Software Architecture Document

One Click Buy E-commerce Application

Capstone project for:

SOEN 6411: Advanced Programming Practices

Submitted By:

Preeti Chouhan (40198661) **Anant Shukla (40231430)**

Master of Applied Computer Science (Fall 2022)

Under the guidance of

Dr. Constantinos Constantinides, P.Eng.

Associate Professor

Document Version: 1.0

Date: November 13th, 2022

Table of Contents

1 OVERVIEW

The project aims to build an e-commerce application which allows users to make online purchases at the ease of their comfort. Besides providing the main functionality the project aims to demonstrate the accepted development procedures and structures for project development.

The source code of the project is available on GitHub at: https://github.com/anantshukla/One-Click-E-Commerce-MarketplaceApp . Please mail anant.shukla@mail.concordia.ca if there are issues accessing the repository.

1.1 Project Description

E-commerce is fast gaining its popularity as the accepted way of most of the business paradigms. The objective of the project is to develop a general-purpose e-commerce website where a user can buy products from multiple categories at the comfort of their home via internet. However, for the purpose of simplicity we mainly aim to achieve the same task with few categories.

An online store is a virtual store where a user can buy products online by going through the available catalog. The user will have to first create his account via signup page. And then he will have to login with username and password selected while creating the account in order to access the catalog of the available products.

Therefore, the major functionalities of the project are:

- Any user can register/ create a new account.
- Registered users can only login with username and password given while creating their accounts.
- Only registered users can access the available product catalog.
- Only registered users can access the details of a particular product.

The major components of the projects are:

- Demonstrating usage of proper design patterns and object-relational structures.
- Implement refactoring strategies on the code.
- Performing unit testing.
- Populating the database via an API.
- Parameterized Queries.

1.2 Technology Stack Used

Tech Stack	**Description**
NodeJS - ExpressJS	Backend – REST API
REACT JS	Frontend
SQLite	Database
Jest, Supertest and Istanbul for Test Reporting	Testing Framework/Reporting

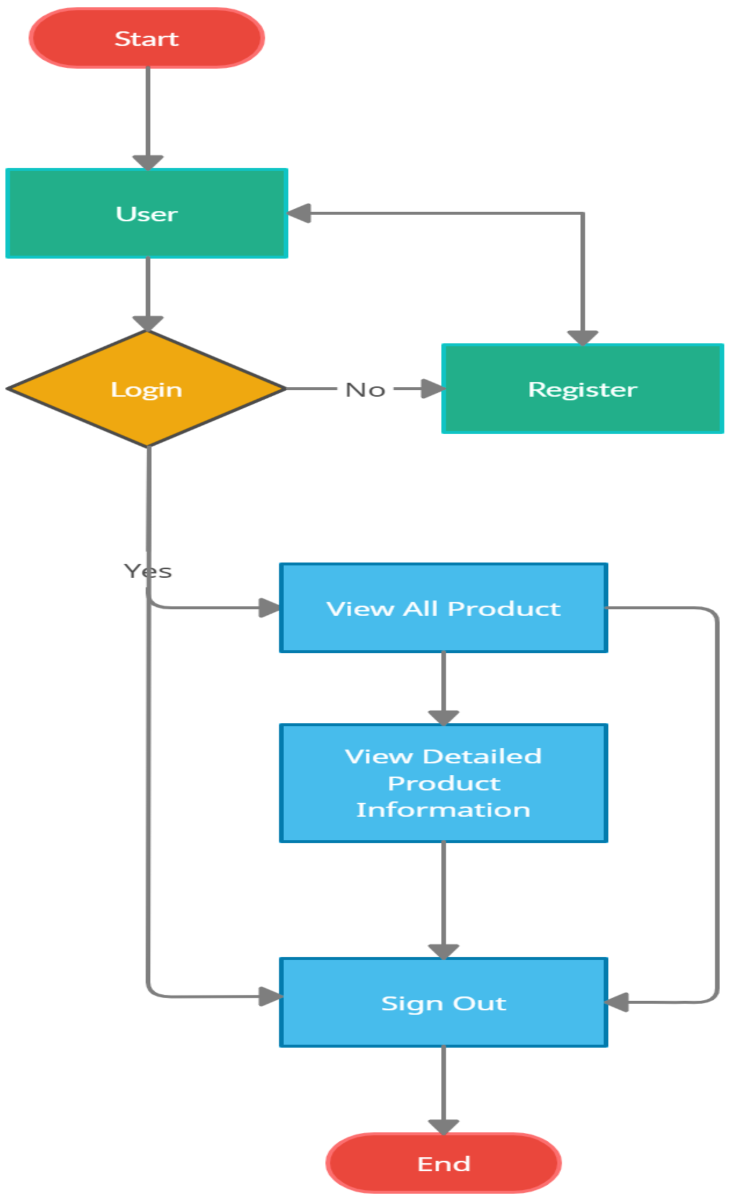
1.3 Development Tools

For development of the application, we used the following development tools:

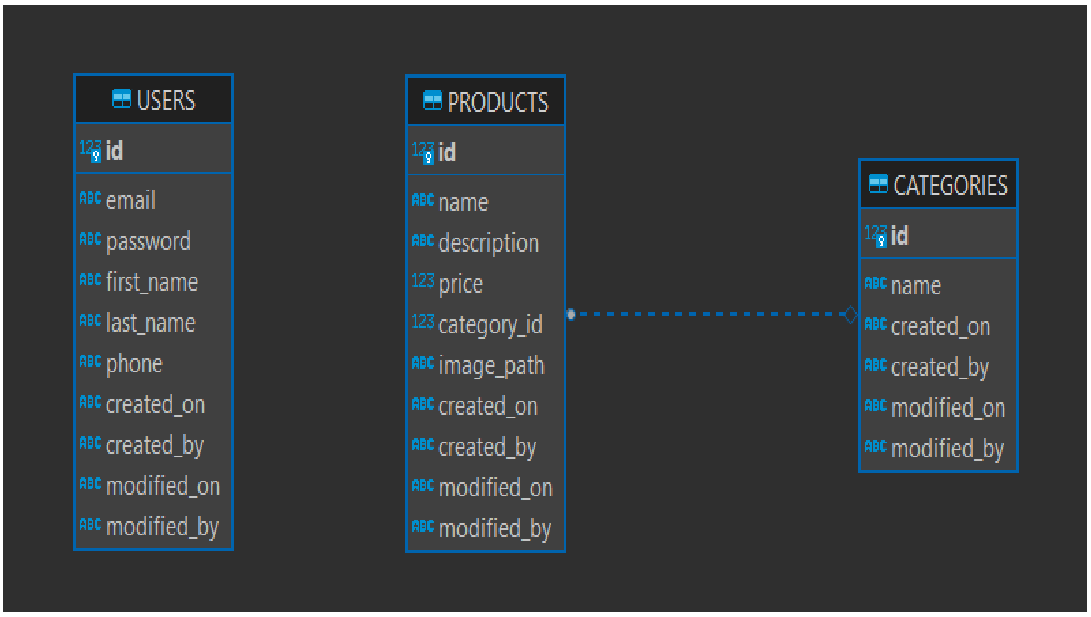
- Visual Studio Code: IDE for code development.
- DBeaver: For running SQL Scripts
- Git for Windows: Source Code Management
- GitHub: Remote Source Control Management
- Postman: Testing tool for API's

