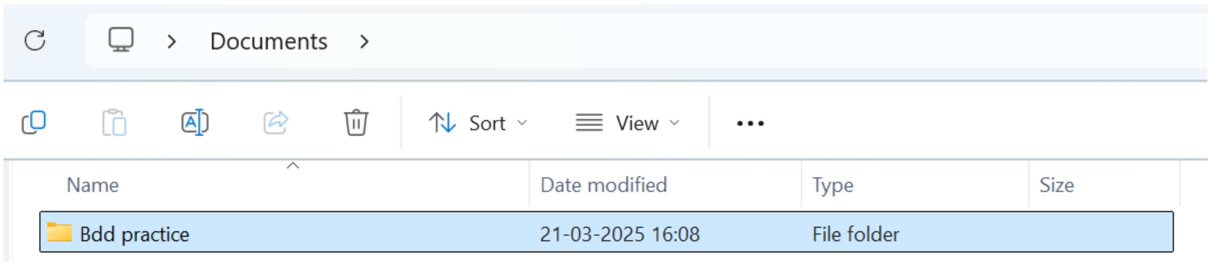
BDD Integration in Playwright with TypeScript

Step 1: Create a Project

1. Create a Folder

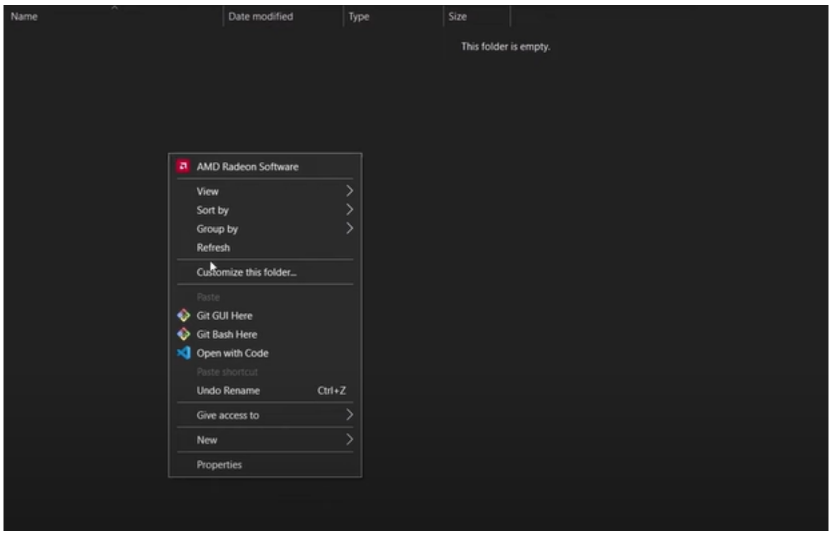
- Create a folder under **Documents** and name it BDD Practice.



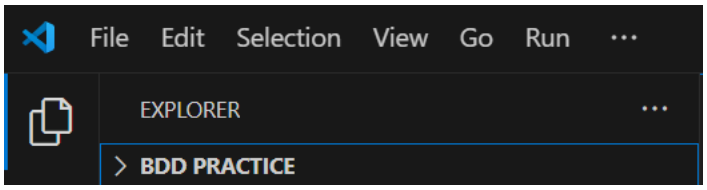
Name	Date modified	Type	Size
Bdd practice	21-03-2025 16:08	File folder	

2. Open the Folder with VS Code

- Open the folder in **VS Code**.



- The empty folder will appear in VS Code.

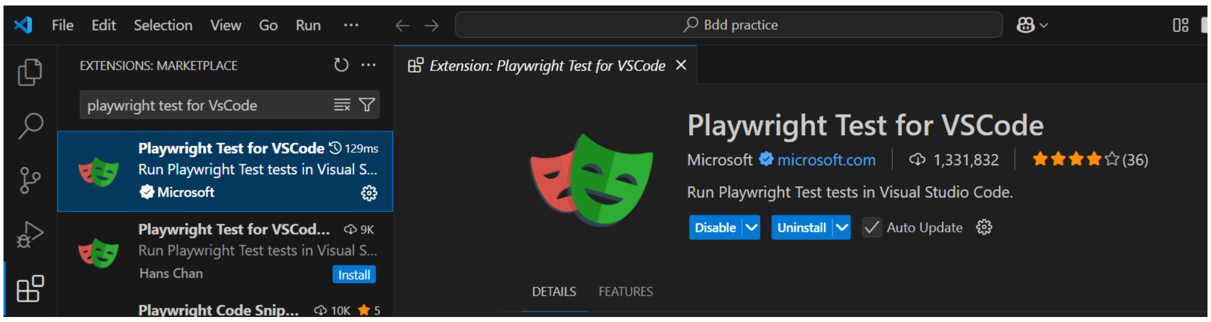


Created by: Yogesh Pandian

Step 2: Create a Playwright Project

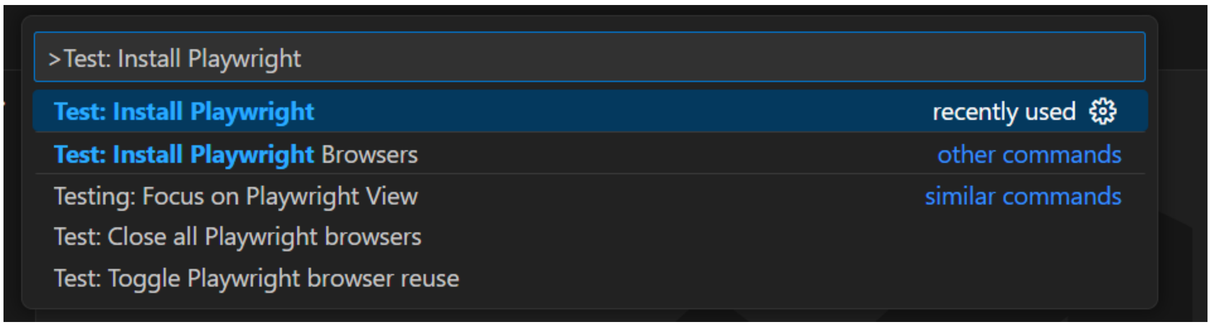
1. Install the Playwright Extension

- In **VS Code**, navigate to the **Extensions** tab and search for **"Playwright Test for VS Code"**.
- Install the extension.

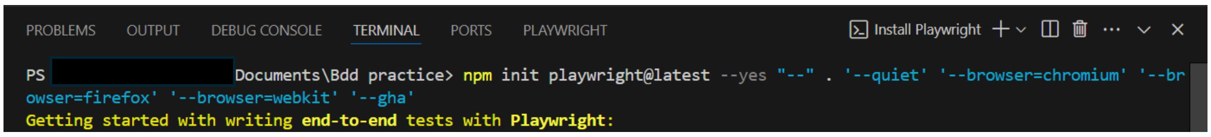

File
Edit
Selection
View
Go
Run
Bdd practice
EXTENSIONS: MARKETPLACE
playwright test for VsCode
Playwright Test for VSCode 129ms
Run Playwright Test tests in Visual S...
Microsoft
Playwright Test for VSCod... 9K
Run Playwright Test tests in Visual S...
Hans Chan
Install
Playwright Code Snip... 10K 5
Extension: Playwright Test for VSCode
Playwright Test for VSCode
Microsoft microsoft.com 1,331,832 (36)
Run Playwright Test tests in Visual Studio Code.
Disable
Uninstall
Auto Update
DETAILS
FEATURES

2. Install Playwright and Related Dependencies

- Press **Ctrl+Shift+P** to open the Command Palette.
- Search for **"Test: Install Playwright"** and select it.
- Choose the following browsers: **Chromium**, **Firefox** and **WebKit**.
- Add the **GitHub Action Workflow**.


>Test: Install Playwright
Test: Install Playwright recently used
Test: Install Playwright Browsers other commands
Testing: Focus on Playwright View similar commands
Test: Close all Playwright browsers
Test: Toggle Playwright browser reuse

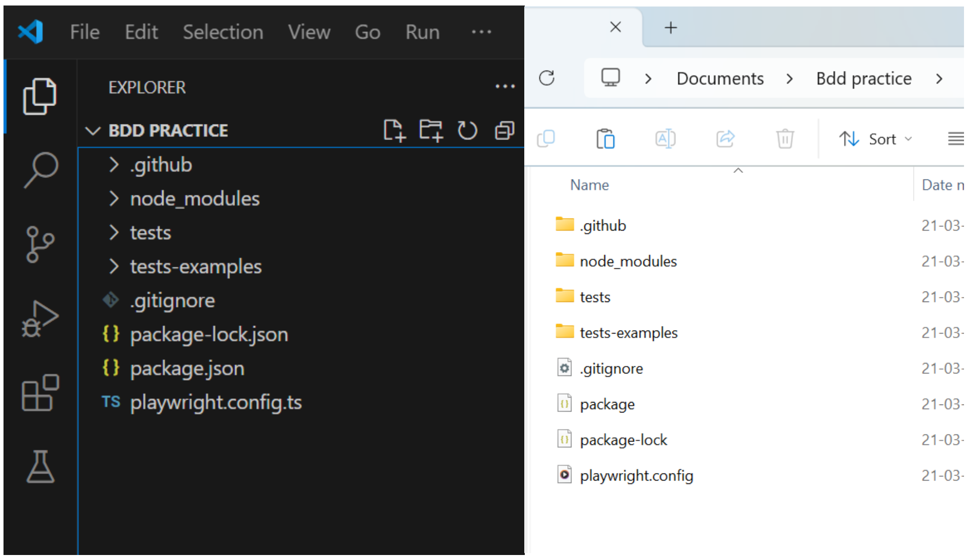
- Once you select all options, click **OK**. The command will auto-populate in the terminal and start the project creation process.
- **npm init playwright@latest --yes "--" . '--quiet' '--browser=chromium' '--browser=firefox' '--browser=webkit' '--gha'**



```
PS [redacted]Documents\Bdd practice> npm init playwright@latest --yes "--" . '--quiet' '--browser=chromium' '--br
owser=firefox' '--browser=webkit' '--gha'
Getting started with writing end-to-end tests with Playwright:
```

Created by: Yogesh Pandian

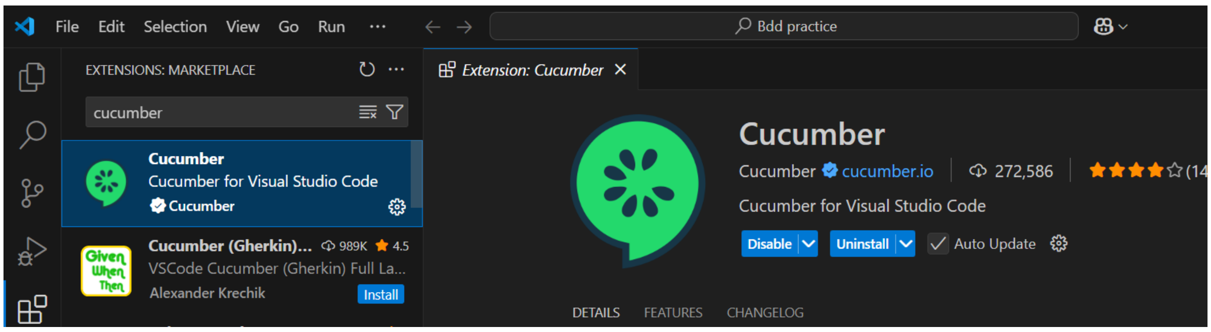
- After the installation, you should see the following folders and files in both **VS Code** and your system project folder:



Step 3: Install Cucumber Plugin

1. Install the Cucumber Extension

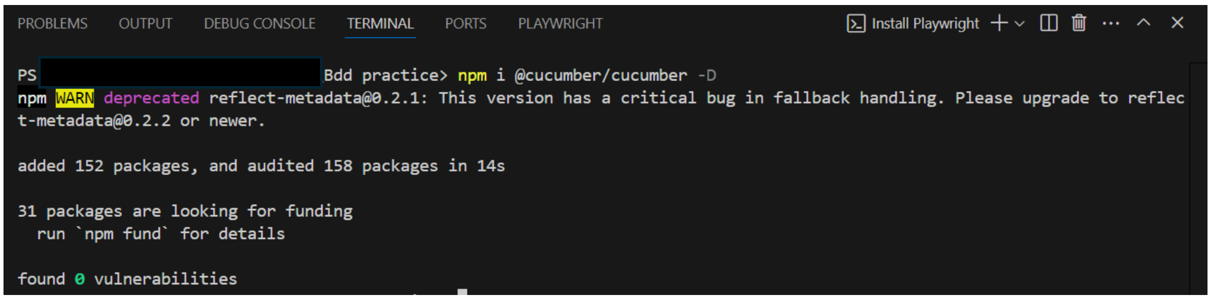
- In **VS Code**, go to the **Extensions** tab and search for **Cucumber**.
- Install the relevant **Cucumber extension** (make sure it's for TypeScript support).



