	Topic name	N/A	N/A
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",
 "groupId":"test-consumer-group",
 "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",
 "groupId":"test-consumer-group",
 "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",
 "groupId":"test-consumer-group",
 "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(${name})  |  user            |
CONDITION: SUCCESS  |  PASSED  |
 |  address.residenceNo  |  100001               |  100001          |
100001              |  PASSED  |
Save:
#|  Property  |  Value                |  Runtime Value          |
 |  id        |  id                   |  532457234857234879594  |
 |  name      |  %SUBSTRING($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(${name},1)  |  SER                  |  SER             |
SER                 |  PASSED  |
Save:
#|  Property  |  Value                  |  Runtime Value          |
 |  id        |  //id                   |  532457234857234879594  |
 |  name      |  %SUBSTRING($//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 |
```

Runtime Scenario with oauth cases:

OAuth2Test.case

```
Case: OAuth2 Authentication Test
Tags: OAuth2, Kafka, IDCS

When send topic message, with OAuth2 authentication
Data:
| $message | {"test": "oauth2 integration", "timestamp": "2026-02-03"} |
| $messageType | JSON |
Validate:
| $status | SUCCESS |

When get topic last message, with OAuth2
Data:
| $messageType | JSON |
Validate:
| $status | SUCCESS |
| test | oauth2 integration |
```

UI Automation Plug-in

The STAP UI Automation Plug-in delivers reliable, low-code browser automation within the STAP ecosystem, streamlining UI testing with intelligent waits, self-healing selectors, and a consistent action interface.

The STAP UI Automation Plug-in extends the STAP Action Plug-in Framework to automate and validate interactions with web user interfaces. It abstracts Selenium-based browser automation behind a consistent action interface to enable low-code, browser-independent, and maintainable tests. To reduce flakiness and improve robustness, the plug-in includes intelligent selectors, dynamic wait strategies, automatic retry logic, DOM stabilization detection, and self-healing selectors. Future releases will add AI-assisted element identification, wait management, and failure recovery.

Key capabilities:

- Low-code, browser-agnostic UI automation integrated with the STAP execution lifecycle.
- Intelligent waits, retries, and self-healing selectors to stabilize tests across UI changes.
- Consistent interface across automation types (REST, SOAP, SSH, UI).

The next section explains UI Actions, which the UI Automation Plug-in uses to perform specific tasks and build complete automated UI workflows.

Supported UI Actions

Supported UI actions are modular, scriptable commands in the UI Plug-in that automate interactions with web application interfaces. Each of the supported UI action represents a single user operation, such as clicking a button, entering text in an input field, selecting a value from a drop-down list, validating the page title or URL, or uploading a file. These actions convert complex Selenium WebDriver logic into concise, reusable steps.

In the UI plug-in, you define the supported UI actions in test scripts or BDD scenarios so you can build end-to-end UI workflows in a clear, data-driven format. The plug-in converts each supported UI Action into the required automation code, resolves dynamic variables, locators, and data, and then runs the action in the browser. This process enables fast, reliable, and maintainable automation of functional UI testing across different web applications.

The supported UI Actions are grouped into the following categories:

- Browser Actions
- Form Actions
- Save Actions
- Validation Actions

Browser Actions

Browser actions control browser navigation and behavior during UI automation in STAP. The browser action in the UI Plug-in provides a single interface to manage navigation, handle dynamic or static URLs, and perform common browser operations.

It supports expression-based navigation, fixed or mapped paths, and standard browser controls such as back, forward, refresh, alerts, and wait operations.

Table 7-3 Browser Actions

Action	Description	Arguments	Example
open	The `$open` action enables dynamic and static navigation to web URLs.	`uri`	**Dynamic:** Data: `| $open \| @{%CONCAT($uri,/dashboard/)} |` `| $open \| @{%CONCAT($apiBase,endpoint,/${userId})} |` `| $open | @uri |` `| $open | /manualEntry/ |` Ensures dynamic navigation by concatenating base URLs, paths, and parameters and appending them to the base URL defined in the environment. **Fixed:** Data: `| $open | @uri |` Looks up the URI in the `locatorMap` and joins it with the base URL defined in the environment. **Literal/Static:** Data: `| $open | /manualEntry/ |`
back	Goes back to the previous webpage from the current page.	`uri`	Data: `| $back | @uri |`
forward	Goes forward to the next webpage from the current page.	`uri`	Data: `| $forward | @uri |`
refresh	Refreshes the current webpage.	`uri`	Data: `| $refresh | @uri |`
alert	Accepts an alert.	`mockLocator`	Data: `| $alert` `| $mockLocator |`
waitFor	Waits for a specified time in milliseconds.	`mockLocator`, `time` (ms)	Data: `| $waitFor` `| $mockLocator,7000 |`

Form Actions

Form actions perform interactions with form fields and page elements during UI automation. The form actions toolkit in the UI Plug-in supports standard input operations as well as advanced interactions such as scrolling, keyboard actions, and file uploads. It also supports dynamic parameters and provides diagnostics to help ensure reliable execution.

Table 7-4 Form Actions

Action	Description	Arguments	Example	Notes
input	Enters a value in an input field.	Xpath, Input string	`Data:` `| $input | $username, user1 |`	`username` is the `Xpath` variable and `user1` is the input string.
copyAndPasteInput	Copies a value from a source input field and pastes it into a destination input field.	Source Xpath, Destination Xpath	`Data:` `| $copyAndPasteInput | $ordernum, //input[@name='Service_Id'] |`	`ordernum` is the input `XPath` variable. The value is pasted into the destination `XPath` with name `Service_Id`.
selectFromDropdown	Selects a particular field from a drop-down list.	Input Xpath, value to select, keys	`Data:` `| $selectFromDropdown | $dropdown, Residential |`	`dropdown` is the input `Xpath` variable and `Residential` is the value to select from the drop-down list.
click	Clicks a web element.	Xpath of the web element	`Data:` `| $click | $submit |`	`submit` is the `Xpath` variable of the web element to be clicked.
doubleClick	Performs a double-click on a specified element (button, row, div, and so on).	Xpath of the web element	`Data:` `| $doubleClick | $doubleClickBox |`	`doubleClickBox` is the `Xpath` variable of the web element to be clicked.
save	Simulates the Ctrl+S key combination and waits for the operation to complete.	`uri`	`Data:` `| $save | @uri |`	`$save` is the method and `uri` is the URL.
pressKeysSequentially	Presses keyboard keys one after another in sequence.	`mockLocator`, keys combined using '+'.	`Data:` `| $pressKeysSequentially | $mockLocator, ARROW_DOWN+ENTER |`	`$mockLocator` is a placeholder locator and `ARROW_DOWN+ENTER` are the keys which have to be pressed one by one.

Table 7-4 (Cont.) Form Actions

Action	Description	Arguments	Example	Notes
pressKeysTogether	Presses a combination of keyboard keys at the same time.	`mockLocator`, keys combined using '+'.	`Data:` `| $pressKeysTogether | $mockLocator, ARROW_DOWN+ENTER |`	`$mockLocator` is a placeholder locator and `ARROW_DOWN+ENTER` are the keys which has to be pressed together.
scrollVertical	Scrolls the page vertically by pixels or to a position (top or bottom).	Direct value such as `600`, or `top`, `start`, `end`, `bottom`.	`Data:` `| $scrollVertical | start |` `| $scrollVertical | 700 |`	**Valid parameters:** • Number (positive/ negative): pixels to scroll (for example, `600` or `-500`). • `top` or `start` – scroll to the top of the page. • `end` or `bottom` – scroll to the bottom of the page.
scrollHorizontal	Scrolls the page horizontally (X axis). For example, wide tables or sections.	Direct value such as `1200`. Specials such as `start/left` (X=0); `end/right` to far right.	`Data:` `| $scrollHorizontal | 700 |` `| $scrollHorizontal | start |` `| $scrollHorizontal | end |`	**Valid parameters:** • Number (pixels): how far to scroll right. • `start/left` • `end/right`
scrollToElement	Scrolls the page to bring a specific UI element into view and optionally aligns it to the top or bottom of the viewport.	Option1: Xpath only Option2: Xpath, Input string with alignment.	`Data:` `| $scrollToElement | $target2 |` `| $scrollToElement | $target2,top |` `| $scrollToElement | $target3,bottom |`	**Valid Parameters:** `target#` is the `Xpath` variable of the web element. **Variable:** Only `Xpath` or with alignment variable `${elementKey},bottom`.

Table 7-4 (Cont.) Form Actions

Action	Description	Arguments	Example	Notes
infiniteScroll	Scrolls the page repeatedly to simulate infinite scrolling or page loading.	input value, input value, input value	`Data:` `| $infiniteScroll | 7,400,400 | // repeat 7 times, 400px each, 400ms delay`	`| $infiniteScroll | ${scrollRepeat},${scrollStep},${scrollDelay} |` **Direct format:** `repeat,stepPx,delayMs` **Valid parameters:** • Number of times to scroll (Integer, for example 7). • Pixels to scroll in each step (for example `400`). • Delay in milliseconds between scrolls (for example `400`).
uploadFile	Uploads a file to a specified input field (for example, `<input type="file">`).	• Variable representing the XPath (or CSS selector) for the file input element, typically the Browse button. • Absolute path of the file to be uploaded.	`Data:` `| $uploadFile | $fileInputId, /path/to/file.txt |`	• The framework automatically locates the element and performs the upload using Selenium. • This action supports automation of file attachments and is compatible with BDD scenarios and data-driven tests.

Save Actions

The UI Plug-in provides save actions that help manage test state and retrieve values from the UI. These patterns support both explicit value assignment and value capture from UI elements, which improves test flexibility, reduces failures caused by changes, and supports reliable validation.

Context Save

Use the Context Save action to store values directly in the test context so that subsequent steps can reference them during execution.

Table 7-5 Save Actions

Action	Description	Arguments	Example	Notes
save	Saves a value in the context as a key-value pair.	`Expected value` when setting the variable.	`When set variable:` `Save:` `| userNameInput | tesuser |`	Saves the value `tesuser` in the context under the key `userNameInput`. You can reference this value in subsequent steps.

Save from UI

Use the Save from UI action to capture a value directly from a UI element and store it in the test context for reuse in later steps.

Table 7-6 Save from UI

Action	Description	Arguments	Example	Notes
save	Captures a value from a UI element and stores it in a specified variable using the element's locator (for example, XPath).	• **Locator:** The XPath or identifier for the UI element from which the value is to be saved. • **Identifier:** `$VALUE`.	`Save:` `| ProductName | xpath://*[@id="abc"], $VALUE |` `or` `Save:` `| ProductName | $productName, $VALUE |`	• Saves the value from the `$productName` variable into `ProductName`. • `$VALUE` is an identifier that the UI Plug-in uses to extract a value from a UI element or a source variable. • Use this action to preserve UI values for validation or for use in later steps.

Validation Actions

Validation actions verify key browser and element states during UI tests to ensure the application behaves as expected and meets user requirements.

Table 7-7 Validation Actions

Action	Description	Arguments	Example	Notes
url	Validates the expected URL of the webpage against the actual URL.	Expected URL	`Validate:` `| $url | https://localhost:3000 |`	This action validates the current URL of the webpage.
title	Validates the expected title of the webpage against the actual title.	Expected title	`Validate:` `| $title | Communications Website |`	This action validates the title of the webpage.
visible	Validates whether a particular web element is visible.	Locator (Xpath) of the web element	`Validate:` `| $visible | $ordernum |`	`ordernum` is the input `XPath` variable.

UI Plug-in Testing

Use the UI Automation Plug-in to validate end-to-end user journeys, confirm UI behavior, and capture visual evidence as part of continuous testing. The plug-in operates as an action handler within the platform, parsing UI actions, resolving elements, interacting with the browser driver, and returning structured results to the core engine.

With the testing approach in mind, let's walk through the required setup and execution steps.

Prerequisites

- Installed browser and matching WebDriver (ChromeDriver or GeckoDriver (for Firefox) or msEdgeDriver).
- Access to the AUT, including network routes, credentials, and test data.
- STAP Engine and STAP platform - access and permissions.
- Java runtime (if required by your environment) with adequate heap for UI tests.
- File system access for WebDriver and download directories.
- Stable test URLs and dedicated test accounts.

Steps to Run UI Test Automation Using UI Plug-in

1. Configure the UI Plug-in Environment

To configure the UI-plug-in, create a **uiPlugin-environment.properties** file with the following properties.

uiPlugin-environment.properties