1.4 Data Model

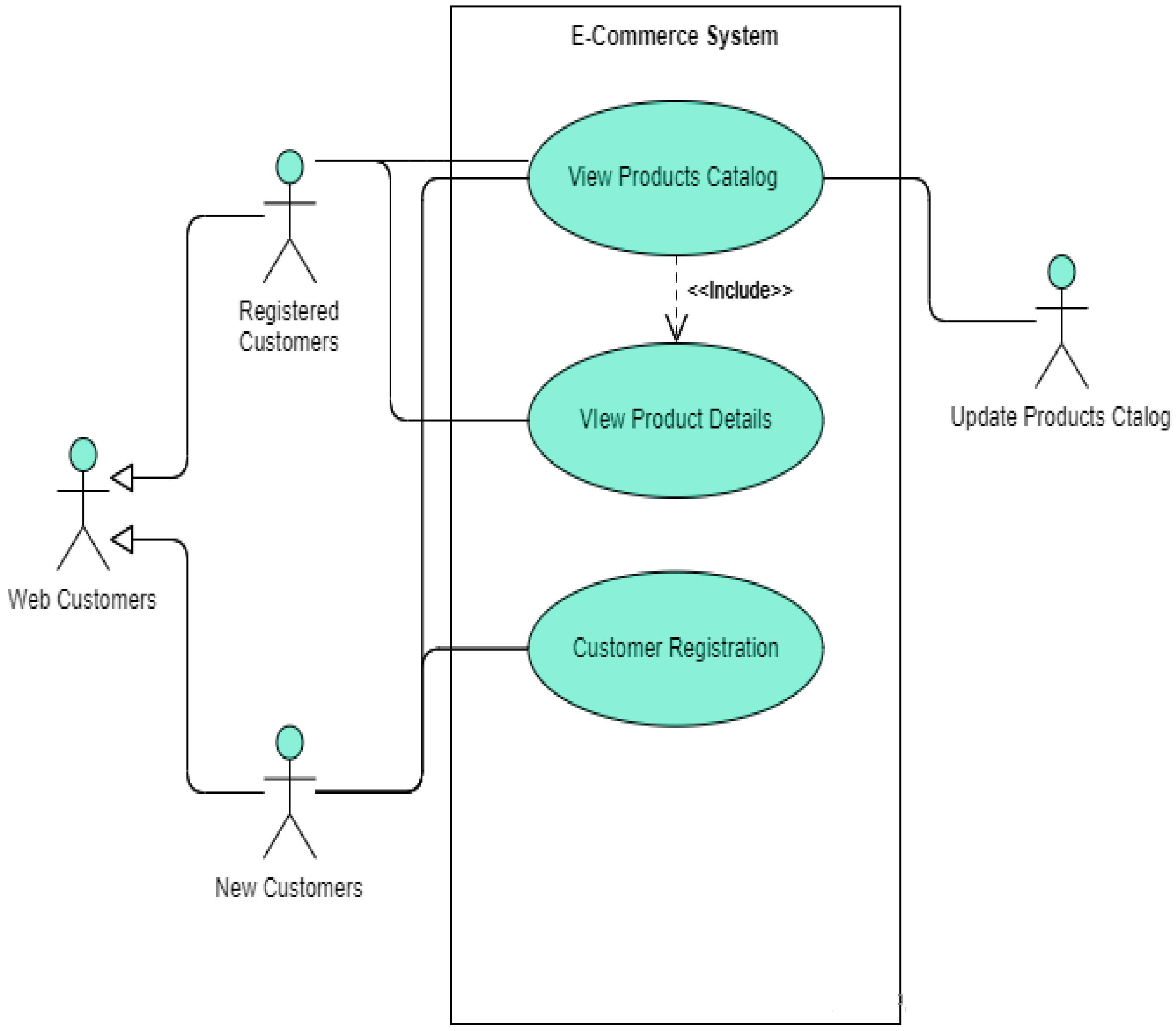
1.4.1 FLOW CHART



1.4.2 Entity Relationship Diagram



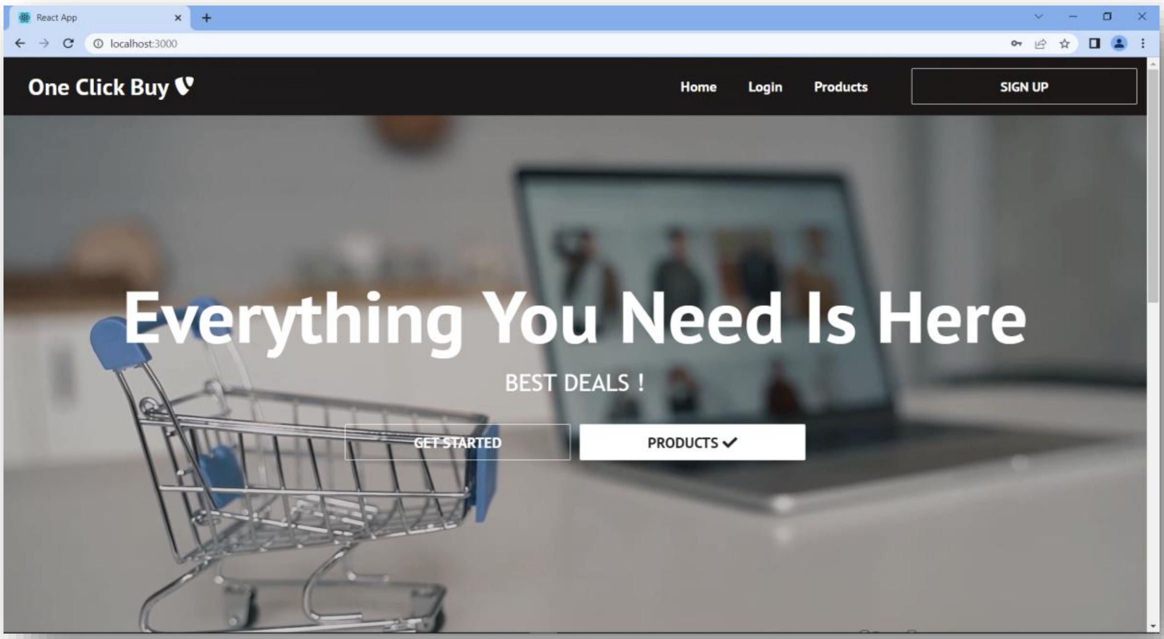
1.4.3 Use Case Model

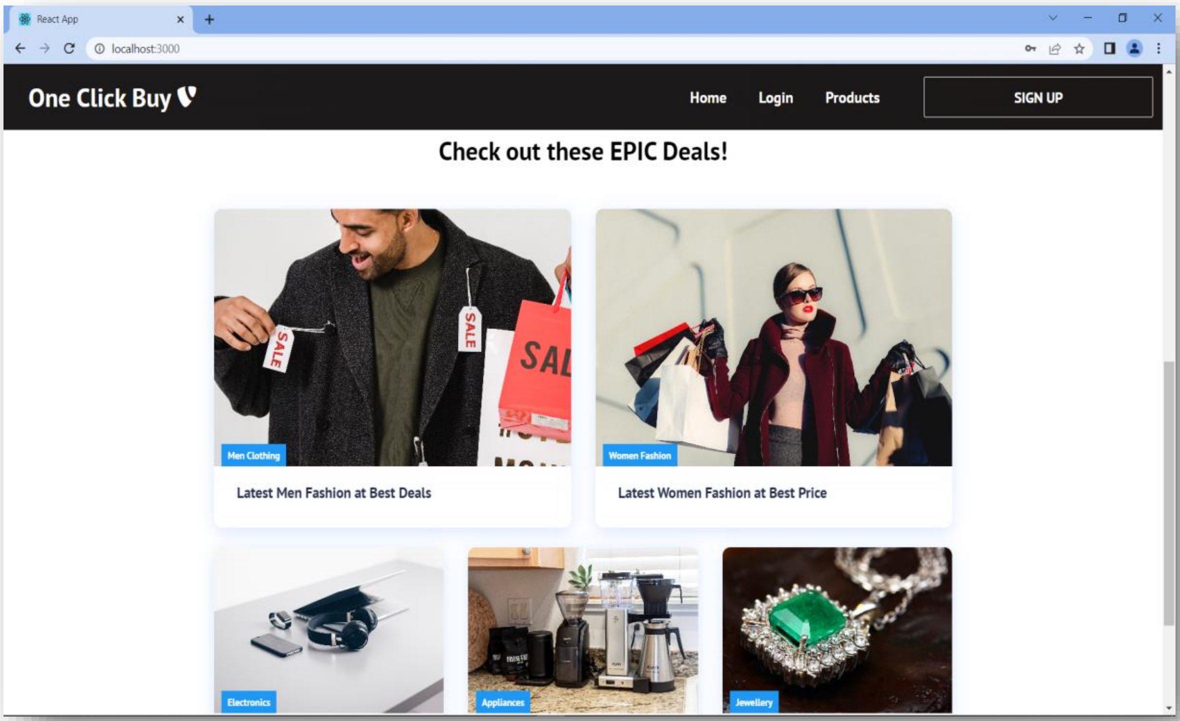


2 APPLICATION SNAPSHOTS

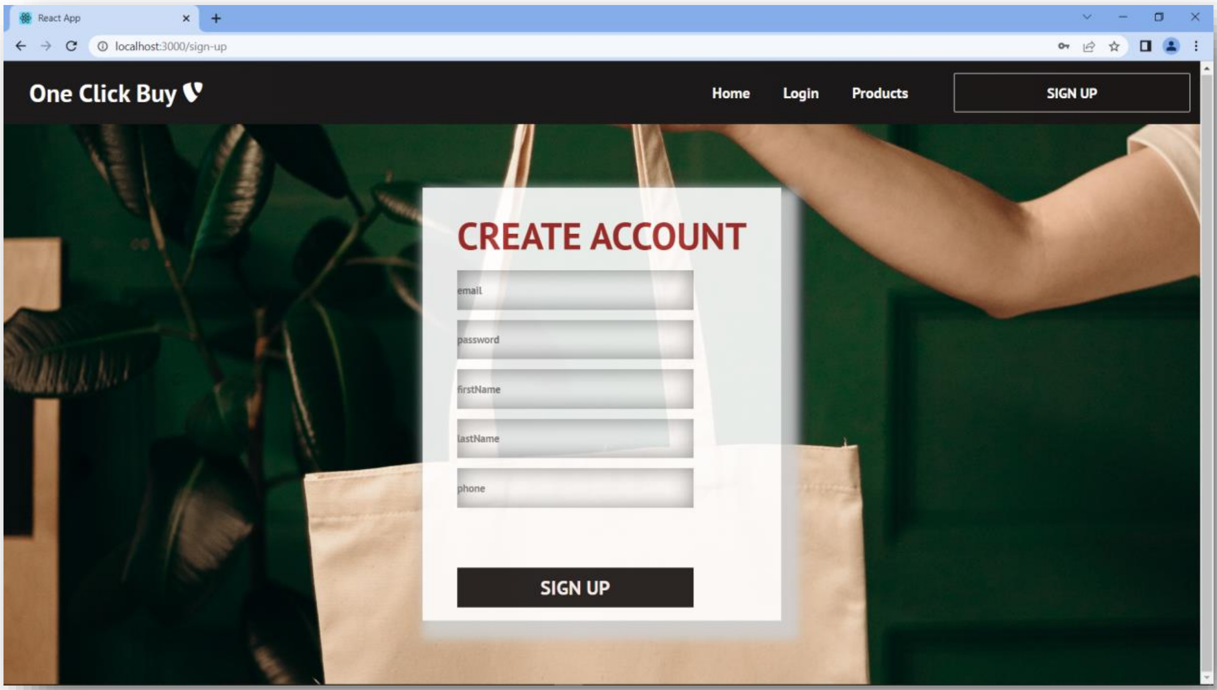
This section contains screenshots of the application with which the user interacts. The UI is build using React.js mainly.