2. Install Cucumber Dependencies

- To install **Cucumber** dependencies, run the following command in the terminal:

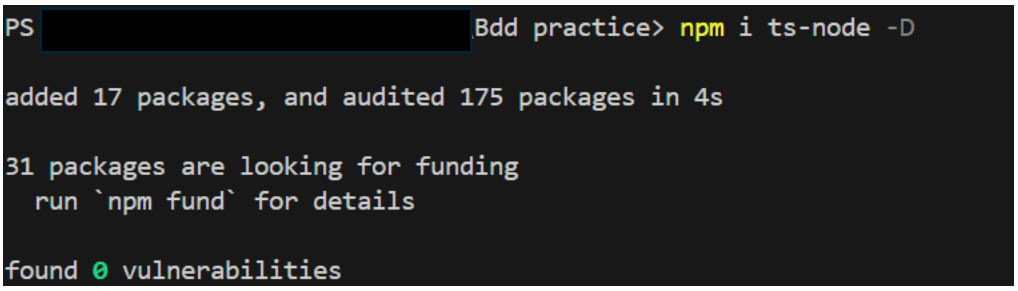
  npm i @cucumber/cucumber -D



```
PS [redacted] Bdd practice> npm i @cucumber/cucumber -D
npm WARN deprecated reflect-metadata@0.2.1: This version has a critical bug in fallback handling. Please upgrade to reflec
t-metadata@0.2.2 or newer.

added 152 packages, and audited 158 packages in 14s

31 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

- Additionally, install **ts-node** for TypeScript execution:

  npm i ts-node -D



```
PS [redacted] Bdd practice> npm i ts-node -D

added 17 packages, and audited 175 packages in 4s

31 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Step 4: TypeScript and Project Configuration (tsconfig.json)

Note: Step 4 is mostly theoretical. If you find it boring, feel free to skip it — just make sure that tsconfig.json is added correctly.

1. Why Do We Need tsconfig.json?

**We use a tsconfig.json file to tell TypeScript how to behave in our project.
It controls:**

- Which version of JavaScript to use.
- How to organize compiled files.
- Which folders to include or ignore.
- Rules to make sure your code is clean and correct.

Without this file, TypeScript won't know how to process your code properly.

---

2. tsconfig.json Explanation

This file controls how TypeScript compiles your .ts files.

Key Points:

- **compilerOptions:** Defines how TypeScript should work.
  - **target:** Converts TypeScript to ES2020 JavaScript.
  - **module:** Uses the CommonJS module system (used by Node.js).
  - **strict:** Enables strict type-checking (helps catch errors early).
  - **esModuleInterop:** Allows better compatibility with import statements.
  - **outDir:** Sets the output folder for compiled files (e.g., dist).
- **include and exclude:**
  - **include:** Tells TypeScript to look inside the src folder.
  - **exclude:** Ignores node_modules to avoid unnecessary processing.

Example tsconfig.json:

```
{
 "compilerOptions": {
  "target": "ES2020",
  "module": "CommonJS",
  "strict": true,
  "esModuleInterop": true,
  "outDir": "dist"
 },
 "include": ["src/**/*"],
 "exclude": ["node_modules"]
}
```

Important:

1. package.json Explanation

This file manages project scripts and dependencies.

Key Parts:

- **scripts:**
    - **Example:** "test": "playwright test" – runs your Playwright tests.
- **dependencies:**
    - **Lists required packages such as:**
        - @playwright/test
        - @cucumber/cucumber
        - ts-node (if used)

To Use It:

- **Run** npm install **to install all required packages.**
- **Run** npm test **to execute the test script.**

- **tsconfig.json → Configures how TypeScript compiles your code.**
- **package.json → Manages your test scripts and libraries.**

Final Steps

Once all the required files and dependencies are installed:

- You are ready to start writing BDD-style tests using Playwright, Cucumber, and TypeScript.
- Begin by creating your feature files and writing step definitions.

Created by: Yogesh Pandian

Summary

Playwright Installation Commands:

1. **npm init playwright@latest --yes "--" . '--quiet' '--browser=chromium' '--browser=firefox' '--browser=webkit' '--gha'**
   - **Purpose**: Initializes a new Playwright project with Chromium, Firefox, and WebKit browsers, and sets up GitHub Action workflows.

Cucumber Installation Commands:

1. **npm i @cucumber/cucumber -D**
   - **Purpose**: Installs **Cucumber** as a development dependency for BDD-style testing.
2. **npm i ts-node -D**
   - **Purpose**: Installs **ts-node** to enable running TypeScript directly in the Node.js environment.

Additional Knowledge: (Ignore this part If you find it boring)

Purpose of npm i ts-node -D (Basics)

Without **ts-node**, you would need to manually compile TypeScript files into JavaScript using the TypeScript compiler (tsc) before running them. However, with **ts-node**, you can run TypeScript files directly in Node.js without the need for manual compilation.

Command:

npm i ts-node -D

Example Use Case: (Basics)

After installing ts-node, you can run TypeScript files directly, which is useful in Playwright tests or other TypeScript-based projects.

1. Using tsc (TypeScript Compiler)

First, let's explain how you would use the TypeScript compiler (tsc) without ts-node.

Steps:

- **Install TypeScript:**

  ```
  npm i typescript -D
  ```

- **Create a TypeScript file (e.g., example.ts)**:

  ```
  console.log("Hello from TypeScript!");
  ```

- **Compile the TypeScript file** using tsc:

  ```
  npx tsc example.ts
  ```

  This generates a JavaScript file (e.g., example.js).

- **Run the compiled JavaScript file** using Node.js:

  ```
  node example.js
  ```

  This will output:

  ```
  Hello from TypeScript!
  ```

2. Using ts-node (Running TypeScript Directly)

Alternatively, if you install **ts-node**, you can run the TypeScript file directly without needing to compile it manually first.

Steps:

- **Install ts-node**:

  ```
  npm i ts-node -D
  ```

- **Create the same TypeScript file (e.g., example.ts)**:

  ```
  console.log("Hello from TypeScript!");
  ```

- **Run the TypeScript file directly** with ts-node:

  ```
  npx ts-node example.ts
  ```

This will output:

```
Hello from TypeScript!
```

Comparison: tsc vs ts-node

- **With tsc:**
  - You need to manually compile the TypeScript code first (**npx tsc example.ts**), then run the generated JavaScript file using Node.js (**node example.js**).
- **With ts-node:** (tsx)
  - You can **skip the manual compilation** step and directly run the TypeScript file (**npx ts-node example.ts**).

Why Install ts-node?

- It **enables you to run TypeScript files directly** in a Node.js environment.
- It's commonly used in projects with **Playwright**, **Cucumber** and other tools that require running TypeScript scripts without needing to manually compile them.

Playwright BDD Project Folder Structure

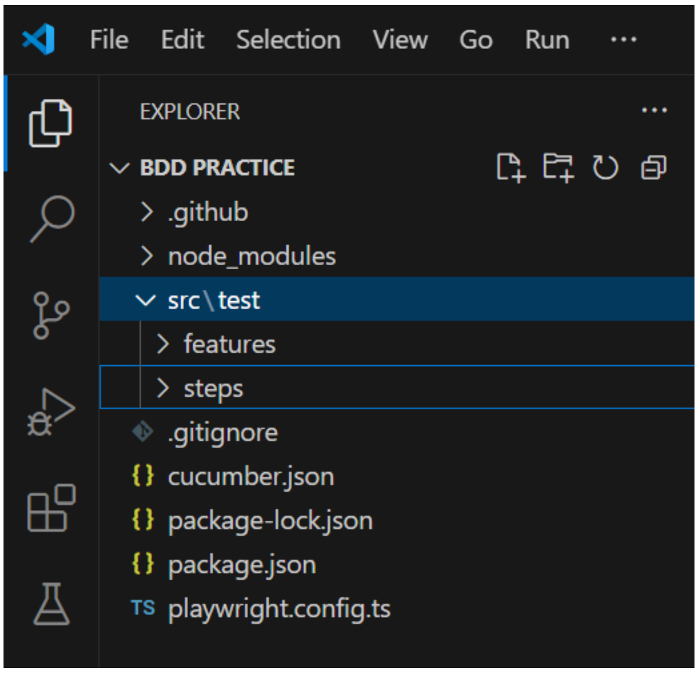
Once the installation is complete, we need to set up the following **project structure**:

Step 1: Create a src/test folder
Inside the **src/test** folder, we will define three layers of Cucumber:

Feature Layer: Where we write our use cases in simple, plain English (Gherkin syntax).

Step Definition Layer: Where we implement the actual code in TypeScript. The step definitions map the steps in the feature file to actual executable code.

