

# SIT 725 Prac 6 Testing

# Contents

1. Understanding TDD
2. Mocha and chai at a glance
3. Understanding Basic Mocha Specs
4. Adding Mocha and chai
5. Routes
6. Testing
7. Conclusion
8. Questions

# Understanding TDD

One of the most important aspects of Agile development is Test Driven Development (TDD). TDD can improve code quality, speed up the development process, promote programmer confidence, and improve error identification.

Due to a lack of testing methodologies, supportive tools, and frameworks, it was difficult to automate web applications in the past, thus developers had to rely solely on manual testing, which was a painful and time-consuming procedure. TDD, on the other hand, can now be used to test entire standalone services and REST APIs.

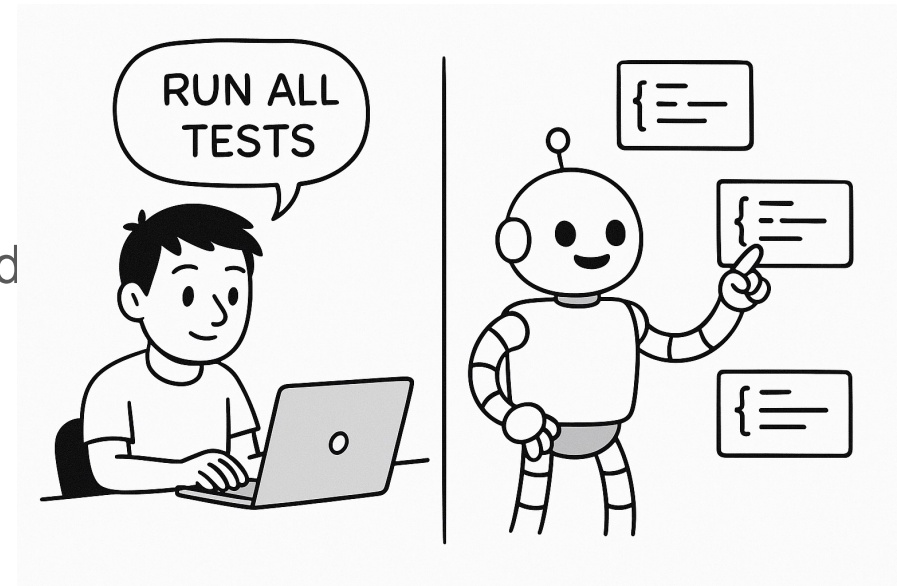
Testing is one of the primary processes in the Software Development Life Cycle (SDLC) that must be completed before the end product can be deployed and used. To attain quality and confidence, a variety of Software Testing Services, such as Unit Testing, Integration Testing, Smoke, Sanity, and Regression Testing, are used on the product.

# Automated Testing





- Automated testing is when test scripts check your code without manual effort.
- It helps ensure your code behaves as expected every time you make a change.
- Think of it like a safety net for your code.

# Manual vs Automated Testing

- Manual: Time-consuming, easy to forget edge cases, harder to scale.
- Automated: Fast, consistent, repeatable, and scalable.
- Automated testing is ideal for large and evolving codebases.



# Common Types of Automated Tests

-  Unit Tests: Test small, isolated functions (e.g., addTwoNumbers).
-  Integration Tests: Test how components work together.
-  API Tests: Check if REST endpoints return correct responses.
-  End-to-End (E2E): Simulate real user workflows.

# Mocha and Chai at a glance





[Mocha](#) is a popular JavaScript testing framework that runs on Node.js and in the browser. It simplifies asynchronous testing. It generates accurate test reports as well as stack traces for any uncaught exceptions.

[Chai](#), on the other hand, is an assertion library that can be used in conjunction with any JavaScript testing framework.



# Mocha & Chai: What Types of Testing?

Mocha (test framework) + Chai (assertion library) can be used for various types of testing:

-  Unit Testing: Test individual functions or methods.
-  Integration Testing: Test how multiple modules/components work together.
-  API Testing: Combine with Supertest or Chai HTTP to test REST endpoints.
-  End-to-End (E2E) Testing: Possible with Puppeteer or Selenium, but tools like Cypress or Playwright are better.



# Understanding Basic Mocha Specs

When dealing with mocha we have majorly three things to understand the most

- **assert:** The word 'assert' aids in determining the test status. It determines whether or not the test was successful.
- **describe:** 'describe' is a function that holds a collection of tests, or a test suite as we might call it. It has two parameters: the first is a descriptive name that describes the method's usefulness, and the second is the function, which basically comprises one or more tests. It's also possible to specify a nested 'describe'.
- **it:** It's a function that contains the actual test or test steps that must be run. It also has two parameters: the first is the test's meaningful name, and the second is the function, which contains the test's body or stages.

Now let's get into creating our own TDD application.

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

describe('Array', function () {
  describe('#indexOf()', function () {
    it('should return -1 when the value is not present', function () {
      assert.strictEqual([1, 2, 3].indexOf(4), -1);
    });
  });
});
```

# Understanding Basic Mocha Specs contd...

