# Hands-on Lab: Testing React with Jasmine

Skills Network

**Estimated time needed:** 45 minutes

In this lab, you will learn how to test your logic in React with Jasmine.

The lab starts with a Create React Application and then adds the logic and relevant tests.

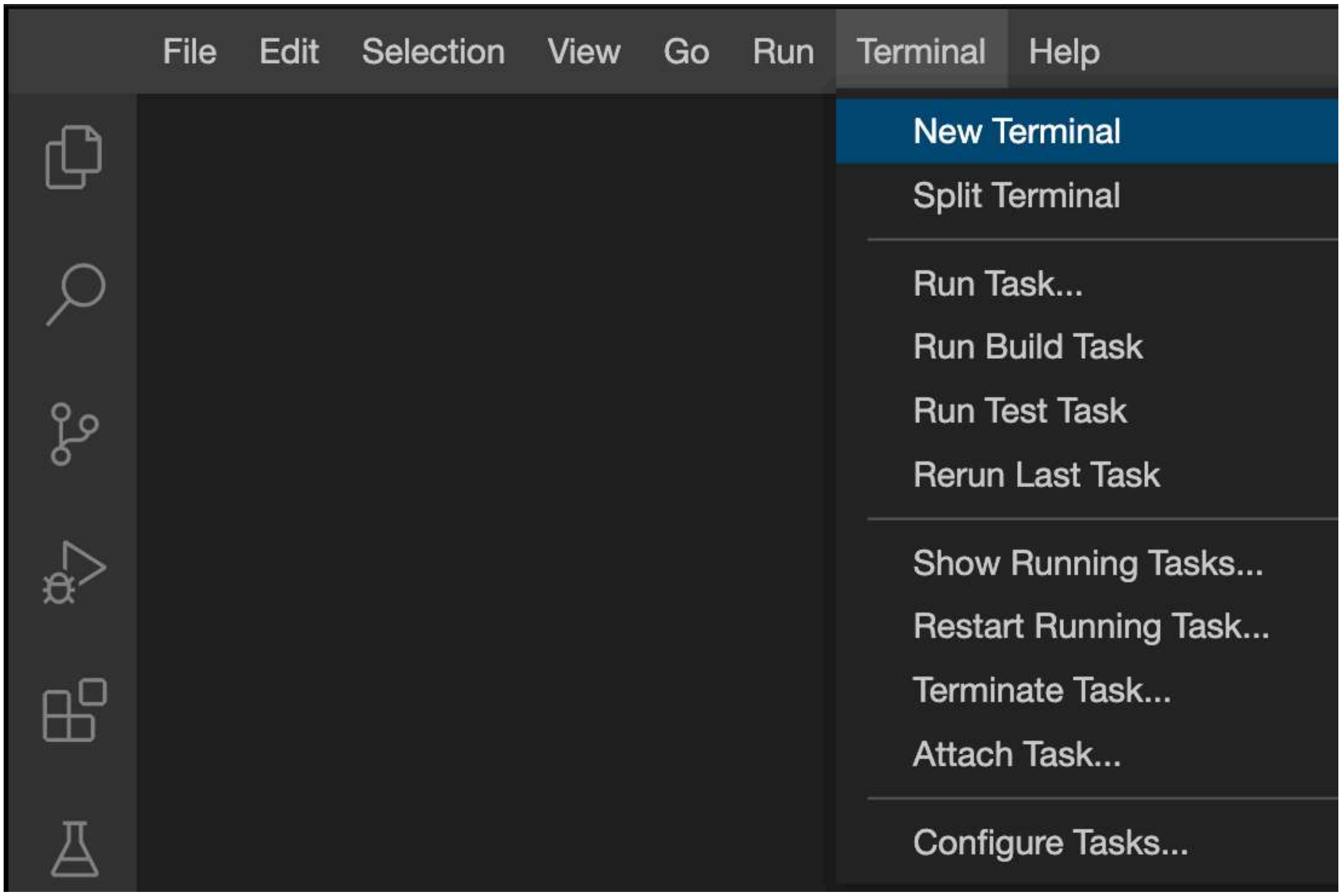
## Objectives:

After completing this lab, you will be able to:

- Write unit tests using Jasmine
- Run a test suite using the command line

## Set-up: Get the source code

1. Open a terminal window by using the menu in the editor: **Terminal** > **New Terminal**.



2. If you are not currently in the project folder, copy and paste the following code to change to your project folder.

```
cd /home/project
```

Copied! Executed!

