

# API Automation Testing with Cucumber.js

This API Automation Framework is built in BDD approach using Cucumber.js.

## Cucumber.js

Cucumber.js is JavaScript & Node.js implementation of Behaviour Driven Development test framework. Cucumber.js uses Gherkin language for describing the test scenarios in BDD manner.

This framework validates the API response received from the server with the expected values in the data file and also has the capability to validate the same in the database.

## Cucumber best practices

To work with Cucumber, we need following files:

- **Feature file:** text file where the acceptance criteria are written in Gherkin format (**Given, When, Then**). These acceptance criteria could be seen as the tests we are going to prepare.
- **Step Definition:** files in the programming language where Cucumber will be able to associate what actions to execute associated with each step of each acceptance criterion defined in the different features.
- **Others:** These are the helper files which contain the core programming logic. Methods written in this files are called in Step Definition files which helps in achieving the required functionality.

Following practices should be followed while writing the cucumber features:

### 1. Write declarative features

Scenarios should be written like a user would describe them. Do not write programming details in scenarios.

## 2. Avoid conjunctive steps

When you encounter a Cucumber step that contains two actions conjuncted with an “and”, you should probably break it into two steps. Sticking to one action per step makes your steps more modular and increases reusability.

## 3. Reuse step definitions

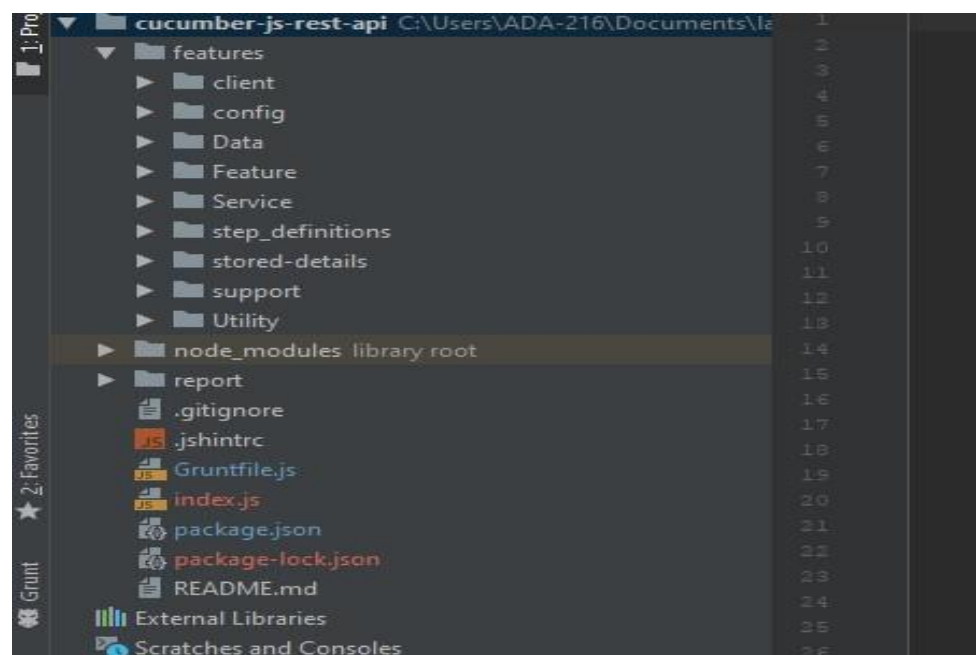
In Cucumber you can reuse steps in other steps. This comes in handy when a step extends another step’s behaviour or defines a superior behaviour that consists of multiple steps. You should try to reuse steps as often as possible. This will improve the maintainability of your app: If you need to change a certain behaviour, you just need to change a single step definition.

## 4. Use backgrounds wisely

If you use the same steps at the beginning of all scenarios of a feature, put them into the feature’s Background. Background steps are run before each scenario. But take care that you don’t put too many steps in there as your scenarios may become hard to understand

## Folder structure

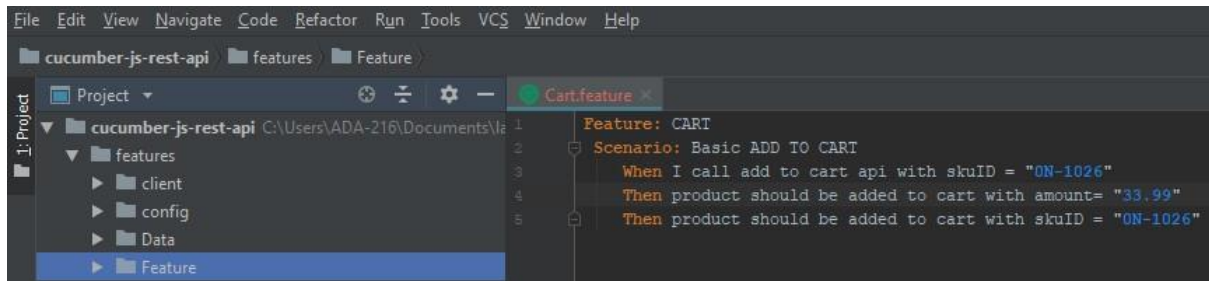
Fig. below shows the default folder structure of codebase.