Runner Layer: To execute our tests. In Playwright BDD, we will use the runner file in JSON format. Unlike Java, where we have a TestRunner class with @CucumberOptions, we will maintain the options in a **JSON file(Later we will switch to .js will see one by one)**.



```
.github
node_modules
src\test
  features
  steps
.gitignore
cucumber.json
package-lock.json
package.json
playwright.config.ts
```

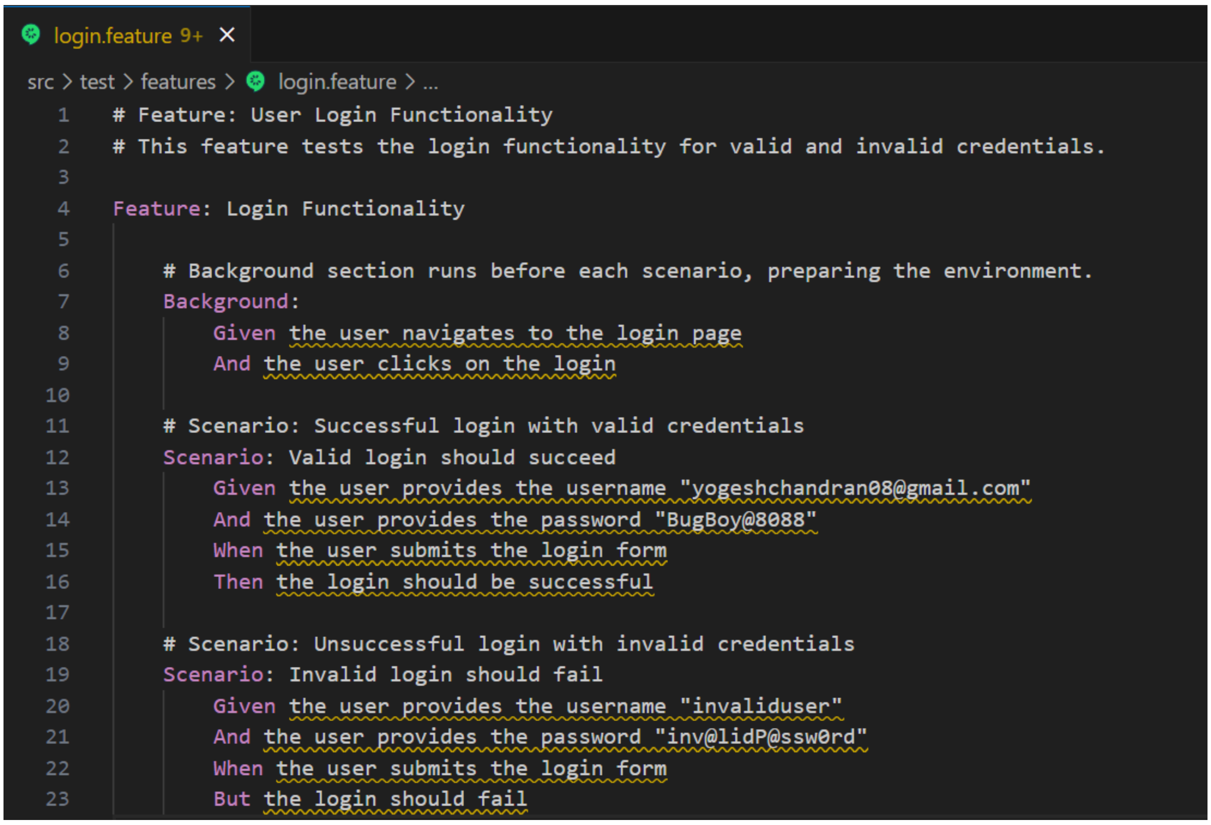
Note: We will **not** create a custom runner. Instead, we will use the default runner provided by the @cucumber/cucumber package.

---

1. Creating the Feature File for Login Functionality

Feature File:

Start by creating a **feature file** for the login functionality. A feature file describes the desired behaviour of the application in Gherkin syntax (plain English).



```
# Feature: User Login Functionality
# This feature tests the login functionality for valid and invalid credentials.

Feature: Login Functionality

    # Background section runs before each scenario, preparing the environment.
    Background:
        Given the user navigates to the login page
        And the user clicks on the login

    # Scenario: Successful login with valid credentials
    Scenario: Valid login should succeed
        Given the user provides the username "yogeshchandran08@gmail.com"
        And the user provides the password "BugBoy@8088"
        When the user submits the login form
        Then the login should be successful

    # Scenario: Unsuccessful login with invalid credentials
    Scenario: Invalid login should fail
        Given the user provides the username "invaliduser"
        And the user provides the password "inv@lidP@ssw0rd"
        When the user submits the login form
        But the login should fail
```

Generating Code for the Feature File:

To generate the code snippets for the feature file, we will need a **runner JSON file**. This will allow us to generate code snippets in normal text for mapping between the feature and the implementation.

Creating the cucumber.json File

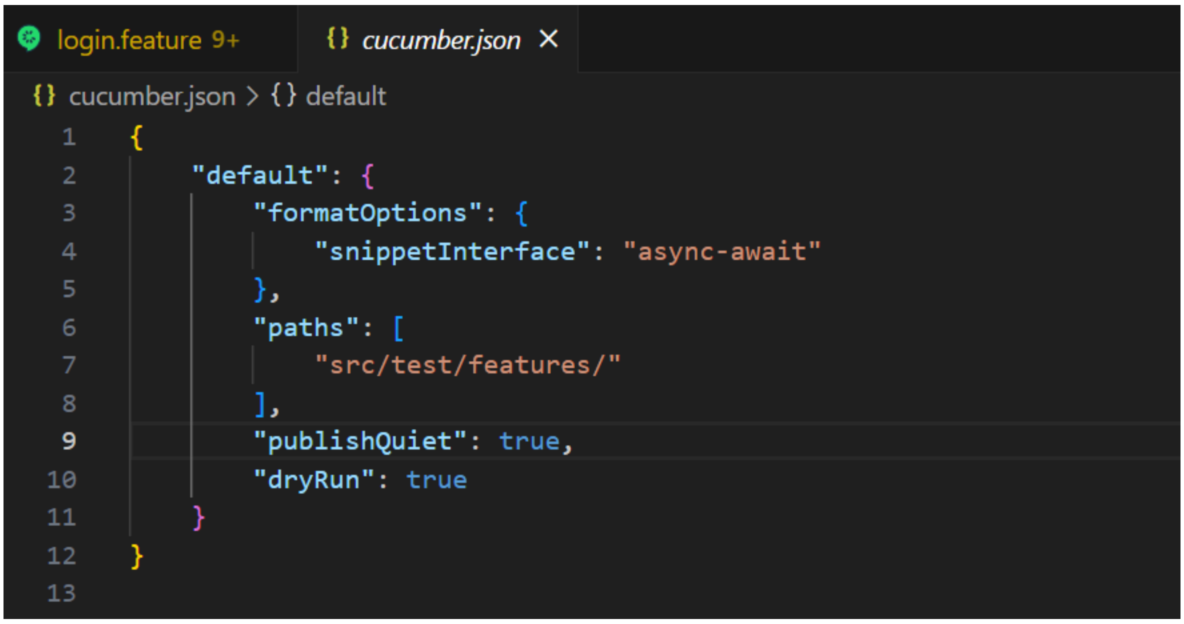
To configure the runner, we need to create a cucumber.json file in the root directory of the project.

Here's an example cucumber.json file:

```
{
 "default": {
  "paths": [
   "src/test/features/"
  ]
 }
}
```

Explanation:

- **default**: This key overrides the default functionality of Cucumber.
- **paths**: This specifies the location of the feature files. The paths should be in an array format. In this case, it points to the src/test/features/ directory where the feature files are located.



```json
{
    "default": {
        "formatOptions": {
            "snippetInterface": "async-await"
        },
        "paths": [
            "src/test/features/"
        ],
        "publishQuiet": true,
        "dryRun": true
    }
}
```

Configuring the Runner

Now, we need to configure the test runner in the package.json file:

Edit package.json:
Add the following property inside the scripts section of package.json to run Cucumber tests.

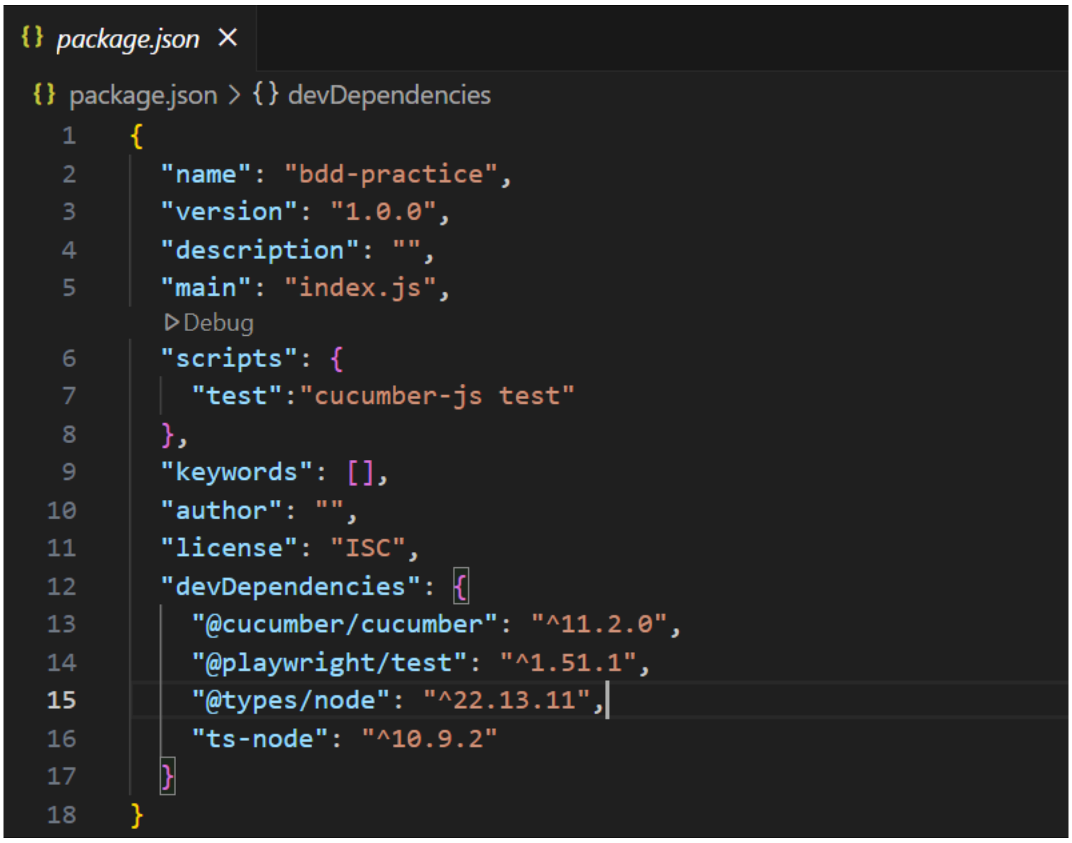
```
"scripts": {
 "test": "cucumber-js test"
 }
```

Created by: Yogesh Pandian

Explanation:

We are using **cucumber-js test** to run the Cucumber tests.

Note: Since we are using TypeScript, we need **ts-node** to convert TypeScript to JavaScript. This is why ts-node is added as a dev dependency in package.json.



package.json > {} devDependencies

```json
{
  "name": "bdd-practice",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test":"cucumber-js test"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@cucumber/cucumber": "^11.2.0",
    "@playwright/test": "^1.51.1",
    "@types/node": "^22.13.11",
    "ts-node": "^10.9.2"
  }
}
```

Generating the Snippet

Now that everything is set up, let's generate the snippet for our feature file:

1. **Run the command**: Open the terminal (Ctrl + J) and run the following command:

   npm test

2. **What happens?**:

   - The npm test command will check for the feature file and its corresponding step definitions.
   - Since we haven't created the step definitions yet, **Cucumber will generate the snippets in the console**.

3. **Snippet Format**:

   - The snippet will be displayed in JavaScript format by default.

- If you wish to use **async functions** (which are necessary in Playwright), you will need to configure the snippet format accordingly.

Customizing the Snippet Format for Playwright

To ensure the snippets are in **async-await** format (since Playwright uses asynchronous functions), we need to make a small change in the cucumber.json file.