```
#Name of the application
name=STAP

#Type Of application
type=UI

#Base url of application
url=
```

```
#short description of application
description=stap-ui platform

#browser on which automation will be performed.
# Supported browsers : FIREFOX, CHROME
browser=firefox
# UI Automation Configuration
# Environment based automation-configuration allowing users to customize the
behavior of the plugin through a properties file at runtime.
# The following properties control the behavior of the UI automation
framework.# Maximum time in milliseconds to wait for an element to be
available.
# DefaultValue:5000
element.wait.maxTimeout=

# Maximum time in milliseconds to wait for a page to load.
# DefaultValue:90000
page.load.maxTimeout=

# Minimum time in milliseconds to wait between actions to confirm if the
previous action completes or not.
# DefaultValue:500
action.min.wait=

# The maximum time in milliseconds the framework will wait for an element to
be available.
# DefaultValue:15000
action.max.wait=

# Time in milliseconds to pause between actions.
# DefaultValue:400
action.delay.wait=

# Maximum number of retries for a failed action.
# DefaultValue:5
max.retries=

# Time in milliseconds to wait before retrying a failed action.
# DefaultValue:2000
retry.delay=

# Training model to be used for UI automation. Can be either 'DEFAULT' or
'LLM'.
# DefaultValue:Default
training.model=
```

File location: $testWS/config/environments/uiPlugin-environment.properties

2. **UI-Product and Browser Configuration**

The next step is to configure the browser. This is done by creating a browser-specific properties file (for example, **chrome.properties, firefox.properties**) with settings for the browser.

The **driver.path** property is mandatory and should be set to the location of the webdriver file.

The other properties in the browser properties file have default values that can be used or modified as needed.

chrome.properties

```
#browser-driver path (Mandatory)
driver.path=

# Launch browser in headless mode, i.e (no browser window will appear,
Browser operations run in the background).
headless=false

# Set window size
window.size=1920,1080

# Open browser in incognito
incognito=false

# Disable extensions
disable.extensions=true

# Disable pop-up blocking
# TRUE: Allows pop-ups to open, which may be necessary for certain web
application tests involving pop-ups or new windows.
disable.popup.blocking=true

# Custom user agent (leave blank to use default; Websites can detect the
browser and platform from the User-Agent. Customizing it is useful for
simulating different devices or browsers or bypassing certain restrictions.)
user.agent=CustomUserAgent

# Download directory (use absolute path)
download.directory=

# Disable GPU
disable.gpu=true

# Disable notifications (Stops "Allow/block notification" prompts from
appearing, reducing flakiness in automated tests.)
disable.notifications=true

# --- PROXY CONFIG (recommended, simple) ---

# Main proxy setting: protocol://host:port (e.g. http://proxyhost:8080,
socks5://socksproxy:1080)
proxy.server=

# Proxy protocol type: http, socks5, socks4, ftp, ssl, direct, or system
proxy.type=

# Comma-separated hosts to bypass the proxy (NO_PROXY equivalent)
proxy.bypass=

# For most uses, simply set proxy.server and proxy.type. Example:
#   proxy.server=http://proxyhost:8080
```

```
#   proxy.type=http

# Advanced: Separate proxies per protocol (If multiple proxies required, but
supported for rare cases):
#   proxy.http.host=
#   proxy.ssl.host=
#   proxy.ftp.host=
#   proxy.socks.host=

# Normally, DO NOT use the above advanced keys unless your network requires
it!
```

File Location: $testWS.config.plugins.ui.browsers

> **Note**
>
> The UI Automation Plug-in loads a browser driver during execution that could introduce performance overheads.

3. **Create Scenario Files**

The scenario files define the test cases for UI testing. A scenario file typically includes:

- Scenario: The name of the scenario.
- Description: A brief description of the scenario.
- Tags: Relevant tags for the scenario.
- Case: The specific case being tested.
- When statements: The steps to be performed during the test.

For more information see, "Creating Scenarios".

Scenarios-Library.scenario

```
Scenario: Scenarios library functionality
Description: Actions library functionality
Tags: STAP, Selenium, Scenarios

Case: Scenarios Library Page

When on STAP UI login page, provide login details and submit
Data:
| $open | @uri |
| $input | $username,tesuser |
| $input | $password,****** |
| $pressKeysSequentially | $mockLocator,ENTER |

When on STAP UI scenarios page, selecting scenarios and displaying the steps
Data:
| $click | $menu |
| $click | $scenarios |
| $click | $firstScenario |
| $click | $collapseAllCases |
| $click | $expandAllCases |
```

```
| $click | $collapseAllScenarios |
| $click | $expandAllScenarios |
```

Organizing Scenario Files

To maintain consistency and simplify test case management, scenario files should follow a structured folder hierarchy as outlined below.

Scenario File Location Convention

Scenario files should be placed under:

```
$testWS/scenarios/{ProductName}/{productPage}/{functionality.scenario}
```

- **$testWS**: Your test workspace root directory
- **ProductName**: The product/component being tested (for example, STAP)
- **productPage**: The specific UI page or module under test (for example, actions)
- **functionality.scenario**: The scenario file for a specific feature or test case (for example, actions_library_functionality.scenario)

Example:

```
$testWS/
└──── scenarios/
    └──── {ProductName}/
        └──── {productPage}/
            └──── {functionality}.scenario
```

Scenarios_Library.scenario

```
$testWS/
└──── scenarios/
    └──── STAP/
        └──── Scenarios-Library/
            └──── Scenarios-Library.scenario
```

4. **Create Action Files and Page Properties Files**

After creating the scenario file, create the corresponding action files and page properties files.

Action files define the metadata for a specific action. An action file typically includes:

- **path**: The path to the action.
- **name**: The name of the action.
- **description**: A brief description of the action.
- **actionType**: The type of action (UI), same as mentioned in uiPlugin-environment.properties.
- **product**: The product name, same as mentioned in uiPlugin-environment.properties (case-sensitive)
- **pageElementConfig**: The configuration for the page elements.
- **tags**: Relevant tags for the action.

Example

stap.scenarios.action.json

```
{
    "path":"Stap/scenarios",
    "name":"on STAP UI scenarios page",
    "bdd":"on STAP UI scenarios page",
    "description":"on STAP UI scenarios page",
    "actionType":"UI",
    "product":"STAP",
    "pageElementConfig":"scenarios",
    "tags":["selenium","stap-ui"]
}
```

Page properties files define the locators for the page elements. The file name should be in the format **<pageElementConfig>.page.properties**, where **<pageElementConfig>** is the value of the **pageElementConfig** property in the action file.

Example (actions.page.properties):

scenarios.page.properties

```
page.wait=2000,5000
page.next=history_page

uri = /

menu=xpath://*[@id="drawerToggleButton"]/button
menu.wait=2000,4000

scenarios=xpath://*[@id="ScenarioLibrary"]/a
scenarios.wait=2000,4000

searchScenarios=xpath://input[@placeholder='search...' and contains(@class,
'oj-text-field-input')]
searchScenarios.wait=2000,4000

mockLocator=/
mockLocator.wait=2000,4000

expandAllScenarios=xpath://oj-button[.//span[text()='Expand all']]//button
expandAllScenarios.wait=2000,4000

collapseAllScenarios=xpath://oj-taas-libraries-oj-taas-scenario-library//oj-
button//span[text()='Collapse all']
collapseAllScenarios.wait=2000,4000

expandAllCases=xpath://oj-taas-libraries-oj-taas-scenario-library//oj-button//
span[text()='Expand cases']
expandAllCases.wait=2000,4000

collapseAllCases=xpath://oj-taas-libraries-oj-taas-scenario-library//oj-
button//span[text()='Collapse cases']
collapseAllCases.wait=2000,4000
```

