Zaplet Scenario Writing Guide

Table of Contents

Introduction

Scenarios in Zaplet are a sequence of HTTP requests defined in a YAML file that allow you to automate and replicate API testing. Scenarios can contain variables, conditions, expected responses, and other elements to create flexible and powerful tests.

The main advantages of using scenarios:

- Reproducible API testing
- Sequential execution of related requests
- Extraction and use of data from previous responses
- Response validation
- Conditional step execution

Zaplet scenario files have the `.zpl` extension and use YAML syntax.

Scenario File Structure

Main Elements

A scenario file consists of the following main elements:

```yaml
name: Scenario Name
description: Scenario Description
repeat: 1
continue_on_error: false

environment:
  key1: value1
  key2: value2

steps:
  - name: Step 1
    description: Description of Step 1
    request:
      # Request definition
    expected_response:
      # Expected response
    variables:
      # Variables to extract

  - name: Step 2
    # ...
```

Required elements:

- **name**: scenario name (string)
- **steps**: list of steps (array)

Optional elements:

- **description**: scenario description (string)
- **repeat**: number of times to repeat the scenario (integer or `infinite`)
- **continue_on_error**: continue execution on error (boolean)
- **environment**: global variables (object)

YAML Format

Scenarios use YAML syntax, which has the following basic rules:

- Indentation is used to indicate nesting (usually 2 spaces)
- `-` is used to denote array elements
- `key: value` is used to define key-value pairs
- Strings can be enclosed in single (`'`) or double (`"`) quotes
- Multi-line strings can be defined using `|` or `>`

Example:

```yaml
# This is a comment
simple_string: value
quoted_string: "value with special characters"
multiline_string: |
  First line
  Second line
  Third line
array:
  - item1
  - item2
  - item3
nested_object:
  key1: value1
  key2: value2
```

Scenario Steps

Step Structure

Each step in a scenario represents a separate HTTP request with additional parameters:

```yaml
- name: Step Name
  description: Step Description
  request:
    # Request definition
  expected_response:
    # Expected response (optional)
  variables:
    # Variables to extract (optional)
  condition: # Execution condition (optional)
  delay: 1000 # Delay before execution in milliseconds (optional)
```

Required step elements:

- **name**: step name
- **request**: HTTP request definition

Optional step elements:

- **description**: step description
- **expected_response**: expected response for validation
- **variables**: variable definitions to extract from the response
- **condition**: step execution condition
- **delay**: delay before step execution in milliseconds

Request Definition

A request is defined using the following parameters:

```yaml
request:
  method: GET  # HTTP method (GET, POST, PUT, DELETE, PATCH, HEAD, OPTIONS)
  url: https://api.example.com/endpoint  # Request URL
  headers:  # Request headers (optional)
    Content-Type: application/json
    Authorization: Bearer token
  body: '{"key": "value"}'  # Request body (optional)
  query_params:  # Query parameters (optional)
    param1: value1
    param2: value2
  timeout: 30  # Request timeout in seconds (optional)
```

Required parameters:

- **url**: request URL
- **method**: HTTP method (default GET)

Optional parameters:

- **headers**: request headers
- **body**: request body
- **query_params**: query parameters
- **timeout**: request timeout in seconds

Examples of different request types:

GET request:

```
request:
  method: GET
  url: https://api.example.com/users
  headers:
    Authorization: Bearer token
```

POST request with JSON data:

```
request:
  method: POST
  url: https://api.example.com/users
  headers:
    Content-Type: application/json
    Authorization: Bearer token
  body: '{"name": "John", "email": "john@example.com"}'
```

PUT request:

```
request:
  method: PUT
  url: https://api.example.com/users/1
  headers:
    Content-Type: application/json
    Authorization: Bearer token
  body: '{"name": "John Updated", "email": "john@example.com"}'
```

DELETE request:

```
request:
  method: DELETE
  url: https://api.example.com/users/1
  headers:
    Authorization: Bearer token
```

Expected Responses

To verify the response to a request, you can define an expected response:

```
expected_response:
  status_code: 200  # Expected status code
  headers:  # Expected headers (optional)
    Content-Type: application/json
  body: '{"success": true}'  # Expected response body (optional)
```

All elements of the expected response are optional, but at least one of them should be specified for validation.