Add the following configuration under formatOptions:

```
{
 "default": {
  "formatOptions": {
   "snippetInterface": "async-await"
  },
  "paths": [
   "src/test/features/"
  ],
  "publishQuiet": true,
  "dryRun": true
 }
}
```

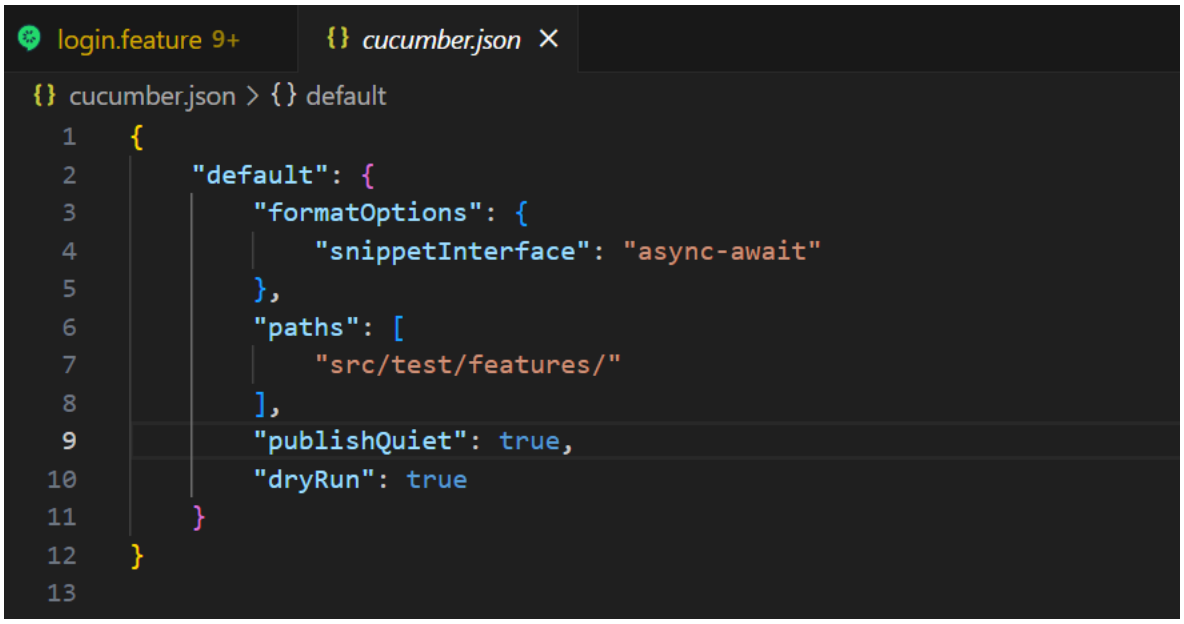
Explanation of New Properties: **(Again Theoertical part. If you feel bored, you can ignore this part but make sure your cucumber.json file should look exactly like this)**

- **snippetInterface:**
  - Set to "async-await" to ensure that the generated snippets use asynchronous functions.
- **publishQuiet:** (Now it is depreicated)
  - Set to true to suppress non-essential output in the terminal.
- **dryRun:**
  - Set to true to run a dry run, meaning it will check the feature and step definitions but not actually execute the tests.

Final cucumber.json Configuration

Here is the final version of the cucumber.json file:

```
{
 "default": {
  "formatOptions": {
   "snippetInterface": "async-await"
  },
  "paths": [
   "src/test/features/"
  ],
  "publishQuiet": true,
  "dryRun": true
 }
}
```



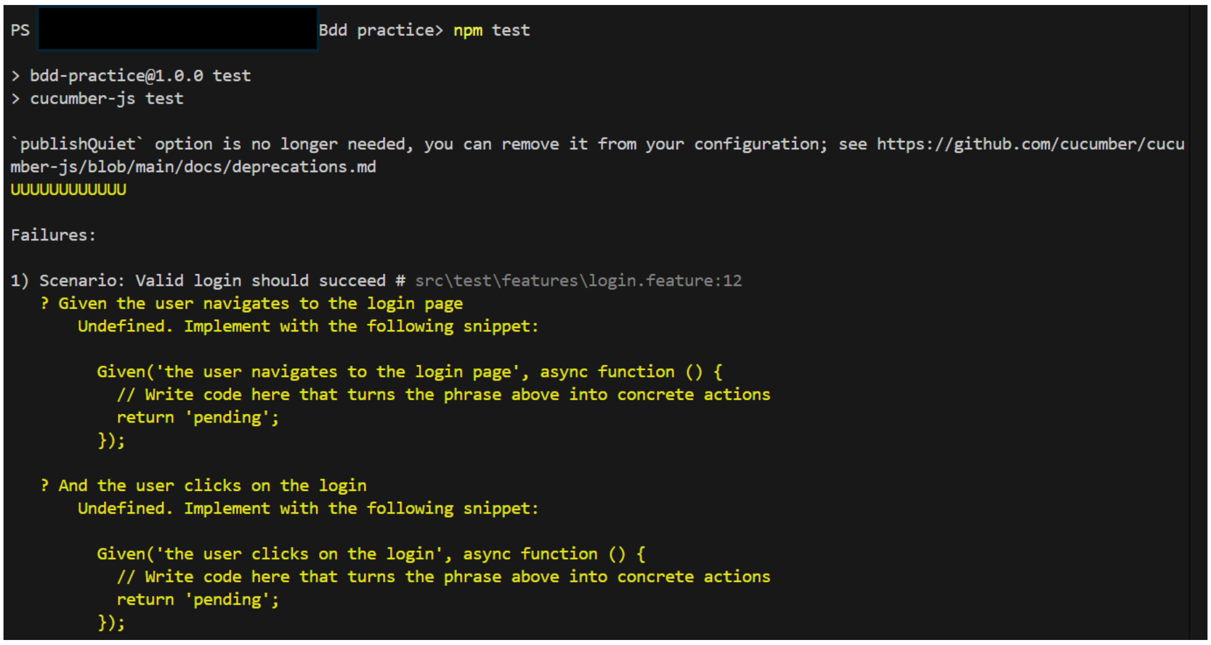
```
{
    "default": {
        "formatOptions": {
            "snippetInterface": "async-await"
        },
        "paths": [
            "src/test/features/"
        ],
        "publishQuiet": true,
        "dryRun": true
    }
}
```

Key Points:

- **paths** specifies the directory where feature files are located.
- **formatOptions.snippetInterface** ensures the snippets are compatible with async functions.
- **publishQuiet** suppresses unnecessary logs.
- **dryRun** helps in verifying the test setup without executing tests.

Execute the test by using the below command:

npm test



```
PS [redacted] Bdd practice> npm test

> bdd-practice@1.0.0 test
> cucumber-js test

`publishQuiet` option is no longer needed, you can remove it from your configuration; see https://github.com/cucumber/cucu
mber-js/blob/main/docs/deprecations.md
UUUUUUUUUUUU

Failures:

1) Scenario: Valid login should succeed # src\test\features\login.feature:12
   ? Given the user navigates to the login page
       Undefined. Implement with the following snippet:

         Given('the user navigates to the login page', async function () {
           // Write code here that turns the phrase above into concrete actions
           return 'pending';
         });

   ? And the user clicks on the login
       Undefined. Implement with the following snippet:

         Given('the user clicks on the login', async function () {
           // Write code here that turns the phrase above into concrete actions
           return 'pending';
         });
```

Copy the snippet from console and we can use the same in stepdefenition.

Summary:

By following the above steps:

- We set up the folder structure for a Playwright BDD project.
- We created a cucumber.json configuration to manage our feature files, step definitions and snippets.
- We configured Playwright with asynchronous code snippets, ensuring compatibility with Playwright's async functionality.

Step 2: Configuring and Implementing Step Definitions for Playwright BDD

Feature File and Path Configuration

At this point, we already have the feature file (login.feature) ready and we have set the path of the feature file in the cucumber.json file. Additionally, in the package.json file, we added the script to run the Cucumber test using cucumber-js test.

By adding ts-node to the package.json file, we ensured that TypeScript files could be converted without using tsc. Running the npm test command generated code snippets for the unimplemented steps from the feature file.

Adding Step Definitions **(loginSteps.ts)**

Now that we have the feature file and generated snippets, we need to implement the step definitions in a separate TypeScript file.

1. **Create loginSteps.ts**:
   - This file will hold the actual implementation of the steps defined in the feature file.
2. **Add the Snippets to loginSteps.ts**:
   - Copy the generated code snippets from the terminal after running npm test and paste them into the loginSteps.ts file.

Updating cucumber.json File for Step Definitions

Up until now, we've only mentioned the path of the feature file in the cucumber.json file. We also need to specify the path of the step definition files.

1. **Add the require Property**:
   - The require property in the cucumber.json file will tell Cucumber where to find the step definitions.

     Here is how to add it:

     ```
     "require": [
       "src/test/steps/*.ts"
     ]
     ```
   - This tells Cucumber to look for TypeScript files in the src/test/steps directory for step definitions.

2. **Add the requireModule Property**:
   - The requireModule property is necessary to solve issues with importing TypeScript files and to handle the TypeScript import statements properly. We need to add the following:

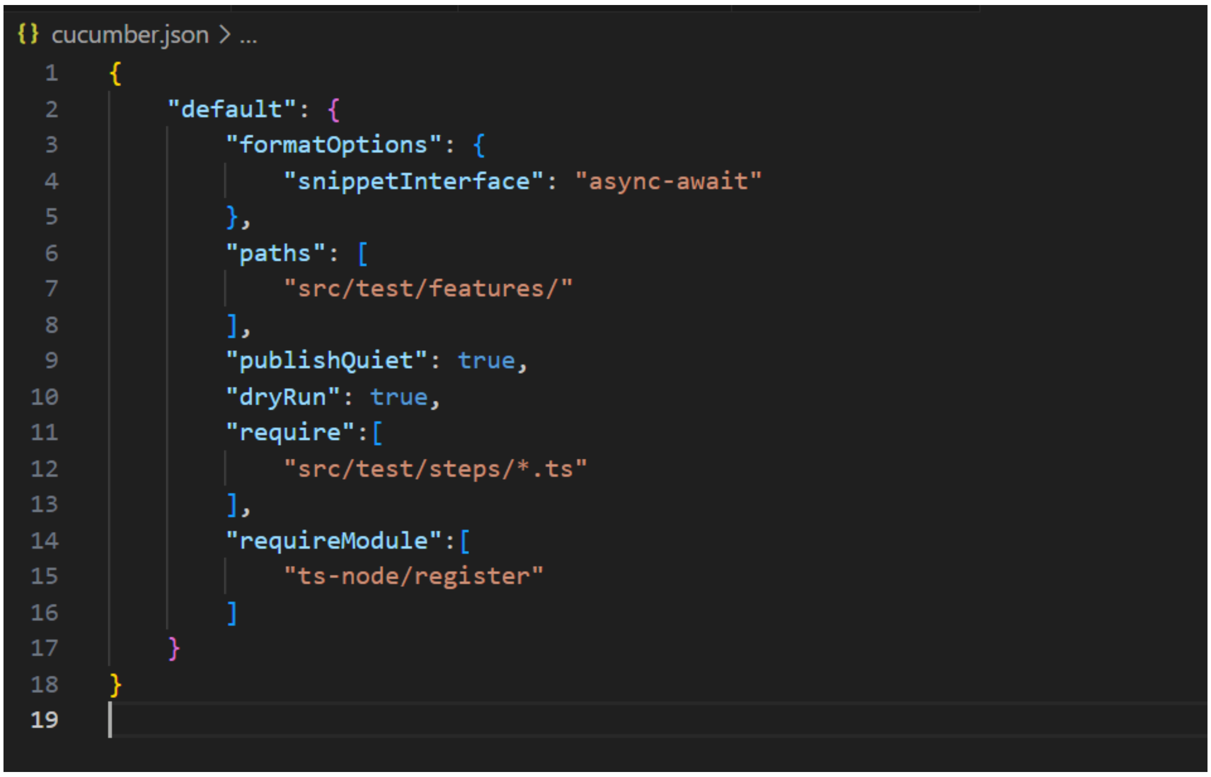
"requireModule": [

 "ts-node/register"