- **HOME PAGE**

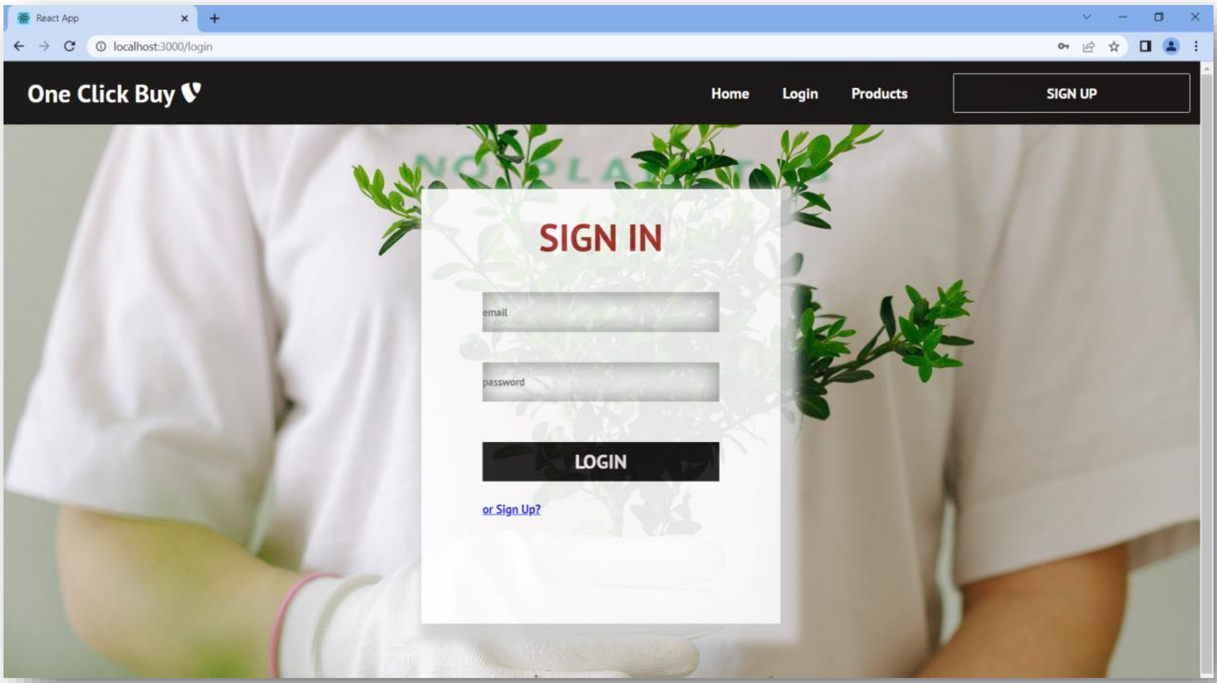




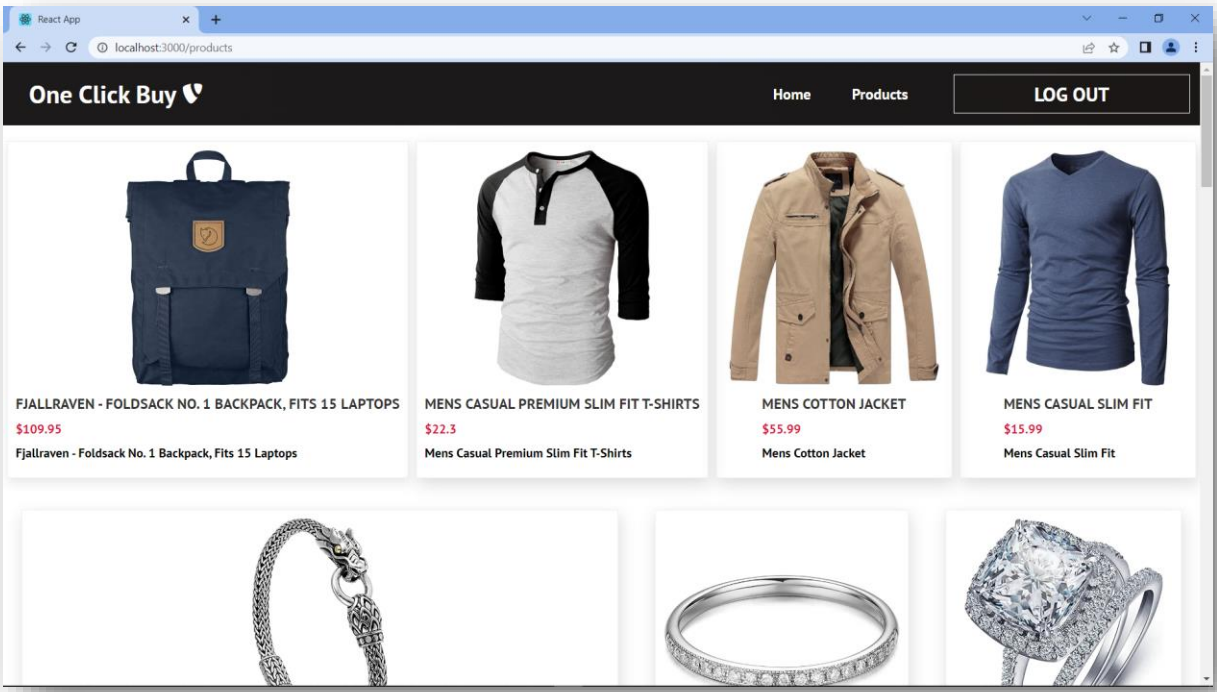
- **SIGN UP PAGE**



- **SIGN IN PAGE**



- **PRODUCT CATALOG PAGE**



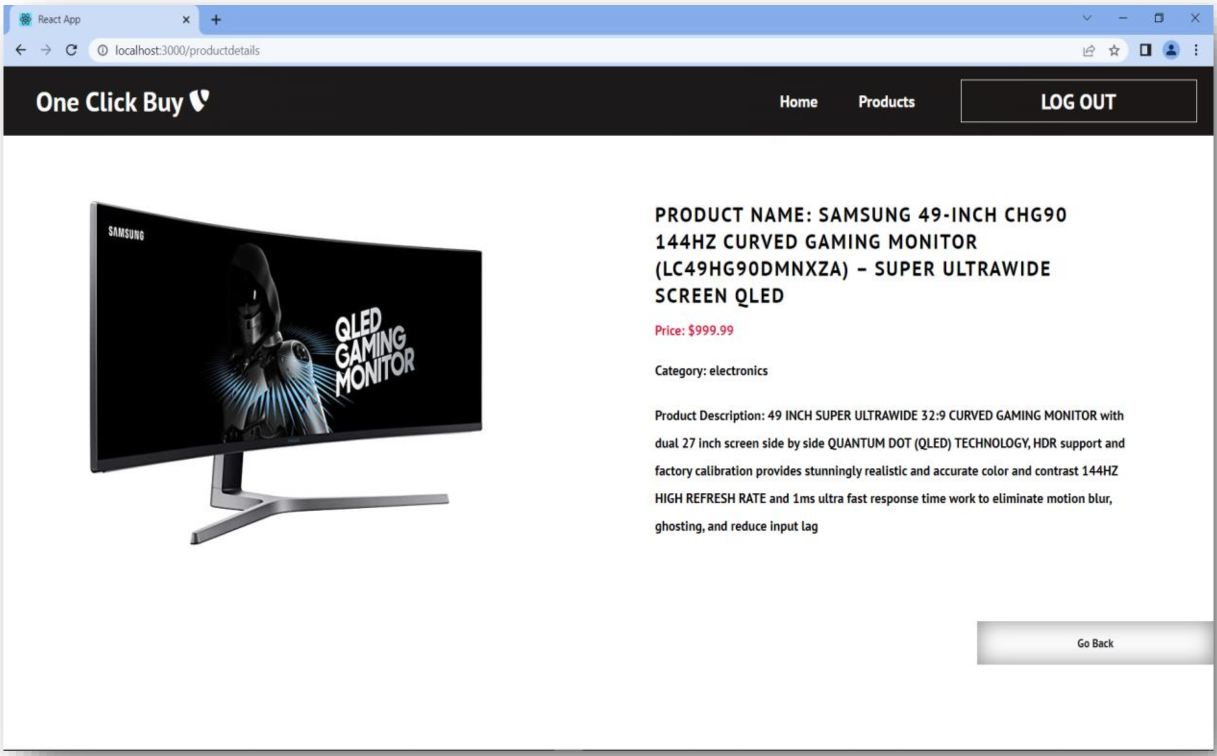
FJALLRAVEN - FOLDSACK NO. 1 BACKPACK, FITS 15 LAPTOPS
$109.95
Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops

MENS CASUAL PREMIUM SLIM FIT T-SHIRTS
$22.3
Mens Casual Premium Slim Fit T-Shirts

MENS COTTON JACKET
$55.99
Mens Cotton Jacket

MENS CASUAL SLIM FIT
$15.99
Mens Casual Slim Fit

- **PRODUCT DETAILS PAGE**



PRODUCT NAME: SAMSUNG 49-INCH CHG90 144HZ CURVED GAMING MONITOR (LC49HG90DMNXZA) – SUPER ULTRAWIDE SCREEN QLED

Price: $999.99

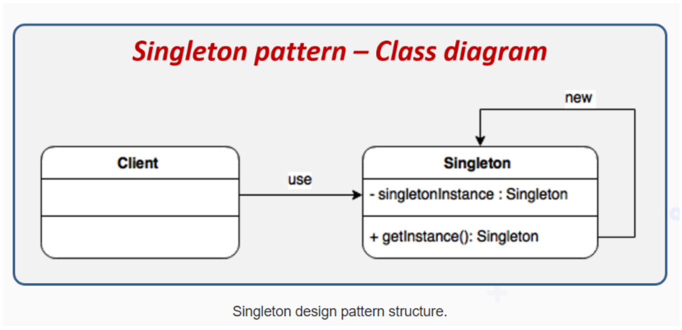
Category: electronics

Product Description: 49 INCH SUPER ULTRAWIDE 32:9 CURVED GAMING MONITOR with dual 27 inch screen side by side QUANTUM DOT (QLED) TECHNOLOGY, HDR support and factory calibration provides stunningly realistic and accurate color and contrast 144HZ HIGH REFRESH RATE and 1ms ultra fast response time work to eliminate motion blur, ghosting, and reduce input lag

