SIT 725 Prac 7
Testing

Contents

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.

Test-Driven Development (TDD) is a software development approach where you write tests before you write the actual code.

Mocha and Chai at a glance

Mocha and Chai are popular JavaScript testing frameworks often used together for Test-Driven Development (TDD) and Behavior-Driven Development (BDD).

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.

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.

Adding Mocha and chai

So we are going to continue with the same project as for last prac. We create folder in our project where all the test files goes.

```
mkdir test
```

Next we need to install two packages mocha and

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

```
PS C:\Users\new\OneDrive - Deakin University\Desktop - W\SIT725\practive-6\sit725-2024-t1-prac6> npm install --save mocha chai request
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
or details.
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142

added 198 packages, and audited 199 packages in 6s

30 packages are looking for funding
  run `npm fund` for details

5 moderate severity vulnerabilities

To address issues that do not require attention, run:
  npm audit fix

Some issues need review, and may require choosing
a different dependency.

Run `npm audit` for details.
```

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”
```

```
"scripts": {
  "start": "node server.js",
  "test": "mocha --reporter spec"
}
```

Routes

Next we update our server.js to add an api for us to use for testing.

Here is the example of AddTwoNumber

```
app.get('/addTwoNumbers/:firstNumber/:secondNumber', function(req,res,next){
  var firstNumber = parseInt(req.params.firstNumber)
  var secondNumber = parseInt(req.params.secondNumber)
  var result = firstNumber + secondNumber || null
  if(result == null) {
    res.json({result: result, statusCode: 400}).status(400)
  }
  else { res.json({result: result, statusCode: 200}).status(200) }
})
```

This route accepts two parameters, **firstNumber** and **secondNumber**, and returns the sum of the two numbers. If the parameters are not provided or cannot be parsed as integers, a status code of 400 (Bad Request) is returned along with a JSON object containing the error message. Otherwise, the sum is calculated and returned along with a status code of 200 (OK).

Routes Cont...

Once you added this your server.js should look like this

This code demonstrates a typical server.js file for a Node.js application using Express. It includes setting up Express, defining routes, handling HTTP requests, and implementing WebSocket functionality with Socket.IO.

```
let express = require("express");
let app = express();
let dbConnect = require("./dbConnect");
//dbConnect.dbConnect()
//var app = require('express')();
let http = require('http').createServer(app);
let io = require('socket.io')(http);
//const MongoClient = require('mongodb').MongoClient;
// routes
let projectsRoute = require('./routes/projects')
var port = process.env.PORT || 8080;
app.use(express.json());
app.use(express.static(__dirname + '/public'));
app.use('/api/projects',projectsRoute)
app.get("/test", function (request, response) {
  var user_name = request.query.user_name;
  response.end("Hello " + user_name + "!");
});
app.get('/addTwoNumbers/:firstNumber/:secondNumber',  function(req,res,next){
  var firstNumber = parseInt(req.params.firstNumber)
  var secondNumber = parseInt(req.params.secondNumber)
  var result = firstNumber + secondNumber || null
  if(result == null) {
    res.json({result: result, statusCode: 400}).status(400)
  }
  else { res.json({result: result, statusCode: 200}).status(200) }
})
//socket test
io.on('connection', (socket) => {
  console.log('a user connected');
  socket.on('disconnect', () => {
    console.log('user disconnected');
  });
  setInterval(()=>{
    socket.emit('number', parseInt(Math.random()*10));
  }, 1000);
});
http.listen(port,()=>{
  console.log("Listening on port ", port);
});
```

Testing

Next we come down to creating actual test cases for our project. So we go into the test folder that we created for our project and we add a file test.js to it. Once done that we add a few test cases for our api that we created.

Execute the test cases by running **npm test** in the terminal. This command will trigger Mocha to run the test file (**test.js**) and report the results

