Learning Objectives

Learners will be able to...

- **Define UI testing**
- **Identify common tools used for UI testing**
- **Create UI tests with Selenium**

info

Make Sure You Know

You should be familiar with writing async-await functions.

Limitations

We are not using a specific testing framework with Selenium. Instead, we are going to use basic `assert` statements.

UI Testing with Selenium

Testing a User Interface

The most popular and commonly used software testing tools are:

1. Selenium: An open-source software testing tool used for automating web browsers.
2. Appium: An open-source mobile app testing tool used for automating native, hybrid and mobile web apps.
3. Ranorex: A commercial software testing tool used for automating desktop, mobile and web applications.
4. TestComplete: A commercial software testing tool used for creating automated tests for desktop, web and mobile applications.
5. LoadRunner: A commercial software testing tool used for creating automated performance tests.
6. SoapUI: An open-source software testing tool used for testing web services.
7. JMeter: An open-source software testing tool used for performance testing(this one was discussed in a previous module)

Selenium is a popular open-source automated testing framework for web applications. It can be used to automate tests for end-user interface (UI) usability, compatibility, performance, and other aspects of the application. Selenium can also be used to perform cross-browser testing and test for user acceptance. It is commonly used to test a range of different web browsers, including Chrome, Firefox, Safari, and Edge.

Selenium UI testing works by automating the user interactions with a web application in order to validate that the application is working as expected. This is done by creating automated scripts that simulate a user's interactions with the application. These scripts are then executed to check the functionality of the application and to ensure that it is performing correctly. The scripts can also be used to take screenshots and record videos of the application to help identify any issues or problems. Once the tests are completed, the results are then analyzed and any issues are reported.

It supports a number of scripting languages like C#, Java, Perl, Ruby, JavaScript, etc. Depending on the application to be tested, one can choose the script accordingly.

Cross-Browser Testing

Testing Web Applications with Browsers

Cross-browser testing is a process of testing an application or website on multiple web browsers, to ensure consistent behavior across different browsers. This allows developers to ensure that the application works correctly for all users, regardless of the browser they are using.

Selenium makes it possible to run automated cross-browser UI tests, using a combination of the Selenium WebDriver and various browser drivers. This allows developers to quickly and easily create tests that can be run across multiple browsers, without having to write custom code for each individual browser. The Selenium WebDriver is a browser automation tool that can be used to control a web browser. This allows developers to simulate user actions and test the functionality of the application.

Selenium also provides a range of tools that can be used to inspect the UI of an application and generate reports. These tools can be used to identify any potential issues with the application's usability, compatibility, and performance. This allows developers to quickly identify and fix any issues before they become a problem for users.

Overall, Selenium is a powerful tool that can be used to perform automated cross-browser UI testing. It is a great tool for ensuring that an application works correctly across all browsers, and for identifying any potential usability issues.

UI Testing Process

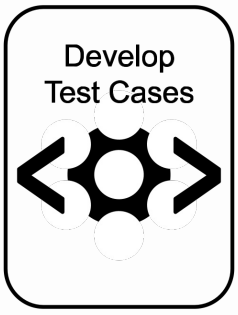
Testing Process

UI testing with Selenium involves the following steps:

Step 1

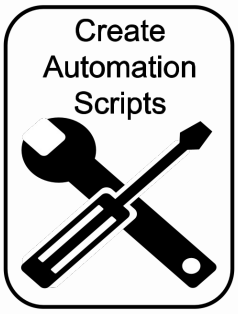
Set up the test environment by configuring the automation framework and the test data.

Step 2



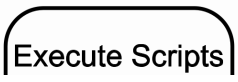
Develop the test cases based on the requirements and design.

Step 3

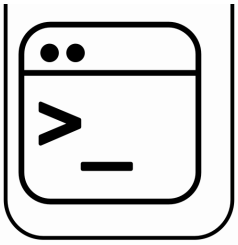


Use Selenium and other tools to create the automation scripts that will be used to execute the tests.

