Learning Objectives

Learners will be able to…

- **Differentiate behavior-driven development (BDD) from other forms of testing**
- **Identify the benefits of using BDD**
- **Identify common tools for BDD testing**
- **Create BDD tests with Cucumber.js**

info

Make Sure You Know

You should be familiar with JavaScript as well as creating directories and files with the terminal.

Limitations

This is only a brief introduction to behavior-driven development and Cucumber.js.

Behavior-Driven Development

Why Behavior-Driven Development?

Behavior-driven development (BDD) is an approach to development that improves communication between business and technical teams to create software with business value.

This assignment is for both technical and business professionals and explores how BDD can benefit projects of all sizes, and how to implement it with confidence. It's structured to reflect the flow of the BDD process.

When launching a new digital project, there can be a disconnect between:
* the business' ability to define the desired outcomes
* the developer's understanding of what needs to be built
* the business' understanding of the technical challenges its requirements may present.

Behavior-driven development helps achieve all of the above. Ultimately, it helps your business and technical team to deliver software that fulfills your business goals.

What is BDD?

Defining Behavior-Driven Development

Behavior-driven development (BDD) is a software development process that seeks to bridge the gap between engineers and business professionals and ensure software projects remain focused on delivering what the business actually needs. Developed by experienced technology and organizational consultant Dan North, BDD is designed to improve communication between engineers and business professionals while ensuring software development meets user needs.

The BDD process begins with examples written in a language that is understood by all parties, which then serves as the basis of automated tests. This ensures the system works as defined by the business throughout the project lifetime and that business value and user requirements are never secondary during development.

BDD has several key benefits. All development work can be traced back directly to business objectives, which allows software development to be tailored to the needs of the user. This also results in improved quality code that reduces the cost of maintenance and minimizes project risk. Additionally, BDD facilitates efficient prioritization, meaning business-critical features are delivered first, and all parties involved have a shared understanding of the project, which ensures everyone (technical or not) has thorough visibility into the project's progression.

BDD is a critical tool for any software development project. It allows for efficient communication and collaboration between engineers and business professionals, which ensures all development projects remain focused on delivering what the business actually needs while meeting all requirements of the user. Ultimately, BDD is an invaluable tool for any software development project, and its use should be strongly considered.

BDD as a Part of TDD

Comparing BDD and TDD

Behavior-driven development (BDD) is an approach to writing automated tests that is meaningful to business people. It is a departure from the traditional Test Driven Development (TDD) approach, which uses technical terminology for test names. BDD uses a language that is easily understood by non-technical users, and often follows a structure of “given-when-then” to describe the tests.

The “given” part of BDD tests is where preconditions are declared. This could be something like a customer type or an action. The “when” part contains the action to be tested. The “then” part is the verification of the outcome. An example of a BDD test could be written as “given a premium customer, when we get the discount, then it should return ten percent.”

Unlike TDD tests which generally have a one-word descriptive name, BDD tests are more like a sentence that is easily understood by a non-technical user. This makes it easier for new developers to understand the business rule that is being tested.

As an example, you could name a TDD-style test “getPremiumCustomerDiscountTest”, whereas in BDD you would name it something like “givenAPremiumCustomer_WhenWeGetTheDiscount_ItShouldReturnTenPercent”.

Steps of BDD

Behavior-driven development is a process of writing automated tests that is useful to business people and follows a given-when-then structure.

1. First, identify the business rules that need to be tested.
2. Next, write out the test in a meaningful language that is easy to understand by non-technical users. This could be written as a sentence, using the given-when-then format.
3. Once the test is written, it is important to ensure that the test is valid and will accurately test the business rule.
4. Finally, the test is automated and run to ensure that the desired outcome is achieved.

BDD is a great way to ensure that the tests are written with the end user in mind. They are easier to read and understand, which means they are more likely to be maintained and updated over time.