]

Explanation:

- require: Specifies the path of the step definition files.
- requireModule: Registers ts-node to handle TypeScript files and imports.


cucumber.json > ...

```json
{
    "default": {
        "formatOptions": {
            "snippetInterface": "async-await"
        },
        "paths": [
            "src/test/features/"
        ],
        "publishQuiet": true,
        "dryRun": true,
        "require":[
            "src/test/steps/*.ts"
        ],
        "requireModule":[
            "ts-node/register"
        ]
    }
}
```

Final cucumber.json Configuration

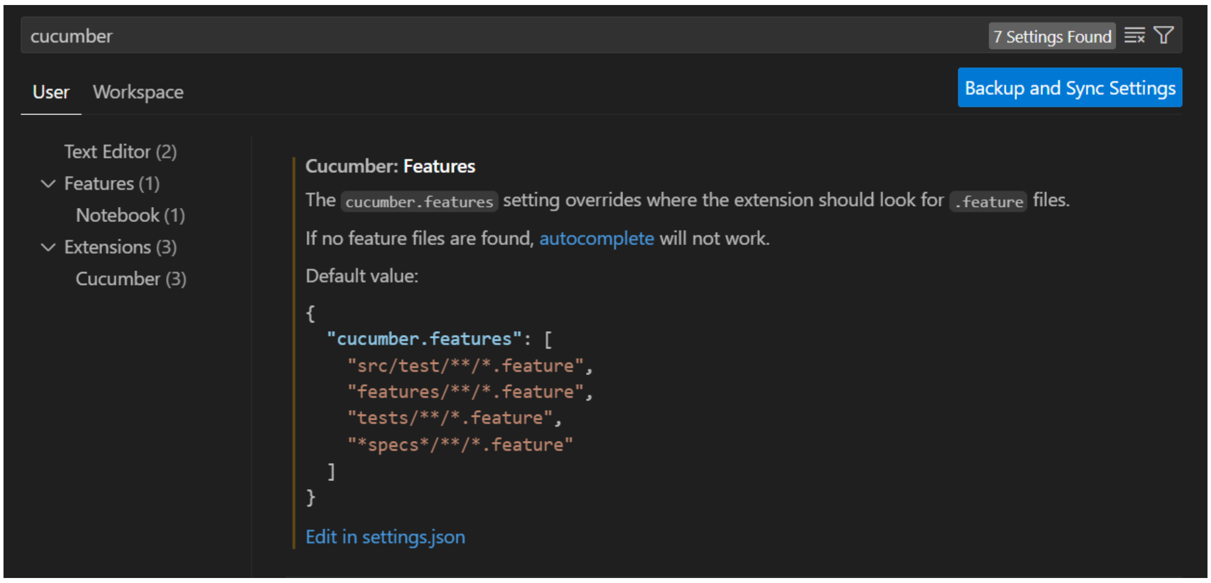
Here's the final version of the cucumber.json file after adding the paths for the step definitions and ts-node registration:

```
{
 "default": {
  "formatOptions": {
   "snippetInterface": "async-await"
  },
  "paths": [
   "src/test/features/"
  ],
  "require": [
   "src/test/steps/*.ts"
  ],
  "requireModule": [
   "ts-node/register"
  ],
  "publishQuiet": true,
  "dryRun": true
 }
}
```

Step 3: Configuring VS Code for Step Definitions and Feature Files

Before proceeding with the implementation of the step definitions, it's important to configure your **VS Code settings** for smooth development.

1. **Open Settings**:
   - Press Ctrl + , to open the settings page in VS Code.
2. **Search for cucumber**:
   - In the settings search bar, type "cucumber."
3. **Edit in settings.json**:
   - Click on **Cucumber** under Extensions and then click **Edit in settings.json**.



Cucumber: **Features**

The `cucumber.features` setting overrides where the extension should look for `.feature` files.

If no feature files are found, autocomplete will not work.

Default value:

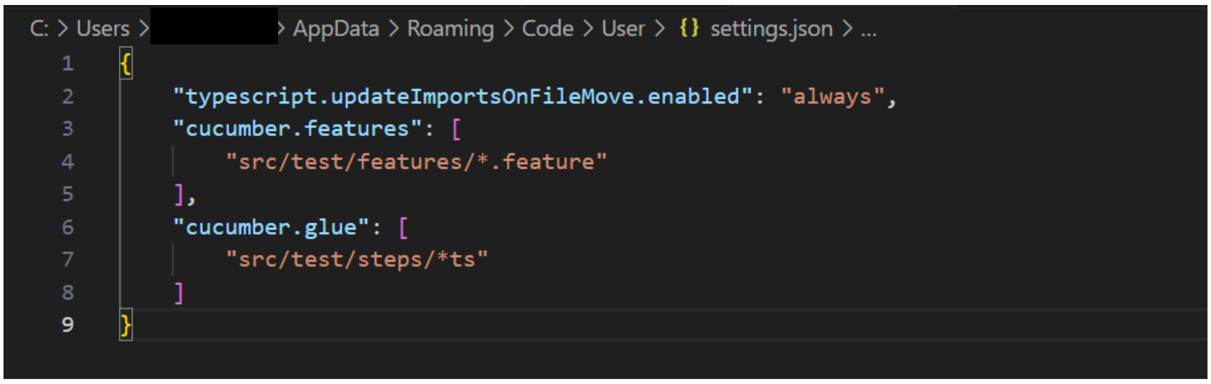
```
{
  "cucumber.features": [
    "src/test/**/*.feature",
    "features/**/*.feature",
    "tests/**/*.feature",
    "*specs*/**/*.feature"
  ]
}
```

Edit in settings.json

4. **Set the Path for Feature and Step Definitions**:
    - In the settings.json file, you need to define the paths of your feature and step definition folders to make it easier for VS Code to detect and work with them.

Example configuration:

```
{
 "cucumber.features": ["src/test/features/*.feature"],
 "cucumber.glue": ["src/test/steps/*.ts"]
}
```



C: > Users > [redacted] > AppData > Roaming > Code > User > {} settings.json > ...

```
{
    "typescript.updateImportsOnFileMove.enabled": "always",
    "cucumber.features": [
        "src/test/features/*.feature"
    ],
    "cucumber.glue": [
        "src/test/steps/*ts"
    ]
}
```

Explanation:

- **cucumber.features**: Specifies the path to the feature files.
- **cucumber.glue**: Specifies the path to the step definition files.

Created by: Yogesh Pandian

});

Step 4: Implementing Step Definitions in loginSteps.ts

Now, let’s implement the step definitions for the login.feature file.

1. Import Gherkin Keywords from @cucumber/cucumber

- In the loginSteps.ts file, you may see errors if Gherkin keywords are not imported.
- Add the following import statement at the top of the file:

import { Given, When, Then } from '@cucumber/cucumber';

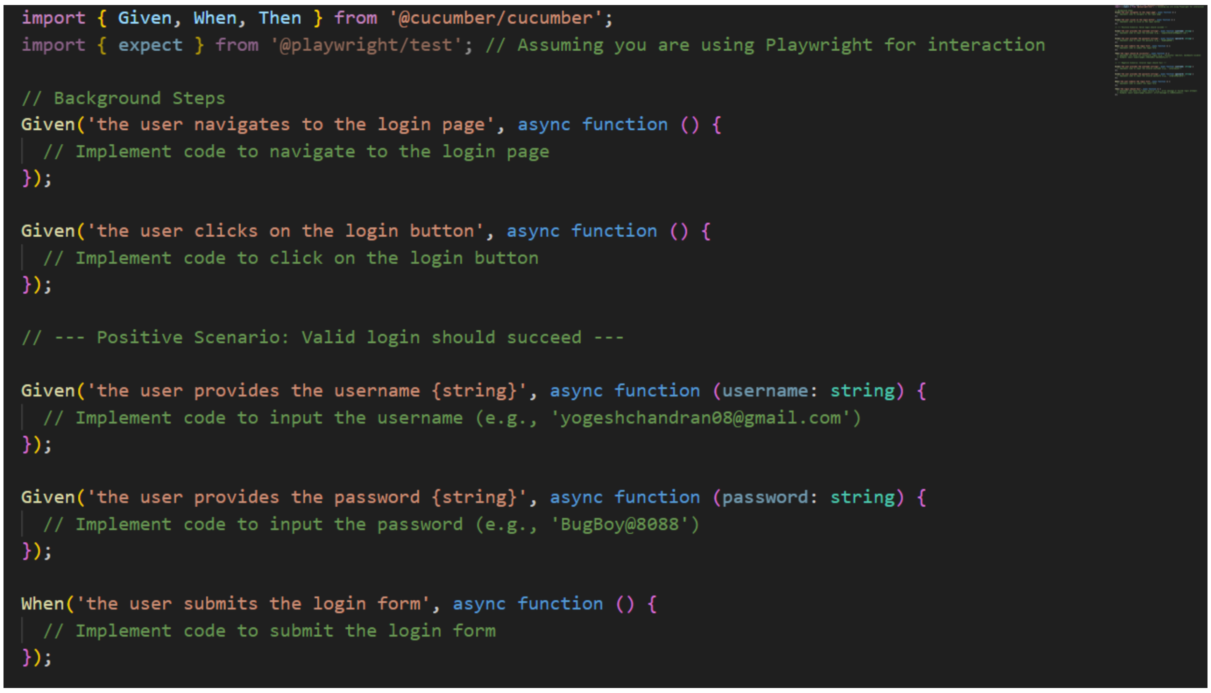
---

2. Implement the Snippets

- Use the auto-generated snippets to implement logic for each step.
- Below is an example of how loginSteps.ts can be implemented:

```
import { Given, When, Then } from '@cucumber/cucumber';

// Background Steps

Given('the user navigates to the login page', async function () {

  // Implement navigation to login page

});

Given('the user clicks on the login button', async function () {

  // Implement click on the login button

});


// Scenario 1: Valid login should succeed

Given('the user provides the username {string}', async function (username: string) {

  // Implement entering the username

});

Given('the user provides the password {string}', async function (password: string) {

  // Implement entering the password

});
```

```
When('the user submits the login form', async function () {
  // Implement form submission
});
Then('the login should be successful', async function () {
  // Implement checking for successful login
});

// Scenario 2: Invalid login should fail
Given('the user provides the username {string}', async function (username: string) {
  // Implement entering the invalid username
});
Given('the user provides the password {string}', async function (password: string) {
  // Implement entering the invalid password
});
When('the user submits the login form', async function () {
  // Implement form submission
});
Then('the login should fail', async function () {
  // Implement checking for failed login
});
```



```
import { Given, When, Then } from '@cucumber/cucumber';
import { expect } from '@playwright/test'; // Assuming you are using Playwright for interaction

// Background Steps
Given('the user navigates to the login page', async function () {
  // Implement code to navigate to the login page
});

Given('the user clicks on the login button', async function () {
  // Implement code to click on the login button
});

// --- Positive Scenario: Valid login should succeed ---

Given('the user provides the username {string}', async function (username: string) {
  // Implement code to input the username (e.g., 'yogeshchandran08@gmail.com')
});

Given('the user provides the password {string}', async function (password: string) {
  // Implement code to input the password (e.g., 'BugBoy@8088')
});

When('the user submits the login form', async function () {
  // Implement code to submit the login form
});
```

Running the Tests

After implementing the step definitions:

1. Open your terminal.
2. Run the command npm test to execute your Playwright BDD tests.

Cucumber will:

- **Find the feature files.**
- **Find the step definitions.**
- **Execute the tests and generate the necessary output.**

Conclusion

At this point:

- We have set up the **folder structure** with feature files, step definitions, and the required configuration.
- The **cucumber.json** file was updated to include paths for step definitions and module registration for TypeScript.
- The **step definitions** were implemented for the login functionality.