Step 4

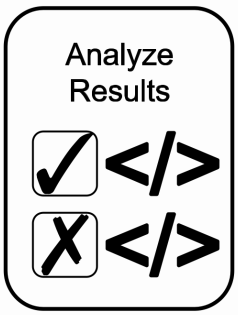


Execute the scripts by running them in the browser



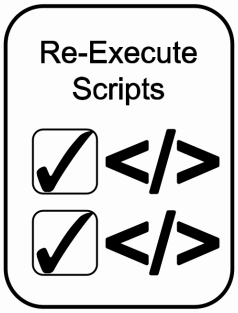
or by using a Selenium Grid.

Step 5



Analyze the results of the tests. If there are discrepancies, report any issues.

Step 6



Re-execute the tests as needed to ensure that the UI is functioning as expected.

Step 7

Clean up the test environment after the tests have been executed.

Setting Up Selenium

Installing Selenium

info

Important

Selenium and the other necessary components have already been installed on this system. The following instructions are here so you understand how you would set up Selenium on your own computer.

Before we can start using Selenium, we need to first install it. Codio uses the Ubuntu distribution of GNU/Linux. If you want to install Selenium on your personal machines, you may need to follow different instructions. Please see the Selenium documentation for more information.

You can use the following instructions to install and set up Selenium for UI tests. Let's start by installing the JDK. Copy and paste the following command into the terminal.

```
sudo apt-get install default-jdk
```

Some of the packages needed for this setup require the use of the Node Package Manager (NPM) for installation. If you do not already have it installed, here is how to do so:

```
sudo apt-get install npm
```

Once NPM is on your system, install the Selenium WebDriver.

```
npm install selenium-webdriver
```

Note, sometimes you may need to install the following packages before installing Selenium WebDriver.

```
npm install bufferutil@^4.0.1 utf-8-validate@^5.0.2
```

Installing ChromeDriver and Chromium

In addition to Selenium, we need to install ChromeDriver and Chromium. ChromeDriver is a tool to help you automate testing web apps across different browsers. Download the installation files with `curl` command and rename the file `chromedriver_linux64.zip`. **Remember**, these instructions are for a GNU/Linux environment. You may need to download different files on your personal machine.

```
curl
http://chromedriver.storage.googleapis.com/108.0.5359.71/chromed
river_linux64.zip --output  chromedriver_linux64.zip
```

Once the installation file has been downloaded, unzip the file.

```
unzip chromedriver_linux64.zip
```

Next, move the extracted folder to the `/usr/local/bin` directory:

```
mv chromedriver /usr/local/bin
```

Be sure to make the ChromeDriver executable:

```
chmod +x /usr/local/bin/chromedriver
```

Add the ChromeDriver to your PATH:

```
export PATH=$PATH:/usr/local/bin/chromedriver
```

With ChromeDriver installed, we can now install Chromium. Chromium is the open-source foundation of the Chrome browser. It is a full-featured browser that does not have some of the proprietary features found in Chrome. Start by updating the package list:

```
sudo apt update
```

Then install the `chromium` package:

```
sudo apt install chromium
```

First Test

Writing our First Test

Now that Selenium and its other dependencies are installed, we are ready to start writing tests. Start by importing the various classes and objects that we need to complete the test. The `Builder` class creates a new WebDriver instance. The `By` class helps us to locate elements on a webpage. The `Key` class represents non-text keys on the keyboard. The `until` module allows us to specify how long the WebDrive should wait. The `selenium-webdriver/chrome` module allows us to define a WebDriver client for the Chrome browser. In our case, we are using Chromium as the browser.

```
const {Builder, By, Key, until} = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');
```

Our test is going to be an asynchronous function named `example`. Instantiate the variable `driver`. Then set up a try-catch block. For now, leave the `try` portion empty. Catch and then log any errors. Under the `finally` section, wait until the WebDriver instance has finished running, and then quit.

```
const {Builder, By, Key, until} = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');

(async function example() {
  let driver;
  try {

  } catch (err) {
    console.error(err);
  } finally {
    if (driver) {
      await driver.quit();
    }
  }
})();
```