## 1. Feature

“Feature” is the folder where the test are to be placed followed by the specific folder. This contains the cucumber .feature files in Gherkin language.

Fig. below shows “Cart.feature” file written using Gherkin language.



## 2. Data

“Data” folder contains the various data files for the test in JSON format.

## 3. Utility

“Utility” folder contains helpers to read/write config files and files which helps in parsing JSON objects.

## 4. Support

“Support” folder contains environment details on which we have to test the API

## 5. Config

“Config” folder contains various database and other folder path configurations.

## 6. Stored-Details

“Stored-Details” folder contains the user’s details which are registered in the system through API call.

## 7. Service

“Service” folder contains services for API calls, Authentication and Database connections.

## 8. Step-definations

“step-definations” folder contains functions written in JavaScript for each step in feature files.

## 9. Report

“report” folder contains the reports in HTML and JSON format for the last execution.

## 10. logs

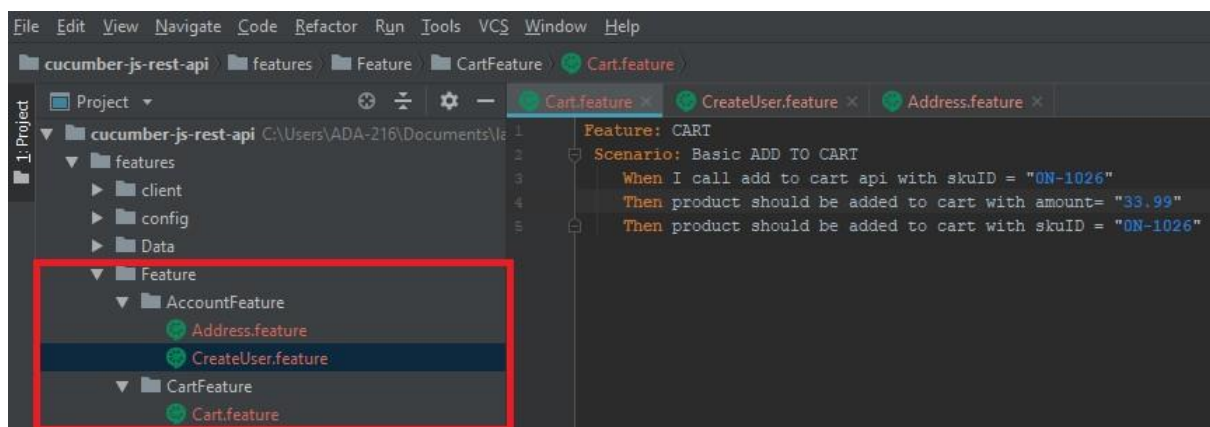
“logs” folder contains the log files generated by framework. These logs contain all the request/response for API and the other utility logs.

## Feature Covered for Demo

The test flow that we have covered for demo are

1. CreateUser.feature
2. RegisterDuplicateUser.feature
3. Address.feature
4. Cart.feature

Fig. below shows features file.