Variables

Variables in Zaplet scenarios allow you to:

- Store common values for use in multiple steps
- Extract values from responses to requests

- Create dynamic requests depending on previous responses

Environment and Global Variables

Global scenario variables are defined in the `environment` section:

```
environment:
  base_url: https://api.example.com
  auth_token: my_secret_token
  api_version: v1
```

These variables are available in all steps of the scenario.

Using Variables

Variables can be used in the following places:

- Request URL
- Request headers
- Request body
- Execution conditions

Syntax for using variables: `${variable_name}`

Examples of using variables:

In URL:

```
request:
  method: GET
  url: ${base_url}/users/${user_id}
```

In headers:

```
request:
  method: GET
  url: ${base_url}/users
  headers:
    Authorization: Bearer ${auth_token}
```

In request body:

```
request:
  method: POST
  url: ${base_url}/users
  headers:
    Content-Type: application/json
  body: '{"name": "${user_name}", "email": "${user_email}"}'
```

Extracting Variables from Responses

Variables can be extracted from request responses using the `variables` section:

```
variables:
  user_id: $.id
  access_token: $.data.access_token
  total_count: $.meta.total
  content_type: header.Content-Type
  status: status_code
  response: body
```

Each variable has a name and an extraction rule that defines how the value will be extracted from the response.

JSONPath

To extract values from JSON responses, JSONPath syntax is used:

```
variables:
  user_id: $.id               # Value of the id field at the root of JSON
  token: $.data.access_token  # Nested field data.access_token
  first_item: $.items[0].name # Name of the first element in the items array
  count: $.meta.total         # Value of the meta.total field
```

Examples of JSONPath expressions:

- `$.field` - field at the root of JSON
- `$.parent.child` - nested field
- `$.items[0]` - first element of an array
- `$.items[0].name` - field of the first array element

Extracting Headers

To extract values from response headers, use the `header.` prefix:

```
variables:
  content_type: header.Content-Type
  location: header.Location
  custom_header: header.X-Custom-Header
```

Extracting with Regular Expressions

To extract values from the response text using regular expressions:

```
variables:
  token: regex:token=([a-zA-Z0-9]+)
  number: regex:"id":(\d+)
```

In this case, the value of the first capture group in the regular expression will be extracted from the response.

Conditional Execution

Zaplet allows you to execute scenario steps conditionally, depending on the values of variables or the results of previous steps.

Condition Syntax

Conditions are defined using the `condition` parameter in the step:

```
- name: Execute request under certain condition
  condition: status_code == 200
  request:
    # ...
```

Comparison Operators

Supported comparison operators:

- == - equal to
- != - not equal to
- > - greater than
- < - less than
- >= - greater than or equal to
- <= - less than or equal to

Condition Examples

```
condition: user_id != ""      # Execute if user_id is not empty
condition: status_code == 200 # Execute if status code equals 200
condition: count > 0          # Execute if count is greater than 0
condition: token != null      # Execute if token is not null
```

Execution Control

Scenario Repetition

A scenario can be executed multiple times using the `repeat` parameter:

```
name: Repeating scenario
repeat: 5  # Repeat the scenario 5 times
# ...
```

For infinite repetition:

```
name: Infinite scenario
repeat: infinite  # Repeat the scenario infinitely
# ...
```

Delays Between Steps

To add a delay before executing a step, use the `delay` parameter:

```
- name: Step with delay
  delay: 1000  # Delay of 1000 milliseconds (1 second)
  request:
    # ...
```

Error Handling

By default, scenario execution stops at the first error. To change this behavior, use the `continue_on_error` parameter:

```
name: Scenario with continuation on errors
continue_on_error: true
# ...
```

Response Validation

Response validation allows you to check whether the response matches the expected one, and if necessary, interrupt the scenario execution.

Status Code Validation

To validate the HTTP status code:

```
expected_response:
  status_code: 200
```

Headers Validation

To validate the presence and value of certain headers:

```
expected_response:
  headers:
    Content-Type: application/json
    Cache-Control: no-cache
```

Response Body Validation

To validate the response body:

```
expected_response:
  body: '{"success": true}'
```

If the response body is in JSON format, structural comparison is performed. This means that the order of fields and formatting do not matter.