In the `try` section, let's create our WebDriver client. This is done by instantiating a `Builder` object. Set the browser to Chrome (Chromium in our case) and tell the client that we are running a headless browser. That

means our browser does not have a graphical interface. Then create the client with the `.build()` method.

```
const {Builder, By, Key, until} = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');

(async function example() {
  let driver;
  try {
    driver = await new Builder()
      .forBrowser('chrome')
      .setChromeOptions(new chrome.Options().headless())
      .build();
  } catch (err) {
      console.error(err);
  } finally {
    if (driver) {
      await driver.quit();
    }
  }
})();
```

Our test is rather simple. It is going to the Google search page, enter the term `selenium`, and then simulate pressing the `RETURN` key on the keyboard. It then verifies that the title of the page is `selenium - Google Search`. We are also going to give the wait command a one-second timeout. At each step of the process, our test waits until the previous task is complete before continuing.

```
const {Builder, By, Key, until} = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');

(async function example() {
  let driver;
  try {
    driver = await new Builder()
      .forBrowser('chrome')
      .setChromeOptions(new chrome.Options().headless())
      .build();
    await driver.get('https://www.google.com/');
    await driver.findElement(By.name('q')).sendKeys('selenium',
        Key.RETURN);
    await driver.wait(until.titleIs('selenium - Google Search'),
        1000);
  } catch (err) {
      console.error(err);
  } finally {
      if (driver) {
        await driver.quit();
      }
  }
})();
```

We did not add any feedback with this script. As long as there are no errors in the JavaScript, you will see a message that the command was successfully executed.

Assertions in Selenium

Assertions

Our first UI test in Selenium works as expected. However, we need to prove that the results from Selenium align with our expected behavior. To do this, we are going to use assertions.

In UI testing, we use the same assertions as in unit testing because Selenium assertions are taken from the Jest library. You can find more details about Jest assertions in the documentation.

Before we can use any assertions in our test, you have to first import the `assert` module.

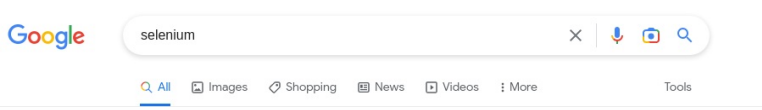
```
const assert = require ("assert");
```

The first step is to find the element on the web page using an XPath (see the next page). Once the element is visible, we can extract the value of the element. Then we can make an assertion that the value in the element is the same as an expected value.

```
let element = await driver.findElement(By.xpath("path-string"));
await driver.wait(until.elementIsVisible(element), 1000);
let elementValue = await element.getAttribute('value');
assert.strictEqual(elementValue, "expected value");
```

Try It Out

Using the above information, let's create an assertion for the Selenium script that searches for `'selenium'` in Google. Our Selenium test from the previous page simulates a Google search for `selenium`. If we were working with a proper browser, we would see results like the image below. At the top of the search results page would be another search box with `selenium` in it. We want to assert that this is the case.



depicts the top part of a google result page for selenium. there is a Google search box with "selenium" in it.

Start by requiring the `assert` module.

```
const {Builder, By, Key, until} = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');
const assert = require ('assert');
```

At the end of the `try` block, we are going to start our assertion. Create the variable `searchElement` that represents the search box element on the results page. Use the element name `'q'` to specify the element with the search box. Then have the WebDriver wait until `searchElement` is visible.

```
(async function example() {
  let driver;
  try {
    driver = await new Builder()
      .forBrowser('chrome')
      .setChromeOptions(new chrome.Options().headless())
      .build();
    await driver.get('https://www.google.com/');
    await driver.findElement(By.name('q')).sendKeys('selenium',
        Key.RETURN);
    await driver.wait(until.titleIs('selenium - Google Search'),
        1000);

    // assertions
    let searchElement = await driver.findElement(By.name('q'));
    await driver.wait(until.elementIsVisible(searchElement),
        1000);

  } catch (err) {
    console.error(err);
  } finally {
    if (driver) {
      await driver.quit();
    }
  }
})();
```