Created by: Yogesh Pandian

- The **VS Code settings** were updated to ensure that the editor is correctly configured for the Playwright BDD setup.

You are now ready to run your Playwright BDD tests and expand on your test scenarios!

Step 5: Writing Playwright Code in Step Definition File (loginSteps.ts)

Now let's start writing our Playwright code in the step definition file.

1. Import Playwright

At the beginning of your file, import the necessary Playwright components:

import { chromium, Page, Browser } from "@playwright/test";

What each component means:

- **chromium**: The browser type we are using (like Chrome).
- **Page**: Represents a browser tab where the actual interaction happens.
- **Browser**: Represents the browser instance that controls the tab.

---

2. Define Variables for Browser and Page

Use let to define the variables since their values will be assigned later during execution:

let page: Page;

let browser: Browser;

Why use let?

- We use let because the variables are not initialized immediately.
- let allows us to assign values later during test execution.

---

3. Step to Launch the Application

Implement the step to navigate to the login page:

```
Given('the user navigates to the login page', async function () {
  browser = await chromium.launch({ headless: false }); // Launch browser visibly
  page = await browser.newPage();                  // Open a new tab
  await page.goto("http://www.automationpractice.pl/index.php?id_category=3&controller=category");
});
```

---

4. Understanding Page Fixture

- **Page** is like a tab in the browser.
- It lets you perform actions like clicking, typing, and verifying elements on the web page.

---

5. What is async and await?

- **async**: Marks a function as asynchronous (it can run tasks that take time).
- **await**: Waits for the task to complete before moving to the next line.

Why are they used here?

- **async** is used in step definitions because browser actions take time.
- **await** ensures the browser finishes the current task (like loading a page) before continuing.

Step 6: Continue Writing Playwright Steps

Now, with this setup, you're ready to continue writing your Playwright steps!

You can proceed to:

- Click the login button
- Enter credentials
- Verify login status

All of this can be done using the same **page fixture** and **await** wherever needed.

Created by: Yogesh Pandian

Using Assertions in Playwright

Playwright provides a built-in **expect** function for assertions.

Why use expect in Playwright?

- Playwright has **auto-wait** capabilities.
- When using expect, there's **no need for explicit waits**.
- It improves **test stability** and **readability**.

Example:

```
import { expect } from "@playwright/test";


await expect(message).toBeVisible();
```

Problem Statement 1: Duplicate Step Definitions

Scenario:

- You have multiple feature files (e.g., login.feature, addtocart.feature, etc.).
- You also have multiple step definition files (e.g., loginStepDef.ts, addtocartStepDef.ts, etc.).

Problem:

The login steps are repeated in each file, leading to:

  - Code duplication
  - Ambiguous step definitions
  - Maintenance issues

Goal:

Avoid repeating login steps across files while keeping them **accessible** to all feature files.

Solution to Problem Statement 1: Use Hooks and Page Fixture

You can solve this problem by:

- Creating a **hooks file** that contains shared logic.
- Defining **common pre-conditions and post-conditions**.
- Managing a shared **page object** to avoid redefining it in each step file.

This way, you write the login steps **once** and reuse them across all features.

Problem Statement 2: Sharing Page Object Across Tests

Need:

A way to **share variables and logic** like the page object across all feature files.

Solution:

- **Create a hooks folder** under src/test/
  → This handles shared setup (before scenarios) and teardown (after scenarios).
- **Create a pageFixture.ts file**
  → This file exports the shared page object that can be imported in any step definition file.

Step-by-Step Solution for Sharing Page Object and Avoiding Duplication for above problems

1. Create Hooks Folder and Files

Create the following structure in your project:

```
src/
└── test/
    └── hooks/
        ├── hooks.ts
       └── pageFixture.ts
```

---

2. Inside pageFixture.ts

This file defines and exports a pageFixture object that holds the Playwright page. It will be shared across all step definitions.

```
// pageFixture.ts


import { Page } from '@playwright/test';


// Export a shared pageFixture object

export const pageFixture = {

  page: undefined as Page, // Will be initialized in hooks.ts

};
```

Why use undefined for page?

- Playwright will assign the actual page object during test execution.
- The page will be initialized in hooks.ts using BeforeAll.

---

3. Inside hooks.ts

This file handles test setup and teardown. It initializes the page object and makes it available globally via pageFixture.

```
// hooks.ts


import { BeforeAll, AfterAll } from '@cucumber/cucumber';

import { chromium, Page, Browser } from '@playwright/test';

import { pageFixture } from './pageFixture';


let browser: Browser;


BeforeAll(async function () {

  browser = await chromium.launch({ headless: false }); // Launch browser

  const page = await browser.newPage();           // Create new page
  pageFixture.page = page;                    // Assign to fixture
});

AfterAll(async function () {
  if (browser) {
    await browser.close(); // Close browser
  }
});
```

Why use BeforeAll and AfterAll?

- **BeforeAll:** Runs once before all scenarios – sets up the browser and page.
- **AfterAll:** Runs once after all scenarios – ensures proper cleanup.

---

4. Using pageFixture in Step Definitions

Now any step definition file can access the shared page from pageFixture.ts.

Example: loginStepDef.ts

```
// loginStepDef.ts

import { Given } from '@cucumber/cucumber';
import { pageFixture } from '../hooks/pageFixture';

Given('the user navigates to the login page', async function () {
  const page = pageFixture.page;
  await
page.goto('http://www.automationpractice.pl/index.php?id_category=3&controller=category');
});
```

```
Given('the user clicks on the login button', async function () {
  const page = pageFixture.page;
  await page.locator("//a[normalize-space(text())='Sign in']").click();
});
```

Example: addToCartStepDef.ts

```
// addToCartStepDef.ts

import { Given } from '@cucumber/cucumber';
import { pageFixture } from '../hooks/pageFixture';

Given('the user adds an item to the cart', async function () {
  const page = pageFixture.page;
  await page.locator('button.add-to-cart').click();
});
```

How This Solves Your Problem

◈ **Avoids Duplication**

- Common steps (like login) are reused across scenarios.
- No need to recreate the page object in every file.

◈ **Shared Objects**

- pageFixture.page holds one shared browser page accessible in all step definitions.

◈ **Centralized Hooks**

- hooks.ts handles one-time browser launch and cleanup using BeforeAll and AfterAll.

Created by: Yogesh Pandian

Summary

File	**Purpose**
pageFixture.ts	Stores and shares the Playwright page object
hooks.ts	Initializes and tears down browser & page

Benefits

- Clean, reusable test structure
- One-time browser and page setup
- No duplication of page initialization
- Easy maintenance of shared steps

Cucumber Report Generation in Playwright with Screenshots

Step 1: Generate Cucumber Report

Modify cucumber.json:

Add this format property to specify where to generate the HTML report:

```
{
  "format": [
    "html:test-results/cucumber-report.html"
  ]
}
```

Execute the Tests:

Run this command:

```
npm test
```

- A folder named test-results will be created.
- The report will be available at test-results/cucumber-report.html.

Step 2: Add Screenshots to the Report

Optimize Browser Launching

Don't launch browser in every Before / After hook

Instead, use:

```
// hooks.ts


import { BeforeAll, AfterAll, Before, After } from '@cucumber/cucumber';

import { chromium, Browser, BrowserContext, Page } from 'playwright';

import { pageFixture } from './pageFixture';


let browser: Browser;

let context: BrowserContext;


BeforeAll(async function () {

  browser = await chromium.launch({ headless: false });

});


Before(async function () {

  context = await browser.newContext();

  const page = await context.newPage();

  pageFixture.page = page;

});


After(async function () {

  await pageFixture.page.close();

  await context.close();

});
```

Created by: Yogesh Pandian

```
AfterAll(async function () {
  await browser.close();
});
```

Step 3: Capture and Attach Screenshots

Basic Screenshot After Each Scenario

```
import { After } from '@cucumber/cucumber';


After(async function () {
  const img = await pageFixture.page.screenshot({ path: './test-results/screenshots/scenario.png', type: 'png' });
  await this.attach(img, 'image/png');
});
```

Name Screenshot by Scenario Name

```
After(async function ({ pickle }) {
  const img = await pageFixture.page.screenshot({
    path: `./test-results/screenshots/${pickle.name}.png`,
    type: 'png',
  });
  await this.attach(img, 'image/png');
});
```

Step 4: Capture Screenshots Only for Failed Scenarios

```
import { After, Status } from '@cucumber/cucumber';


After(async function ({ pickle, result }) {
  if (result?.status === Status.FAILED) {
    const img = await pageFixture.page.screenshot({
      path: `./test-results/screenshots/${pickle.name}.png`,
      type: 'png',
    });
    await this.attach(img, 'image/png');
  }
  await pageFixture.page.close();
  await context.close();
});
```

Step 5 (Optional): Capture Screenshots After Each Step

```
import { AfterStep } from '@cucumber/cucumber';

AfterStep(async function ({ pickle }) {
  const img = await pageFixture.page.screenshot({
    path: `./test-results/screenshots/${pickle.name}-step.png`,
    type: 'png',
  });
  await this.attach(img, 'image/png');
});
```

Summary

Feature	**Description**
HTML Report	Configure via cucumber.json with format path
Efficient Browser Handling	Use BeforeAll/AfterAll and browser contexts per scenario
Scenario-level Screenshots	Use After hook with this.attach()

Created by: Yogesh Pandian

Feature	Description
Name Screenshots Dynamically	Use pickle.name
Only on Failure	Check result.status === Status.FAILED
Step-level Screenshots (Optional)	Use AfterStep to capture after every step

---

How to Create a Cucumber Reporter with multiple-cucumber-html-reporter

1. Introduction to Cucumber Reporters

- Cucumber comes with a default reporter that provides basic information (pass/fail).
- For advanced dashboards, system metadata, and cleaner UI, use multiple-cucumber-html-reporter.

---

2. Default Cucumber Report

In your **cucumber.js** configuration file, add:

```
module.exports = {
 default: {
  format: [
   "html:test-results/cucumber-report.html",
   "json:test-results/cucumber-report.json",
   "junit:test-results/cucumber-report.xml"
  ]
 }
};
```

- Provides basic status like pass/fail.
- Does not include dashboards or system info.