```
firstScenario=xpath://ul[contains(@class, 'oj-listview-element')]/li[1]
firstScenario.wait=2000,4000
```

Organizing Action and Page Properties Files

To ensure a clean and maintainable project structure, action and page properties files should be organized in a standardized folder hierarchy. This enables easy navigation and scalability as your UI automation project grows.

1. Create a folder named after your product inside the main **actions** directory.
2. Within the product folder, create a subfolder named **UI**.
3. Inside the **UI** folder, create a folder for each specific page (use the page's name).
4. Place both the action file (for example, **xyz.action.json**) and the page properties file (for example, **abc.page.properties**) inside the respective page folder.

Example

```
actions/
└── <productName>/
    └── UI/
        └── <pageName>/
            ├── <productName>.<pageElementConfig>.action.json
            └── <pageElementConfig>.page.properties
```

Example folder structure:

```
actions/
└── STAP/
    └── UI/
        └── scenarios/
            ├── scenarios.page.properties
            └── stap.scenarios.action.json
```

Example **execution.config.json**:

```
{
  "name": "Stap E2E Tests",
  "mode": "training",
  "group": [
    {
      "name": "UI-Plugin testing",
      "scenarios": [
        "uiMock/dropdown"
      ]
    }
  ]
}
```

For subsequent runs, you can change the **mode** value back to **default** in **execution.config.json** to resume the standard UI automation flow. The system attempts the primary locator first; if it fails, it cycles through the stored alternative locators and uses the recorded statistics to enable auto-healing via a fallback mechanism, helping prevent test

failures. When a working locator is found, it is promoted to the top of the list to optimize future executions.

> **Note**
>
> This feature is currently supported only for **STAP-DE** executions and is not available when running from the **Microservice**.

Tiered Configuration System for UI Plugin

The UI Plugin uses a tiered configuration approach to control automation settings, such as timeouts, retries, and delays. Configuration is managed through dedicated property files and code defaults.

Key Property Files:

- **uiPlugin-environment.properties**
- **plugin-global.properties**
  Location: `config/plugins/ui/plugin-global.properties`

Tiered Override Logic: Config lookup now follows this order: environment properties → global properties → code defaults. If a value is missing, blank, or invalid at a higher tier, it is ignored and the next tier is used automatically.

Common Configuration Settings

These properties determine UI automation framework behavior.

Table 7-8 Common Configuration Settings

Property Name	Description	Default Value
element.wait.maxTimeout	Maximum time in milliseconds to wait for an element to become available.	5000
page.load.maxTimeout	Maximum time in milliseconds to wait for a page to finish loading.	90000
action.min.wait	Minimum time in milliseconds to wait between actions to confirm previous completion.	1000
action.max.wait	Maximum time in milliseconds to wait for an action to complete.	12000
action.delay.wait	Time in milliseconds to pause between actions.	5000
max.retries	Maximum number of retries for a failed action.	3
retry.delay	Time in milliseconds to wait before retrying a failed action.	3000
training.model	Specifies training model for UI automation. Use 'DEFAULT' or 'LLM'.	DEFAULT

Configuration Override Order

The system determines each property's value using the following precedence:

- **Environment Properties**: If set and non-blank in the environment-specific config (uiPlugin-environment.properties), this value is used.
- **Global Property File**: If not set in the environment config, the system uses the non-blank value from plugin-global.properties.
- **Code Defaults**: If neither configuration file provides a non-blank value, the system applies the safe, hard-coded default from UIPluginConfigImpl.

Blank or Missing Properties

- If a property key is present but blank (for example, `element.wait.maxTimeout=`), the system treats it as not set and applies the next configuration tier.
- If a property key is missing, the behavior is the same as blank.

All fallback or code defaults are defined and maintained in `UIPluginConfigImpl`, ensuring a single source of truth and centralized documentation for built-in values.

Examples for code defaults:

- `element.wait.maxTimeout: 5000`
- `page.load.maxTimeout: 90000`
- `action.min.wait: 1000`
- `action.max.wait: 12000`
- `action.delay.wait: 5000`
- `max.retries: 3`
- `retry.delay: 3000`

Troubleshooting and Best Practices

For uninterrupted UI tests, follow these best practices:

- The folder containing the **action.json** file should have the same name as the product name (case-sensitive).
- All file names should be in lowercase.
- The product name is case-sensitive inside the **action.json** file and in **uiPlugin-environment.properties**.
- The **action.json** file naming convention is **product_name.page_element_config.action.json** (in lowercase).
- The page properties file naming convention is **page_element_config.page.properties** (in lowercase).

Run UI Tests

To run the UI tests, ensure that the scenarios are added in the execution.config.json file under scenarios section.

```
testWS/
└──── config/
    └──── execution/
        └──── execution.config.json
```

Running UI Automation with the TES Microservice

Use the STAP UI Automation Plug-in with the TES microservice to run browser-based UI tests as STAP jobs. The TES microservice orchestrates execution and must have access to the UI automation assets (scenarios, actions, page properties) and the required browser runtime dependencies.

About UI Automation Dependencies

- **Automation scripts: .scenario** and **.case** files.
- **Environment configuration: uiPlugin-environment.properties** and related environment files.
- **UI automation resources:** UI action JSON files and page properties files (locators).
- **Browser dependencies:** a supported browser binary and a matching WebDriver binary.

The STAP UI Automation Plug-in requires an installed browser and matching WebDriver, network access to the application under test, and file system access for WebDriver and downloads.

Required Assets

UI Automation Assets for TES Execution

Table 7-9 Required Assets

Asset Type	Purpose	Examples
Automation scripts	Define test flows	`*.scenario`, `*.case`
Environment configuration	Defines base URL, browser, and timeouts	`uiPlugin-environment.properties`
UI automation resources	Defines UI actions and locators	`*.action.json`, `*.page.properties`
Browser dependencies	Runs and controls the browser	Browser binary, `geckodriver`, `chromedriver`

Set Up TES for UI Automation

Prerequisites

- TES microservice is deployed and can access the Persistent Volume (PV) used for automation assets.
- A supported browser and matching WebDriver are available to the TES runtime.
- Network connectivity from the TES runtime to the application under test (including any required proxy configuration).

Step 1: Service and TES Pod Setup

- **Deploy all required services**, including TES.
- **Install browser dependencies in the TES runtime (TES pod)** for UI automation. The browser must be available in the same container image (or runtime) that executes TES jobs; otherwise, UI jobs will not be able to launch a browser.
  - **Install one browser per TES image**, unless the deployment team has validated storage and performance for multiple browsers.

- **Ensure all required OS libraries** for the selected browser (Firefox/Chrome) are installed.

> **Note**
>
> Any third-party browser installation method must align with Oracle security and compliance requirements for your environment.

Step 2: Provide UI Automation Resources to TES

1. Store UI automation resources in a PV location that TES can read.
2. Ensure the PV includes:
   - UI action definition files (for example, ***.action.json**).
   - Page properties files (for example, ***.page.properties**) that define locators
3. Use a consistent folder structure for UI actions and page properties files (product folder → UI → page folder → files)

Step 3: Configure the UI Plug-in Environment

1. Create (or update) uiPlugin-environment.properties for the UI target. At minimum, set:
   - **name** (application name)
   - **type=UI**
   - **url** (base URL)
   - **browser** (supported values include Firefox and Chrome)
2. If needed, set UI automation timeout and retry properties (for example, element wait and page load timeouts). These properties can be controlled through the environment file and global UI plug-in properties.

Step 4: Configure the WebDriver (Browser Driver)

1. Place the WebDriver binary (for example, **geckodriver** for Firefox or **chromedriver** for Chrome) in a TES-accessible path (for example, on the PV).
2. Create or update the browser-specific properties file (for example, **firefox.properties** or **chrome.properties**).
3. Set the mandatory property:
   - **driver.path=<absolute path to the driver>**
4. If the environment requires a proxy or proxy bypass configuration, set it in the browser properties file so the browser can reach the application under test

Step 5: Ensure UI Plug-in Configuration Files are Available

1. Ensure TES can load the UI plug-in configuration files, including:
   - **plugin-global.properties** (global UI plug-in settings)
   - Browser properties files (for example, **chrome.properties, firefox.properties**)
2. Verify the UI plug-in tiered override behavior (environment properties → global properties → code defaults) to avoid unexpected timeout or retry behavior.

Step 6: Execute the UI Automation Job

1. Ensure the UI scenarios are included in the execution configuration used by TES (for example, **execution.config.json**)
2. Run the job through the TES execution flow that your deployment uses for STAP job execution.
3. Review job results and logs in the configured results location.

Symptom	Likely Cause	Corrective Action
UI automation cannot launch a browser	Browser binary is not available in the TES runtime	Install the browser in the TES container image or runtime environment (not only on the host).
WebDriver fails to start	`driver.path` is missing or incorrect	Set `driver.path` in the browser properties file and confirm the binary is executable and reachable.
Actions or locators not found	UI action JSON or page properties files are missing or not loaded	Confirm the Persistent Volume (PV) contains the correct `*.action.json` and `*.page.properties` files and that the folder structure matches the UI plug-in conventions.
UI is unreachable from TES runtime	Proxy configuration is missing or incorrect	Update proxy and proxy bypass settings in the browser properties file so the browser can access the UI endpoints.

Best Practices

- Match browser and WebDriver versions and validate compatibility in the TES runtime environment.
- Keep UI automation files lowercase and follow naming conventions for UI action files and page properties files to reduce failures.
- Use PV-backed paths for shared automation resources to simplify updates across deployments.
- Limit access to automation PVs and configuration files based on least privilege, and do not store credentials in plain text.

FAQ

Q: What if we already have existing TES deployment without the browser and we need to install browser for UI Automations?

A. The automation runs within the TES pod. browser binaries must be installed there (not just on the VM or host). If the browser is not present in the TES pod, test jobs will be unable to launch browsers, and automation will fail. Steps to manually install the browser inside an existing TES pod:

- If browser needs to be installed in an already existing TES pod, a new image can be created from the existing TES image. Once the file is updated, build a new image and update the **override-values.yaml** with the new image name and **tag -> setup** the TES pod with helm upgrade.

- Edit the **tes dockerfile** to download the gtk3 support and Firefox browser while building the tes image. Run the below steps when the user is set as ROOT, ideally after the proxies have been set.