Then, once the element is visible, we are going to extract the `value` from `searchElement` and store it in the `searchText` variable. This would be the text inside the search box. To verify the value of `searchText`, log it to the console. Finally, create an assert statement that compares `searchText` to `"selenium"`. Use `strictEqual` since we are comparing strings.

```
    // assertions
    let searchElement = await driver.findElement(By.name('q'));
    await driver.wait(until.elementIsVisible(searchElement),
        1000);
    let searchText = await searchElement.getAttribute('value');
    console.log("Get the text from search element:",
        searchText);
    assert.strictEqual(searchText, "selenium");
```

Running the script should produce the following output:

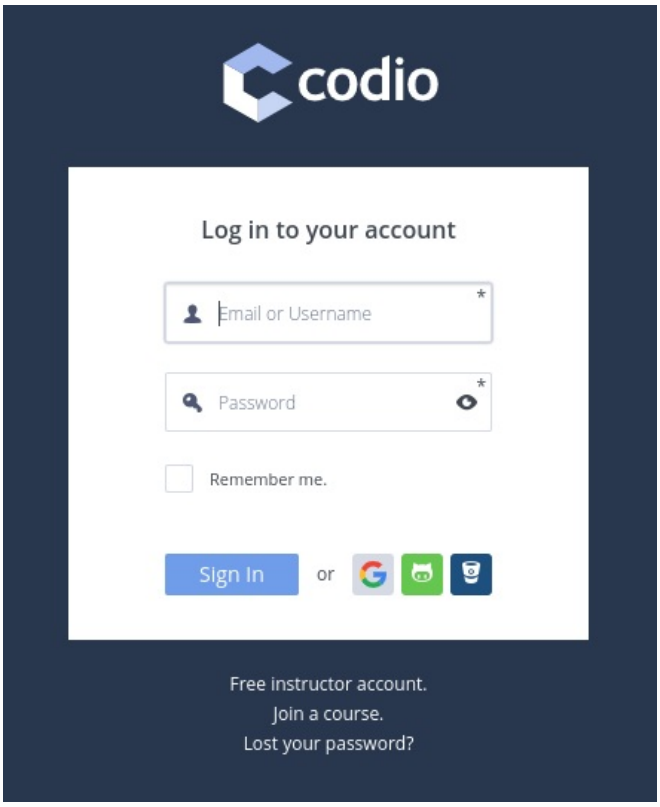
```
Get the text from search element: selenium
```

JavaScript has many libraries that also help with testing and have built-in functions and tools. For example, the Chai assertion library can be used instead of the default assertion module. This way you can pair a library that you may already know or one that has more flexibility with UI testing in Selenium.

Using XPaths and CSS Selectors

Sample using XPaths and CSS Selectors

We can select a specific part of a web page using XPath and CSS selectors. The following examples use the login page for Codio.



depicts the login page (username and password) for Codio

Both the XPath and CSS selectors represent an “address” or a way of identifying elements on the page. From there, you can use Selenium to extract information from these elements. For example, here is the XPath to the Codio logo on the login page.

```
/html/body/main/div/div[1]/img
```

And here is how to locate the same image using CSS selectors.

```
#codio > div > div.auth.auth-login > img
```

Id

Finding an element by its `id` with the XPath is done using: `[@id='example']` and with CSS using: `#`.

```
XPath: //div[@id='example']
CSS: #example
```

Find the section of code with the comment `// Using XPath and CSS` and add the following code snippet below. This example searches for the `Remember me` text using its XPath. Then click the `TRY IT` button.

```
    // Using XPath and CSS
    let labelElement = await
        driver.findElement(By.xpath('/html/body/main/div/div[1]/div/form/div[
        ]/label'));

    await driver.wait(until.elementIsVisible(labelElement),
        1000);
    let labelText = await labelElement.getText();
    console.log(labelText);
```

Class

Finding an element by its `class` with the XPath is done using: `[@class='example']` and with CSS using: `.`.

Example:

```
XPath: //div[@class='example']
CSS: .example
```

Update the section of code with the comment `// Using XPath and CSS` with the following code snippet below. This example searches for the `Log in to your account` text using its CSS selector. Then click the `TRY IT` button.

```
    // Using XPath and CSS
    let titleElement = await driver.findElement(By.css('#codio >
        div > div.auth.auth-login > div > form > p'));
    await driver.wait(until.elementIsVisible(titleElement),
        1000);
    let titleText = await titleElement.getText();
    console.log(titleText);
```

Other Attribute Values

We can easily select the `username` element without adding a `class` or an `id` to the element.