3. Run the following command to clone the Git repository that contains the starter code needed for this lab if the Git repository doesn't already exist.

```
[ ! -d 'test-react-with-jasmine' ] && git clone https://github.com/ibm-developer-skills-network/test-react-with-jasmine.git
```

Copied! Executed!

4. Change to the directory **test-react-with-jasmine** to start working on the lab.

```
cd test-react-with-jasmine
```

Copied! Executed!

5. Install the dependencies.

```
npm install
```

Copied! Executed!

6. Run the React application.

```
npm start
```

Copied! Executed!

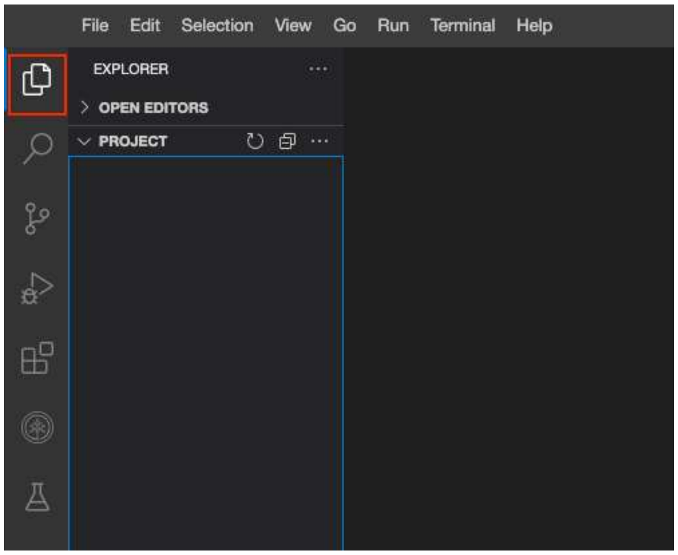
7. To test your application in your browser, run the application first.

Launch Application

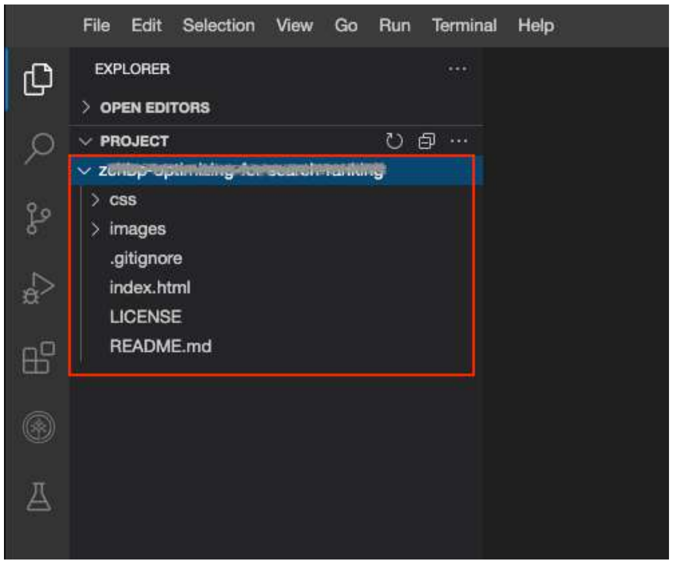
## Working with files in Cloud IDE

If you are new to Cloud IDE, this section will show you how to create and edit files, which are part of your projet, in Cloud IDE.