3 DESIGN PATTERN

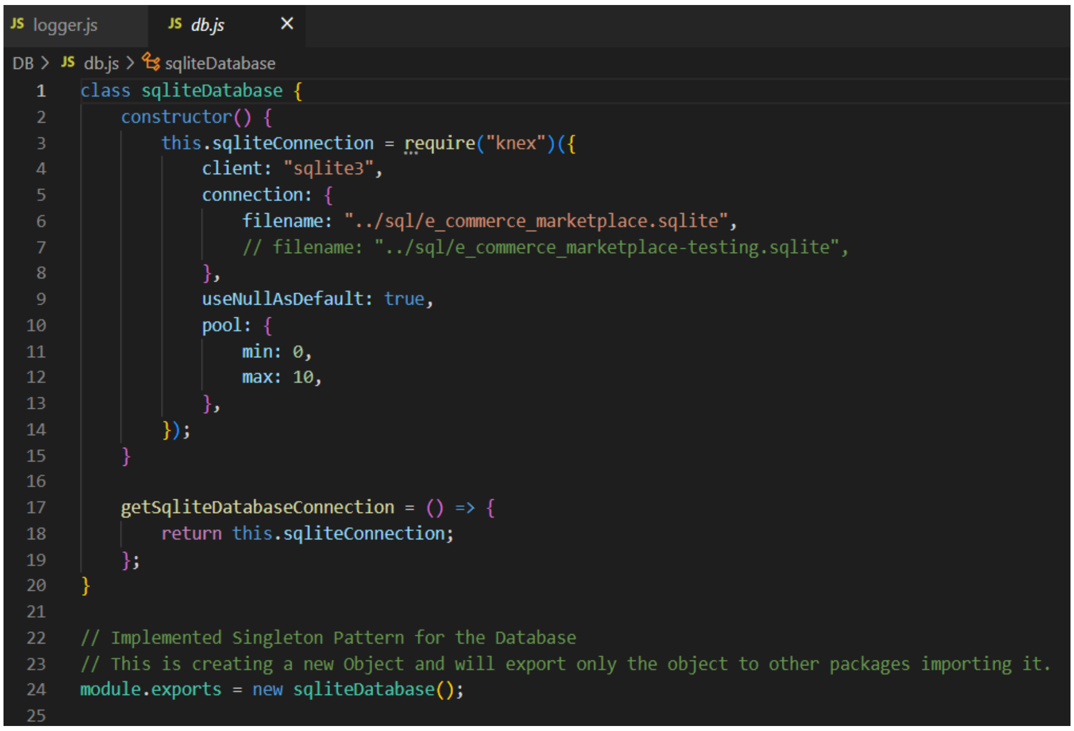
3.1 Singleton Design Pattern

Singleton design pattern is used for creating only one instance of an object no matter how many times the object is instantiated while providing a global access point to this instance.


Singleton pattern – Class diagram
new
Client
use
Singleton
- singletonInstance : Singleton
+ getInstance(): Singleton

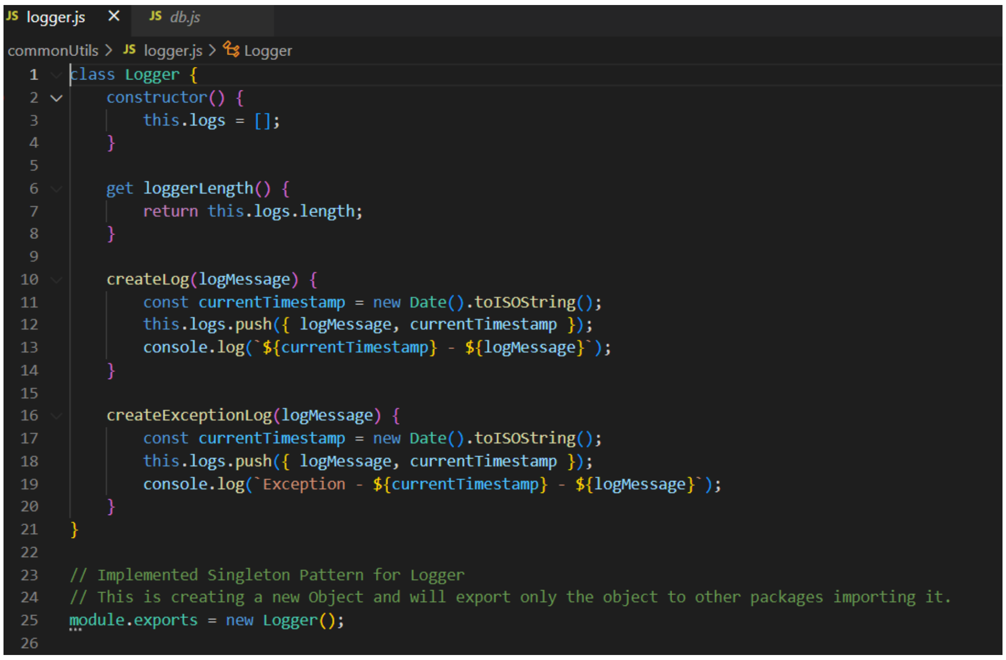
Singleton design pattern structure.

Here we have db.js file where we add a constructor to our singleton class to ensure single instantiation. We record the creation we have logger.js.



```
class sqliteDatabase {
    constructor() {
        this.sqliteConnection = require("knex")({
            client: "sqlite3",
            connection: {
                filename: "../sql/e_commerce_marketplace.sqlite",
                // filename: "../sql/e_commerce_marketplace-testing.sqlite",
            },
            useNullAsDefault: true,
            pool: {
                min: 0,
                max: 10,
            },
        });
    }

    getSqliteDatabaseConnection = () => {
        return this.sqliteConnection;
    };
}

// Implemented Singleton Pattern for the Database
// This is creating a new Object and will export only the object to other packages importing it.
module.exports = new sqliteDatabase();
```

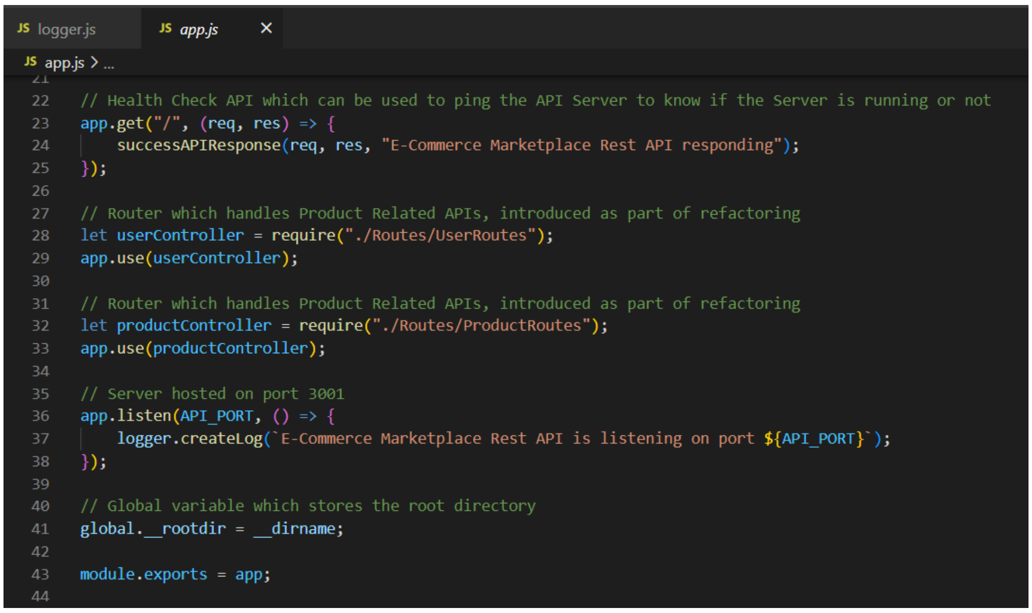
This logger saves information about all the logs that are sent to it and it also logs each message with a timestamp. So, once we create an instance of this object we can use the log method, send it a message, and it will log the timestamp and the message to the terminal as well as save information about that log.



```javascript
class Logger {
    constructor() {
        this.logs = [];
    }

    get loggerLength() {
        return this.logs.length;
    }

    createLog(logMessage) {
        const currentTimestamp = new Date().toISOString();
        this.logs.push({ logMessage, currentTimestamp });
        console.log(`${currentTimestamp} - ${logMessage}`);
    }

    createExceptionLog(logMessage) {
        const currentTimestamp = new Date().toISOString();
        this.logs.push({ logMessage, currentTimestamp });
        console.log(`Exception - ${currentTimestamp} - ${logMessage}`);
    }
}

// Implemented Singleton Pattern for Logger
// This is creating a new Object and will export only the object to other packages importing it.
module.exports = new Logger();
```