```
> sit725-prac6@1.0.0 test
> mocha --reporter spec


  Add Two Numbers
    ✓ returns status 200 to check if api works (54ms)
    ✓ returns statusCode key in body to check if api give right result should be 200
    ✓ returns the result as number
    ✓ returns the result equal to 8
    ✓ returns the result not equal to 15

  Add Two strings
    ✓ should return status 200
    ✓ returns statusCode key in body to check if api gives right result should be 400
    ✓ returns the result as null


  8 passing (107ms)
```

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

describe("Add Two Numbers", function() {
    var url = "http://localhost:8080/addTwoNumbers/3/5";
    it("returns status 200 to check if api works", function(done) {
        request(url, function(error, response, body) {
            expect(response.statusCode).to.equal(200);
            done()
          });
    });
    it("returns statusCode key in body to check if api give right result should be 200", function(done) {
        request(url, function(error, response, body) {
            body = JSON.parse(body)
            expect(body.statusCode).to.equal(200);
            done()
          });
    });
    it("returns the result as number", function(done) {
        request(url, function(error, response, body) {
            body = JSON.parse(body)
            expect(body.result).to.be.a('number');
            done()
          });
    });
    it("returns the result equal to 8", function(done) {
      request(url, function(error, response, body) {
          body = JSON.parse(body)
          expect(body.result).to.equal(8);
          done()
        });
  });
  it("returns the result not equal to 15", function(done) {
    request(url, function(error, response, body) {
        body = JSON.parse(body)
        expect(body.result).to.not.equal(15);
        done()
      });
});
});
```

Testing Cont...

You can also use this link to get all the code for this file.

These additional test cases focus on testing the behavior of the API when non-numeric strings are provided as input for addition. It ensures that the API handles invalid input appropriately and returns the expected status codes and results.

```
describe("Add Two strings", function() {
    var url = "http://localhost:8080/addTwoNumbers/a/b";
    it("should not returns status 200", function(done) {
        request(url, function(error, response, body) {
            expect(response.statusCode).to.equal(200);
            done()
        });
    });
    it("returns statusCode key in body to check if api gives right result should be 400", function(done) {
        request(url, function(error, response, body) {
            body = JSON.parse(body)
            expect(body.statusCode).to.equal(400);
            done()
        });
    });
    it("returns the result as null", function(done) {
        request(url, function(error, response, body) {
            body = JSON.parse(body)
            expect(body.result).to.be.a('null');
            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

```
Node server.js
```

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.

```
navit@DESKTOP-CI4LQR8:/mnt/d/node_projects/sit-725-2021-t2/deakin-crowd-testing$ npm run test

> deakin-crowds@0.0.0 test /mnt/d/node_projects/sit-725-2021-t2/deakin-crowd-testing
> mocha --reporter spec



  Add Two Numbers
    ✓ returns status 200 to check if api works
    ✓ returns statusCode key in body to check if api give right result should be 200
    ✓ returns the result as number
    ✓ returns the result equal to 8
    ✓ returns the result not equal to 15

  Add Two strings
    ✓ should not returns status 200
    ✓ returns statusCode key in body to check if api gives right result should be 400
    ✓ returns the result as null


  8 passing (49ms)