```
RUN dnf -y install firefox gtk3 \
 && dnf clean all \
 && rm -rf /var/cache/dnf
```

- Chrome can be installed with the below commands. It is recommended to install only 1 browser at a time as exclusive impact testing has not been done with two or more browsers being installed and the impact on storage and performance have not been evaluated.

```
#Chrome installation steps

dnf -y install google-chrome-stable gtk3 \
 && dnf clean all \
 && rm -rf /var/cache/dnf
```

Summary
To avoid any failures during UI testing with the STAP Engine, follow these steps and best practices:

- **Configure UI-Plugin**:
  - Create a **uiPlugin-environment.properties** file with the required properties.
  - Ensure the product name is case-sensitive.
- **Configure Browser**:
  - Update browser-specific properties file (for example, **chrome.properties, firefox.properties**).
  - Set the **driver.path** property to the location of the webdriver file.
- **Create Scenario Files**:
  - Define the test cases for UI testing.
  - Use the correct syntax and formatting.
- **Create Action Files and Page Properties Files**:
  - Create action files with the correct metadata.
  - Create page properties files with the correct locators.
  - Follow the naming conventions:
    - **action.json** file: **product_name.page_element_config.action.json** (in lowercase).
    - Page properties file: **page_element_config.page.properties** (in lowercase).

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

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 data sets that emulate production conditions, ensuring comprehensive test coverage and improving testing efficiency.

STAP offers two types of plug-ins for synthetic data generation: Internal and External.

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

For more information about Data Faker, see their website at https://www.datafaker.net/.

For more information on External Generators, refer to Fake Data Plug-in.

Configuration

Synthetic Data Generation plug-ins are assigned or configured with attribute data configuration which is used in STAP BDD automation. To configure and use synthetic data generation plug-ins within the STAP Behavior Driven Development automation framework:

1. Configure the attribute home location in **config.properties**.
   Add the property in the ${WORKSPACE}/config/config.properties file. For more details, see #unique_80.

   ```
   attributeData.home=${Workspace_home}/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 values:

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}`

Plug-in with Internal Generators

This plug-in is a versatile tool for number generation, offering two distinct modes to cater to various needs:

Number Generation

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

Unique Number Generation:

In this mode, the plug-in ensures that every number generated is distinct, providing a sequence of non-repeating values. It is ideal for creating identifiers, serial codes, or any application where uniqueness is essential. Each number is carefully selected to guarantee exclusivity within the generated set.

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, generate 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

Random Number Generation:

Here, the focus is on randomness rather than uniqueness. This mode produces a series of numbers without any specific pattern, making it suitable for simulations, gaming, or statistical modeling. The random numbers can be generated within a defined range, allowing users to customize the output according to their requirements.

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=subscription ID # Short description description=random number #Plugin associated with the attribute plugin=NumberData Plugin type=NUMBER_RANDOM_VALUES # Plugin Properties for generating data minValue=9999900000 maxValue=9999990000	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=NumberDataPlugin 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=UniqueDataPlugin 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 plug-in to use it to generate fake data for automation scenarios. It also supports the output in multiple languages.

For more information on Fake Data Plug-in, 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	Plug-in 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 plug-in 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.

The STAP UI is highly extensible and comes with numerous built-in plugins that enable interaction with various application interfaces, such as REST. For more information about using the STAP UI, see:

- Icons in STAP UI
- Using Keyboard Shortcuts

Icons in the STAP UI

Table 9-1 lists the 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 cannot fit all of the action icons.)
	Stop a Running Job

Table 9-1 (Cont.) STAP UI Icons

Icon	Description
	Copy

Using Keyboard Shortcuts

You can use keyboard shortcuts for many actions in the STAP UI.

Table 9-2 lists the keyboard shortcuts in the STAP UI.

Table 9-2 Keyboard Shortcuts

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 the 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 the page.
Arrow Keys	Moves focus to the item in the appropriate 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.

10

STAP UI Login Methods

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

Topics in this chapter:

Guidelines for Using STAP UI

For information about supported browsers, see STAP Compatibility Matrix.

When using the UI:

- To avoid losing data, do not use browser commands like as Back, Forward, and Refresh. If you accidentally use a browser command, navigate to the dashboard 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 window.
- Ensure that cookies are enabled in your browser window.

About Authorization Modes

There are two modes of authentication available:

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

Logging In to STAP

You log in to the STAP UI in a browser window. To log in:

1. Enter your **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 username and email address associated with the account.
3. Click **Reset**.
   You will receive an email containing instructions for resetting your password.

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 scheduled to run at that point in time.
Jobs	Total number of jobs.
Completed	Total number of jobs that have been completed.
Running	Total number of scenarios running.
Scenarios	Total number of scenarios.
Active	Total number of scheduled jobs that are running.

Monitoring Real-Time Jobs

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

Table 10-2 Monitoring Real-Time 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.
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 currently running. The fields related to the running jobs 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).

Table 10-3 (Cont.) Fields in Running Jobs

Field	Description
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 Administration

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. Administrative users have additional privileges and information.

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.

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.

Administering Users

The administration environment provides a comprehensive list of all user profiles and facilitates user management tasks, including viewing, deleting, and creating users. When you click the **Admin Dashboard** button, if you are an administrative user you can view the user profiles in a table 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 fields such as first name or last name.
Create New User	Opens a drawer with the fields to create a new user.
Cancel	Returns you to the admin user profile page and cancels the operation.

Creating a New User

As an admin user, you can create a new user by clicking the **Create New User** button on the **Admin Dashboard** This opens a drawer 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 the following 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 creating, updating, and managing environments in Oracle Communications Solution Test Automation Platform (STAP) UI.

Topics in this chapter:

About the Environment Page

To access the environments, from the navigation panel, select **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 mappings for specific testing scenarios. This is a critical part of STAP, you can use the Environments page to manage execution environments used across different jobs. Each environment can have zero or more connections based on the scope of the test. These environments limit or direct job execution within STAP.

To create a new environment:

1. On the Environments page, click **Create Environment.**The **Create environment** page is displayed.
2. Enter the **Name**, and if desired also the **Release, Build Number, and Description.**
3. If you want to add connections to the environment, do the following:
   - Click **Add Connection.**

The **Create Connection** window is displayed.

4. Enter the connection Name, Description, Product (product or application name), and Type (for example, REST, SOAP, or 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 returned to the **Create Environment** page.
6. In the **Search connection** text box under **Connection** section, specify the just created connection to ensure 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 returned 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:

1. Open the environments page using one of the following methods:
   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.

   This action opens the **Edit Environments** page, which consists of two sections: **Overview** and **Connections**.
2. Click the edit (✎) icon on each section to edit.
3. In the **Overview** section, edit the **Name, Description, Release**, or**Build Number** as needed.
4. If you want to view or delete existing connections, in the Connections Section, enter the name of the connection in the search field.
5. To add new connections, click the add (+) icon.
6. Click **Update** to save your changes.
7. Click **Cancel** to terminate your changes.

Deleting an Existing Environment

To delete an existing environment:

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 window appears.
2. Click **Delete.**

13

STAP Jobs Management

Learn about creating, managing, and running jobs in Oracle Communications Solution Test Automation Platform (STAP) UI.

Topics in this chapter:

About the Jobs Page

The term *job* represents a package 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, from the navigation panel, select **Jobs**. The Jobs page allows you to view and manage all the previously created jobs in a table. Each row represents a unique job.

Table 13-1 lists the columns for each job.

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:

1. On the Jobs page, click the **Create New Job** button on the top right corner of the page. The fields include:
   - Name
   - Tags (optional)
   - Environment
   - Description (optional)

- Scenarios: Select each scenario that you would like to include in the job.

2. Click **Create Job** to add the job.

Updating an Existing Job

To update an existing job:

1. On the Jobs page, click the edit ( ) icon in the **Actions** column for the row corresponding to the job you want to modify.
   This opens the **Edit Job** page with the existing information for the job displayed.
2. Edit the data as needed.
3. Click **Update** to save the changes and update the jobs table accordingly.
4. Click **Cancel** to terminate the changes made.

Running a Job

You can run a job in two modes:

- Background
- Run and visit Dashboard (for monitoring real-time execution of the job)

Stopping a Running Job

To stop a running job:

1. Navigate to the Dashboard screen.
2. Scroll to the Running Jobs section, where all jobs that are currently running are listed in a table.
3. In the Actions column for the corresponding running job, click the icon to initiate a safe and controlled termination of the selected running job.
4. A confirmation dialog will appear; click **Stop** or **Confirm** to proceed.
5. The job will be terminated gracefully, ensuring all processes are closed safely.

Deleting a Job

To delete an existing job:

1. On 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.**

Viewing Job History

To manage previously run jobs, from the navigation panel, select **History**.