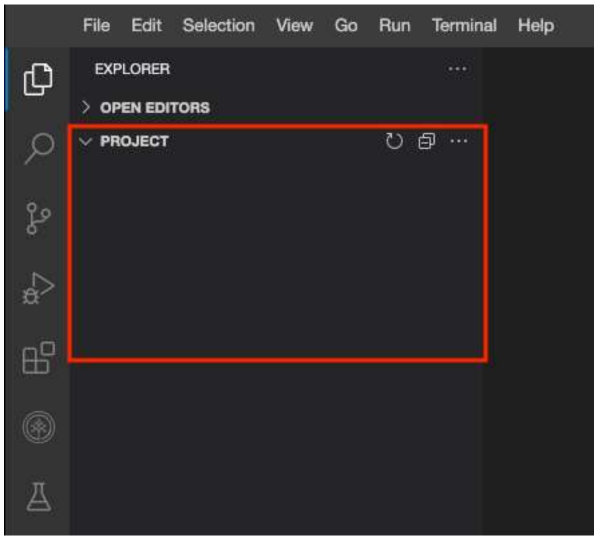
To view your files and directories inside Cloud IDE, click on this files icon to reveal it.



If you have cloned (using `git clone` command) boilerplate/starting code, then it will look like below:

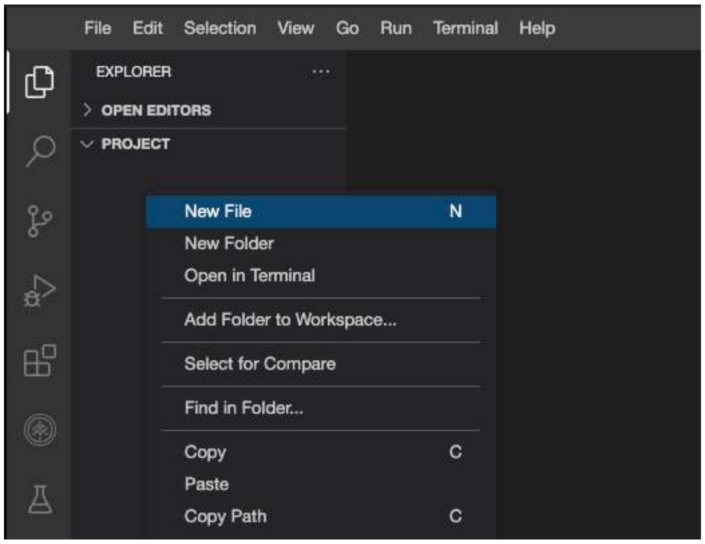


Otherwise a blank project looks like this:



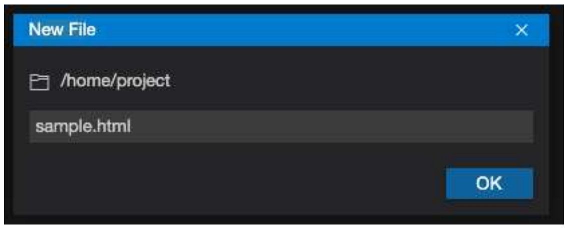
## Create a new file

You can right-click and select the New File option to create a file in your project.

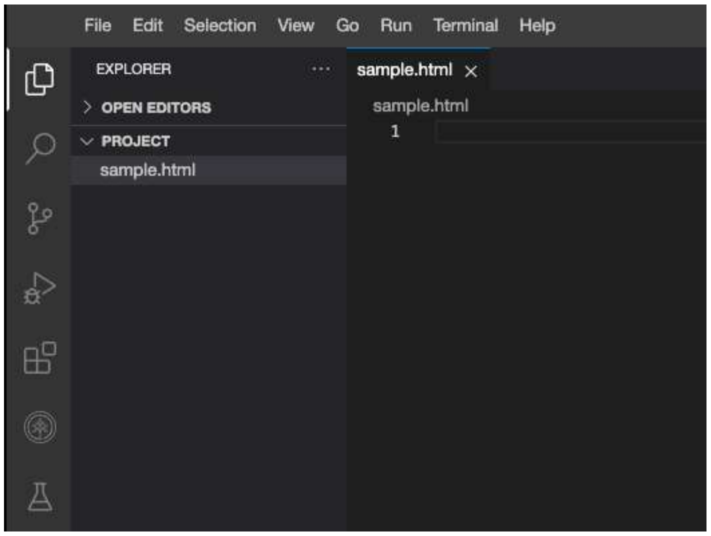


You can also choose `File -> New File` to do the same.