---

3. Why Use multiple-cucumber-html-reporter?

This 3rd-party reporter includes:

- Dashboard UI
- System Info: Who executed, project, release, etc.
- Progress Bar in console

To enhance console output, use this in **cucumber.js**:

```
format: [
  "json:test-results/cucumber-report.json",
  "progress-bar"
]
```

---

4. Generate JSON Results

Update **cucumber.js** like so:

```
format: [
  "json:test-results/cucumber-report.json"
]
```

- This creates the cucumber-report.json file after tests.

---

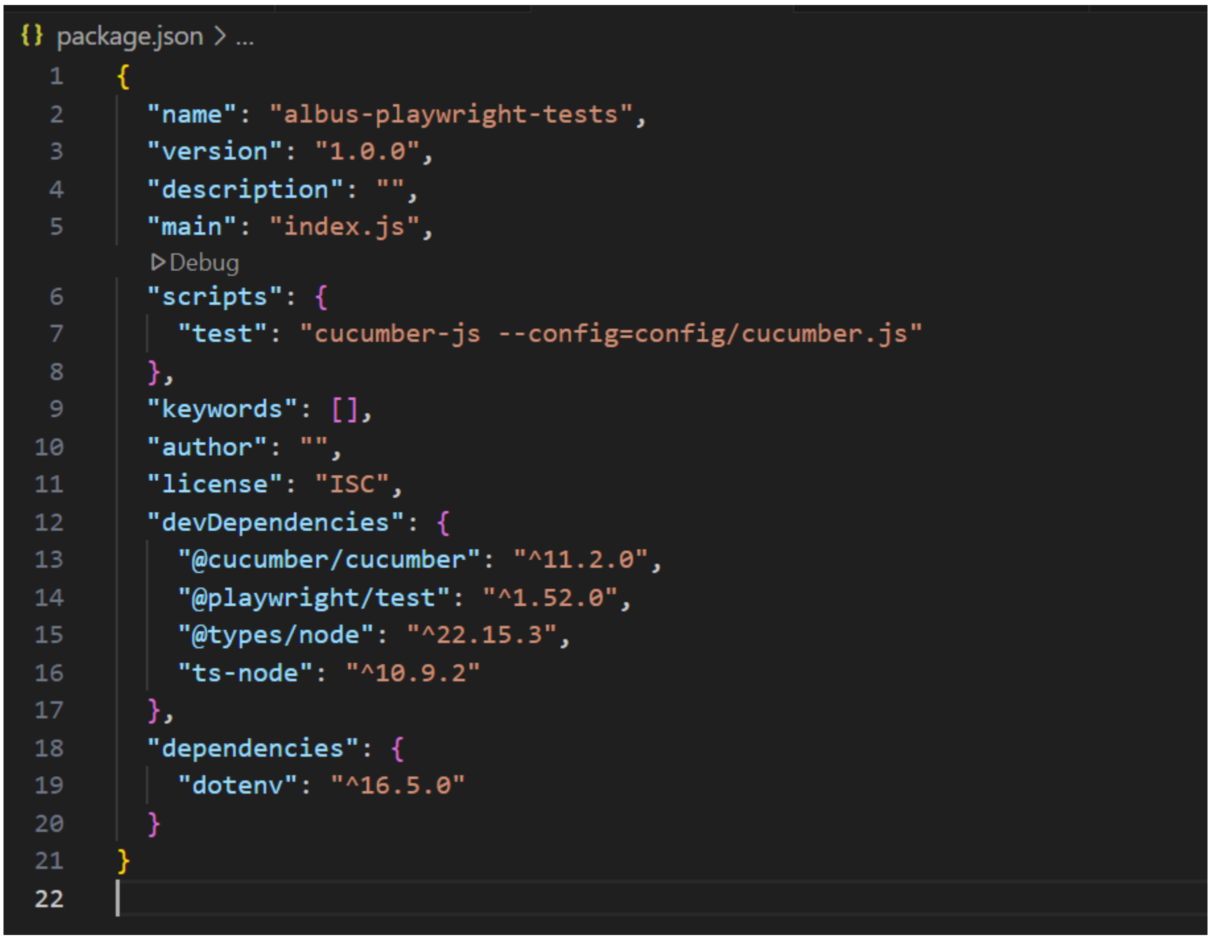
5. Install the Reporter

Install the reporter via npm:

```
npm install multiple-cucumber-html-reporter --save-dev
```

- Check your package.json to confirm it’s under devDependencies.

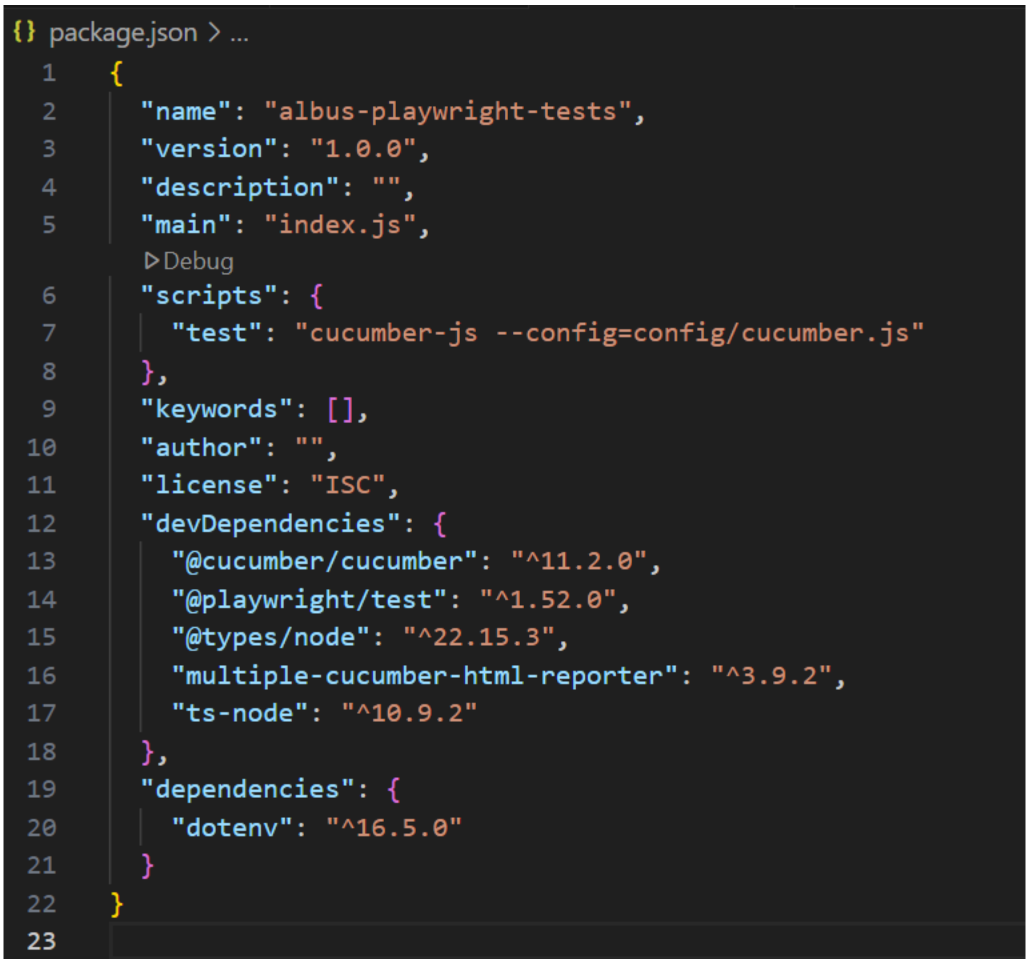
  Before adding multiple cucumber reporter



{} package.json > ...

```json
{
  "name": "albus-playwright-tests",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷Debug
  "scripts": {
    "test": "cucumber-js --config=config/cucumber.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@cucumber/cucumber": "^11.2.0",
    "@playwright/test": "^1.52.0",
    "@types/node": "^22.15.3",
    "ts-node": "^10.9.2"
  },
  "dependencies": {
    "dotenv": "^16.5.0"
  }
}
```

After adding multiple cucumber reporter



```
{} package.json > ...
{
  "name": "albus-playwright-tests",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷Debug
  "scripts": {
    "test": "cucumber-js --config=config/cucumber.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@cucumber/cucumber": "^11.2.0",
    "@playwright/test": "^1.52.0",
    "@types/node": "^22.15.3",
    "multiple-cucumber-html-reporter": "^3.9.2",
    "ts-node": "^10.9.2"
  },
  "dependencies": {
    "dotenv": "^16.5.0"
  }
}

```

---

6. Create the Report Generation Script

Step 1: Create a helper/ folder

mkdir src/helper

Step 2: Create report.ts inside helper/

```
const report = require("multiple-cucumber-html-reporter");


report.generate({
  jsonDir: 'test-results/cucumber-json',
  reportPath: 'test-results/cucumber-report',
  storeScreenshots: true,
  displayDuration: true,
```

Created by: Yogesh Pandian

```
  openReportInBrowser: true,
  screenshotsDirectory: 'test-results/screenshots/',
  pageTitle: 'Cucumber Test Report',
  reportName: 'Cucumber Test Execution Report',
  customData: {
    title: 'Test Execution Information',
    data: [
      { label: 'Project', value: 'Playwright Cucumber Tests' },
      { label: 'Version', value: '1.0.0' },
      { label: 'Execution Date', value: new Date().toLocaleString() }
    ]
  }
});
```

---

7. Link in package.json

Update the scripts section:

```
"scripts": {
  "test": "cucumber-js test",
  "posttest": "npx ts-node src/helper/report.ts"
}
```

- posttest will run after test, generating the HTML report.

---

8. Handling Test Failures

To ensure the report is generated even if tests fail:

```
"scripts": {
  "test": "cucumber-js --config=config/cucumber.js || exit 0",
  "posttest": "npx ts-node src/helper/report/report.ts"
}
```

|| exit 0 allows posttest to run even when tests fail.

Created by: Yogesh Pandian

9.Execution Flow

1. Run tests: npm run test
2. JSON report is generated (cucumber-report.json)
3. posttest triggers report.ts
4. View your report at: test-results/cucumber-report/index.html

---

Parallel & Rerun Setup for Playwright + Cucumber

Step 1: Run Tests in Parallel

In cucumber.json, add:

```
{
 "default": {
  "format": [
   "progress-bar",
   "html:test-results/cucumber-report.html",
   "json:test-results/cucumber-report.json",
   "junit:test-results/cucumber-report.xml"
  ],
  "parallel": 2,  // Run 2 browser instances in parallel
  "require": [
   "src/test/steps/*.ts",
   "src/test/hooks/hooks.ts"
  ],
  "requireModule": [
   "ts-node/register"
  ]
 }
}
```

Created by: Yogesh Pandian

Step 2: Rerun Failed Tests Automatically

1. Generate @rerun.txt

- After running tests, Playwright + Cucumber creates @rerun.txt listing failed scenarios with file paths and line numbers.