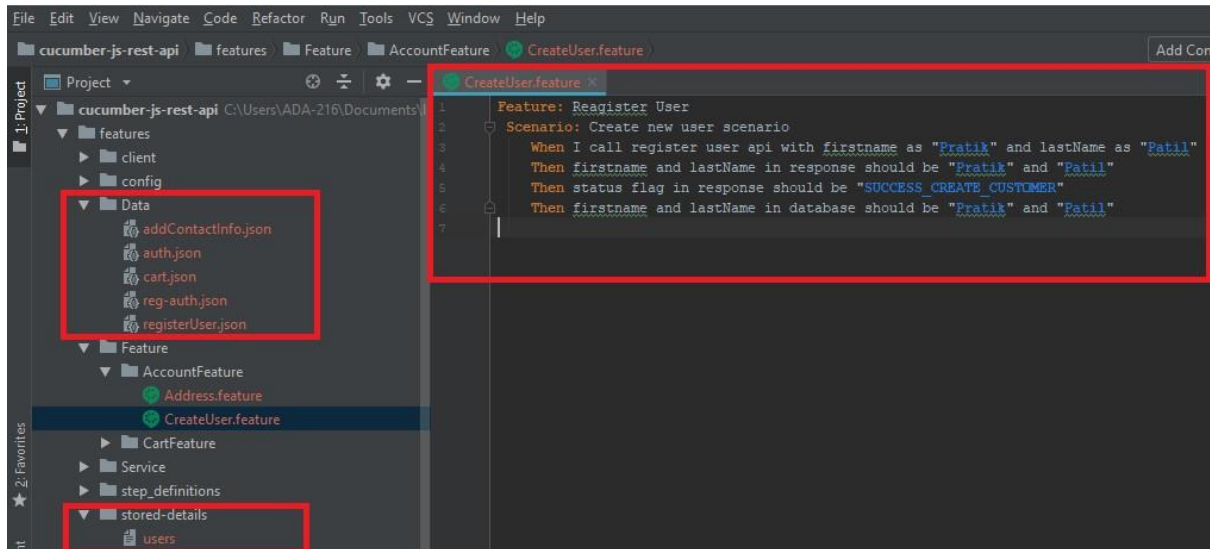
## Feature 1. CreateUser.feature

This is CreateUser.feature file contains test scenario as

- Register new user by calling register user post api with payload
- Then validate response data with “registerUser.json” file present in Data directory.
- Also new user created is stored in “users” file present in stored-details directory.
- Then validate the response status flag.

- Then validate the response data with database data.

Fig. below shows “CreateUser.feature” file, Data directory and stored-details directory.



## Database Validation

The API Response is also validated in the database with automated SQL Queries. Provide the basic information in the config file as below and the script would then connect to the configured database and will bring the result data with the query generated.

Fig. Below is the file /config/database.js where the table names are defined. So in configuration we only specify the collection name instead of complete table name.

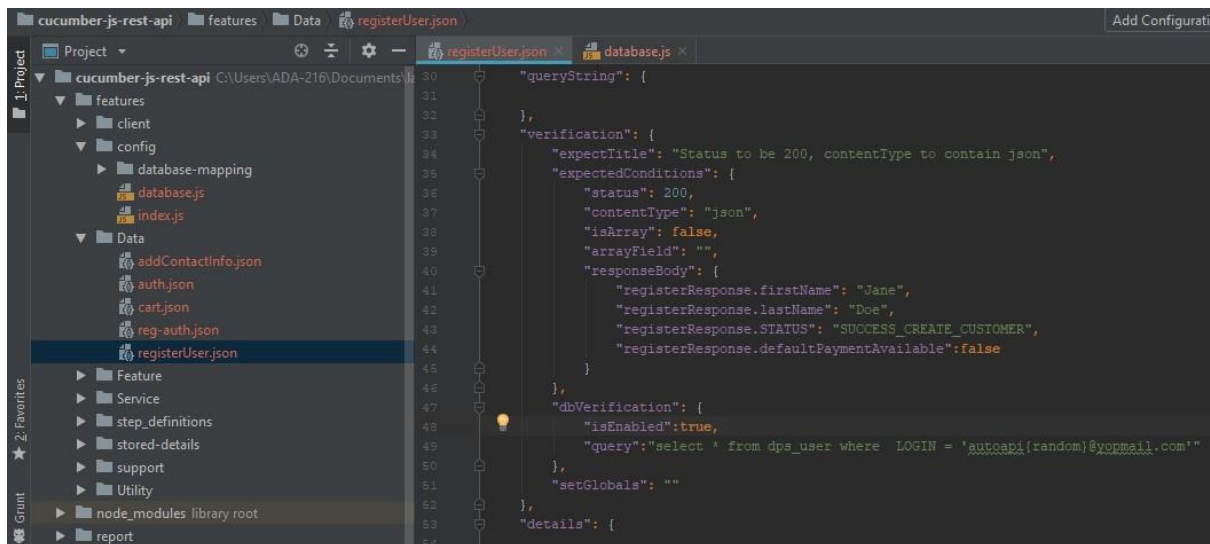
```

databaseCreds: {
  user      : "core"
  password  : "core"
  connectionString : "vsi-dev-chi-db-01-p.int.sparkred.com:1521/vsidevl.sparkred.com"
},

dbName: "CORE" // DB name or schema for QA1
/**
 * [tableConfig Table/Collection names for of the entities]
 * @type {Object}
 */
tableConfig: {
  /**
   * [profile Table/Collection name of profile in database]
   * @type (String)
   */
  profile: "DPS_USER"
  address: "DPS_CONTACT_INFO"
  otherAddress: "DPS_OTHER_ADDR"
  userAddress: "DPS_USER_ADDRESS"
  product: ""
  orders: ""
  item: "DCSPP_ITEM"
  order: "DCSPP_ORDER"
}

