**TDD**

**Course Objectives**

* To understand what TDD is
* How TDD is beneficial to Software Development
* Learn how to write test cases
* Learn what Unit testing is
* Learn what Integration Testing is
* Learn the necessary TDD tools like Mocha, Chai, and Selenium

**Overview**

Our Approach to learning TDD:

1. Testing functions directly from a "test" file. This is for your to focus on just testing without having to worry about folder structure, configurations, or docker.
2. Unit testing the methods of Model classes. This is to slowly introduce you to the "real" thing without having to worry about proper mvc structure, or worrying about controllers, and routes. You will also be exposed to unit testing + docker.
3. Perform integration testing alongside with unit testing. For you to work on an actual express mvc + docker.

## Objectives

* Understand what TDD is
* How different TDD is versus the traditional way of developing software.
* Why it's important to write tests first
* Learn the benefits of using TDD
* Learn when to use and when not to use TDD

# What is TDD?

TDD or Test Driven Development is the practice of **writing tests before we write code.** The idea is we would make **assertions**of what we would like our code to do before we actually write the implementation.

We are "testing" our application all the time: We write the code, **test our app manually (via the browser or the console/terminal)**, make code revisions, repeat. This is how we currently develop our app. But with TDD, the order is different:

1. Design a feature
2. Write a test that fails (RED). This test will model a specific behavior or outcome in your code
3. Write the code that will make your feature work and pass the test (GREEN).
   1. Also make sure that prior tests still pass!
4. Refactor your code and improve your design
5. Move to the next feature and repeat the process

We'll use tools like Selenium and Mocha to automate the testing for us.

**Why Write The Tests First?**

A lot of people see the benefit in writing tests before writing the code. Others would disagree. They think that it doesn't matter whether you write the tests before or after the code. **But in TDD,** **it's really all about adopting a practice for ensuring that you're writing testable code.**

**TDD forces you to consider how you design your code.** If you take an approach of writing tests after you've written a lot of code, there's a chance that you will have written some code throughout that process that's going to be really hard to extract and isolate just to be able to test what you've written.

**How do you "test" an app?**

As mentioned earlier, you are already used to testing your app the traditional way (via the browser or the termina/console). But with TDD, we'll use certain **tools**that our app needs to install, import, and invoke. These tools are going to be invoked in specific test files within your project folder (typically stored in a **/test**folder) and you are going to write the **test code.** You will then need to run a **test**command**,**and see **test results**(in the terminal/console).

Before diving in to the meaty part of the subject, you must already have this mindset: that at this point, **before writing any line of code for that awesome app you are thinking of, you need to have the test code for it first.**



You won't probably enjoy it at first. As developers, and specially during our training, we are usually motivated to building that app as soon as possible and you might probably be thinking that writing tests would just waste your time or just hinders our productivity. Well, that's kinda true... When you're just thinking about creating Hacker Hero assignment apps.

Yes, real world products that gets delivered the fastest are still much preferable, but it must not compromise quality, and maintainability. And TDD just solves that. And in terms of speed, yes it might take you a while in the early stages of the app creation, but those tests you wrote basically just saved you from the hundreds of hours you will definitely be spending on debugging code in the later stages. So speed-wise, TDD still wins.

**So.. You still think TDD is NAH?**

Okay, so I mentioned "test code". That's what usually puts people off.

Do I really have to write a code... just to test.. MY CODE?! That just means instead of writing X lines of code, I'd be writing 2X as much!

Well the answer is, YES! I'm sorry. But don't worry! It might take you a while for you to get comfortable with the test tools, but the concept is really really easy. Here's the whole idea of how testing works.

So, you have this code for your app:

**fairness.js**

class Fairness {

    // a method that simply returns true if a & b is equal, false otherwise.

    is\_fair(a, b) {

        let result = false;

    // do stuff here

        return result;

    }

}

module.exports = Fairness;

And below is just the basic test file, to test your code:

**test.js**

// some necessary import stuff here

// const Fairness = require('./fairness');

describe("Fairness", function() {

    it("should return false if values passed are not equal", function(){

        let fairness = new Fairness();

        let result = fairness.is\_fair(1, 2);

        expect(result).to.be.false;

    });

});

// more test code below

And then you run a command to execute the test: **npm test**or simply **mocha (**if you haven't setup your package.json file to run the **test**script)**.** Test results are very intuitive as well. They are:

1. Pass/Passing
2. Fail - a fail test will return an Assertion error, and will also return some bit of information why the test failed.
3. Error - An Exception other than an Assertion error. Also returns some bit of information why it got an error.
4. Pending

So you see, most of the code in the test file are codes you are already familiar with. Importing stuff, invoking stuff, and expecting the return values of certain methods you invoked. The only bit of code that's different there is the **expect()**method (It's a special method from the **Chai**library - a library that you will be going to use later). You are basically just invoking a method, passing values to this method, and expecting a predicted return value using Chai's **expect()**method. That's all there is.