2. Add rerun Profile in cucumber.json

```
{
 "rerun": {
  "formatOptions": {
   "snippetInterface": "async-await"
  },
  "publishQuiet": true,
  "dryRun": false,
  "require": [
   "src/test/steps/*.ts",
   "src/test/hooks/hooks.ts"
  ],
  "requireModule": [
   "ts-node/register"
  ],
  "format": [
   "progress-bar",
   "html:test-results/cucumber-report.html",
   "json:test-results/cucumber-report.json",
   "junit:test-results/cucumber-report.xml",
   "rerun:@rerun.txt"
  ],
  "parallel": 2
 }
}
```

- The rerun profile uses rerun:@rerun.txt to rerun only failed tests.
- No feature files path specified here — it reads failures from the rerun file.

Step 3: Update package.json Scripts

```
{
 "scripts": {
  "test": "cucumber-js test || exit 0",        // Run all tests, allow failures without stopping
  "posttest": "npx ts-node src/helper/report.ts", // Generate report after tests
  "test:failed": "cucumber-js -p rerun @rerun.txt" // Rerun only failed tests
 }
}
```

Step 4: How to Use

- Run full tests:

```
npm run test
```

- Check @rerun.txt for failed scenarios (auto-generated if any fail).
- Rerun only failed tests:

```
npm run test:failed
```

Step 5: Clean-up

- After rerun, if all failed tests pass, @rerun.txt is cleared automatically.
- This indicates that all issues were fixed.

Summary

Feature	**How to Configure**
Parallel Execution	"parallel": 2 in cucumber.json
Failed Tests Rerun	Use rerun profile with rerun:@rerun.txt in format

Created by: Yogesh Pandian

Feature	How to Configure
Scripts	test, test:failed in package.json
Auto Clean rerun.txt	Happens after rerun if all pass

Dynamic Browser Selection in Playwright with TypeScript

Step 1: Refactor Hooks - Remove Hardcoded Browser

Old code initializing Chromium only:

```
BeforeAll(async function () {
  browser = await chromium.launch({ headless: true });
});
```

Step 2: Create BrowserManager.ts

Location: src/helper/browsers/BrowserManager.ts

```
import { LaunchOptions, chromium, firefox, webkit } from "playwright-core";

const options: LaunchOptions = {
  headless: true
};

export const invokeBrowser = () => {
  const browserType = process.env.BROWSER;

  switch (browserType) {
    case "chrome":
      return chromium.launch(options);
    case "firefox":
      return firefox.launch(options);
    case "webkit":
      return webkit.launch(options);
    default:
      throw new Error("Please set an available browser in the environment variables.");
  }
};
```

Step 3: Use Environment Variable process.env.BROWSER

- **Purpose:** Allows switching browsers dynamically without changing code.
- Example run command:

```
BROWSER=firefox npm run test
```

Step 4: Define Type Declarations for Environment Variables

Create env.d.ts in src/helper/types/ or src/helper/:

```
declare global {
  namespace NodeJS {
    interface ProcessEnv {
      BROWSER: "chrome" | "firefox" | "webkit";
      ENV: "staging" | "prod" | "test";
      BASEURL: string;
      HEAD: "true" | "false";
    }
  }
}
```

- This enables autocomplete and type checking for process.env.

Step 5: Update tsconfig.json to Recognize Custom Types

```
{
 "compilerOptions": {
  "typeRoots": [
   "./node_modules/@types",
   "src/helper/types"
  ]
 }
}
```

- Ensures TypeScript loads your env.d.ts.

Step 6: Update Hooks File to Use invokeBrowser

```
import { invokeBrowser } from "../helper/browsers/BrowserManager";


BeforeAll(async function () {
  getEnv(); // your existing env setup function if any
  browser = await invokeBrowser();
});
```

What	**Why**
Move browser launch to BrowserManager.ts	Centralized browser management
Use process.env.BROWSER	Dynamic browser choice without code change
Create env.d.ts	TypeScript support for env variables
Update tsconfig.json	Load custom type declarations
Update hooks to call invokeBrowser	Run tests on selected browser dynamically

---

Created by: Yogesh Pandian

Why These Steps?

- **process.env**: To configure environment variables outside code, making switching easy.
- **env.d.ts**: Typescript does not know custom env variables by default; this file declares allowed keys and values.
- **tsconfig.json**: Tells TypeScript where to find custom type declarations to avoid errors.
- **Refactoring hooks**: Makes your test setup scalable and maintainable.

How to Handle Multiple Environments in Your Project

Step 1: Create the env Package

- Inside your project folder, create a directory:

src/helper/env/

- Inside it, create a file:

env.ts

Step 2: Install Required Dependencies

Run:

npm install dotenv --save-dev

npm install cross-env --save-dev

- dotenv loads environment variables from files.
- cross-env helps set env vars cross-platform in scripts.

Step 3: Setup env.ts

Write:

import * as dotenv from 'dotenv';

```
export const getEnv = () => {
  dotenv.config({
    override: true,
    path: '' // We'll set this dynamically later
  });
};
```

Step 4: Create .env Files per Environment

Create environment files inside src/helper/env/:

- **Production:** .env.prod

```
BASEURL=https://prod.example.com
BROWSER=chrome
```

- **Staging:** .env.stage

```
BASEURL=https://stage.example.com
BROWSER=firefox
```

(Add other env files similarly for test, dev, etc.)

Step 5: Update env.ts to Load Based on ENV Variable

Change getEnv to:

```
import * as dotenv from 'dotenv';

export const getEnv = () => {
  dotenv.config({
    override: true,
    path: `src/helper/env/.env.${process.env.ENV}`
  });
};
```

Created by: Yogesh Pandian

- This loads the environment file dynamically based on the ENV environment variable.

Step 6: Use Environment Variables in Your Code

Example in login.ts or anywhere:

```
Given("User navigates to the application", async function() {
  await pageFixture.page.goto(process.env.BASEURL as string);
});
```

- process.env.BASEURL picks the base URL for the environment.

Step 7: Update package.json Scripts to Pass ENV

Add/modify scripts like this:

```
"scripts": {
  "pretest": "npx ts-node src/helper/init.ts",
  "test": "cross-env ENV=prod cucumber-js || true",
  "test:stage": "cross-env ENV=stage cucumber-js",
  "posttest": "npx ts-node src/helper/report.ts",
  "test:failed": "cucumber-js -p rerun @rerun.txt"
}
```

- Run tests in production:

```
npm run test
```

- Run tests in staging:

```
npm run test:stage
```

Bonus: Running Tests with Custom ENV on CLI

Alternatively, run directly with:

```
cross-env ENV=stage npm run test
```

Or:

Created by: Yogesh Pandian

```
ENV=stage npm run test
```

(On Windows, cross-env is preferred)

Summary

Step	Purpose
env.ts	Loads environment variables dynamically
.env.* files	Store env-specific configuration
process.env.ENV	Runtime selector for environment
package.json	Scripts pass ENV to switch environments

How to Integrate Winston Logger into Playwright Cucumber Framework

1. Create Logger Utility

- Create folder:
  src/helper/utils/
- Create file:
  logger.ts
- Add this code:

```
import { transports, format } from 'winston';


export function options(scenarioName: string) {
  return {
    transports: [
      new transports.File({
        filename: `test-results/logs/${scenarioName}/log.log`,
        level: 'info',
        format: format.combine(
            format.timestamp({ format: 'MMM-DD-YYYY HH:mm:ss' }),
            format.align(),
            format.printf(info => `${info.level}: ${info.timestamp}: ${info.message}`)
          )
        })
      ]
    };
}
```

Explanation:

- Creates a separate log file per scenario under test-results/logs/{scenarioName}/log.log.
- Logs include timestamp and aligned formatting for readability.

---

2. Update pageFixture.ts

Add a logger property alongside page:

```
import { Page } from "@playwright/test";
import { Logger } from "winston";

export const pageFixture = {
  //@ts-ignore
  page: undefined as Page,
  logger: undefined as Logger
};
```

---

3. Install Winston

Run:

```
npm install winston --save-dev
```

Created by: Yogesh Pandian

4. Configure Logger in hooks.ts

Modify your hooks to initialize the logger per scenario:

```
import { createLogger } from 'winston';

import { options } from '../helper/utils/logger';

import { pageFixture } from '../fixture/pageFixture'; // adjust path as needed


Before(async function({ pickle }) {

  const scenarioName = `${pickle.name}_${pickle.id}` ; // unique per scenario

  const context = await browser.newContext();

  const page = await context.newPage();


  pageFixture.page = page;

  pageFixture.logger = createLogger(options(scenarioName));

});
```

About pickle:

- pickle.name = scenario name
- pickle.id = unique id ensuring distinct log files even if scenario names repeat

---

5. Close Logger After All Scenarios

Add in hooks.ts:

```
AfterAll(async function() {

  await browser.close();

  if (pageFixture.logger) {

    pageFixture.logger.close();

  }

});
```

- Closing the logger ensures no memory leaks or dangling file handles.

Created by: Yogesh Pandian

6. Using Logger in Step Definitions

Anywhere in your steps, you can now use:

```
Given("User navigates to the application", async function() {
  await pageFixture.page.goto(process.env.BASEURL as string);
  pageFixture.logger.info("User navigated to the application");
});


Given("User searches for a {string}", async function(book) {
  pageFixture.logger.info(`Searching for a book: ${book}`);
  await pageFixture.page.locator("input[type='search']").fill(book);
});
```

Recap

Task	**Description**
utils/logger.ts	Configures Winston with scenario-specific files
pageFixture.ts	Holds page and logger objects
hooks.ts	Creates logger instance before each scenario
hooks.ts	Closes logger after all tests
Step definitions	Use pageFixture.logger.info() to log messages

---

Understanding Tags and Why We Use config.js Instead of config.json

What are Tags?

- **Tags** are labels that you add to your test scenarios or features in Cucumber.
- They help you **filter which tests to run**. For example, you can run only tests tagged as @smoke or @regression.

- This makes test execution **more flexible and faster** by running only the tests you want.

Why config.json Does NOT Support Tags Well

- The **config.json** file is a simple JSON file.
- It is mainly for basic configuration, like defining paths, timeouts, or other settings.
- But **JSON files do not support complex logic** like running tests by tags.
- You cannot easily add or modify tag filters dynamically in config.json.

Why We Switch to config.js

- **config.js** is a JavaScript file, so you can write **logic inside it**.
- You can easily add conditions, use variables, and customize which tags to run.
- It supports **dynamic configurations** — you can read environment variables or other inputs to decide which tests to run.
- Overall, it makes your test setup **more powerful and flexible**.

Sample config.js File with Tag Support

```
const commonConfig = {
  defaultTimeout: 60000,
  paths: ['./features/**/*.feature'],
};


const tags = process.env.TAGS || '@smoke';  // Default to @smoke if no tag passed


module.exports = {
  ...commonConfig,
  tags: tags,
};
```

Created by: Yogesh Pandian

- Here, you can run tests with different tags like:

```
TAGS='@regression' npx cucumber-js
```

Quick Comparison: config.json vs config.js

Feature	**config.json**	**config.js**
File Type	JSON (static data)	JavaScript (dynamic code)
Supports Tags?	No	Yes
Can use variables?	No	Yes
Supports Logic/Conditionals	No	Yes
Flexibility	Low	High
Recommended for	Simple settings only	Complex configs & tags

- **Tags** help run specific tests easily.
- **config.json** is simple but cannot handle tags or dynamic configs.
- **config.js** supports tags, variables, and logic.
- Switching to **config.js** makes your framework more flexible and powerful.