The jobs are listed in a table. They are listed in the order of their run date, with the most recent job at the top of the page. The row for each job displays the job name, job start date and time (in the time zone of the deployment server), and the job result (passed or failed). You can also:

- View more details about the job by clicking the view button
- Run the job again by clicking the restart button (pic).

> **Note**
>
> If you cannot see the view or restart buttons, you may need to click the more actions icon (•••) to view or restart the job.

You can access the details of a specific job by clicking its row. The row expands and you can view the details listed in Table 13-2:

Table 13-2 Viewing Job Details

Field	Description
Name	Name of the job.
Type	Type of the job. The default job type is **Instant Job**.
Environment	The environment in which the job was run.
Start Time	The date and time that the job was started.
Duration	The amount of time the job took to run, in seconds and milliseconds.
Result	The status of the job: passed or failed.

Viewing the Scenarios for a Job

You can view detailed information about the scenarios in a previously run job by clicking the view icon ( ) at the end of the row of the respective job in the **History** page.

Table 13-3 lists the information displayed for a scenario.

Table 13-3 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, it could show the number of scenarios failed due to validation errors and the number of scenarios that failed due to configuration errors.
- **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".

To restart the job, click on the **Restart Job** button on the top-right corner.

Viewing the Results of Each Scenario

You can view the results of each individual scenario under the selected job under **Scenarios Result**.

If you have multiple scenarios, you type its name into the Filter field.

> **Note**
>
> This search bar does not support filter tags.

All scenarios in the job are listed with the details listed in the Table 13-4:

Table 13-4 Scenario Results

Field	Description
Name	Name of the scenario.
Duration	The duration of time for which the scenario was run, in seconds and milliseconds.
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.

Viewing the Detailed Report of Scenarios

From the **Scenario Results** page, click **View Detailed Report**. This lets you view each scenario in detail including the tasks that passed, failed, or skipped.

1. The pane on the left lists all the scenarios. 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.
2. To expand a task, click its row. You can also filter tasks using the **Pass, Fail,** and **Skip** filter tabs. By default, all filter tabs are enabled.
   When you click on a task, you can view a list of the steps it contained.
3. Click on the row of each step to view a detailed report for the step.
   This opens a window detailing information about the step.

Table 13-5 lists the details displayed for each step:

Table 13-5 Details displayed for each step

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. The time for the job was displayed in the deployment server time zone.
End Time	The end date and time of the task, in the time zone set in your UI.
Duration	The amount of time the step took to run, in milliseconds.
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
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.

14

Intelligent Search

Learn about how to look for test recommendations to help you create new test jobs rapidly by re-using existing test artifacts.

Searching for Test Artifacts

To search for test artifacts, navigate to **Menu** using the navigation panel from **Dashboard**. Select **Intelligent Search**.

The Intelligent Search page opens and displays a search bar and a Type filter.

The Intelligent Search page includes the following areas:

- **Search Artifacts**: A keyword search field where you enter text to search across available test artifacts.
- **Type**: A drop-down list that narrows results by artifact type (for example, ALL, STEP, or CASE, depending on what is available in your environment).
- **Search History**: A list of your recent search keywords. Select a keyword to run the search again.

To search for the test artifacts, perform the following steps:

1. In **Search Artifacts**, enter a keyword (for example, function, control, or customer).
2. (Optional) Select a value from the **Type** drop-down list to filter results.
3. Review the results list displayed below the search bar.

Each result appears as a row and includes:

- The artifact name/summary text.
- An artifact label on the right (for example, STEP or CASE).
- A **View** icon to open the artifact details.

After you run a search, the page shows matching artifacts in a scrollable list. To refine what you see:

- Change the keyword in **Search Artifacts** and search again.
- Use the **Type** filter to focus on a specific artifact type.
- Select a term from **Search History** to quickly re-run a previous search.

When a result is labeled CASE, you can open the scenario details.

1. Locate a CASE result in the list.
2. Select the View icon for that row.

A details window opens showing the step title (for example, Step: Then get mock response, processing user subscription and notifications) and the step content. The step details can include:

- **Data** shown in a table format.

- **Validate** expectations (for example, expected status values).

From this window you can:

- Select the **Copy** icon  to copy the step content.
- Select **X** to close the window and return to the results list.

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 the product from 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

Setting Up The STAP Environment

Learn about setting up the Oracle Communications Solution Test Automation Platform (STAP) environment to automate scenarios and publish results.

17

Low Code Automation

Learn about creating automation scenarios in Oracle Communications Solution Test Automation Platform.

Topics in this chapter:

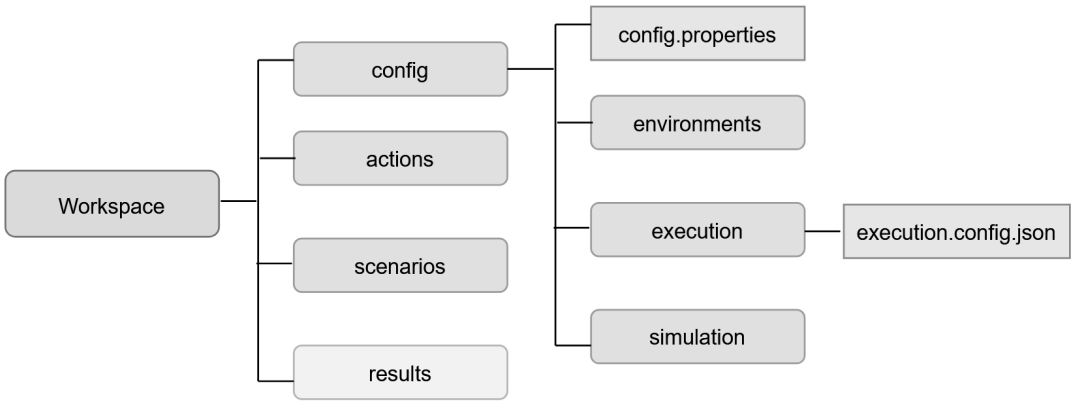
Overview

STAP enables users to automate workflows without writing complex code. You can use the Behavior-Driven Development (BDD) language 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.

It is important for you to use a well-organized project structure to ensure that you manage automation assets effectively.

Figure 17-1 shows the key project directories in STAP and their purposes.

Figure 17-1 Automation Workspace File Structure



The workspace should contain the following top-level folders:

- **action:** Contains action files required for running automation scenarios. For more information, see "Setting Up Actions".
- **config:** Contains the **config.properties** files and directories that contain various configuration settings for your automation project. For more information, see "Setting Up STAP Configuration".
  - **Environment:** Contains environment-specific configurations for your automation tests. For more information, see "Setting Up Environments".

  - **Simulation:** Contains configurations related to test data simulation. For more information, see "Setting Up Simulation".
  - **Execution:** Contains execution configurations that define how automation scenarios are run. For more information, see "Setting Up Execution".
- **scenarios:** Organizes your automation scenarios into a logical folder structure for improved maintenance and navigation. For more information, see "Setting Up Scenarios".
- **results:** STAP automatically publishes all test execution results to this folder. For more information, see "Setting Up Reports".
- **context:** Stores automation context data used to avoid redundant execution of steps during scenario automation. For more information, see "Setting Up Context".

Automating Using the STAP Design Experience

The STAP Design Experience 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 plug-ins. Additionally, the package provides ready-to-use environment templates tailored for specific plug-ins and environments to accelerate the automation process.

Before using the STAP Design Experience package, ensure you have set it up on your system. For more information, see "Setting Up The STAP Design Experience" in *Deployment Guide*.

Ensure you have securely stored your automation project in a third-party version control that includes initializing a repository, tracking changes, and collaborating efficiently.

The following is an end-to-end process of how to set up and run automation using the STAP Design Experience package.

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 "Setting Up STAP Configuration". 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 "Setting Up Environments".
   - **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 **$results/reports/<timestamp>**. After the execution is completed, the report will be automatically opened at the url specified in **results.publish.url** or Open the **index.html** to view the execution report.Configure the results storage location in **config.properties**. For more information, see "Setting Up Reports".
   - **Tomcat Configuration for publishing results:**Publishing results to STAP is mandatory to access the report at the specified URL. Execution results are published

to a web server (for example, Tomcat) to enable easy viewing of reports. Configure **config.properties** as shown below to enable result publishing.

```
#Provide results.publish.file=<tomcat server location>/webapps/
STAPReports/reports/SE2EReports/results.js
results.publish.file=<Tomcat installation directory>/webapps/
STAPReports/reports/SE2EReports/results.js
#Provide the Webserver URL to open the report here ex:- http://
localhost:800
results.publish.url=http://localhost:1800
```

- **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 you to execute specific steps while simulating previously run ones using the context. Configure the context storage location in **config.properties**. For more information, see "Setting Up Context".
- **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".

2. **Compile and Run Automation:** Use the Command Line Interface to compile and run automation. For more information, see "Publishing Data using Command Line Interface".
3. **View Reports:** You can view the reports of the scenarios run in the Results folder. For more information, see "Setting Up Reports".
4. **Publish Scenarios:** Once the automation is complete, you can publish scenarios. For more information, see "Publishing Data using Command Line Interface".
5. **Generating Reports:** You can publish your automation reports in an HTML or PDF format. For more information, see "Generating Automation Reports".

18

Setting Up The STAP Environment

Learn about setting up the Oracle Communications Solution Test Automation Platform (STAP) environment in your system.

Setting Up STAP Configuration

Learn about setting up the configuration for STAP. This includes the **config** folder that contains the main configuration properties **config.properties** file. Ensure the attribute home location is set for the **config.properties** file. For more information on how to set the the attribute home location, see "STAP Synthetic Data Generation".

Using the configuration.properties File

The **config** folder contains the primary configuration file titled **config.properties**. This file contains the configurations required to run STAP.