So now you're probably worried, and thought, "Well, if I have to write the test first, then that means I would have to know in advance what certain parameters are required, and what certain values a specific method returns?" Well, in your previous training, you would already know that when building an app, as much as possible, each methods should only do one specific thing.

For example, when creating a new user, you would normally have a method called createUser(params){} that returns either of the values below:

1. It returns true if user creation was successful
2. It returns an object that contains the user creation status, etc if user creation was successful.
3. It returns false if user creation failed.
4. It returns an object that contains the user creation status, error messages, etc if user creation failed.

In your test code, you would probably expect **createUser(params)**method to just return booleans initially. And that's perfectly okay. But as your app evolves, you would probably think returning a JSON object with specific attributes and values would be better. When you have these changes for your app, just make sure you modify your test code first, before actually modifying your actual code. If you take an approach of modifying tests after you've modified a lot of code,  there's a chance that you will have written some code throughout that process that's going to be really hard to extract and isolate just to be able to test what you've written.

**Benefits of Using TDD?**

* Higher Code Quality
* The code you've written is covered by tests
* You can change or extend your code with much more confidence
* Reduces the time required for project development
* Cost-efficient in the long run
* Your tests will serve as the detailed project documentation for your code base

**What TDD Can't Help You**

* TDD can't make code design decisions for you. You still have to design the code yourself.
* You can still write bad code with TDD. It's just far less likely. Because with TDD approach, you know that the piece of code you've written should behave as expected.

### When to use TDD and when not to

Test-Driven Development is a very valuable technique to use when you intend to go a bit more slowly in order to create something that is as bug-free as possible. TDD is most valuable when you are building a production system and you know that any defect will be very expensive to your company -- such as when building Amazon.com, for example. TDD is also a great choice when writing code that you expect will be important and will be used for a very long time -- such as when building the public-facing API for your company, for example. However, it is not as useful when exploring new technology areas, since there is no real cost of an undiscovered defect, and since this code will be discarded once the technology is well-understood.

With this in mind, TDD is generally not worth the cost when working through prior Hacker Hero assignments.

**Moving Forward**

You are not allowed to write a line of code before you write a **failing test**. Write failing tests first. An ideal scenario would be:

1. You write a **failing test**
2. You see the test fail (You read the exceptions and you see why it fails)
3. You write a very small code just enough to get that test to pass
   1. You make sure that other tests pass as well.
4. Once that test passes, you can then do code clean up (refactor). But never write more code than is required to pass the failing test
5. Repeat.

If you've never done this before, it will feel a little awkward. Spending hours and hours writing code only to uncover a dozen bugs and then going back and try to fix all of those bugs is what you are probably used to. This routine is what you're already familiar with but don't worry. **With TDD, you are literally fixing everything that is immediately broken as you work.** The side effect of this is, you will end up writing code that is loosely coupled. Because, again, if you want to test the behaviour of something, you always want to test it in isolation. The result is cleaner code.

## Objectives

* Learn how to write test cases
* Learn what the describe statement is
* Learn what the it statement is
* Learn tools like Chai and Mocha

# Writing Your First Test

Let's write our first test using Mocha testing framework. Open your terminal and type the following:

1. mkdir sample\_mocha
2. cd sample\_mocha
3. npm init (You will be prompted to fill in certain fields. Just hit enter until finish)
4. npm install --save-dev -g mocha
5. npm install --save-dev chai
6. mkdir test
7. Create a test file called **test.js**inside the **test**folder and paste the code below:

const chai = require('chai');

const expect = chai.expect;

describe("Login feature", function(){

    it("data.status should return true if email exists in the database.");

    it("data.result.email should be equal to the inputted email if email exists in the database.");

});

  9. Then simply run ***mocha*** in the terminalto see the test result

The it call identifies each individual tests but by itself it does not tell Mocha anything about how your test suite is structured. How you use the describe call is what gives structure to your test suite. Here are some of the things that using describe to structure your test suite does for you. Here's an example of a test suite, simplified for the purpose of discussion:

When you run **mocha**in the terminal, you will see **pending**as the test result. **pending**is rarely used. Typically when we work on an app, we always want to write a **fail**test first. A fail test looks like like the one below:

const chai         = require('chai');

const expect      = chai.expect;

const UserModel    = require('../models/user.model');

describe("Login feature", function() {

it('data.status should return true if email exists in the database.', async function(){

let email = 'jrosales@village88.com';

let userModel = new UserModel();

let data = await userModel.getUser(email);

/\* Check and expected data.status is equal to true \*/

expect(data.status).to.equal(true);

});

});

This will return **fail**because we really haven't created the UserModel or basically the code used to **retrieve the user from the database**yet. To make the test **pass,**you have to actually **write the code** for it to pass.