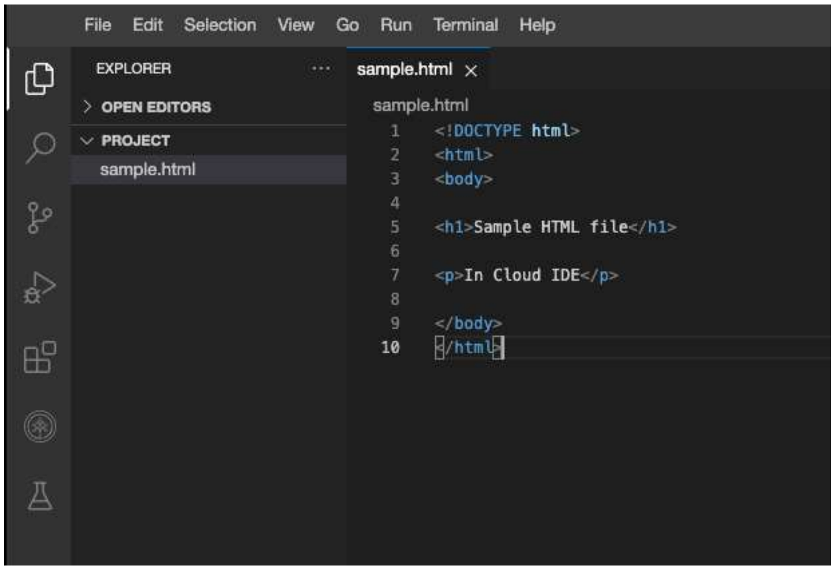
It will then prompt you to enter name of this new file. In the example below, we are creating `sample.html`.



Clicking on the file name `sample.html` in the directory structure will open the file on the right pane. You can create all different types of files; for example `FILE_NAME.js` for JavaScript file.



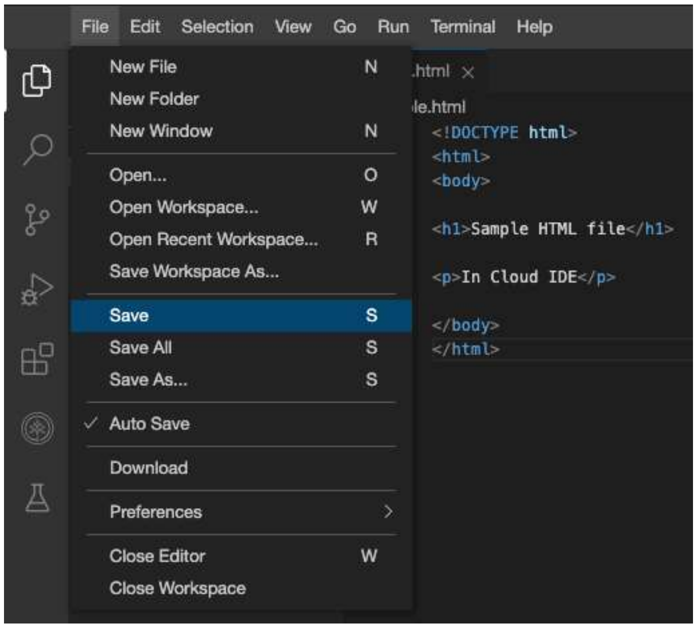
In the example, we just pasted some basic html code and then saved the file.



```html
<!DOCTYPE html>
<html>
<body>

<h1>Sample HTML file</h1>

<p>In Cloud IDE</p>

</body>
</html>
```

And saving it by:

- Going in the menu.
- Press ⌘ `Cmd + S` on Mac or `CTRL + S` on Windows.
- Or it can Autosave it for you too.



```
<!DOCTYPE html>
<html>
<body>

<h1>Sample HTML file</h1>

<p>In Cloud IDE</p>

</body>
</html>
```

## Exercise 1: Install Jasmine with npm

The first step is to install the `jasmine` npm package. This module allows you to run Jasmine specs for your Node.js code. The output will be displayed in your terminal by default.