Cucumber.js

BDD Tools

There are many tools that you can use for the BDD process. Many of the tools provide syntax for creating scenarios in a simple language and then map to a specific programming language. SpecFlow works with the .NET framework, JBehave works with Java, and Gospecify works with the Go language. We are going to focus on Cucumber, which works with JavaScript.

Introduction to Cucumber.js

Cucumber.js is a testing library that allows you to write your tests in plain language. It follows the given-when-then structure, but as you'll see the tests are very readable, even by business users. This allows your tests to be a point of communication and collaboration. They can even serve as documentation that is automatically up-to-date.

This can have far-reaching consequences. Whenever there is an issue or a feature request, both developers and business people can work on the same document. Developers can then translate this document into a test and implement the feature.

Cucumber.js (and similar tools in other languages) uses something called the Gherkin language. It's a Domain Specific Language but it's easily read. Here's what a test could look like:

```
Feature: Login Form
  As a user
  I want to be able to log in to the website
  So that I can access private content

Scenario: Login with blank username and password
  Given I have not entered a username or password
  When I submit the login form
  Then I should see an error message saying "Missing username or
password"

Scenario: Login with invalid username and password
  Given I have entered an invalid username and password
  When I submit the login form
  Then I should see an error message saying "Invalid username or
password"
```

Then you should add the correct “translation” from English to machine language. The tests are equal to the project documentation in this case, and it is well suited to small projects and websites without lots of logic changes. These tests can be read easily by anyone without any technical background.

Setting Up Cucumber

info

Chrome Already Installed

We need a browser installed for our Cucumber tests. As this is can be a complicated task due tc changing versions of Chromium, this has been done for you. If you are interested in how you v Chromium and the Chrome driver installed on your own system, click on the dropdown menu

▼ **Installing Chrome**

These instructions are based on the Ubuntu operating system. You may need to alter these inst slightly for your own operating system.

First you need to run the following commands to make sure you have the last available update system.

```
sudo apt update
sudo apt upgrade
```

Then download the Chromium browser. Pay attention to the version number, we need it for ar step.

```
sudo apt install chromium-browser
```

Next, we are going to download the Chrome driver. The version we download needs to match t version of Chromium. In this example, the version is `111.0.5563`. Replace this with the appropr version number.

```
wget -qP "/tmp/"
        "https://chromedriver.storage.googleapis.com/111.0.5563.64/chromedriver_linu
        ip"
```

If you system does not have the `unzip` utility, install it with the following command:

```
sudo apt install unzip
```

Unzip the Chrome driver and install it on the system.

```
sudo unzip -o /tmp/chromedriver_linux64.zip -d /usr/bin
```

With this done, you are ready to install Cucumber.

Installing Cucumber

Installing Cucumber can be done through NPM. Start by initializing a new project.

```
npm init
```

Answer the questions posed to you. In most cases, the default answer is sufficient. However, when it comes to the testing script, we want to make a change. Enter the information below for the testing script, which will run when we execute the `npm test` command.

```
test command: cucumber-js
```

Then install version 9.0.0 of Cucumber:

```
npm install @cucumber/cucumber@9.0.0
```

Finally, let’s take a look at the contents of our `package.json` file. Use the `cat` command to print the contents of the file to the terminal.

```
cat package.json
```

Your `package.json` file should look similar the example below for testing with Cucumber:

```
{
  "name": "first-example",
  "version": "1.0.0",
  "description": "",
  "main": "isEven.js",
  "scripts": {
    "test": "cucumber-js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@cucumber/cucumber": "^9.0.0"
  }
}
```

Cucumber Directories and Files

Testing Structure

We are going to set up a few directories to organize our testing files. This structure of directories and files is required by Cucumber. Start by creating the directory `features`:

```
mkdir features
```

Cucumber requires that testing scenarios are written in files with the `.feature` file extension. These files must reside in the `features` directory. Create the `feature_test.feature` file inside of the `features` directory. For now, we are just going to create the file. We will write the actual contents on the next page.

```
touch features/feature_test.feature
```

We also need to create the `support` directory inside of the `features` directory. This is where the JavaScript test files must reside.

```
mkdir features/support
```