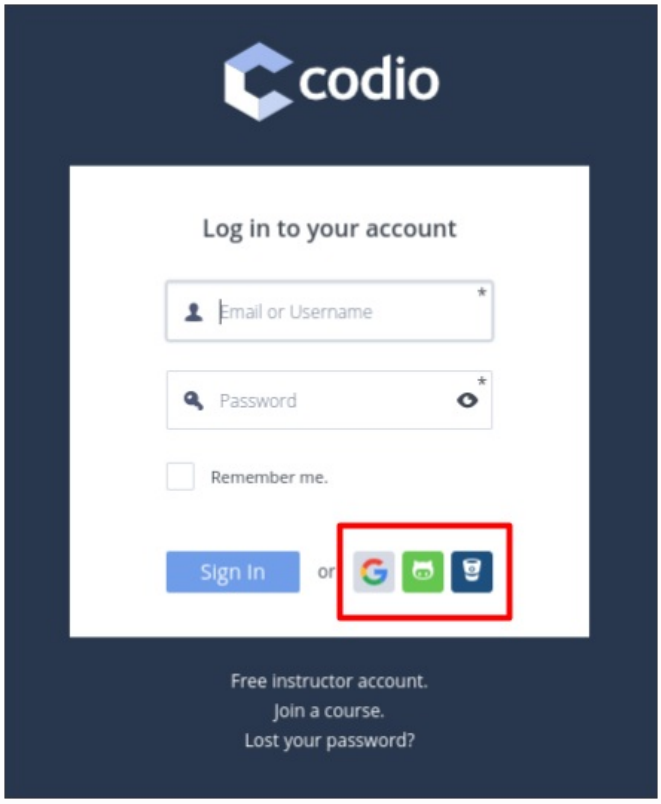
```
XPath: //input[@name='username']
CSS: input[name='username']
```

Update the section of code with the comment `// Using XPath and CSS` with the following code snippet below. This example searches for the `Email or Username` text using its XPath. Then click the `TRY IT` button.

```
    // Using XPath and CSS
    let userElement = await
        driver.findElement(By.xpath('//input[@name="user"]'));
    await driver.wait(until.elementIsVisible(userElement),
        1000);
    let placeholderText = await
        userElement.getAttribute('placeholder');
    console.log(placeholderText);
```

Choosing a Specific Match

There are three buttons to sign in with another service.



depicts the login page with a red box around the three buttons for logging in with a different service

They are all children of a parent `<div>` which has the class `oauthSigninButtons`. Here is their structure:

```
<div class="oauthSigninButtons u-clearfix">
    <a href="/service/raw?
oauthSignIn=1&amp;provider=google&amp;requestId=1675780678718&am
p;prevUrl=/home" class="plainLink oauthSigninButtons-oauth-btn
oauthSigninButtons-oauth-btn--google" title="Log in with
Google"><img class="oauthSigninButtons-oauth-btn--google-icon"
src="https://static-
assets.codio.com/dashboard/images/auth/google.14137f89247c1ab0.s
vg" alt="google"></a>
    <a href="/service/raw?
oauthSignIn=1&amp;provider=github&amp;requestId=1675780678718&am
p;prevUrl=/home" class="plainLink oauthSigninButtons-oauth-btn
oauthSigninButtons-oauth-btn--github" title="Log in with
Github"><i class="icon ss-icon icon-github ss-social-regular"
type="social">github</i></a>
    <a href="/service/raw?
oauthSignIn=1&amp;provider=bitbucket&amp;requestId=1675780678718
&amp;prevUrl=/home" class="plainLink oauthSigninButtons-oauth-
btn oauthSigninButtons-oauth-btn--bitbucket" title="Log in with
Bitbucket"><i class="icon ss-icon icon-bitbucket ss-social-
regular" type="social">bitbucket</i></a>
</div>
```

If we want to select a specific child from the parent element, we can use `nth-child()`. Put a number between the parenthesis to indicate which child you want to select. The Bitbucket link is the third child, so we would use:

```
CSS: .oauthSigninButtons :nth-child(3)
```