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/reports
```

```
results.publish
#Provide the results.js file path where the results will be published Ex:-
results.publish.file=C:\software\Servers\apache-
Server-1\webapps\STAPReports\reports\SE2EReports\results.js
results.publish.file=
#Provide the Webserver URL to open the report here ex:- http://localhost:800
results.publish.url=
#--------------------------------------------------------------------------------
--
# 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
```

Setting Up Environments

The environments folder contains the various testing environments in STAP. This folder is a subfolder under the configuration folder.

> **Note**
>
> You can organize environment configurations into distinct folders within the **environment_configurations** sub directory.

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 location for adding the environment details in the **config.properties** folder:

```
environments.home=${WORKSPACE}/config/environments
```

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 its own **environment.properties** file.

Setting Up Execution

The execution folder contains an **execution.config.json** configuration file. This file contains details of scenarios to run. For more information about scenarios, see "Setting Up Scenarios".

Scenarios are run in various execution groups. You can modify this file to group multiple automation scenarios and run them at the same time. A minimum of one group is required to run a scenario.

The **execution.config.json** contains these components:

- **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. If no name is provided, a unique group ID is assigned.
- **execution (Optional)**
  Execution can be configured as **serial** or **parallel** for each group or subgroup. This parameter controls whether the scenarios within the group run one after another (serial) or at the same time (parallel). If not defined, groups default to serial execution.
- **description (Optional):**
  Provides a textual description of the scenario or group.
- **release (Optional):**
  Indicates the associated release identifier or version, which can help organize or filter results by release.
- **milestone (Optional):**
  Specifies the milestone connected to the group or scenario.
- **build (Optional):**
  Specifies the build ID for the scenarios.
- **level (Optional):**
  Indicates thehierarchy level for the scenarios.
- **reportTitle (Optional):**
  Sets a custom title that will appear in generated reports. If this field is not set, the default report title is **Automation Report**.

The following is an example of the **execution.config.json** file when a single scenario is run:

```
{
        "name": "STAP Sample Tests",
        "execution": "serial",
        "group" : [
                {
                    "name" : "STAP Sample Tests",
                    "scenarios" : [
                        "1.ABC"

                ]
                }
        ]
}
```

Creating different groups of scenarios allows independent execution of groups, and the failure of one group does not halt the execution of others.

When grouping multiple scenarios, each group contains a scenarios list, which specifies the parent folder names where **.scenario** files reside.

If **.scenario** files are located in nested folders, the parent folder names of the scenarios folder should be specified.

> **Note**
>
> If multiple **.scenario** files exist in a single folder, only the first **.scenario** file is run.

The following is an example of the **execution.config.json** file when multiple scenarios are run:

```
{
    "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"
                    ]
                }
            ]
        }
    ]
}
```

Setting Up Scenarios

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

You can use tags to categorize scenarios for easy identification. For more information, see "Using Tags to Filter Components".

For more information about scenarios in STAP and how to create them, see "Creating Scenarios".

Scenarios Folder

The scenarios folder contains the different scenarios to be run. Each scenario is stored in a separate folder within this folder.

Each scenarios folder has the following components:

- **Header.info**: Contains the scenario details in the following format:

  ```
  Scenario: Name of the E2E Scenario
  Description: Description of the E2E Scenario
  Tags: Tag1, Tag2
  ```

- **Case files**: Each **.case** file covers a specific logical step in a scenario. For more information on the contents of the **.case** file, see "Creating Scenarios".

  > **Note**
  >
  > You only create a separate **.case** file if your scenario contains multiple cases. If you have a single case, you can define it within the **.scenario** file.

- **scenario.config**: Contains the list of **.case** files to be merged to create the **.scenario** file at run time. This is only applicable for multiple **.case** files. Use the following format when creating a scenario configuration file:

  ```
  Header.info
  1.Launch.case
  2.Buying.case
  3.fusionCDM.case
  4.BRM.case
  5.Care.case
  ```

The following is the configuration for adding the scenario details in the **config.properties** folder:

```
scenarios.home=${WORKSPACE}/scenarios
```

For more information on creating scenarios, see Creating Scenarios.

Setting Up Simulation

Enter a short description of your topic here (optional).

This is the start of your topic.

Setting Up Actions

The Action component provides all input required to the respective plug-in. This input specifies how and with what data the plug-in should run the action. For more information on plug-ins, see "STAP Action Plug-ins".

The structure of the Action folder is in the following hierarchy:

- **Product Folder:** The folder for the product containing the respective plug-in.
- **Plug-In Folder:** The type of plug-in. For example, **REST**.
- **Path Folder:** The folder containing it's respective actions. For example, **bill**.
- **Action:** The action file. Use lowercase letters with hyphens to separate words in action file names. File names should be self-descriptive and end with the **.action.json** extension. For example, **create-bill-by-ID.json**.

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".

You can 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". Ensure to provide a default request/data for the action.

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
}
```

Setting Up Context

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 parameters used in the configuration are described below:

- **context.home:** Defines the directory where context data is stored.
- **context.load:** Determines whether to load context data while running (`YES/NO`).
- **context.save:** Specifies whether to save context data for a scenario, useful for debugging (`YES/NO`).

- **context.global.load:** Controls whether global context data (shared across scenarios) should be loaded.
- **context.global.save:** Controls whether global context data (shared across scenarios) should be saved.

Configuration Sample:

```
context.home=${WORKSPACE}/context
context.load=NO
context.save=NO
context.global.load=NO
context.global.save=NO
```

Context manages two types of variables:

- **Local Variables:** These are available only during the execution of a single scenario. They are not preserved outside the scenario run.
- **Global Variables:** Global variables are prefixed with an underscore and exist for the duration of the entire job run, across scenarios and steps.

For more information on variables, see "Setting Up The STAP Environment".

Setting Up Reports

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 for each test run are created in a timestamped folder under **$results/reports/<timestamp>**. Once the test run completes, the report is automatically opened using the URL specified in **results.publish.url**. Alternatively, you can open **index.html** in the corresponding timestamped folder to view the test run report.

The following is the configuration for adding the result details in the **config.properties** folder:

```
#-------------------------------------------------------------------------
--
# Results storage location
#-------------------------------------------------------------------------
--
results.home=${WORKSPACE}/results
```

19

Creating Scenarios

Learn how to create scenarios to be tested and automated in Oracle Communications Solution Test Automation Platform.

There are two ways to create a scenario:

1. **Using A Single Case File:** For a simple scenario, you can create a single **.scenario** file which contains the case details. The following format shows how to create a **.scenario** file with a single case:

```
Scenario: Name of the E2E Scenario

Description: Description of the E2E Scenario
Description can be of multiple lines>

Tags: Tag1, Tag2

Case: Case Name
Description: Case Description
Tags: Tag1, Tag2
Given/When/Then/And Step description

Data:
| name  | value |
| name  | value |

Validate:
| name  | value |
| name  | value |

Save:
| Path | Variable |
```

2. **Using Multiple Case Files:** For larger scenarios containing complex multi-product or end-to-end scenarios, you can split it into multiple **.case** files, These are configured in the **scenario.config** configuration file. To set up the case file, see "Case".
For more information on setting up the scenario folder, see "Scenarios Folder".

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.

Each case file looks like this:

```
Case: Case Name
Description: Case Description
Tags: Tag1, Tag2
Given/When/Then/And Step description

Data:
| name  | value |
| name  | value |

Validate:
| name  | value |
| name  | value |

Save:
| Path | Variable |
```

If your scenario contains just one case, you do not create a separate **.case** file. Instead, you define the case within the **.scenario** file. For more information, see "Creating Scenarios".

You can use tags to categorize cases for easy identification and filtering based on various contexts like use case, feature, or functionality. For more information, see "Using Tags to Filter Components".

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.

The step uses the BDD syntax of 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.

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.

You might use the following information to set up tags:

- Product Name
- Feature Name
- Use Case ID/Name
- Release
- Test Type (for example, Functional, Regression, Performance)
- Priority (for example, High, Medium, Low)
- Customer
- Topology/Setup/Environment
- Group/Category

20

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".

21

Publishing Data

Learn about publishing data and reports in Oracle Communications Solution Test Automation Platform (STAP).

Topics in this document:

- Publishing Actions, Scenarios, and Environments Using the Command-Line Interface
- Generating Automation Reports

Publishing Actions, Scenarios, and Environments Using the Command-Line Interface

You can use the STAP utility to publish actions, scenarios, and environments to the cloud, where they can be executed as jobs. This allows you to run tests remotely, share results, and use cloud resources instead of your local machine.

> **Note**
>
> To publish components, you must add the TDS environment details in your environment configurations.

To publish an action, run this command:

```
./stap --publish -action "workspace=path"
```

To publish a scenario, run this command:

```
./stap --publish -scenario "workspace=path"
```

To publish an environment, run this command:

```
./stap --publish -environment "workspace=path"
```

The following example shows a successful environment publish operation. Key indicators of success are highlighted below:

```
./stap --publish -environment workspace=sampleWorkSpace
============================================================
STAP Automation Platform CLI
Version : version
============================================================
WARNING: Runtime environment or build system does not support multi-release
JARs. This will impact location-based features.
```

```
Workspace Location : /home/opc/STAP/sampleWorkSpace
================================================================================
==========
CONFIGURING ENVIRONMENT PUBLISH UTILITY
================================================================================
==========

Loading REST environment
Loading configuration /home/opc/STAP/sampleWorkSpace/config/config.properties
Loading environment connections from /home/opc/STAP/sampleWorkSpace/config/
environments

================================================================================
==========
PUBLISH ENVIRONMENT
================================================================================
==========

adding basic...bXVqaWJ1ci5zaGFpa0BvcmFjbGUuY29tOndlbGNvbWUx
End Point=http://123.456.7.890:12345/environment/complete

{"name":"Publish env test","description":"STAP
environment","build":"1.0","release":"3.0 Productize","connection":[]}
Path : http://123.456.7.890:12345/environment/complete
Target : http://123.456.7.890:12345/environment/complete

================================================================================
==========
PUBLISH RESULT
================================================================================
==========
Status : SUCCESS

Response:
{"_id":1}
================================================================================
==========
```

Updating an Existing Scenario

When publishing a scenario, STAP also supports updating an existing scenario. To allow updates, you must explicitly enable updates in the `publish.properties` file located under the **scenario** folder.