Create the `steps.js` file (for our tests) inside of the `support` directory. Just as before, we will write the actual code for the file on the next page.

```
touch features/support/steps.js
```

This last file is not required, but it can improve the user experience. By default, Cucumber will post a message in the terminal about sharing your report. It detracts from the test output. We want to silence this message. Create the `cucumber.js` file.

```
touch cucumber.js
```

Use the link below to open the newly created file:

Finally, copy the code below into the file. This will “turn off” the message about sharing your Cucumber report.

```
module.exports = { default: '--publish-quiet' }
```

Support Directory and Files

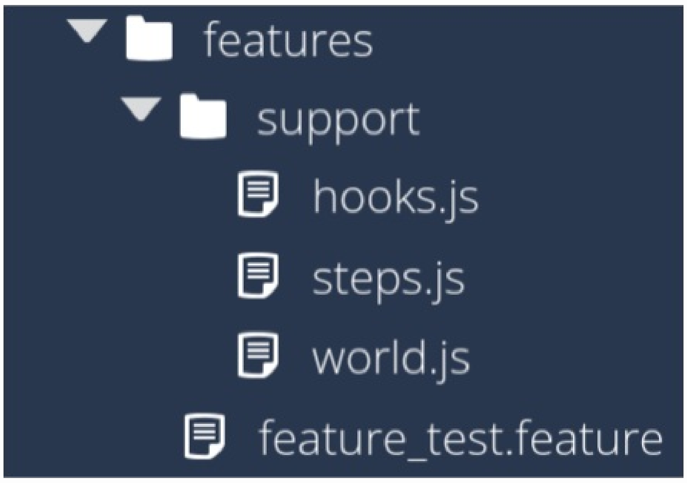
info

Important

We will not be using the `hooks.js` or `world.js` files for this first project. However, more robust projects (like our second example) make use of these files which also reside in the `support` directory.

You are not expected to copy and paste the code samples below. They are shown for informative purposes.

As your Cucumber tests grow in complexity, you may find yourself needing to customize certain aspects of the testing process outside of the `steps.js` file. These customizations are commonly done in additional files in the `support` directory.



```
features
  support
    hooks.js
    steps.js
    world.js
  feature_test.feature
```

The image shows a file tree with the support directory inside the features directory. Inside the support directory are the files world.js, steps.js, and hooks.js.

The `hooks.js` file is used to create (and later destroy) the testing environment before each scenario. Often tasks are performed before and/or after specified steps in the testing process. Please see this documentation for more information about the `hooks.js` file.

Here is a rather simple `hooks.js` file. We will be using this in our second example. The `Before` hook is run before any tests. We are maximizing our browser window used for testing. The `After` hook runs after the tests are executed. We are stopping the browser used for UI testing.

```
// hooks.js file

const {Before, After} = require('cucumber');

Before(function() {
  return this.driver.manage().window().maximize();
});

After(function() {
  return this.driver.quit();
});

module.exports = function() {
  this.After(function() {
    return this.driver.quit();
  });
};
```

The “world” is an abstraction that represents the Cucumber testing environment. We want to customize our environment a bit. In particular, we want a headless version of Chrome to be our browser when we reference `this.driver`. This is why we are creating the `world.js` file.

```
// world.js file

const {Builder} = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');

function CustomWorld() {
  this.driver = new
Builder().forBrowser('chrome').setChromeOptions(new
chrome.Options().addArguments('--headless')).build();
}

module.exports = function() {
  this.World = CustomWorld;
  this.setDefaultTimeout(30000);
};
```

A difference between the `hooks.js` and `world.js` files is that the `world.js` file is run once when the project starts. The `hooks.js` file, howerver, runs on every event.