Add the following code to extract the `title` attribute from the Bitbucket button and log it to the console.

```
    // Using XPath and CSS
let linkElement = await
    driver.findElement(By.css('.oauthSigninButtons :nth-
    child(3)'), 10000);
await driver.wait(until.elementIsVisible(linkElement),
    1000);
let linkTitle = await linkElement.getAttribute('title');
console.log(linkTitle);
```

The above example selects a child element regardless of its type. For example, if you wanted to select the second link (as opposed to the second element), you would use `a:nth-of-type(2)` instead. The CSS would look like:

```
CSS: .oauthSigninButtons a:nth-of-type(2)
```

Let's extract the `title` attribute for the Github button by selecting the second child element that is a link.

```
    // Using XPath and CSS
    let linkElement = await
        driver.findElement(By.css('.oauthSigninButtons a:nth-of-
        type(2)'), 10000);
    await driver.wait(until.elementIsVisible(linkElement),
        1000);
    let linkTitle = await linkElement.getAttribute('title');
    console.log(linkTitle);
```

Child or Sub-Child

If an element could be inside another element or one of its child elements, it's defined in XPath using `//` and in CSS with whitespace.

Examples:

```
XPath: //div//a
CSS: div a
```

Towards the top of the page, you can see the XPath for the Codio logo. Instead of using the full path, we can truncate it. Instead, use `//img[@class="auth-form-logo"]` which states there is an `img` child element with the class name `"auth-form-logo"`. This is enough information to find the correct element. Add the code snippet below and press the `TRY IT` button. Selenium will not return the logo itself. Instead it will return the URL that hosts the image because that is what is in the HTML document.

```
    // Using XPath and CSS
    let imgElement = await
        driver.findElement(By.xpath('//div//img[@class="auth-
        form-logo"]'));
    await driver.wait(until.elementIsVisible(imgElement), 1000);
    let imgSource = await imgElement.getAttribute('src');
    console.log(imgSource);
```

Direct Child

HTML pages are structured like XML, with children nested inside of parents. If you can locate, for example, the first link within a `div`, you can construct a string to reach it. A direct child in XPath is defined by the use of a `/`, while on CSS, it's defined using `>`.

```
XPath: //div/a
CSS: div > a
```

Below the “Remember me” check box is a row of buttons. They are contained in a `<div>` with the class name `actions`. The `<span>` with the text `or` is a direct child of this `<div>`. We can access it with the following CSS selector. Add the code snippet below and press the `TRY IT` button.

```
// Using XPath and CSS
let spanElement = await
    driver.findElement(By.css('div[class="actions"] >
    span'));
await driver.wait(until.elementIsVisible(spanElement),
    1000);
let spanText = await spanElement.getText();
console.log(spanText);
```

Putting It Together

Testing with XPath

Let's see how we can combine assertions, Selenium, as well as extracting information from elements with XPath. We are going to enter incorrect information for the username/email as well as the password. We want to verify the website is returning the correct error message.

Start by setting up the basic structure of our UI testing. Import the needed modules. Then create an asynchronous function with try and catch blocks. In the try block, set up our WebDriver and open the Codio login page. In the catch block, log any errors.

```
const {Builder, By, Key, until} = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');
const assert = require('assert');

(async function example() {
  let driver;
  try {
    driver = await new Builder()
      .forBrowser('chrome')
      .setChromeOptions(new chrome.Options().headless())
      .build();
    await driver.manage().setTimeouts( { implicit: 10000 } );

    await driver.get('https://codio.com/p/login');
    console.log("Open site 'codio.com'");

  } catch (err) {
    console.error(err);
  } finally {
    if (driver) {
      await driver.quit();
    }
  }
})();
```

After loading the login page, find the element that has `user` for the `name` attribute. Then enter `fakename` in the input field. Log some text to provide feedback to the user.

```
    await driver.get('https://codio.com/p/login');
    console.log("Open site 'codio.com'");

    let nameInputEl = await driver.findElement(By.name('user'));
    await nameInputEl.sendKeys('fakename');
    console.log("Find login field and set name:", "fakename");
```