For example:

```
#
#Fri Jan 09 09:13:09 IST 2026
id=1
update=YES
version=21a
```

By default, the `update` value is set to `NO`. When `update` is set to `YES` and you publish the scenario again, STAP treats the operation as a scenario update.

When a scenario is updated, the following actions occur automatically:

- The existing scenario is marked as inactive.
- A new scenario ID is generated for the updated scenario.
- All existing jobs that reference the old scenario are automatically updated to use the new scenario ID.

This behavior ensures that jobs always run against the latest active version of the scenario without requiring any manual updates to the jobs.

If updates are not enabled in the `publish.properties` file, publishing of the scenario is skipped and the existing scenario remains unchanged.

As an example, to update and publish a scenario, run the following command:

```
./stap --publish -scenario “workspace=./sampleWorkSpace”
```

The command produces the following output:

```
Updated 1.FunctionsAndOperators scenario with new Id: 2
-------------------------------------------------------

=================================================
PUBLISH SCENARIOS SUMMARY:
=================================================
Total Scenarios : 1
1.FunctionsAndOperators : Updated
=================================================
Scenarios Published : 0
Scenarios Updated : 1
Scenarios Skipped : 0
Scenarios Failed : 0
=================================================
STAP Publish Scenario Successfully Completed.
```

After a successful update, the `update` value is automatically reset to `NO` in the `publish.properties` file, as shown below:

```
#
#Fri Jan 09 09:14:09 IST 2026
id=2
update=NO
version=21a
```

This prevents unintended updates in subsequent publish operations.

Generating Automation Reports

Learn about generating automation reports in a PDF format or using a web server in an HTML format.

Topics in this section:

- Publishing PDF Reports

- Publishing Reports Using Third-Party Web Servers

Publishing PDF Reports

You can use the **PDF Generator Adapter** in the STAP to generate PDF reports. The **PDF Generator Adapter** is a configurable module in the STAP **Design Experience** that generates PDF reports from structured data and HTML templates. It supports summary and evidence report formats and can create single or multiple documents.

You can generate the following report types using the **PDF Generator Adapter**:

- **Summary Report:** Provides an overview of key results. It includes high-level information about each scenario with it's duration and status, along with an overall summary chart. For more information, see "Summary Report".
- **Evidence Report:** Provides a detailed report of each scenario run, alongside information of each case within the scenario, with request and response data. For more information, see "Evidence Report".

These reports can either have a single file or multiple files based on their configuration.

Setting Up The PDF Adapter In Your Workspace

Before configuring the method of generating PDF reports, ensure the PDF Generator adapter is set up correctly in your workspace directory. The folder **summaryPDFGenerator** is shipped with the STAP DE package, under the **adapters** folder in **config** folder..

summaryPDFGenerator contains the following components:

- **.properties file**. You can generate PDF reports using the STAP DE, the command-line interface, or the TES microservice. Each method requires a different properties file:
  - **pdfGenerator.config.properties:** The configuration file for generating reports using the STAP DE.
  - **pdf-adapter.properties:** The configuration file for generating reports using the command-line interface.
  - **pdfGenerator.config.properties:** The configuration file generating reports using the TES microservice.
    The property **pdf.generate** within the **.properties** file determines the reports to generate: evidence or summary. To customize this, create a comma-separated list of the reports you want to generate:

    ```
    pdf.generate=evidenceReport,summaryReport
    ```
- **Config Folder:** Contains a sub-folder titled **Configs**, which contains configuration files specific to each PDF report generated in JSON format:
  - **evidenceReport.pdf.config.json:** The JSON input file that provides structured data for the evidence report.
  - **summaryReport.pdf.config.json:** The JSON input file that provides structured data for the summary report.
- **Templates Folder:** Template files for the summary report, titled **summaryReport.template**, and the evidence report, titled **evidenceReport.template**. These can either be in an HTML or an FTL format. By default, they are in HTML format.
- **Output Folder:** The generated evidence and summary reports in PDF format.
- **Resources Folder:** Contains the static components of the PDF report: the company or project logo in PNG format, and the report font in TTF format. By default, the resources file

ships with Oracle's logo and default font. However, you can change the logo and font by replacing the PNG and TTF files with your custom files in the same format.

The final directory structure looks like this:

- Configuration Properties file (dependent on the report generation method)
- Config folder
  - Configs Folder
    - evidenceReport.pdf.config.json
    - summaryReport.pdf.config.json
- Templates folder
  - summaryReport.template.html
  - evidenceReport.template.html
- Output folder
  - EvidenceReport.pdf
  - SummaryReport.pdf
- Resources folder
  - logo.png
  - font.ttf

After ensuring the structure of the **PDF Generator Adapter** is set correctly in your workspace, you can generate PDF reports using three methods:

- To publish PDF reports using the STAP DE, see "Generating PDF Reports With The STAP DE".
- To publish PDF reports using the command-line interface, see "Generating PDF Reports With The Command-Line Interface".
- To publish PDF reports using the TES microservice, see "Generating PDF Reports With The TES Microservice".

Generating PDF Reports With The STAP DE

To generate PDF reports using the STAP DE:

1. Start WireMock:

   ```
   sh myWorkSpace/WireMock/startWireMock.sh
   ```

2. Run the PDF Generator jar file:

   ```
   sh run.sh
   ```

   This lets the **PDF Generator Adapter** read the files within the adapter's folder, alongside the scenario execution result JSON file generated in the **data** folder when the scenario is run.

3. PDF reports of the scenarios run are generated in the **Output** folder.

Generating PDF Reports With The Command-Line Interface

To generate PDF reports using the command-line interface:

1. Start WireMock:

```
sh myWorkSpace/WireMock/startWireMock.sh
```

2. Run the PDF Generator jar file:

```
sh pdfGenerate.sh myWorkspace jsonResultDirectory
```

Where:

- *pdfGenerate.sh* is the PDF Generator Adapter JAR file
- *myWorkspace* is your workspace directory
- *jsonResultDirectory* is the path to the scenario's results JSON file in its Data folder.

This lets the **PDF Generator Adapter** read the files within the adapter's folder, alongside the scenario execution result JSON file generated in the **data** folder when the scenario is run.

3. PDF reports of the scenarios run are generated in the **Output** folder.

Generating PDF Reports With The TES Microservice

When generating PDF reports using the TES microservice, you do not need to perform any additional steps.

A separate configuration JSON file titled **adapter.config.json** is present within the TES folder:

```
${TES_HOME}/config/adapters/adapters.config.json
```

PDF reports for all jobs run will by default be saved in the **Output** folder.

Viewing PDF Reports

Summary and Evidence reports are pre-structured. For more information about the components of Summary Report, see "Summary Report". For more information about the components of Evidence Report, see "Evidence Report".

Summary Report

The Summary Report contains these fields:

- **Cover Page:** The first page of the summary report.
- **Summary:**
  - **Summary Table:** A table summarizing metrics of all scenarios run.
  - **Summary Chart:** A visual representation of the scenarios run.
- **Test Scenarios:** Report of each test scenario run.

Cover Page

Table 21-1 shows the fields of the cover page of the summary report.

Table 21-1 Cover Page Fields

Field	Description
Company Title	The title of the company or project. By default, it is set to **Oracle**.
Report Type	The Type of report. By default, it is set to **STAP Automation Report**.

Table 21-1 (Cont.) Cover Page Fields

Field	Description
Author	The author of the report.
Creation Date	The date the report is created.
Last Updated	The date the report is updated last.
Version	The version of the report.
Approvals	Names of approvers for the report. You can add approvers in the configuration JSON file. If there are none, the rows are blank.

Summary

Contains the **Summary Table** and **Summary Chart**.

Table 21-2 shows the fields of the summary table in the summary report.

Table 21-2 Summary Table

Field	Description
Name	The name of the scenario.
Status	The status of the scenario.
Pass	Number of scenarios passed.
Fail	Number of scenarios failed.
Error	Number of scenarios containing errors.
Skip	Number of scenarios skipped.
Start Time	The date and time that the test was started.
End Time	The date and time that the test was complete.
Duration	The amount of time the test took to run, in milliseconds.

Summary Chart

Shows a visual representation of the number of scenarios passed and failed in a pie chart format.

Test Scenarios

Table 21-3 shows the fields of the test scenarios table in test scenarios.

Table 21-3 Test Scenarios

Field	Description
Name	Name of the scenario.
Description	Description of the scenario.
Duration	The amount of time the scenario took to run, in seconds and milliseconds.
Status	The status of the scenario: passed or failed.
Tags	Any tags set for the scenario.

Evidence Report

The Evidence Report contains these components:

- **Cover Page:** The first page of the evidence report. For more information, see "Cover Page".
- **Scenario Summary:** Summarizes metrics of all scenarios run. For more information, see "Scenario Summary".
- **Test Case:** Details of each test case run. For more information, see "Test Case".
- **Test Case Summary:** Summarizes metrics of all test cases run. For more information, see "Test Case Summary".

Cover Page

Table 21-4 shows the components of the cover page of the summary report.

Table 21-4 Cover Page Components

Component	Description
Company Title	The title of the company or project. By default, it is set to **Oracle**.
Report Type	The Type of report. By default, it is set to **Evidence Report**.
Author	The author of the report.
Creation Date	The date the report is created.
Last Updated	The date the report is updated last.
Version	The version of the report.
Approvals	Names of approvers for the report. You can add approvers in the configuration JSON file. If there are none, the rows are blank.

Scenario Sumary

Provides an overall summary of the scenario run. Table 21-5 shows the components of the scenario summary.