```

Fig. Below is the code where dbVerification is set to true.



```

{
  "queryString": {
    "query": "select * from dps_user where LOGIN = 'autoapi{random}@yopmail.com'"
  },
  "verification": {
    "expectTitle": "Status to be 200, contentType to contain json",
    "expectedConditions": [
      {
        "status": 200,
        "contentType": "json",
        "isArray": false,
        "arrayField": "",
        "responseBody": {
          "registerResponse.firstName": "Jane",
          "registerResponse.lastName": "Doe",
          "registerResponse.STATUS": "SUCCESS_CREATE_CUSTOMER",
          "registerResponse.defaultPaymentAvailable": false
        }
      }
    ]
  },
  "dbVerification": {
    "isEnabled": true,
    "query": "select * from dps_user where LOGIN = 'autoapi{random}@yopmail.com'"
  },
  "setGlobals": ""
}

```

## Execution of test

'npm test' to run tests.

Fig. Below shows the execution output on console for registerUser.json test.

```

2020-01-03T15:44:59.893 [INFO] default - Request body { firstName: 'Pratik',
lastName: 'Patil',
email: 'autoapi1578046499812@yopmail.com',
zipCode: '07013',
phoneNumberType: 'HOME',
phoneNumber: '878001136',
password: 'password',
confirmPassword: 'password' }
2020-01-03T15:45:01.238 [INFO] default - Api Response for url https://qa1.vitaminshoppe.com/rest/model/vitaminshoppe/ca/actor/VSIProfileActor/register
{ "registerResponse": { "lastName": "Patil", "profileImagePath": "", "profileId": "u987562600", "thread1": null, "customerNumber": "827219172", "defaultStore": "", "emailOptInFlag": "0", "zipCode": "07013", "defaultPaymentAvailable": false, "thread2": "190", "firstName": "Pratik", "STATUS": "SUCCESS_CREATE_CUSTOMER" } }
..[2020-01-03T15:45:03.797] [INFO] default - Connection successful
2020-01-03T15:45:03.799 [INFO] default - QUERY is ::::: select * from dps_user where LOGIN = 'autoapi1578046499812@yopmail.com'
2020-01-03T15:45:04.082 [INFO] default - db response is [
{
  "ID": "u987562600",
  "LOGIN": "autoapi1578046499812@yopmail.com",
  "AUTO_LOGIN": 1,
  "PASSWORD": "5f4dcc3b5aa765d61d8327deb882cf99",
  "MEMBER": 0,
  "FIRST_NAME": "Pratik",
  "MIDDLE_NAME": null,
  "LAST_NAME": "Patil",
  "USER_TYPE": 0,
  "LOCALE": null,
  "LASTACTIVITY_DATE": "2020-01-02T23:45:01.137Z",
  "REGISTRATION_DATE": "2020-01-02T23:45:00.615Z",
  "EMAIL": "autoapi1578046499812@yopmail.com",
  "EMAIL_STATUS": 0,
  "RECEIVE_EMAIL": 1,
  "LAST_EMAILED": null,
  "GENDER": 0,
  "DATE_OF_BIRTH": null,
  "SECURITYSTATUS": null,
  "DESCRIPTION": null,
  "LASTPWDUPDATE": "2020-01-02T23:45:00.617Z",
  "GENERATEDPWD": 0,
  "PASSWORD_SALT": null,
  "REALM_ID": null
}
]
}

```

## Parameterization

Cucumber supports Data Driven Testing using **Scenario Outline** and **Examples** keywords. Creating a feature file with **Scenario Outline** and **Example** keywords will help to reduce the code and testing multiple scenarios with different values.

### Scenario Outline

This keyword lets you run the same scenario for two or more different input data. It basically replaces value assigned in the variable from the input values mentioned in the **Examples** input data set. Each row in the input data set acts as a scenario.

### Examples

All scenario outlines are followed by **Examples** part that contains the set of data that has to be passed during the execution of tests.