Before you progress, let's stop our React application and get the cursor back in Terminal by pressing `CTRL + C`.

You can use npm to install Jasmine locally in your project:

```
npm install --save-dev jasmine
```

Copied! Executed!

With the above local installation, you can invoke the CLI tool using `npx jasmine ...` commands.

## Exercise 2: Initialize Jasmine

Run the following command to initialize the project for Jasmine:

```
npx jasmine init
```

Copied! Executed!

By initializing Jasmine, it creates the `spec/support/jasmine.json` file, which acts as a configuration file for Jasmine.

```
{
  // Spec directory path relative to the current working dir when Jasmine is executed.
  "spec_dir": "spec",

  // Array of filepaths (and globs) relative to spec_dir to include and exclude
  "spec_files": [
    "**/*[sS]pec.?(m)js"
  ],

  // Array of filepaths (and globs) relative to spec_dir to include before Jasmine specs
  "helpers": [
    "helpers/**/*.?(m)js"
  ],

  // Configuration of the Jasmine environment
  // "env" is optional, as are all of its properties.
  "env": {
    // Stop execution of a spec after the first expectation failure in it
    "stopSpecOnExpectationFailure": false,

    // Run specs in semi-random order
    "random": false
  }
}
```

Copied!

Optionally, you can also generate example spec and source files:

```
npx jasmine examples
```

Copied! Executed!

You should now be able to write your first suite.

But first, we will make a change in `package.json` to set `jasmine` as the test script.

```
"scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "jasmine",
    "eject": "react-scripts eject"
},
```

Copied!

# Exercise 3: Write first suite

First, let's create a module that we can test.

Create a new file called `src/helloWorld.js` with the following code:

```
function helloWorld() {
    return "hello world";
}

module.exports = helloWorld;
```

Copied!

This is a simple module that returns `hello world` when called.

To test this module, we will create a Jasmine spec file called `spec/helloWorldSpec.js` with the following code:

```
const helloWorld = require('../src/helloWorld.js');

describe("helloWorld", () => {
    it("returns hello world", () => {
      var actual = helloWorld();
      expect(actual).toBe("hello world");
    });
  })
```

Copied!

## Jasmine functions

The following function definitions provide usage details.

### describe

The `describe` function is used to group related specs and is usually found at the top level of each test file. The string parameter is used to name the collection of specs and will be concatenated with specs to form a spec's full name. This makes it easier to find specs in a large suite. If you name them correctly, your specifications will read as full sentences in traditional Behavior-driven-development (BDD) style.

### it

Specs are defined by calling the global Jasmine function `it`, which, like `describe`, takes a string and a function. The string is the title of the spec, and the function is the spec or test. A spec contains one or more expectations that test the state of the code. In Jasmine, an expectation is an assertion that is either true or false. A passing spec contains all true expectations. A failing spec contains one or more false expectations.

### expect

Expectations are built with the function `expect`, which takes a value called the actual. It is linked with a Matcher function, which takes the expected value.

Jasmine has a rich set of matchers included, and you can find the full list in the API docs.

# Exercise 4: Let's run a test

To run the unit test on hello world, we can either run

```
npx jasmine
```

Copied! Executed!

or

```
npm test
```

Copied! Executed!

You will notice the following outcome:

```
1 spec, 0 failures
```

This means that the test has passed.

If you notice, under `Started`, one green dot `.` represents the passed test. A failed test will have a red F.

# Exercise 5: New module and test

Let's add two new modules.

`src/factorial.js`

```
const factorial = (inputNumber) => {
  if (inputNumber < 0)
    throw new Error("We don't allow factorial of a negative number.");

  if (inputNumber === 1 || inputNumber === 0) return 1;

  return inputNumber * exports.factorial(inputNumber - 1);
};

exports.factorial = factorial;
```

Copied!