Advanced Techniques

Request Chaining

One of the most powerful features of Zaplet scenarios is the ability to execute a sequence of related requests, where each subsequent request uses data obtained from previous responses:

```
steps:
  - name: Authentication
    request:
      method: POST
      url: ${base_url}/auth
      body: '{"username": "user", "password": "pass"}'
    variables:
      token: $.access_token

  - name: Data retrieval
    request:
      method: GET
      url: ${base_url}/data
      headers:
        Authorization: Bearer ${token}
```

Dynamic URL Formation

Variables can be used to dynamically form URLs:

```
steps:
  - name: Get resource list
    request:
      method: GET
      url: ${base_url}/resources
    variables:
      resource_id: $.data[0].id

  - name: Get specific resource
    request:
      method: GET
      url: ${base_url}/resources/${resource_id}
```

Dynamic Headers

Request headers can also be dynamic:

```
steps:
  - name: Authentication
    request:
      method: POST
      url: ${base_url}/auth
      body: '{"username": "user", "password": "pass"}'
    variables:
      token: $.access_token

  - name: Request with token
    request:
      method: GET
      url: ${base_url}/protected-resource
      headers:
        Authorization: Bearer ${token}
        X-Request-ID: ${request_id}
```

Response Transformation

Variables can be used to transform and manipulate data between requests:

```
steps:
  - name: Get data
    request:
      method: GET
      url: ${base_url}/data
    variables:
      item_id: $.data.id

  - name: Update data
    request:
      method: PUT
      url: ${base_url}/data/${item_id}
      body: '{"id": "${item_id}", "status": "updated"}'
```

Scenario Examples

Authentication and Data Retrieval

```yaml
name: Authentication and User Data Retrieval
description: Authentication via API and retrieving user profile
repeat: 1
continue_on_error: false

environment:
  base_url: https://api.example.com
  username: test_user
  password: test_password

steps:
  - name: Authentication
    description: Getting authentication token
    request:
      method: POST
      url: ${base_url}/auth/login
      headers:
        Content-Type: application/json
      body: '{"username": "${username}", "password": "${password}"}'
    variables:
      token: $.data.access_token
      user_id: $.data.user_id

  - name: Retrieve user profile
    description: Request to get profile data
    request:
      method: GET
      url: ${base_url}/users/${user_id}
      headers:
        Authorization: Bearer ${token}
    expected_response:
      status_code: 200
      body: '{"order_id": "${order_id}", "status": "confirmed"}'
    condition: status == "confirmed"
```

Recommendations and Best Practices

Scenario File Organization

1. **Use a logical directory structure**:

```
scenarios/
├── auth/
│   ├── login.zpl
│   └── refresh_token.zpl
├── users/
│   ├── create_user.zpl
│   └── update_user.zpl
└── orders/
    ├── create_order.zpl
    └── process_order.zpl
```

2. **File names should reflect the content**:
    - Use snake_case for file names
    - Give descriptive names: `create_user.zpl` instead of `scenario1.zpl`
3. **Split large scenarios**:
    - Large scenarios are better split into several smaller ones
    - Use common environment variables

Variable Management

1. **Use meaningful variable names**:
    - `user_id` instead of `id`
    - `access_token` instead of `token`
2. **Group related variables in the environment**:

```
environment:
  # API configuration
  base_url: https://api.example.com
  api_version: v1

  # User data
  username: test_user
  password: test_password
```

3. **Define reusable values as variables**:
    - Use variables for recurring values
    - This simplifies maintenance and updating of scenarios

Scenario Debugging

1. **Start with simple scenarios**:
    - Create and debug a simple scenario with a single step
    - Gradually add more steps and complexity
2. **Enable detailed logging**:
    - Set the logging level to debug or `trace` for debugging
    - Check logs to identify errors and issues
3. **Check variables**:
    - Add intermediate steps to check variable values
    - Use conditions to verify expected values
4. **Isolate problems**:
    - When an error occurs, isolate the problematic step
    - Test the request manually using Zaplet commands

This guide covers the main aspects of writing scenarios for Zaplet. As you use the tool, you will discover additional features and techniques that you can apply to test your API._code: 200 condition: token != ""

````
### CRUD Operations