```
// Define a test suite named 'Array'
// This groups together related tests about array behavior
describe('Array', function () {

  // Define a nested suite for the `indexOf()` method of arrays
  describe('#indexOf()', function () {

    // Define an individual test case
    // Description explains what this test is checking for
    it('should return -1 when the value is not present', function () {

      // Use 'assert.strictEqual' to check if the result of [1, 2, 3].indexOf(4) is exactly -1
      // 'indexOf(4)' means: look for the value 4 in the array [1, 2, 3]
      // Since 4 is not found, indexOf should return -1, and the assertion will pass
      assert.strictEqual([1, 2, 3].indexOf(4), -1);
    });
  });
});
```

Should this  
test pass or  
not?



# Understanding Basic Mocha Specs contd...

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

describe('Array', function () {
  describe('#indexOf()', function () {
    it('should return -1 when the value is not present', function () {
      assert.strictEqual([1, 2, 3].indexOf(2), -1);
    });
  });
});
```

Should this  
test pass or  
not?



# A simple example using Mocha & Chai

Create an express app as you did in the last pracs. I am going to reuse code from the CalcApp (the calculator app from week 4)

The express app has

- A server.js file (backend)
- An index.html file in the public/ folder (frontend form to add two numbers)
- A route `/add?a=1&b=2` to calculate the sum

# Adding Mocha and chai

Then we create folder in our project where all the test files goes.

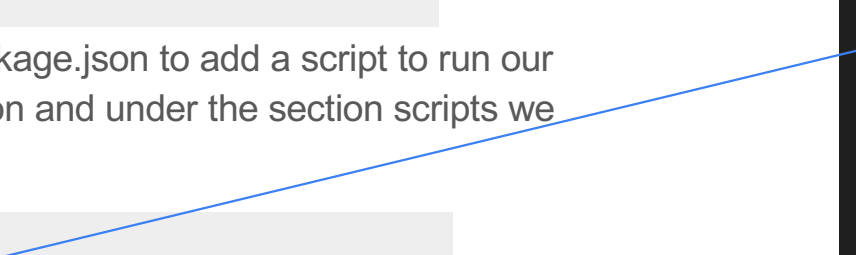
```
mkdir test
```

Next we need to install two packages `mocha` and `chai`

```
npm install --save-dev mocha chai request
```

After this we need to update our package.json to add a script to run our test files. We go into our package.json and under the section scripts we add this

```
"test": "mocha --reporter spec"
```



```
week6_1 > {} package.json > ...
1  {
2    "name": "week6_1",
3    "version": "1.0.0",
4    "main": "index.js",
5    "scripts": {
6      "test": "mocha --reporter spec"
7    },
8    "author": "",
9    "license": "ISC",
10   "description": "",
11   "dependencies": {
12     "express": "^5.1.0"
13   },
14   "devDependencies": {
15     "chai": "^5.2.0",
16     "mocha": "^11.1.0",
17     "request": "^2.88.2"
18   }
19 }
20
```

## Ensure Your server.js Handles /add

```
app.get('/add', (req, res) => {  
  const a = parseFloat(req.query.a);  
  const b = parseFloat(req.query.b);  
  
  if (isNaN(a) || isNaN(b)) {  
    return res.status(400).send("Invalid input");  
  }  
  
  const sum = a + b;  
  
  res.send(`The sum of ${a} and ${b} is: ${sum}`);  
});
```

# Write Mocha + Chai tests (test/calculator.test.js)

```
const expect = require("chai").expect;
const request = require("request");

describe("Sum Calculator API", function () {
  const baseUrl = "http://localhost:3000";

  it("returns status 200 to check if api works", function(done) {
    request(baseUrl, function(error, response, body) {
      expect(response.statusCode).to.equal(200);
      done();
    });
  });

  it("should return correct sum for valid numbers", function (done) {
    request.get(`${baseUrl}/add?a=10&b=5`, function (error, response, body) {
      expect(response.statusCode).to.equal(200);
      expect(body).to.include("15"); // Response contains the sum in plain text or HTML
      done();
    });
  });

  it("should handle missing parameters", function (done) {
    request.get(`${baseUrl}/add?a=10`, function (error, response, body) {
      expect(response.statusCode).to.not.equal(200); // Expect error
      done();
    });
  });

  it("should return error for non-numeric input", function (done) {
    request.get(`${baseUrl}/add?a=hello&b=world`, function (error, response, body) {
      expect(response.statusCode).to.not.equal(200);
      done();
    });
  });
});
```

# Testing Cont...

Once we are done with that we are all set up to run our test cases. For doing that first we need to start our node server

```
npm start
```

Then we need to open another terminal and go to our project folder once we are there we simply run our tests by running the command

```
npm test
```



# Testing Cont...

After running the command your test file should run and you should get an output in your terminal which should look something like this.

```
[niroshinie.fernando@DCCP20R2XG week6_1 % npm test

> week6_1@1.0.0 test
> mocha --reporter spec

Sum Calculator API
  ✓ returns status 200 to check if api works
  ✓ should return correct sum for valid numbers
  ✓ should handle missing parameters
  ✓ should return error for non-numeric input

4 passing (18ms)

niroshinie.fernando@DCCP20R2XG week6_1 %
```

Code: [https://github.com/niroshini/sit725/tree/main/week6\\_1](https://github.com/niroshini/sit725/tree/main/week6_1)

# Conclusion

So that is just a basic example of how you can create your own test scripts for unit testing your REST Api's

**Thanks**