3.2 Router Design Pattern

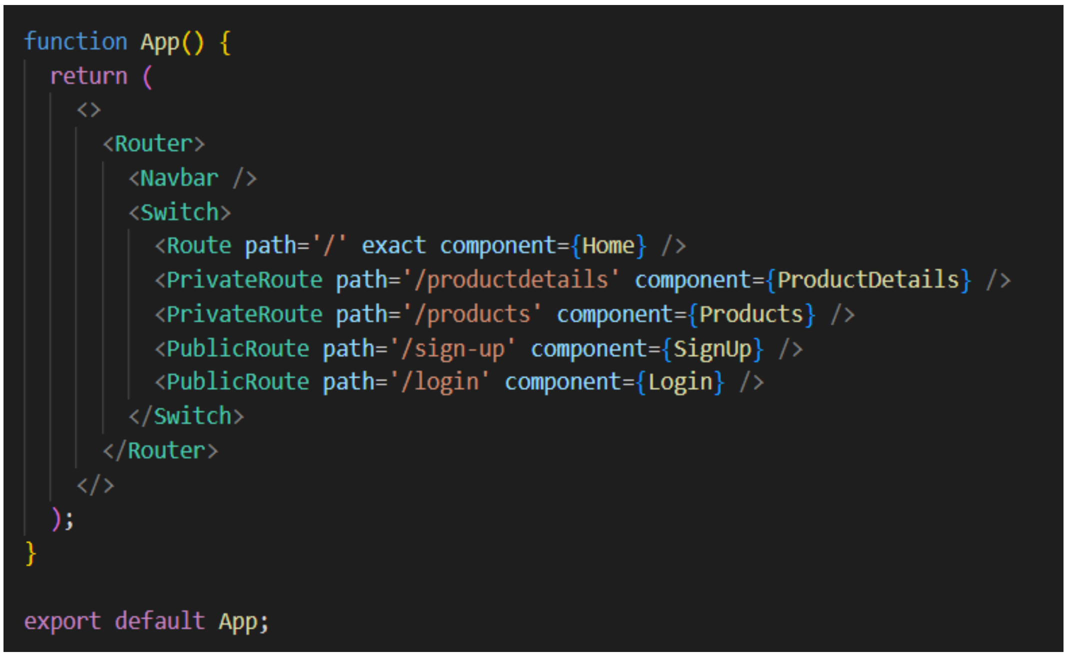
No matter how many routes a backend API supports, every single request needs to be processed by a single request handler function and this is a router design design pattern. We have kept all the routes together in app.js. We have achieved the same pattern in front end side with React where we are handling multiple routes via single file.



```javascript
// Health Check API which can be used to ping the API Server to know if the Server is running or not
app.get("/", (req, res) => {
    successAPIResponse(req, res, "E-Commerce Marketplace Rest API responding");
});

// Router which handles Product Related APIs, introduced as part of refactoring
let userController = require("./Routes/UserRoutes");
app.use(userController);

// Router which handles Product Related APIs, introduced as part of refactoring
let productController = require("./Routes/ProductRoutes");
app.use(productController);

// Server hosted on port 3001
app.listen(API_PORT, () => {
    logger.createLog(`E-Commerce Marketplace Rest API is listening on port ${API_PORT}`);
});

// Global variable which stores the root directory
global.__rootdir = __dirname;

module.exports = app;
```

3.3 Protected Routes

Protected routes pattern allows us to make sure that a user can access certain functionalities only when he logs into the application. Also, it allows us to make sure that user once logged in cannot login again to have another session. We have maintained this with user Session which gets removed when user logs off.



```
function App() {
  return (
    <>
      <Router>
        <Navbar />
        <Switch>
          <Route path='/' exact component={Home} />
          <PrivateRoute path='/productdetails' component={ProductDetails} />
          <PrivateRoute path='/products' component={Products} />
          <PublicRoute path='/sign-up' component={SignUp} />
          <PublicRoute path='/login' component={Login} />
        </Switch>
      </Router>
    </>
  );
}

export default App;
```

4 Refactoring

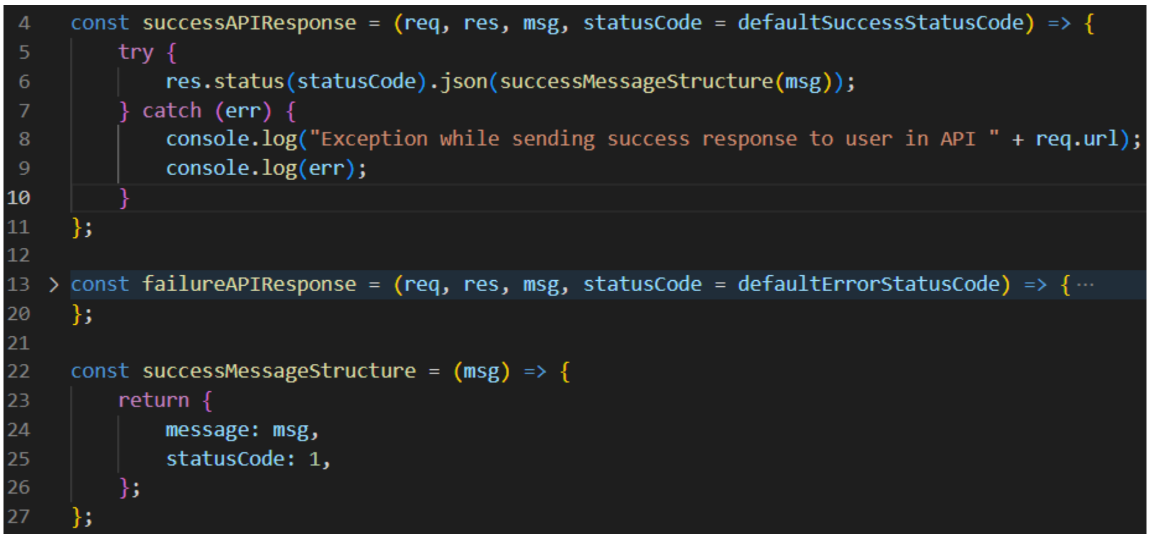
The goal of refactoring is to make sure that the code developed is maintainable and that multiple developers working on the project can simultaneously work on the code without creating conflicts. By refactoring, we can improve the structure, design and implementation of software. Refactoring helps improves code readability too.

4.1 Extract Method

With the help of this refactoring technique can can move code that is reused across multiple functions to a separate new method which passes variables as parameters.

This has the following advantages:

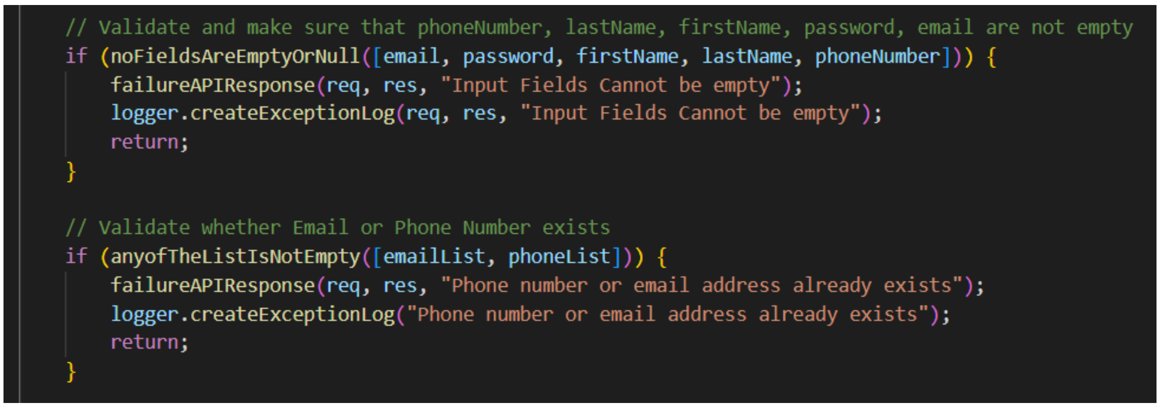
- Reuse code across multiple methods
- Help reduce function length
- Helps maintain common output patterns. This can be seen in the screenshot below, where we have created common interfaces for successAPIResponse and failureAPIResponse which is used by all REST APIs to send a response. This way we always maintain a common structure, as seen in successMessageStructure.



```
const successAPIResponse = (req, res, msg, statusCode = defaultSuccessStatusCode) => {
    try {
        res.status(statusCode).json(successMessageStructure(msg));
    } catch (err) {
        console.log("Exception while sending success response to user in API " + req.url);
        console.log(err);
    }
};

const failureAPIResponse = (req, res, msg, statusCode = defaultErrorStatusCode) => {···
};

const successMessageStructure = (msg) => {
    return {
        message: msg,
        statusCode: 1,
    };
};
```

4.2 Replacing Nested Conditions with Guard Clauses

In order to avoid a huge nest of if-else conditions, we have replaced these with Guard Clauses. Doing so helps make the code significantly easier to read.

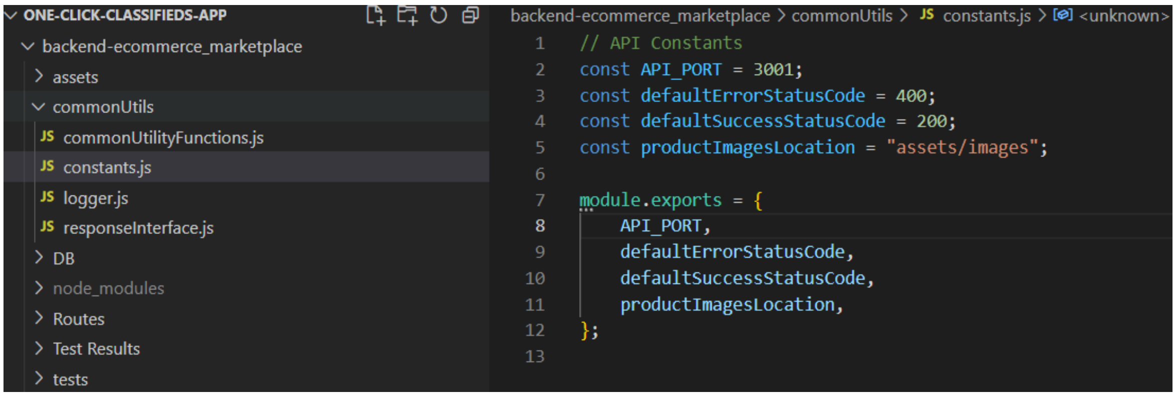


```
// Validate and make sure that phoneNumber, lastName, firstName, password, email are not empty
if (noFieldsAreEmptyOrNull([email, password, firstName, lastName, phoneNumber])) {
    failureAPIResponse(req, res, "Input Fields Cannot be empty");
    logger.createExceptionLog(req, res, "Input Fields Cannot be empty");
    return;
}

// Validate whether Email or Phone Number exists
if (anyofTheListIsNotEmpty([emailList, phoneList])) {
    failureAPIResponse(req, res, "Phone number or email address already exists");
    logger.createExceptionLog("Phone number or email address already exists");
    return;
}
```

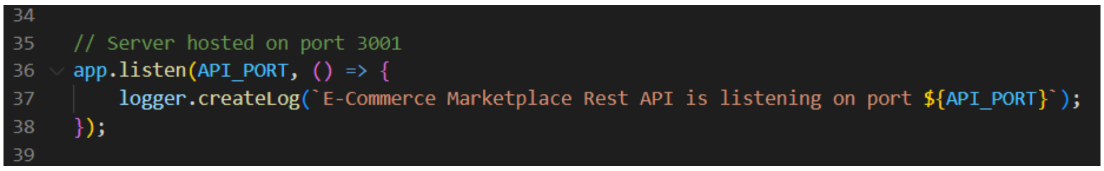
4.3 Replace Magic Number with Symbolic Constants

Magic Numbers should never be used in the code. Magic Numbers are numbers that are used in the code without any explanation. These cause confusion to people who are reading the code, as it might not make sense to the reader that what is the significance of the number.