Table 21-5 Scenario Summary

Component	Description
Status	The status of the scenario.
Description	The description of the scenario.
Tags	Any tags set for the scenario.
Start Time	The date and time that the test was started.
End Time	The date and time that the test was complete.
Duration	The amount of time the test took to run, in milliseconds.

Test Case

This section details runtime results of test case within the scenario, and each step run. Table 21-6 describes the components under Test Case.

Table 21-6 Test Case

Field	Description
Test Case	The title of the test case.
Step	The title of the step.
Action Name	The title of the action.
Action Type	The type of action. For example, REST.
Data	The data within the step. This includes its name, description, and ID.
Save	That data to save. This includes name, description, ID.
Validate	The data to validate. This includes its status.

Test Case Summary

Displays the list of cases, along with request and response payloads for the steps within each case. Table 21-6 shows the fields in Test Case Summary.

Table 21-7 Test Case Summary

Field	Description
Case ID	The ID of the case.
Name	The name of the case.
Status	The status of the case: passed or failed.
Start Time	The date and time that the case was started.
End Time	The date and time that the case was complete.
Duration	The amount of time the case took to run, in milliseconds.
Step Name	The name of the step.
Start Time	The date and time that the step was started.
End Time	The date and time that the step was complete.
Duration	The amount of time the step took to run, in milliseconds.
Status	The status of the step: passed or failed.
Request	The request payload for the step.
Response	The response payload for the step.

Publishing Reports Using Third-Party Web Servers

You can publish user-interactive reports of the scenarios run using third-party web servers.

You can publish automation reports using these web servers:

- To publish reports using Tomcat, see "Configuring Tomcat to View Automation Reports".
- To publish reports using NGINX, see "Viewing Automation Reports Using NGINX".
- To publish reports using the Apache HTTP server, see "Viewing Automation Reports Using Apache HTTP Server".

Configuring Tomcat to View Automation Reports

To configure Tomcat to view automation reports:

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. In the *Workspace_home***/config/config.properties** file, edit the following lines to set up the location for the published reports:

   ```
   results.home=${WORKSPACE}/results/reports
   results.publish
   file=${WORKSPACE}/results/results.js
   #Provide the Webserver URL to open the report here ex:- http://
   localhost:800
   results.publish.url=http://localhost:1800
   ```

3. Edit the Tomcat server configuration file *Tomcat_Home*/**conf/server.xml**.

   a. Edit these lines 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. Edit the **Context** element inside the **Host** element to configure the path for 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 for the changes to take effect.

Use a URL in the following format to access the published reports:

`https:`*`//TomcatHost:TomcatPort`***`/stap-reports`**

To access individual automation execution results, click on the link for 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.
3. In the *Workspace_home***/config/config.properties** file, edit the following lines to set up the location for the published reports:

```
results.home=${WORKSPACE}/results/reports
results.publish=${WORKSPACE}/results/results.js
results.publish.url=http://{nginx-host}:{nginx-server}
```

4. Configure the path for the automation reports by editing the following lines in the **nginx.conf** file:

```
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.
6. Compile and run the scenarios using STAP, the reports will be automatically opened in the browser at the url: `${host}:{server}`

Use a URL in the following format to access the published reports:

`https:`*`//NGINXhost:NGINXport/`*

To access individual automation execution results, click on the link for 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. In the *Workspace_home***/config/config.properties** file, edit the following lines to set up the location for the published reports:

```
results.home=${WORKSPACE}/results/reports
results.publish=YES
results.publish.file=${WORKSPACE}/results/results.js
```

4. Configure the path for the automation reports in Apache HTTP Server's **httpd.conf** file by running the following:

```
DocumentRoot "${STAP_HOME}/sampleWorkSpace/results/"
<Directory "${STAP_HOME}/sampleWorkSpace/results/">
```

5. Restart the Apache HTTP server.

Use a URL in the following format to access the published reports:

https://*ApacheHost:ApachePort/*

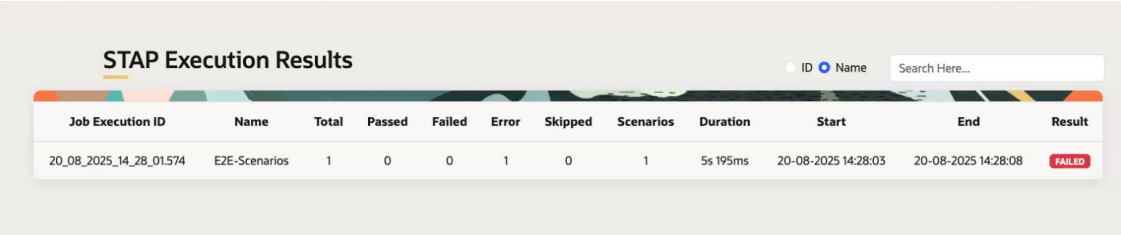
To access individual automation execution results, click on the link for the job.

Viewing HTML Reports

After configuring HTML reports using a web server, you can access it them a user-interactive format.

Upon launching a report, it opens an index page titled **STAP Execution Results** with the total number of jobs run listed. You use the search bar to search for a particular job. To search for a job using its execution ID, select **ID**. To search for a job using its name, select **Name**.

Figure 21-1 Web Server Report Index Page



Each job row has these fields:

Table 21-8 Web Server Report Index Page

Field	Description
Job Execution ID	The ID of the job run.
Name	The job's name.
Total	Number of scenarios in the job.
Passed	Number of scenarios passed.
Failed	Number of scenarios failed.
Error	Number of scenarios containing errors.
Skipped	Number of scenarios skipped.
Duration	The amount of time the job took to run, in seconds and milliseconds.
Start	The date and time that the test was started.
End	The date and time that the test was complete.
Result	The result of the scenario: passed or failed.

To view more details about the job run, click on its respective row. The **STAP Automation Report** opens.

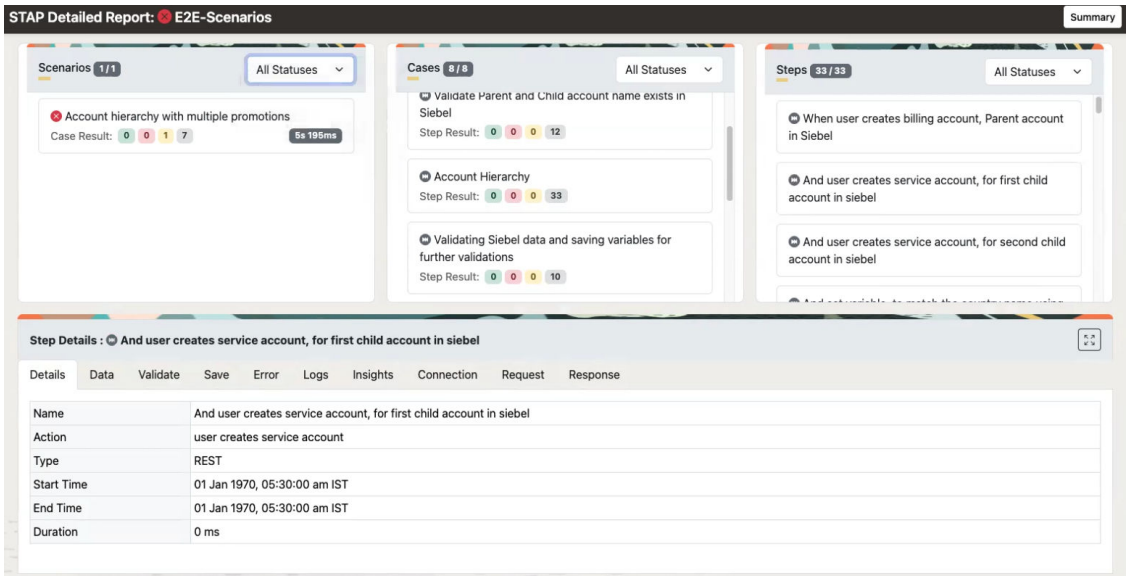
This report displays these metrics:

- Total number of scenarios in the job, including the number of scenarios passed, failed, skipped, or those containing errors. Additionally, it shows the total percentage of scenarios passed, and the amount of time taken to run the job.
- The total number of scenarios in the job, including the number of scenarios passed, failed, skipped, or those containing errors in a pie chart format.
- A failure analysis pie chart that presents the number of scenarios failed, and those with errors.
- The amount of time taken to run each scenario in a horizontal graph.

To return to the index page, click **Home** on the top right corner.

To view the detailed report of each scenario, click on the scenario name in the list under **Scenario Summary Report**. Upon clicking on a scenario report, you can view detailed metrics of the scenario, each case within the scenario, and each step within the case.

Figure 21-2 Scenario Summary Report



Each case row displays a color-coded status of the steps run under it:

Table 21-9 Color Coded Summary

Color	Description
Green	The number of steps passed.
Red	The number of steps failed.
Yellow	The number of steps containing errors.
Grey	The number of steps skipped.

By default, the report displays data for all statuses: passed, failed, skipped, and errors. To filter a case or step using its status, select the status that you want to view under the drop-down menu titled **Statuses**.

To view details of a particular step, select its corresponding case by clicking on it under **Cases**, and click on the step you want to view under **Steps**. You can view the following metrics:

Table 21-10 Detailed Case Report

Field	Description
Details	The step's details: its name, action name, type, start time, end time, and duration.
Data	Displays data configured in the step's BDD. If no data is configured, this section is blank.
Validate	Displays validations created under the step in BDD.
Save	Displays saved variables and values present in the step.
Error	Details of an error when the action was performed. If there is no error, the column is blank.
Logs	Displays a detailed report of each action performed.
Insights	Display screenshots after each step of UI automation.
Connection	Provides details about the endpoint server and its credentials.
Request	The request body of the action.
Response	The response body of the action.

To go back to the **STAP Automation Report**, click on **Summary** on the top-right corner.