And `spec/factorialSpec.js`

```
const factorial = require("../src/factorial.js");

describe("factorial", function () {
  var calc;

  //This will be called before running each spec
  beforeEach(function () {
    calc = factorial;
  });

  describe("when calc is used to peform factorial operations", function () {
    it("should be able to calculate factorial of 0", function () {
      expect(calc.factorial(1)).toEqual(1);
    });

    it("should be able to calculate factorial of 1", function () {
      expect(calc.factorial(1)).toEqual(1);
    });

    it("should be able to calculate factorial of 5", function () {
      expect(calc.factorial(5)).toEqual(120);
    });

    it("should be able to throw error in factorial operation when the number is negative", function () {
      expect(function () {
        calc.factorial(-2);
      }).toThrowError(Error);
    });
  });
});
```

Copied!

Let's also add a test to throw an error if passed argument is not a number.

```
    it("should be able to throw error in factorial operation when the input is not a number", function () {
      expect(function () {
        calc.factorial(true);
      }).toThrowError(Error);
    });
```

Copied!

Now, run the tests by executing

```
npx jasmine
```

Copied! Executed!

As a result, the following message will be generated:

```
1. Started
2. .....F
3. Failures:
4. 1) factorial when calc is used to peform factorial operations should be able to throw error in factorial operation when the input is not a number
5.   Message:
6.     Expected function to throw an Error.
7. ...
8. 6 specs, 1 failure
```

Copied!

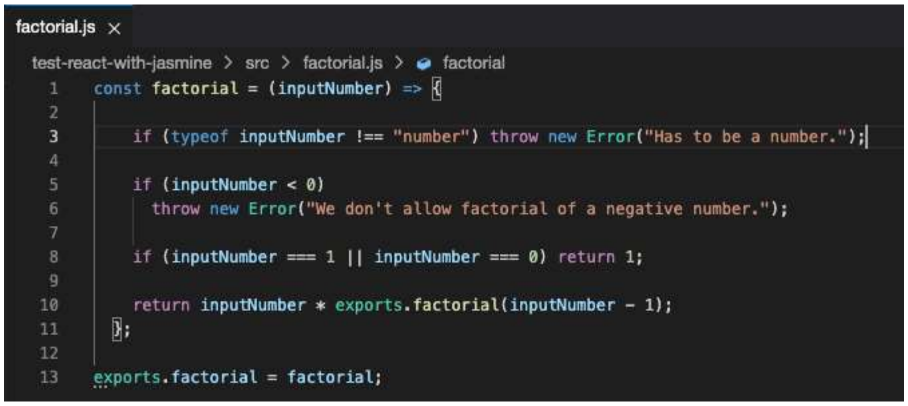
To pass this test, we must modify our code by inserting the following line at the beginning of the factorial function in `src/factorial.js`

```
1. 1
```

```
1.   if (typeof inputNumber !== "number") throw new Error("Has to be a number.");
```

Copied!

And it will look like below:



```
const factorial = (inputNumber) => {

    if (typeof inputNumber !== "number") throw new Error("Has to be a number.");

    if (inputNumber < 0)
      throw new Error("We don't allow factorial of a negative number.");

    if (inputNumber === 1 || inputNumber === 0) return 1;

    return inputNumber * exports.factorial(inputNumber - 1);
  };

exports.factorial = factorial;
```

# Exercise 6: Run the test again

Rerun your tests by executing:

```
1. 1
```

```
1. npx jasmine
```

Copied! Executed!

The following message will be generated, where 6 dots `......` represent 6 passed tests.

```
1. 1
2. 2
3. 3
```

```
1. Started
2. ......
3. 6 specs, 0 failures
```

Copied!

**Congratulations! You have completed the lab for Testing React with Jasmine.**

## Summary:

In this lab, you used Jasmine to test two modules written in React.

## Author(s)

**Muhammad Yahya**

## Changelog

| Date | Version | Changed by | Change Description |
|---|---|---|---|
| 2023-03-29 | 0.3 | Steve Hord | Replaced SN logo |
| 2023-03-24 | 0.2 | Steve Hord | QA pass with edits |
| 2023-02-27 | 0.1 | Muhammad Yahya | Initial version created |