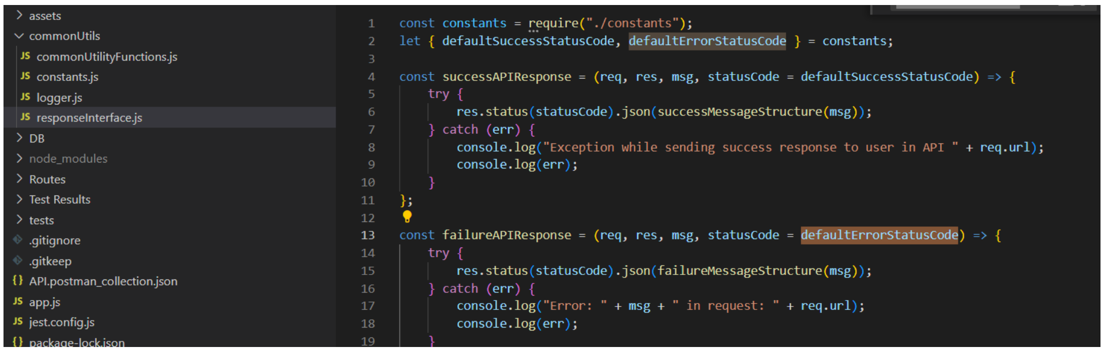
In order to prevent this confusion, we should use constants with proper naming conventions to define such variables.


ONE-CLICK-CLASSIFIEDS-APP
backend-ecommerce_marketplace
assets
commonUtils
commonUtilityFunctions.js
constants.js
logger.js
responseInterface.js
DB
node_modules
Routes
Test Results
tests
backend-ecommerce_marketplace > commonUtils > constants.js > <unknown>

```
// API Constants
const API_PORT = 3001;
const defaultErrorStatusCode = 400;
const defaultSuccessStatusCode = 200;
const productImagesLocation = "assets/images";

module.exports = {
    API_PORT,
    defaultErrorStatusCode,
    defaultSuccessStatusCode,
    productImagesLocation,
};
```



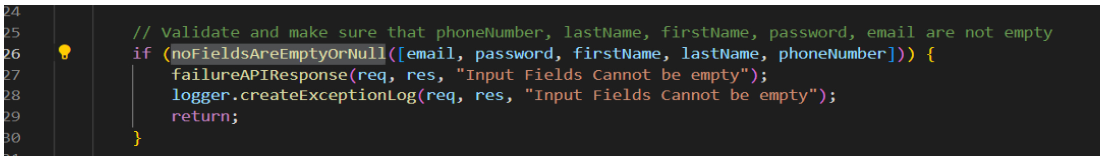
```
// Server hosted on port 3001
app.listen(API_PORT, () => {
    logger.createLog(`E-Commerce Marketplace Rest API is listening on port ${API_PORT}`);
});
```



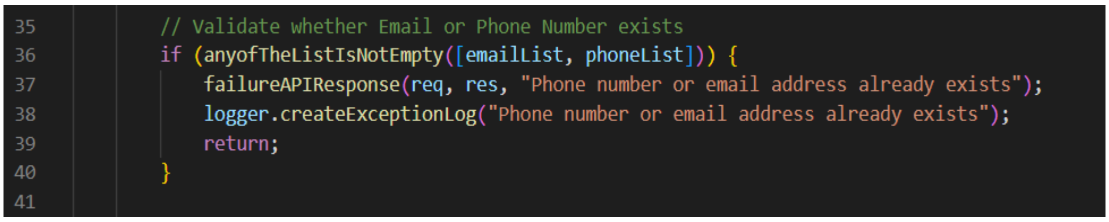
```
const constants = require("./constants");
let { defaultSuccessStatusCode, defaultErrorStatusCode } = constants;

const successAPIResponse = (req, res, msg, statusCode = defaultSuccessStatusCode) => {
    try {
        res.status(statusCode).json(successMessageStructure(msg));
    } catch (err) {
        console.log("Exception while sending success response to user in API " + req.url);
        console.log(err);
    }
};

const failureAPIResponse = (req, res, msg, statusCode = defaultErrorStatusCode) => {
    try {
        res.status(statusCode).json(failureMessageStructure(msg));
    } catch (err) {
        console.log("Error: " + msg + " in request: " + req.url);
        console.log(err);
    }
```

4.4 Consolidate Conditional Expression

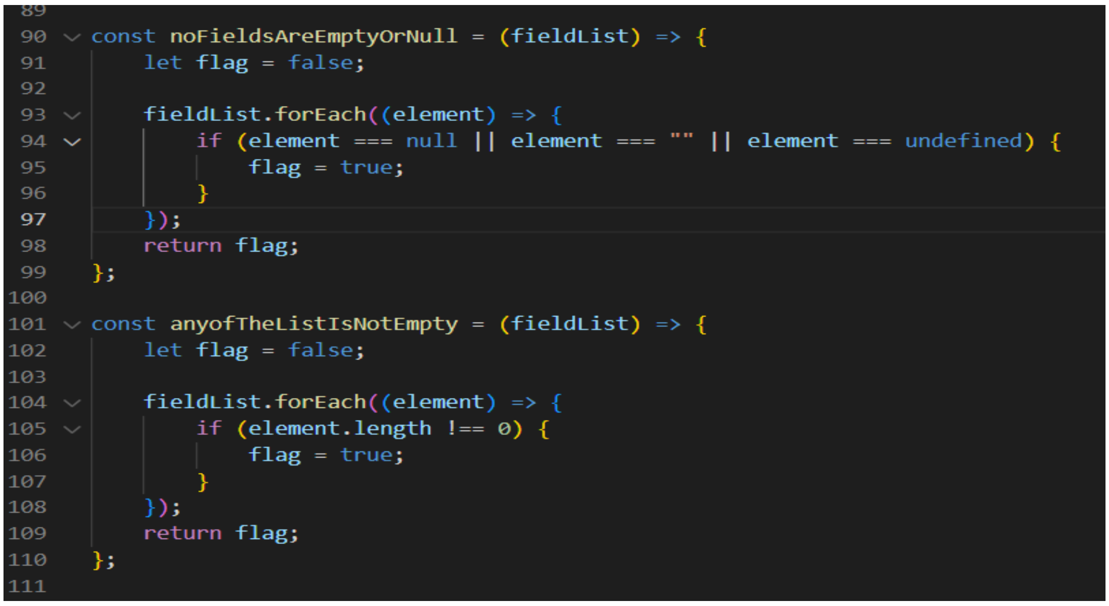
This refactoring technique is used to consolidate multiple conditional statements that produce the same result. This helps make the code more readable.



```
    // Validate and make sure that phoneNumber, lastName, firstName, password, email are not empty
    if (noFieldsAreEmptyOrNull([email, password, firstName, lastName, phoneNumber])) {
        failureAPIResponse(req, res, "Input Fields Cannot be empty");
        logger.createExceptionLog(req, res, "Input Fields Cannot be empty");
        return;
    }
```



```
    // Validate whether Email or Phone Number exists
    if (anyofTheListIsNotEmpty([emailList, phoneList])) {
        failureAPIResponse(req, res, "Phone number or email address already exists");
        logger.createExceptionLog("Phone number or email address already exists");
        return;
    }
```

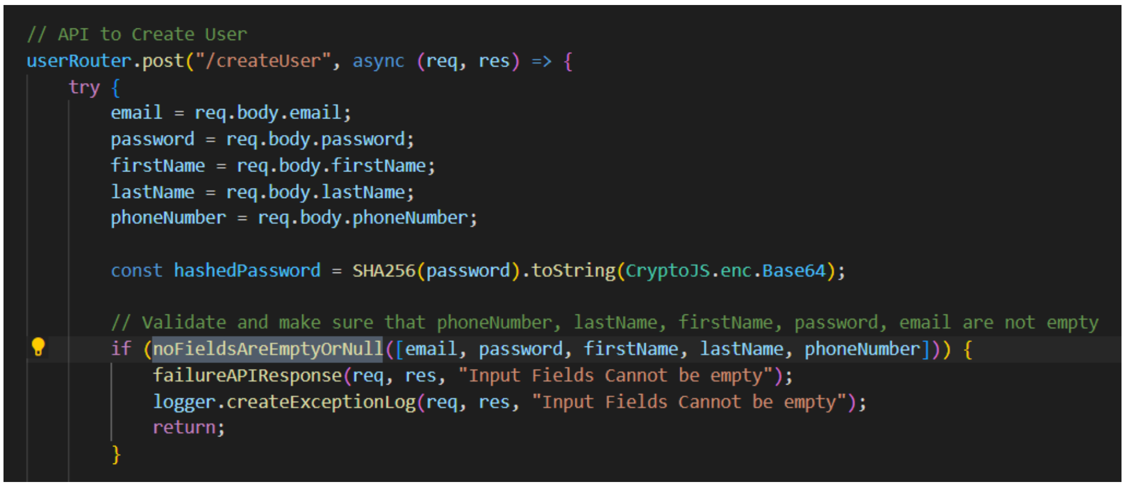


```
const noFieldsAreEmptyOrNull = (fieldList) => {
    let flag = false;

    fieldList.forEach((element) => {
        if (element === null || element === "" || element === undefined) {
            flag = true;
        }
    });
    return flag;
};

const anyofTheListIsNotEmpty = (fieldList) => {
    let flag = false;

    fieldList.forEach((element) => {
        if (element.length !== 0) {
            flag = true;
        }
    });
    return flag;
};
```