## Objectives

* To be able to apply the TDD concept
* To struggle on writing test first, before writing a single code.
* To be comfortable with Chai's expect method
* To be comfortable in using the mocha command.

# Hacker Hero Challenges

Look at the first 10 challenges at Hacker Hero: Basic Algorithm course.  This time, you would solve it using JavaScript but do it using TDD approach.  Remember to create test cases first and make sure test cases don’t pass first.  Then write the function.  For each challenge, create a separate file containing the test cases.  Then demonstrate how you could either individually run the test file or run all of them at once.  Once you did this, write a short reflection on how this approach compared with the non TDD approach.  How do you think this would help in the real world environment?

Note:

* You don't need to create a node server for this assignment. Just create separate javascript files for each of these challenges.
* You don't need to dockerize the challenges for this assignment.

START WORKING ON THIS

## Objectives

* Learn what unit testing is
* Learn how to run tests in your container
* Learn how to come up with test statements
* Learn which code we have to unit test
* Learn which code we don't have to unit test
* Learn how to effectively write it statements

# Unit Testing

Mocha is the testing framework. Chai is optional because Node already has an assertion library. But node's assertion library is really not as powerful as Chai so we use chai.

When we unit test our code, **we don't have to unit test each methods in our routes and controllers.** What we will be focusing on is to unit test the methods in our Models. The reason is because we will be able to test them later using Selenium. Routes and Controllers typically requires us to make **http requests**in order for us to test them**.**Though we can do that using **chai-http**, the best approach is to use Selenium.

Common chai methods we are going to use:

expect(object)

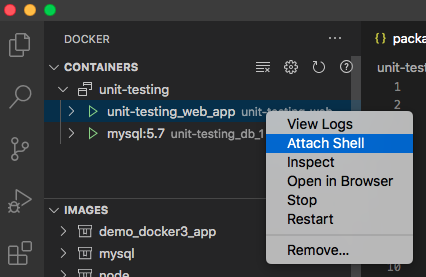
    .to.equal(expected)

    .to.eql(expected) // deep equality

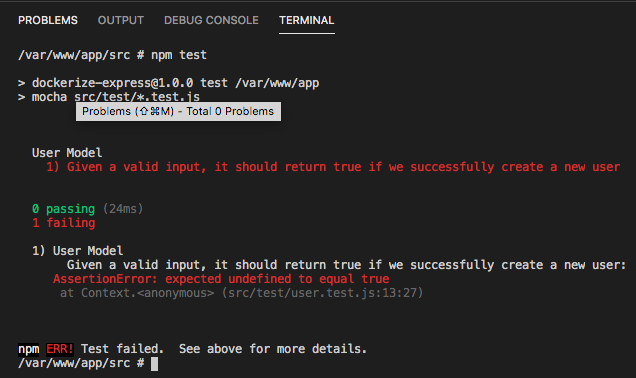
To understand unit testing, go ahead and download this simple express app: [unit-testing](https://github.com/HHOliver88/unit-testing)

Drag and drop the files to VS Code and run this app by right-clicking on docker-compose.yml and choose **Compose Up.**  
Inside the **/test**folder, you should see a single test: **"Given a valid input, it should return true if we successfully create a new user"**

To run a test in our container, we will need to **attach a terminal**to our **app**service. Attaching to a terminal is basically like having a terminal or command prompt access to our desired container. To attach a terminal, go to your Docker extension, look for the unit-testing container, right click on the **unit-testing\_web\_app** service and choose **Attach Shell:**



You can then run **npm test**in the shell to run our tests. You should be able to see a result like below:



**npm test** just basically triggers a **mocha** command. We set this up in **package.json** where we have:

"test": "./node\_modules/.bin/mocha src/test/\*.test.js"

The test result is something we expected. We expect it to **fail** because when you look at the UserModel, though it has the **createUser()** method expecting a params, it doesn't have the necessary code to make our assertion **pass.**Let's make it **pass** by copy and pasting the code below:

async createUser(params){

let result = false;

try{

let create\_user\_query = Mysql.format(`INSERT INTO users(name,email,password) VALUES(?,?,?);`, [params.name, params.email, params.password]);

let create\_user\_result = await executeQuery(create\_user\_query);

if(create\_user\_result){

result = true;