After entering a fake username, we are going to enter a fake password. Find the element with the class name `ReactPasswordStrength-input`. Use the `sendKeys()` method to enter `fakepassword` into the field. Log another message to the user.

```
    let nameInputEl = await driver.findElement(By.name('user'));
    await nameInputEl.sendKeys('fakename');
    console.log("Find login field and set name:", "fakename");

    let passwordInputEl = await
        driver.findElement(By.className('ReactPasswordStrength-
        input'));
    await passwordInputEl.sendKeys('fakepassword');
    console.log("Find password field and set password" ,
        "fakepassword");
```

Try to log in with the fake credentials by clicking the `Sign In` button. We do this selecting the button with `Sign In` as the text attribute. Use the `.click()` method to simulate a button press. Once again, log a message to the user.

```
    let passwordInputEl = await
        driver.findElement(By.className('ReactPasswordStrength-
        input'));
    await passwordInputEl.sendKeys('fakepassword');
    console.log("Find password field and set password" ,
        "fakepassword");

    let signInButtonEl = await
        driver.findElement(By.xpath("//button[text()='Sign
        In']"));
    await signInButtonEl.click();
    console.log("Find and click 'Sign In' button");
```

We want to capture the error message from the website. Use the appropriate XPath to select the element with the authentication error. Once this element is visible, extract its text. Log the extracted message.

```
    let signInButtonEl = await
        driver.findElement(By.xpath("//button[text()='Sign
        In']"));
    await signInButtonEl.click();
    console.log("Find and click 'Sign In' button");

    let errorEl = await
        driver.findElement(By.xpath("//div[@class='alert auth-
        alert alert--error']"));
    await driver.wait(until.elementIsVisible(errorEl));
    var displayedMessage = await errorEl.getText();
    console.log("Get the error message:", displayedMessage);
```

Finally, we are ready to use an assertion statement to verify the website is working properly. Use the `.strictEqual()` method to compare the string captured from the element with the expected string. Log a message to the user.

```
    let errorEl = await
        driver.findElement(By.xpath("//div[@class='alert auth-
        alert alert--error']"));
    await driver.wait(until.elementIsVisible(errorEl));
    var displayedMessage = await errorEl.getText();
    console.log("Get the error message:", displayedMessage);

    assert.strictEqual(displayedMessage, "User name or password
        is incorrect");
    console.log("Check that error message text is similar to
        'User name or password is incorrect'");
```

▼ Code

Your final code should look like this:

```
const {Builder, By, Key, until} = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');
const assert = require('assert');

(async function example() {
  let driver;
  try {
    driver = await new Builder()
      .forBrowser('chrome')
      .setChromeOptions(new chrome.Options().headless())
      .build();
    await driver.manage().setTimeouts( { implicit: 10000 } );

    await driver.get('https://codio.com/p/login');
    console.log("Open site 'codio.com'");
```

```
    let nameInputEl = await driver.findElement(By.name('user'));
    await nameInputEl.sendKeys('fakename');
    console.log("Find login field and set name:", "fakename");

    let passwordInputEl = await
        driver.findElement(By.className('ReactPasswordStrength-
        input'));
    await passwordInputEl.sendKeys('fakepassword');
    console.log("Find password field and set password" ,
        "fakepassword");

    let signInButtonEl = await
        driver.findElement(By.xpath("//button[text()='Sign
        In']"));
    await signInButtonEl.click();
    console.log("Find and click 'Sign In' button");

    let errorEl = await
        driver.findElement(By.xpath("//div[@class='alert auth-
        alert alert--error']"));
    await driver.wait(until.elementIsVisible(errorEl));
    var displayedMessage = await errorEl.getText();
    console.log("Get the error message:", displayedMessage);

    assert.strictEqual(displayedMessage, "User name or password
        is incorrect");
    console.log("Check that error message text is similar to
        'User name or password is incorrect'");

  } catch (err) {
    console.error(err);
  } finally {
    if (driver) {
      await driver.quit();
    }
  }
})();
```

If the assertion passes, you should see only the messages that we logged to the console.