Cucumber Example

Function to Test

Before we can write tests, we need something to test. For our first test, we are going to keep things simple. The function below already exists (**do not copy this**). We are first going to express a test using the previously discussed given-when-then syntax. Then we will transform those ideas into JavaScript that verifies the `isEven` function works as expected.

```
function isEven(num) {
  if (num % 2 == 0) {
    return "Yes";
  } else {
    return "No";
  }
}
```

Creating a Feature and Scenarios

Now that our project is properly structured, we are ready to start filling out the files needed for testing. We are going to start with the `feature_test.feature` file. Cucumber uses the Gherkin language to express tests in a simple manner. Define the purpose of this test using the `Feature:` keyword.

```
Feature: A feature to check if a number is even
```

Next, describe the scenario for tesing an even number with the `Scenario:` keyword. Notice how `Given`, `When`, and `Then` do not use colons. Indentation is also used for the different aspects of the file. This scenario states that when we assume the number 4, calling the `isEven` function with this number will return the string `"Yes"`.

```
  Scenario: Testing an even number
    Given The number is 4
    When asking if the number is even
    Then I should see the output "Yes"
```

Before we move on to the test in JavaScript, let's go ahead and run our Cucumber test by clicking the button below.

Look at the output. Cucumber provides some useful hints about the missing JavaScript tests. It recognizes the three steps in our scenario and gives you function snippets for each step. The snippets use `Given`, `When`, or `Then` which corresponds to the steps in the `.feature` file.

▼ **View test output**

Your test output should look like the sample below. Notice how the `1)` corresponds to the first step in the scenario. Just below that, Cucumber gives the an example function to use for the test. The `callback(null, 'pending');` is a placeholder that allows the test to run but not pass. The body of each function will need to be replaced with the appropriate code.

```
> cucumber-js

UUU

Failures:

1) Scenario: Testing an even number #
features/feature_test.feature:2
   ? Given The number is 4
       Undefined. Implement with the following snippet:

         Given('The number is {int}', function (int) {
         // Given('The number is {float}', function (float) {
           // Write code here that turns the phrase above into
concrete actions
           return 'pending';
         });

   ? When asking if the number is even
       Undefined. Implement with the following snippet:

         When('asking if the number is even', function () {
           // Write code here that turns the phrase above into
concrete actions
           return 'pending';
         });

   ? Then I should see the output "Yes"
       Undefined. Implement with the following snippet:

         Then('I should see the output {string}', function
(string) {
           // Write code here that turns the phrase above into
concrete actions
           return 'pending';
         });


1 scenario (1 undefined)
3 steps (3 undefined)
0m00.003s (executing steps: 0m00.000s)
```

Testing the Scenario

We are now going to create the JavaScript tests. Open the `steps.js` file with the link below.

Start by importing modules needed for the test. We need `Given`, `When`, and `Then` from Cucumber. We also need `assert` to make assertions about the validity of the `isEven` function. Finally, we need the `isEven` function itself.

```
const {Given, When, Then} = require('@cucumber/cucumber');
const assert = require('assert');
const isEven = require('../../isEven');
```

The first step in the `.feature` file is `Given`, which should be the first test in our JavaScript file. Use the snippet as the basic structure for our code. Notice how the function takes the parameter `num`. Cucumber recognizes that `4` in the step can actually be any number. So it generates the function snippet with a parameter that represents `4` from the `Given` step. Create the local variable `this.number` and assign it the value of `num`.

```
Given('The number is {float}', function (float) {
  this.number = float;
});
```

Now add code for the `When` step. There is nothing in the step (like a number or string) that would require a parameter. So there is no parameter for the function. Create the variable `this.actualAnswer` and assign it the value returned by the `isEven(this.number)` function call.

```
Given('The number is {float}', function (float) {
  this.number = float;
});

When('asking if the number is even', function () {
  this.actualAnswer = isEven(this.number);
});
```

Finally, add the code for the `Then` step. Because the string `"Yes"` has quotation marks in the `.feature` file, we need a parameter for this function. Use an assertion to verify if the actual output from the function is equal to the expected output from the scenario.

```
Given('The number is {float}', function (float) {
  this.number = float;
});

When('asking if the number is even', function () {
  this.actualAnswer = isEven(this.number);
});

Then('I should see the output {string}', function (string) {
  assert.equal(this.actualAnswer, string);
});
```

Run the test once again.