}

}catch(err){

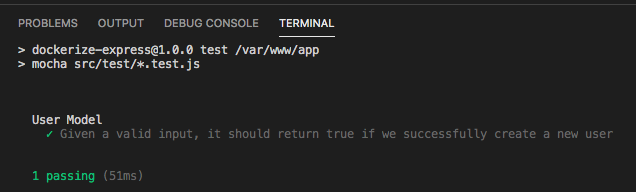
console.log(err);

};

return result;

}

Here's the result if you run **npm test**:



Notice that my code is enough to ONLY make the single assertion to PASS. It doesn't have any other if statements to handle other scenarios like:

* What should it return if user creation failed?
* What should it return if there's an exception error?
* What should it return if other required params are missing?
* etc.

These are questions that you can easily convert into assertions:

* Given \_\_\_\_ it should return \_\_\_\_ if user creation failed
* Given \_\_\_\_ it should return \_\_\_\_ if there's an exception error
* Given \_\_\_\_ it should return \_\_\_\_ if other required params are missing.

## Objectives

* To practice Unit Testing
* To learn how to be creative in writing descriptive test cases
* To be comfortable writing test cases on an app that's already built.

# Testing Models I

Pick any of the Express MVC assignment you’ve completed.  Pick two methods in a model.  Write at least 3 test cases for each of the model method (meaning you should have at least 6 test cases written). Make sure these test cases pass.

Note: We are just unit testing the methods, and are not testing a feature just yet. We just want to make sure that each of our individual methods work as expected.

In this assignment, the test cases are written AFTER the feature is implemented. This is not the TDD approach, but there are certain situations where in TDD is only implemented at the later part of the development. You should be able to handle situations like these as well.

START WORKING ON THIS

## Objectives

* To practice TDD on a close to real-world app feature.
* To reinforce habit of writing tests first before writing code.
* To be exposed on a development setup where pending test cases are provided.
* To reinforce learning of Unit Testing.

# Testing Models II

Create a Model necessary to Log a user in. MySQL Database is provided. You can download the assignment files [here](https://github.com/HHOliver88/tdd-login-assignment). Login credential is:  
email: testuser@test.com  
password: password123

The **pending**test cases are provided in the **./test**folder. Work on each pending tests one by one, and write the necessary code for it to pass before moving on to the other pending test cases.

Make sure you build this app using TDD approach. **You must write the test first, have it failed, then write the necessary code for it to pass.**

Run **npm test** command in a separate shell to see all your test output.

**Important**

1. No need to create Controller or Route files for this assignment. Let's focus on unit testing the Models.

**Instructions**

1. You can download the assignment files [here](https://github.com/HHOliver88/tdd-login-assignment).
2. You will need to run the node server first before running any tests.
3. Make sure you work on this assignment in a docker container. The necessarily files to build and run this app in a docker container is already provided.
4. Once done, you need to zip your app in one file and upload the zipped file in Hacker Hero.

START WORKING ON THI

## Objectives

* Learn what Integration Testing is
* Learn Selenium
* Learn how to add selenium as Docker container
* Learn how to to setup and configure selenium

# Integration Testing with Selenium

The meaning of integration is quite straightforward – Combine the unit tested module one by one and test the functionality of the combined unit. Integration Testing is another level of testing where individual **units**are combined and tested to verify if they are working as they intend to when integrated. No matter how efficiently each unit is running, if they aren’t properly integrated, it will affect the functionality of the software program. As a solution integration testing is implemented.

**How is Integration Testing Done?**

Normally, integration testing is carried out after unit testing. Once all the individual units are created and tested, we start combining those tested modules and start performing the integrated testing. The main goal here is to test the interfaces between the units/modules.

**Selenium**

The tool we are going to use, on top of Mocha is Selenium. There is a neat tool called Selenium IDE. It can auto generate the test code or test instructions for you but we want to learn individual Selenium commands so we will use a tool called **selenium webdriver.**With this tool, we write the test instructions to perform integration testing.

**Selenium WebDriver**

Selenium WebDriver is a web framework that permits you to execute cross-browser tests. This tool is used for automating web-based application testing to verify that it performs as expected.