Fig. Below feature file show the parameterizations.

```
features > Feature > CartFeature > Cart.feature
1 Feature: CART
2 Scenario Outline: Basic ADD TO CART
3   When I call add to cart api with skuID = "<skuID>"
4   Then product should be added to cart with amount= "<amount>" and quantity "<quantity>"
5   Then product should be added to cart with skuID = "<skuID>"
6
7   Examples:
8   | skuID | amount | quantity |
9   | 0N-1026 | 33.99 | 1 |
10  | VTP0003 | 42.99 | 2 |
```

On running the above scenario, “skuID” and “amount” will get filled with new values each time till the Examples has values; this test will run for two times as it has two test data.

Instead of hard coding the test data, variables are defined in the Examples section and used in Scenario Outline section.

## Tools Used

- Cucumber
- Request (To make API calls)
- Chai (Assertions)
- Cucumber-Html-Reported (For Reporting)
- Oracledb (To access database)

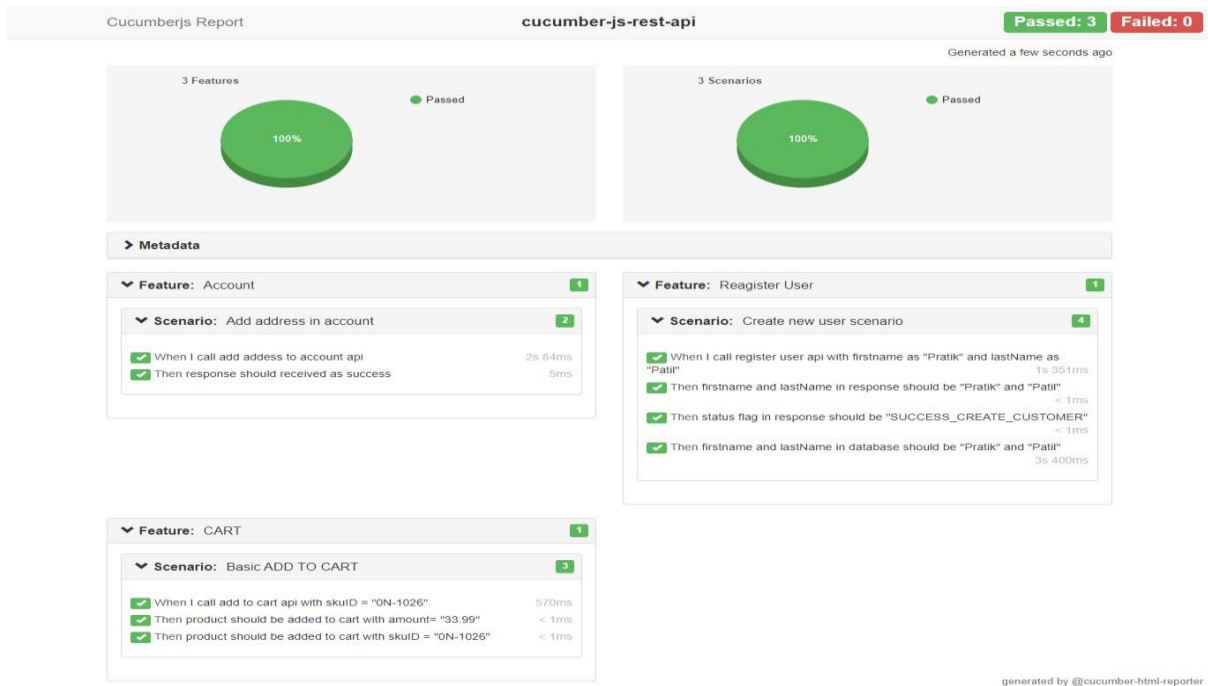
## Reporting

Cucumber uses reporter plugins/modules to produce reports that contain information about what scenarios have passed or failed.

For this project we are using “cucumber-html-reporter” npm modules for reports.



# Cucumberjs Html Report



## Limitations

- Currently this framework supports REST API. If there is a requirement to work with SOAP API then we have to develop it separately.
- Currently this framework is not supporting hiptest. We will have to do lot of rework to make this compatible with hiptest.

## Known issues

- Currently ordering of .feature file execution is achieved by specifying the sequence in package.json. Any newly added .feature file will required to be added in package.json. In future we need to find a better way to order the .feature file execution.
- Since our protractor UI framework is not written in cucumber BDD format we cannot merge this framework with UI framework.

## Prerequisites

This project requires recent version of the Node.js installed in system.

To develop any API feature it requires a hosted API with details of request/response parameters.

## Setup Details

To install and run the tests, follow below steps,

1. Download cucumber-js-rest-api project  
`git clone https://github.com/vitaminshoppe/BDD-API-AUTOMATION.git`
2. npm install
3. npm test to run tests