All the tests should pass. Your output should look something like this:

```
> cucumber-js

...

1 scenario (1 passed)
3 steps (3 passed)
0m00.026s (executing steps: 0m00.001s)
```

Adding Another Scenario

Our original function can return two different strings. However, our test only works with an even number. Let’s add another scenario to our `.feature` file for an odd number. Open the `feature_test.feature` file with the link below.

The structure for the second scenario is almost identical to the first. We are going to make a few substitutions. Change the scenario to testing an odd number, use `5` in the `Given` statement, and change the `Then` statement to expect `"No"`.

```
  Scenario: Testing an odd number
    Given The number is 5
    When asking if the number is even
    Then I should see the output "No"
```

Because our JavaScript functions expect parameters, Cucumber will take the values from the second scenario and pass them the appropriate function. There is no need to create more tests for the new scenario.

We should see that all our tests pass. The results are organized by scenario and steps.

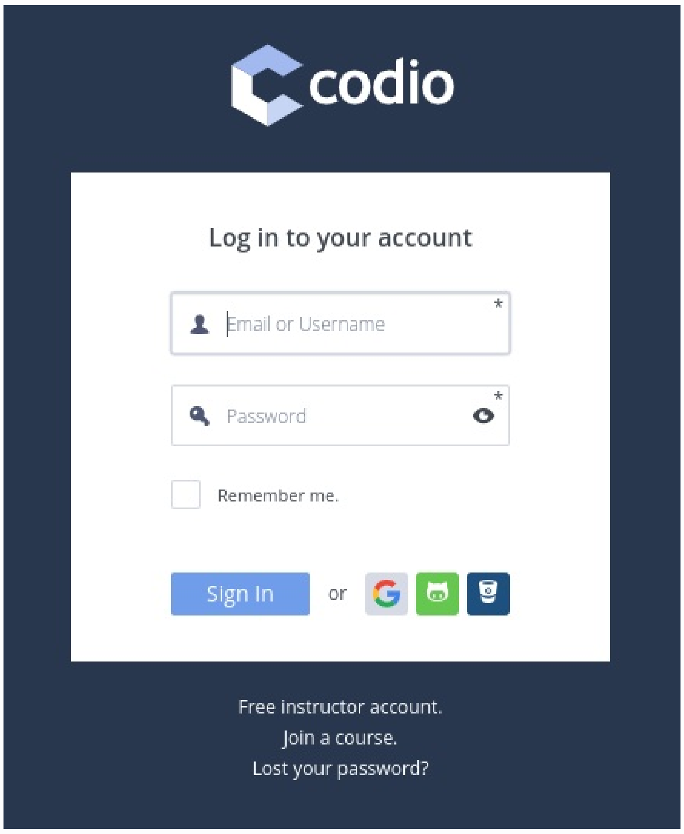
```
> cucumber-js


......

2 scenarios (2 passed)
6 steps (6 passed)
0m00.023s (executing steps: 0m00.001s)
```

UI Testing with Cucumber

For this example, we are going to test a UI with Cucumber. This time, however, all of the directories and files have been created for you. We are going to use the login page for Codio.



depicts the login page (username and password) for Codio

The website is configured so that you cannot click the `Sign In` button without entering information for both the username and password fields. We are going to verify this behavior with Cucumber.

Creating the Scenarios

We need three different scenarios to test — logging in without a password, logging in without a username, and both the email and password fields have information.