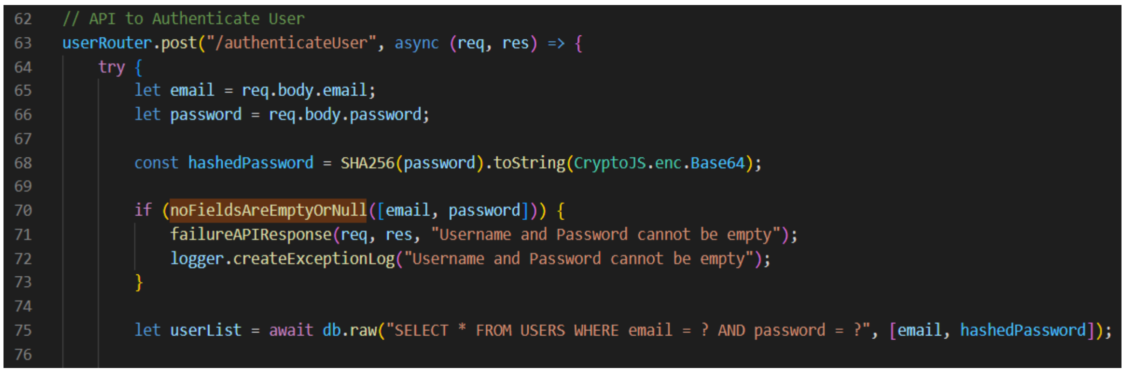
4.5 Consolidate Duplicate Conditional Fragments

This refactoring is important when we have similar conditional statements repeating over a span of multiple function, this repetition of code is not a good practice, because if due to a design changes, a certain field has some changes, then this conditional change will have to be cascaded to all the functions.

In order to prevent this code reputation, we can have consolidated conditional fragments of the code to a common function.



```
// API to Create User
userRouter.post("/createUser", async (req, res) => {
    try {
        email = req.body.email;
        password = req.body.password;
        firstName = req.body.firstName;
        lastName = req.body.lastName;
        phoneNumber = req.body.phoneNumber;

        const hashedPassword = SHA256(password).toString(CryptoJS.enc.Base64);

        // Validate and make sure that phoneNumber, lastName, firstName, password, email are not empty
        if (noFieldsAreEmptyOrNull([email, password, firstName, lastName, phoneNumber])) {
            failureAPIResponse(req, res, "Input Fields Cannot be empty");
            logger.createExceptionLog(req, res, "Input Fields Cannot be empty");
            return;
        }
```

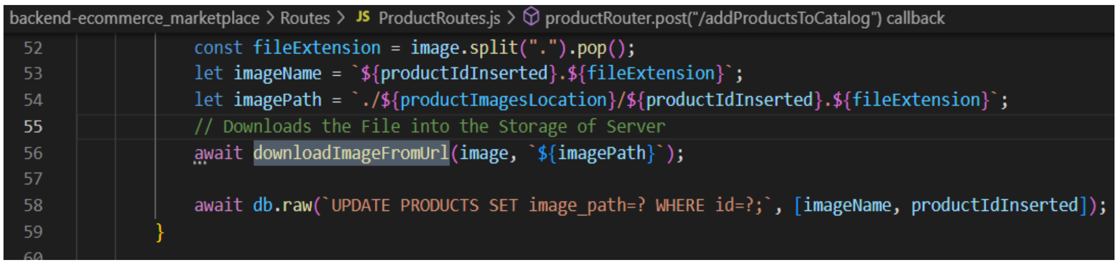


```
// API to Authenticate User
userRouter.post("/authenticateUser", async (req, res) => {
    try {
        let email = req.body.email;
        let password = req.body.password;

        const hashedPassword = SHA256(password).toString(CryptoJS.enc.Base64);

        if (noFieldsAreEmptyOrNull([email, password])) {
            failureAPIResponse(req, res, "Username and Password cannot be empty");
            logger.createExceptionLog("Username and Password cannot be empty");
        }

        let userList = await db.raw("SELECT * FROM USERS WHERE email = ? AND password = ?", [email, hashedPassword]);

```

4.6 Divide Methods into Smaller Computational Blocks

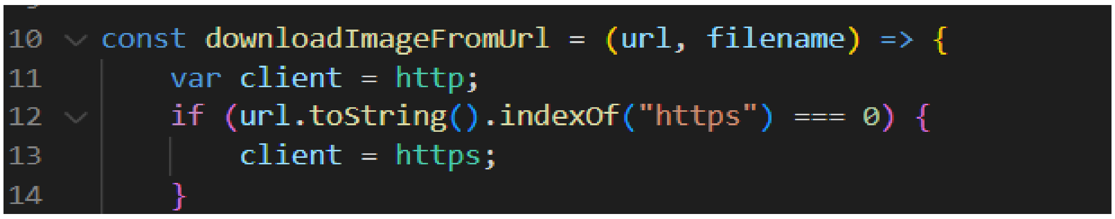
This refactoring technique involves Dividing Methods into smaller blocks of code. This has the following advantages:

- Increases Re-usability of the code: smaller blocks of code can be re-used for by other methods too. Even if the code is not re-used now. It is better to divide code into smaller blocks as this may be used later and helps prevent duplicate code later.
- Increases Code-Readability: Helps the reader understand what is the purpose of each block of code (method).
- Prevents increasing of code size to uncontrollable size that leads to tangled local variables that are hard to isolate at a later stage.



backend-ecommerce_marketplace > Routes > JS ProductRoutes.js > productRouter.post("/addProductsToCatalog") callback

```
            const fileExtension = image.split(".").pop();
            let imageName = `${productIdInserted}.${fileExtension}`;
            let imagePath = `./${productImagesLocation}/${productIdInserted}.${fileExtension}`;
            // Downloads the File into the Storage of Server
            await downloadImageFromUrl(image, `${imagePath}`);

            await db.raw(`UPDATE PRODUCTS SET image_path=? WHERE id=?;`, [imageName, productIdInserted]);
        }
```



```
const downloadImageFromUrl = (url, filename) => {
    var client = http;
    if (url.toString().indexOf("https") === 0) {
        client = https;
    }
```

5 Parameterized Database Queries

A parameterized query is a query in which we use placeholders for parameters and then the parameter values are supplied at execution time. Parameterized Queries are written to prevent SQL Injection and provide flexibility to the developer as we can develop these queries in advance and re-use them as per the requirement.

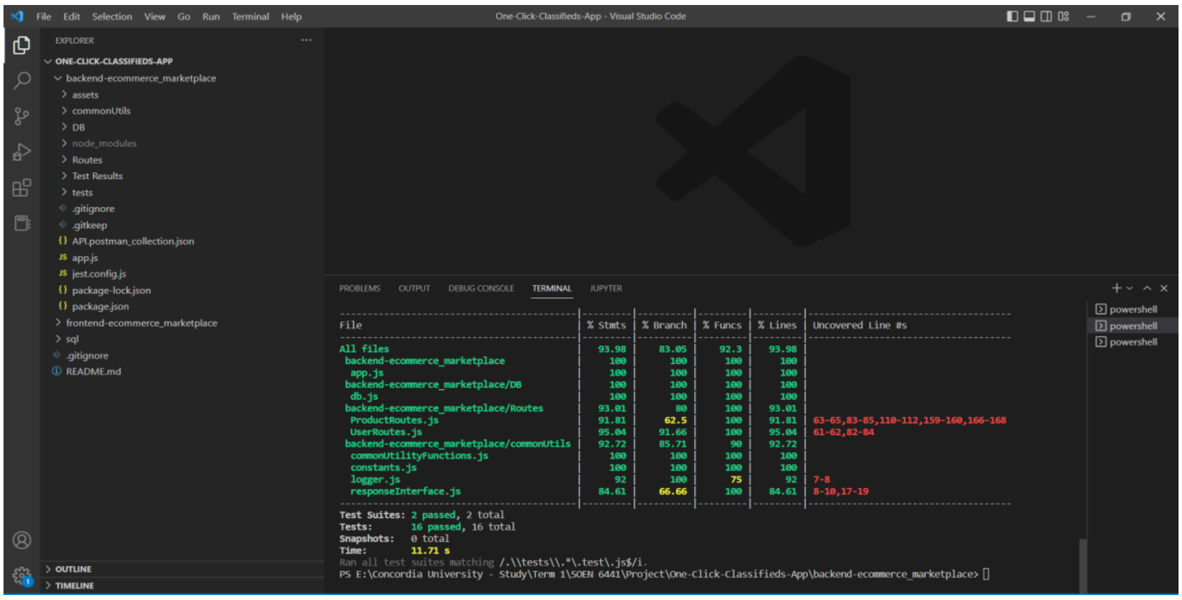
Also, depending upon the flavor of SQL used, some databases also provide caching for parameterized queries (example: SQL Server’s plan cache).

All database queries in our codebase are parameterized.