navit@DESKTOP-CI4LQR8:/mnt/d/node_projects/sit-725-2021-t2/deakin-crowd-testing$
```

Conclusion

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

The example provided demonstrates the process of creating test scripts for unit testing REST APIs. By following these steps, you can ensure the reliability and correctness of your API endpoints:

1.**Write Test Cases**:
Create test cases to cover various scenarios and functionalities of your API endpoints.
2.**Execute Tests**:
Run the test scripts using testing frameworks like Mocha and assertion libraries like Chai to validate the behavior of your API.
3.**Analyze Results**:
Analyze the test results to identify any failures or unexpected behaviors in your API.
4.**Iterate and Improve**:
Make necessary adjustments to your API implementation based on test results to improve functionality and reliability.

By incorporating unit testing into your development workflow, you can confidently deploy robust and bug-free REST APIs.

Ontrack task 6.1

Intro

This week's focus is on testing software, exploring the reasons behind testing and the methods involved. Testing plays a crucial role in software development as it helps as essential for identifying and resolving software defects, validating functionality, and ensuring the overall quality and reliability of the software. It helps build confidence in the software's performance and aids in delivering robust and reliable applications to end-users. identify defects, verify functionality, and ensure the overall quality of the software.

Slides

The slides and lecture recording are available on Microsoft Teams channel and the unit site as well.

Instruction

Once you have completed viewing the lecture, write down a 200 words summary of what you believe are the key points regarding the testing your web application for the group project (considering unit testing, integration and end-to end testing).

Submissions details and Delivery

After completing the task, please proceed to upload the converted PDF file on OnTrack.

Ontrack task 6.2

Instructions

Testing plays a crucial role in ensuring the reliability and functionality of software. It is a vital component that can demonstrate the HD (High Distinction) worthiness of your group project. Pay close attention as we delve into the world of testing and its significance in software development.

Part 1: Take advantage of the provided slides and actively participate in the practical session to gain insights into basic testing strategies. This will equip you with essential knowledge and techniques to design and implement effective tests for your software.

Part 2: Continuing from previous weeks, import your weekly project into a new repository. It's time to make the necessary modifications and incorporate testing into your project. Implementing tests will help verify the correctness of your code, identify potential issues or bugs, and enhance the overall quality of your software.

By applying testing practices to your project, you can proactively address potential problems, maintain code integrity, and ensure the robustness of your application. Embracing testing as an integral part of your software development process is a key step towards delivering a high-quality, reliable, and successful project.

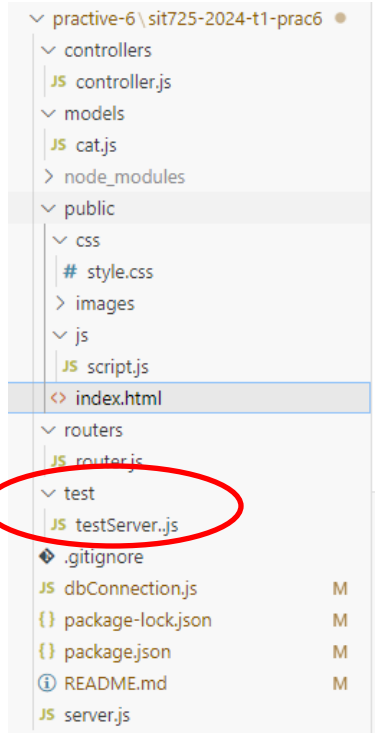
Submission details and Delivery

- Once you are done, push your code into your repo, giving the repository the following name. sit725-2023-t1-prac6,
- Share your repo link and provide screenshots of the running code as evidence, convert it to .pdf and upload it to the OnTrack.

What to do :

1. Import your prac 5 code into prac 6
2. Modify your document and make it look like prac6
3. Create test folder
4. Write the code for test.js
5. Test the code

Example of Ontrack task 6.2




```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS C:\Users\new\OneDrive - Deakin University\Desktop - W\SIT725\practive-6\sit725-2024-t1-prac6> npm test

> sit725-2023-t1-prac3@1.0.0 test
> mocha --reporter spec

(node:8360) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instead.
(Use `node --trace-deprecation ...` to show where the warning was created)


  test GET api
    ✓ returns statusCode of 200

  test POST api
    ✓ post cat to DB

  test DELETE api
    ✓ delete a cat


  3 passing (36ms)

PS C:\Users\new\OneDrive - Deakin University\Desktop - W\SIT725\practive-6\sit725-2024-t1-prac6>
```

Steps:

1. mkdir test
2. npm install --save mocha chai request
3. Update package.json-mocha --reporter spec
4. Write the code for test.js
5. Node server.js
6. Npm test

Thanks