Start by giving the feature a description. Then create the first scenario. A user goes to Codio's login page, enters just the username, and finds that the `Sign In` button is disabled.

```
Feature: A feature to check Login on Codio website

  Scenario: Trying to log in without a password
    Given A user opens Codio login page
    When The username field filled up with "Username"
    Then The "Sign In" button should be disabled
```

Now create the second scenario. A user goes to Codio's login page, enters just the password, and finds that the `Sign In` button is disabled.

```
  Scenario: Trying to log in without a username
    Given A user opens Codio login page
    When The password field filled up with "Password"
    Then The "Sign In" button should be disabled
```

The final scenario entails a user going to Codio's login page, entering both a username and password, and finding that the `Sign In` button is enabled.

```
  Scenario: Trying to log in with a username and a password
    Given A user opens Codio login page
    When The username field filled up with "Username"
    And The password field filled up with "Password"
    Then The "Sign In" button should be enabled
```

▼ **Scenario Code**

The code for your `.feature` file should look like this with all three scenarios:

```
Feature: A feature to check Login on Codio website

  Scenario: Trying to log in without a password
    Given A user opens Codio login page
    When The username field filled up with "Username"
    Then The "Sign In" button should be disabled

  Scenario: Trying to log in without a username
    Given A user opens Codio login page
    When The password field filled up with "Password"
    Then The "Sign In" button should be disabled

  Scenario: Trying to log in with a username and a password
    Given A user opens Codio login page
    When The username field filled up with "Username"
    And The password field filled up with "Password"
    Then The "Sign In" button should be enabled
```

Creating the Tests

Use the link below to open the `steps.js` file.

Start our test by importing the required modules. We need `Given`, `When`, `Then`, and `setWorldConstructor` from Cucumber. We also need the `assert` module for our test the UI element. To parse the webpage, we are going to use the Selenium WebDriver. Finally, we need to set a value for timing out.

```
const {Given, When, Then, setWorldConstructor} =
        require('@cucumber/cucumber');
const {By, until, Builder} = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');
const assert = require('assert');
const timeOut = 10000;
```

Since we are testing UI elements of a website, we need to configure Cucumber to work with a headless version of Chrome. This is done with the `setWorldConstructor` that we imported. Create `this.driver` to be the headless version of Chrome.

```
setWorldConstructor(function(options) {
  this.driver = new
        Builder().forBrowser('chrome').setChromeOptions(new
        chrome.Options().addArguments('--headless')).build();
});
```

Next, we are going to create the `Given`, `When`, and `Then` functions. **Important**, since we are working with web pages, the functions for each step need to be asynchronous. For the `Given` step, load the Codio login page. To confirm this is working as expected, wait until the page loads and then get the text from the `form-title title` class. Log this text to the console.

```
Given('A user opens Codio login page', async function () {
  await this.driver.get('https://codio.com/p/login');
  let formTitle = await
        this.driver.findElement(By.className('form-title
        title'));
  await this.driver.wait(until.elementIsVisible(formTitle),
        timeOut);
  await console.log('    ...Page opened, and form title is \'' +
        await formTitle.getText() + '\'');
});
```

We need two different `When` functions. The first `When` function takes a string from the scenario as a parameter. Find the element with the name `user` and wait for it to be visible. Enter the parameter (`"Username"` in this case) into the field. Then log the value of the field to the console.

```
When('The username field filled up with {string}', async
        function (string) {
  let field = await this.driver.findElement(By.name('user'));
  await this.driver.wait(until.elementIsVisible(field),
        timeOut);
  await field.sendKeys(string);
  await console.log('    ...Username field filled with name \''
        + await field.getAttribute('value') + '\'');
});
```

The second `When` function takes a parameter from the scenario as well. Find the element with the CSS class `ReactPasswordStrength-input` and wait for it to be visible. Enter the parameter (`""` in this case) into the field. Then log the value of the field to the console.

```
When('The password field filled up with {string}', async
        function (string) {
  let field = await
        this.driver.findElement(By.css('.ReactPasswordStrength-
        input'));
  await this.driver.wait(until.elementIsVisible(field),
        timeOut);
  await field.sendKeys(string);
  await console.log('    ...Password field filled with password
        \'' + await field.getAttribute('value') + '\'');
});
```