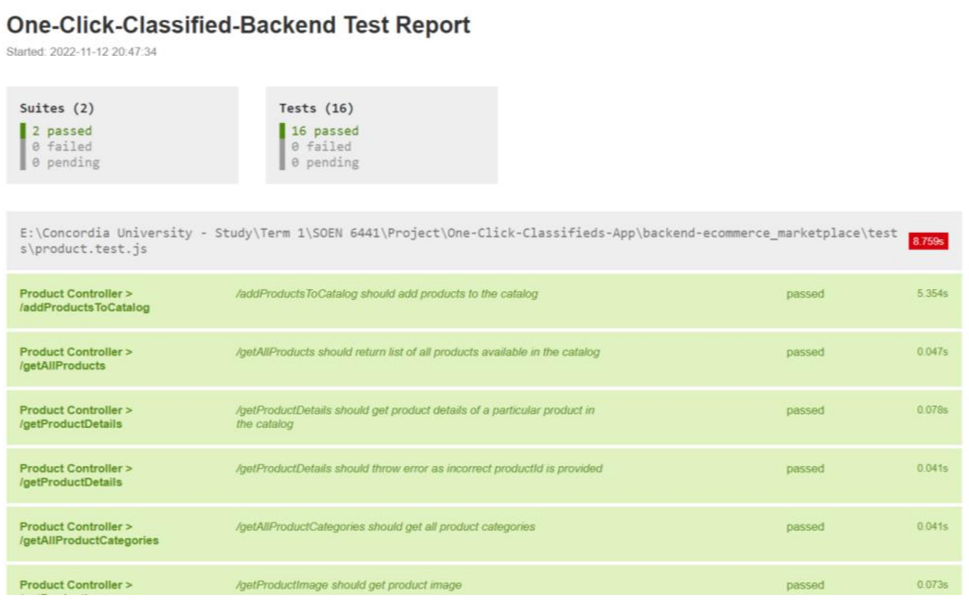
6 Testing

For the purpose of testing, we have used unit testing to test our API endpoints. We have used the jest and supertest to test the functionality of the code and Istanbul for reporting the test results in an html format.
Following are the screenshots of the testing results:

6.1 Testing Results


File Edit Selection View Go Run Terminal Help
One-Click-Classifieds-App - Visual Studio Code
EXPLORER
ONE-CLICK-CLASSIFIEDS-APP
backend-ecommerce_marketplace
assets
commonUtils
DB
node_modules
Routes
Test Results
tests
.gitignore
.gitkeep
API.postman_collection.json
app.js
jest.config.js
package-lock.json
package.json
frontend-ecommerce_marketplace
sql
.gitignore
README.md
OUTLINE
TIMELINE
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
powershell
powershell
powershell

```
-----------------------------------------|---------|----------|---------|---------|----------------------------------------
File                                     | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------------------------------|---------|----------|---------|---------|----------------------------------------
All files                                |   93.98 |    83.05 |    92.3 |   93.98 |
 backend-ecommerce_marketplace           |     100 |      100 |     100 |     100 |
  app.js                                 |     100 |      100 |     100 |     100 |
 backend-ecommerce_marketplace/DB        |     100 |      100 |     100 |     100 |
  db.js                                  |     100 |      100 |     100 |     100 |
 backend-ecommerce_marketplace/Routes    |   93.01 |       80 |     100 |   93.01 |
  ProductRoutes.js                       |   91.81 |     62.5 |     100 |   91.81 | 63-65,83-85,110-112,159-160,166-168
  UserRoutes.js                          |   95.04 |    91.66 |     100 |   95.04 | 61-62,82-84
 backend-ecommerce_marketplace/commonUtils |   92.72 |    85.71 |      90 |   92.72 |
  commonUtilityFunctions.js              |     100 |      100 |     100 |     100 |
  constants.js                           |     100 |      100 |     100 |     100 |
  logger.js                              |      92 |      100 |      75 |      92 | 7-8
  responseInterface.js                   |   84.61 |    66.66 |     100 |   84.61 | 8-10,17-19
-----------------------------------------|---------|----------|---------|---------|----------------------------------------
Test Suites: 2 passed, 2 total
Tests:       16 passed, 16 total
Snapshots:   0 total
Time:        11.71 s
Ran all test suites matching /.\\tests\\.*\.test\.js$/i.
PS E:\Concordia University - Study\Term 1\SOEN 6441\Project\One-Click-Classifieds-App\backend-ecommerce_marketplace> 
```



One-Click-Classified-Backend Test Report

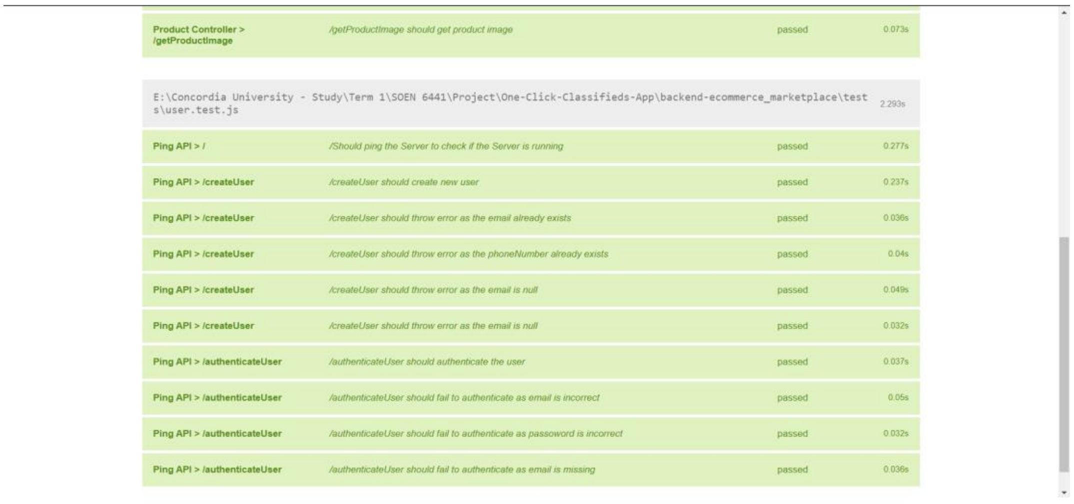
Started: 2022-11-12 20:47:34

```
Suites (2)
2 passed
0 failed
0 pending
```

```
Tests (16)
16 passed
0 failed
0 pending
```

E:\Concordia University - Study\Term 1\SOEN 6441\Project\One-Click-Classifieds-App\backend-ecommerce_marketplace\tests\product.test.js 8.759s

Product Controller > /addProductsToCatalog	/addProductsToCatalog should add products to the catalog	passed	5.354s
Product Controller > /getAllProducts	/getAllProducts should return list of all products available in the catalog	passed	0.047s
Product Controller > /getProductDetails	/getProductDetails should get product details of a particular product in the catalog	passed	0.078s
Product Controller > /getProductDetails	/getProductDetails should throw error as incorrect productId is provided	passed	0.041s
Product Controller > /getAllProductCategories	/getAllProductCategories should get all product categories	passed	0.041s
Product Controller >	/getProductImage should get product image	passed	0.073s

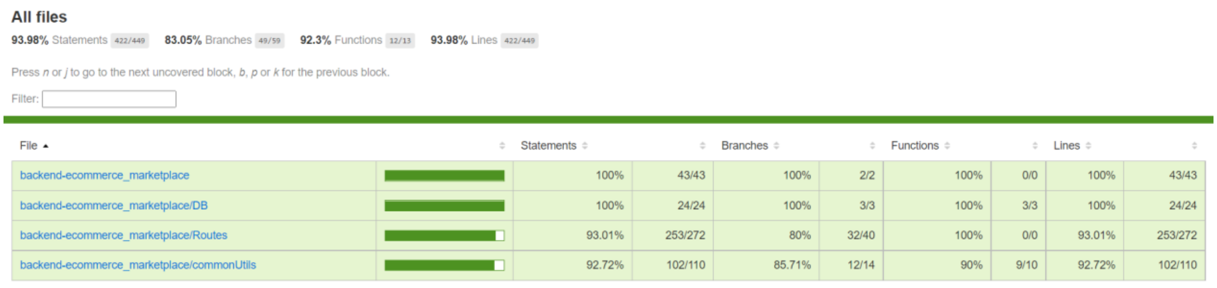


Product Controller > /getProductImage	/getProductImage should get product image	passed	0.073s

E:\Concordia University - Study\Term 1\SOEN 6441\Project\One-Click-Classifieds-App\backend-ecommerce_marketplace\tests\user.test.js 2.293s

Ping API > /	/Should ping the Server to check if the Server is running	passed	0.277s
Ping API > /createUser	/createUser should create new user	passed	0.237s
Ping API > /createUser	/createUser should throw error as the email already exists	passed	0.036s
Ping API > /createUser	/createUser should throw error as the phoneNumber already exists	passed	0.04s
Ping API > /createUser	/createUser should throw error as the email is null	passed	0.049s
Ping API > /createUser	/createUser should throw error as the email is null	passed	0.032s
Ping API > /authenticateUser	/authenticateUser should authenticate the user	passed	0.037s
Ping API > /authenticateUser	/authenticateUser should fail to authenticate as email is incorrect	passed	0.05s
Ping API > /authenticateUser	/authenticateUser should fail to authenticate as passoword is incorrect	passed	0.032s
Ping API > /authenticateUser	/authenticateUser should fail to authenticate as email is missing	passed	0.036s

6.2 Unit Test - Code Coverage Information



All files

93.98% Statements 422/449 **83.05%** Branches 49/59 **92.3%** Functions 12/13 **93.98%** Lines 422/449

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File ▲		Statements		Branches		Functions		Lines	
backend-ecommerce_marketplace		100%	43/43	100%	2/2	100%	0/0	100%	43/43
backend-ecommerce_marketplace/DB		100%	24/24	100%	3/3	100%	3/3	100%	24/24
backend-ecommerce_marketplace/Routes		93.01%	253/272	80%	32/40	100%	0/0	93.01%	253/272
backend-ecommerce_marketplace/commonUtils		92.72%	102/110	85.71%	12/14	90%	9/10	92.72%	102/110

7 References

[1] “React – a JavaScript library for building user interfaces,” – *A JavaScript library for building user interfaces*. [Online]. Available: https://reactjs.org/. [Accessed: 01-Nov-2022]

[2] *Fake store API*. [Online]. Available: https://fakestoreapi.com/products. [Accessed: 01-Nov-2022]

[3] “Software design pattern,” *Wikipedia*, 21-Oct-2022. [Online]. Available: https://en.wikipedia.org/wiki/Software_design_pattern. [Accessed: 10-Nov-2022]

[4] “Unit testing,” *Wikipedia*, 27-Aug-2022. [Online]. Available: https://en.wikipedia.org/wiki/Unit_testing. [Accessed: 11-Nov-2022]

[5] “Refactoring.Guru”. Retrieved November 13, 2022, [Online]. Available: https://refactoring.guru/ [Accessed: 07-Nov-2022]