To integrate and initialize Selenium to your express app, follow the steps below:

1. In any of your dockerized apps (preferrably the latest dockerized app that you worked on), right click on docker-compose.yml and choose **Compose Up**
2. Attach Shell to your **web\_app**service
3. Type in **cd ..**or cd in to /var/www/app (where your package.json is saved)
4. Run **npm install selenium-webdriver --save-dev**
5. Add a new service to **docker-compose.yml** file

selenium\_chrome:

image: selenium/standalone-chrome:4.0.0

restart: always

volumes:

- /dev/shm:/dev/shm

ports:

- "4444:4444"

  4. Add **selenium\_chrome**as the service that our app **depends\_on:**

web\_app:

depends\_on:

- db

- selenium\_chrome

  5. Require selenium in your test file:

const chai = require('chai');

const expect = chai.expect;

const {Builder, By, Capabilities} = require('selenium-webdriver');

const caps = new Capabilities();

caps.setPageLoadStrategy("normal");

let driver;

  3. In our **describe**(before any **it**statements), we add the code below to initialize selenium

describe('Login feature', function(){

this.timeout(30000);

it("some test here", async function(){

try {

driver = await new Builder().

withCapabilities(caps).

forBrowser('chrome').

usingServer('http://selenium\_chrome:4444/wd/hub').

build();

// Navigate to Url

await driver.get("http://web\_app:3000");

let message = await (await driver.findElement(By.id("title"))).getText();

console.log("The message is", message);

expect(message).to.equal('Hello Selenium');

} catch (e) {

console.log(e);

throw new Error("error");

} finally {

if(driver){

await driver.quit();

}

}

})

});

And that's it! For the commonly used Selenium test commands, proceed to the next tab.

Note: You may need to restart your container since you've made changes to your docker-compose.yml file. Just right click on docker-compose and choose **Compose Restart.**

You can download the sample dockerized selenium files [here](https://github.com/codingdojo88oliver/selenium-demo).

## Objectives

* Learn the basic Selenium testing methods
* Learn how to implement integration testing in combination with unit testing

# selenium-webdriver

Selenium WebDriver is a very powerful automation tool. Instead of you having to click certain buttons, type in certain values in specific input fields to manually test your app, Selenium automates this for you by running certain selenium commands. For example, say you want to test the /login page of your app. Instead of opening a browser, and type in the URL, just have this in your test file:

await driver.get("http://app:3000/login");

And to set the size of the browser, just have:

await driver.manage().window().setRect(1363, 1417);

And once this page is loaded (without selenium actually loading a chrome browser), you can look for a specific button and trigger a click event:

await driver.findElement(By.id("login\_btn")).click();

Can you imagine now how we are going to go about implementing this? This is an ideal scenario:

1. We set up an express app (Docker and TDD-ready)
2. We write a unit test that fails
3. We write the code to make it pass
4. We repeat steps 2 and 3 until we are ready to perform an integration test.
5. We write the integration test for a certain feature (Let's say Login feature)
6. This will probably fail
7. We refactor our code to make the integration test pass.
8. We make sure that the changes made don't affect other prior unit tests.
9. If both unit and integration tests pass, repeat step 2.

Below are the common methods you'll use in making your automated testing with Selenium to work:

await driver.manage().window().setRect();

// Initializes the browser. You can specify the height and width via the setRect() method.

// These are the common commands to run if you expect the click event to process something and selenium needs to wait.

await driver.findElement(By.id("some\_btn")).click();

await driver.sleep(1000);

await driver.navigate().refresh();

Other common **find** methods:

await driver.findElement(By.id("some\_id"))

await driver.findElements(By.css("ul#todos>li"))

await driver.findElement(By.css("ul#todos>li:last-child"))

To input something to a form field:

element = await driver.findElement(By.id("email"));

await element.click();

await element.sendKeys("jrosales@village88.com");

To get a text from an element:

let todo\_item = await driver.findElement(By.css("ul#todos>li:last-child")).getText();

And finally, just use **expect**method to make assertions for your test.

## Objectives

* Learn how to put all things together: Unit Testing, Integration testing.
* To come up with descriptive test cases
* To be exposed on a development setup where features were written first, before the tests.

# Login

Download the app with a working login feature here: [selenium-tdd-assignment](https://github.com/HHOliver88/selenium-login-assignment)

In this assignment, come up with the **unit tests** as well as the **integration tests** for the login feature.

Tips:

1. Perform unit testing first. Make sure to only test the Model
2. Then do integration test.

Make sure you run the app first and familiarise how the login works to come up with the necessary test cases.

# TDD Course Evaluation

Your feedback helps us to know how we can improve our training courses. Please give your honest feedback for the questions in this Evaluation form: <https://docs.google.com/forms/d/e/1FAIpQLSdUb-Tix4GK62lHofOnwcn9dZ1DtAc2CasM6sdmY4b7_QgOfQ/viewform>