We need two functions for the `Then` steps. One that checks if the button is enabled and another to check if it is disabled. For our first function, find the `Sign In` button and wait until it is visible. Log the button's `disabled` attribute to the console. The last step is to take the `isEnabled` status and assert that the button is disabled.

```
Then('The {string} button should be disabled', async function
        (string) {
  let button = await
        this.driver.findElement(By.xpath("//button[text()='Sign
        In']"));
  await this.driver.wait(until.elementIsVisible(button),
        timeOut);
  await console.log('    ...Sign In button\'s \'disabled\'
        attribute is ' + await button.getAttribute("disabled"));
  await button.isEnabled().then(function(isEnabled) {
    assert.ok(!isEnabled, 'Button is disabled');
  });
});
```

The final function is for the `Then` step that expects the `Sign In` button to be enabled. Find the button and wait until it is visible. Log the button's `disabled` attribute to the console. The last step is to take the `isEnabled` status and assert that the button is enabled.

```
Then('The {string} button should be enabled', async function
        (string) {
  let button = await
        this.driver.findElement(By.xpath("//button[text()='Sign
        In']"));
  await this.driver.wait(until.elementIsVisible(button),
        timeOut);
  await console.log('    ...Sign In button\'s \'disabled\'
        attribute is ' + await button.getAttribute("disabled"));
  await button.isEnabled().then(function(isEnabled) {
    assert.ok(isEnabled, 'Button is enabled');
  });
});
```

▼ **JavaScript Code**

Here is what your code should look like when done creating the tests:

```
const {Given, When, Then, setWorldConstructor} =
        require('@cucumber/cucumber');
const {By, until, Builder} = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');
const assert = require('assert');
const timeOut = 10000;


setWorldConstructor(function(options) {
  this.driver = new
        Builder().forBrowser('chrome').setChromeOptions(new
        chrome.Options().addArguments('--headless')).build();
});


Given('A user opens Codio login page', async function () {
  await this.driver.get('https://codio.com/p/login');
  let formTitle = await
        this.driver.findElement(By.className('form-title
        title'));
  await this.driver.wait(until.elementIsVisible(formTitle),
        timeOut);
  await console.log('    ...Page opened, and form title is \'' +
        await formTitle.getText() + '\'');
});


When('The username field filled up with {string}', async
        function (string) {
  let field = await this.driver.findElement(By.name('user'));
  await this.driver.wait(until.elementIsVisible(field),
        timeOut);
  await field.sendKeys(string);
  await console.log('    ...Username field filled with name \''
        + await field.getAttribute('value') + '\'');
});


When('The password field filled up with {string}', async
        function (string) {
  let field = await
        this.driver.findElement(By.css('.ReactPasswordStrength-
        input'));
  await this.driver.wait(until.elementIsVisible(field),
        timeOut);
  await field.sendKeys(string);
  await console.log('    ...Password field filled with password
        \'' + await field.getAttribute('value') + '\'');
});


Then('The {string} button should be disabled', async function
        (string) {
  let button = await
        this.driver.findElement(By.xpath("//button[text()='Sign
        In']"));
  await this.driver.wait(until.elementIsVisible(button),
        timeOut);
  await console.log('    ...Sign In button\'s \'disabled\'
        attribute is ' + await button.getAttribute("disabled"));
  await button.isEnabled().then(function(isEnabled) {
    assert.ok(!isEnabled, 'Button is disabled');
  });
});


Then('The {string} button should be enabled', async function
        (string) {
  let button = await
        this.driver.findElement(By.xpath("//button[text()='Sign
        In']"));
    await this.driver.wait(until.elementIsVisible(button),
          timeOut);
    await console.log('    ...Sign In button\'s \'disabled\'
          attribute is ' + await button.getAttribute("disabled"));
    await button.isEnabled().then(function(isEnabled) {
      assert.ok(isEnabled, 'Button is enabled');
    });
  });
```

The test will produce a fair amount of output, as there should be feedback for each step. If you scroll to the bottom of the output, you should see something similar to the example below. All 10 steps should pass.

```
tex -hide-clipboard 3 scenarios (3 passed) 10 steps (10 passed)
0m15.851s (executing steps: 0m15.804s)
```