```yaml
name: CRUD Operations
description: Create, read, update, and delete a resource
repeat: 1
continue_on_error: false

environment:
  base_url: https://api.example.com
  auth_token: my_test_token

steps:
  - name: Create resource
    description: Create a new resource
    request:
      method: POST
      url: ${base_url}/resources
      headers:
        Authorization: Bearer ${auth_token}
        Content-Type: application/json
      body: '{"name": "Test Resource", "description": "Test description"}'
    variables:
      resource_id: $.id

  - name: Retrieve resource
    description: Check the created resource
    request:
      method: GET
      url: ${base_url}/resources/${resource_id}
      headers:
        Authorization: Bearer ${auth_token}
    expected_response:
      status_code: 200

  - name: Update resource
    description: Change resource data
    request:
      method: PUT
      url: ${base_url}/resources/${resource_id}
      headers:
        Authorization: Bearer ${auth_token}
        Content-Type: application/json
      body: '{"name": "Updated Resource", "description": "Updated
description"}'
    expected_response:
      status_code: 200
````
````

```yaml
- name: Verify update
  description: Check the updated data
  request:
    method: GET
    url: ${base_url}/resources/${resource_id}
    headers:
      Authorization: Bearer ${auth_token}
  expected_response:
    status_code: 200
  variables:
    resource_name: $.name

- name: Delete resource
  description: Delete the created resource
  request:
    method: DELETE
    url: ${base_url}/resources/${resource_id}
    headers:
      Authorization: Bearer ${auth_token}
  expected_response:
    status_code: 204
  condition: resource_name == "Updated Resource"
```

Pagination and List Processing

```yaml
name: Pagination and List Processing
description: Example of working with pagination and processing data lists
repeat: 1
continue_on_error: false

environment:
  base_url: https://api.example.com
  auth_token: my_test_token

steps:
  - name: First page
    description: Getting the first page of data
    request:
      method: GET
      url: ${base_url}/items?page=1&limit=10
      headers:
        Authorization: Bearer ${auth_token}
    variables:
      total_pages: $.meta.total_pages
      first_item_id: $.data[0].id

  - name: Second page
    description: Getting the second page of data
    request:
      method: GET
      url: ${base_url}/items?page=2&limit=10
      headers:
        Authorization: Bearer ${auth_token}
    condition: total_pages > 1
    variables:
      second_page_item_id: $.data[0].id

  - name: Retrieve specific item
    description: Getting data for a specific item
    request:
      method: GET
      url: ${base_url}/items/${first_item_id}
      headers:
        Authorization: Bearer ${auth_token}
    expected_response:
      status_code: 200
```

Multi-step Business Process

```yaml
name: Multi-step Business Process
description: Example of simulating a business process via API
repeat: 1

environment:
  base_url: https://api.example.com
  auth_token: my_test_token
```

```
steps:
  - name: Create order
    description: Initialize a new order
    request:
      method: POST
      url: ${base_url}/orders
      headers:
        Authorization: Bearer ${auth_token}
        Content-Type: application/json
      body: '{"customer_id": "cust123", "items": [{"product_id": "prod1",
"quantity": 2}]}'
    variables:
      order_id: $.order_id

  - name: Add shipping address
    description: Specify shipping address for the order
    request:
      method: PUT
      url: ${base_url}/orders/${order_id}/shipping
      headers:
        Authorization: Bearer ${auth_token}
        Content-Type: application/json
      body: '{"address": "123 Main St", "city": "Example City", "postal_code":
"12345"}'
    expected_response:
      status_code: 200

  - name: Select payment method
    description: Add payment information
    request:
      method: PUT
      url: ${base_url}/orders/${order_id}/payment
      headers:
        Authorization: Bearer ${auth_token}
        Content-Type: application/json
      body: '{"payment_method": "credit_card", "card_last4": "1234"}'
    expected_response:
      status_code: 200

  - name: Confirm order
    description: Finalize the order
    request:
      method: POST
      url: ${base_url}/orders/${order_id}/confirm
      headers:
        Authorization: Bearer ${auth_token}
    expected_response:
      status_code: 200
    variables:
      status: $.status
```

```
- name: Check order status
  description: Verify successful order creation
  request:
    method: GET
    url: ${base_url}/orders/${order_id}
    headers:
      Authorization: Bearer ${auth_token}
  expected_response:
    status
```