

Name: **Mahindu Bandaranayake**

Student Reference Number: **10749841**

Module Code: **PUSL3120**

Module Name : **Full Stack Development**

Coursework Title:

Deadline Date: **16/01/2023**

Member of staff responsible for coursework:  
**Dr Mark Dixon**

Programme:

**BSc. (Hons) Computer Science University of Plymouth**

Please note that University Academic Regulations are available under Rules and Regulations on the University website [www.plymouth.ac.uk/studenthandbook](http://www.plymouth.ac.uk/studenthandbook).

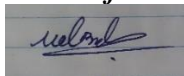
Group work: please list all names of all participants formally associated with this work and state whether the work was undertaken alone or as part of a team. Please note you may be required to identify individual responsibility for component parts.

*We confirm that we have read and understood the Plymouth University regulations relating to Assessment Offences and that we are aware of the possible penalties for any breach of these regulations. We confirm that this is the independent work of the group.*

Signed on behalf of the group:

Individual assignment: *I confirm that I have read and understood the Plymouth University regulations relating to Assessment Offences and that I am aware of the possible penalties for any breach of these regulations. I confirm that this is my own independent work.*

Signed :



Use of translation software: failure to declare that translation software or a similar writing aid has been used will be treated as an assessment offence.

I \*have used/not used translation software.

If used, please state name of software.....

Overall mark \_\_\_\_\_ % Assessors Initials \_\_\_\_\_ Date \_\_\_\_\_

# FULL – STACK DEVELOPMENT

PUSL 3120



## DEVELOPMENT REPORT

***PREPARED BY***

Mahindu Bandaranayake

***SUPERVISED BY***

Dr. Mark Dixon

## **TABLE OF CONTENT**

<b>Topic</b>	<b>Page No</b>
<b>Chapter 1: Requirement Analysis</b>	<b>4</b>
1.1 Development Objectives	
1.2 Application Functionalities	
1.3 Target Users	
<b>Chapter 2: System Design</b>	<b>5</b>
2.1 System Architecture	
2.2 Class Diagram	
2.3 Implementation Structure	
2.4 Technology Strategies	
<b>Chapter 3: Testing Criteria</b>	<b>14</b>
3.1 Unit Testing	
3.2 Integration Testing	
3.3 Usability Testing	
3.4 Reasoning	
<b>Chapter 4 : DevOps Importance</b>	<b>20</b>
4.1 Pipeline usage	
<b>Chapter 5 : Personal Reflection</b>	<b>21</b>
5.1 Issues Encountered	
5.2 Conclusion	
<b>D3 – Video (Presentation Link)</b>	<b>23</b>
<b>Reference</b>	<b>24</b>

# CHAPTER 1

## REQUIREMENT ANALYSIS

### 1.1 Development Objectives

Agriculture is a sector that a country's economy is heavily relying on either exporting or importing certain goods. To enhance the productivity of the crop cultivation and utilize the annual production level by analysing the amount of quantity and types of crops cultivated would be helpful to take the maximum advantage of the harvest without letting it go to waste. By developing a management application related to this field, will benefit in many aspects as mentioned below

- To reduce the supply chain between buyers and vendors
- To reduce the distribution of ownership and centralize the monetary funds
- To direct towards the selling market and utilize the harvest throughout the county without letting the crops rotten (since information of crops can be taken prior to make effective decisions)

Extracted from a local article the following statement validates the issues and challenges mentioned above

*“Inadequacy of awareness program especially on Good Agricultural Practices (GAP) and organic certification, food habits and perceptions on foods of the traditional publics, inadequate linkages between famer-based communities, researchers, extension approaches and farmers and so on”* (Ranathunga *et al.*, 2018). Therefore, to solve these thorough an analysis is required and for that every bit of detail is needed to be collected

### 1.2 Application Functionalities

- ❖ CRUD Operation is initialized in the application (Insert, Update, Edit and Delete features)
- ❖ Retrieving and displaying content option is enabled to rectify the inserted details
- ❖ Navigation through pages
- ❖ Option to navigate to useful other websites to give an idea regarding the elements essential for the form information

### 1.3 Target Users

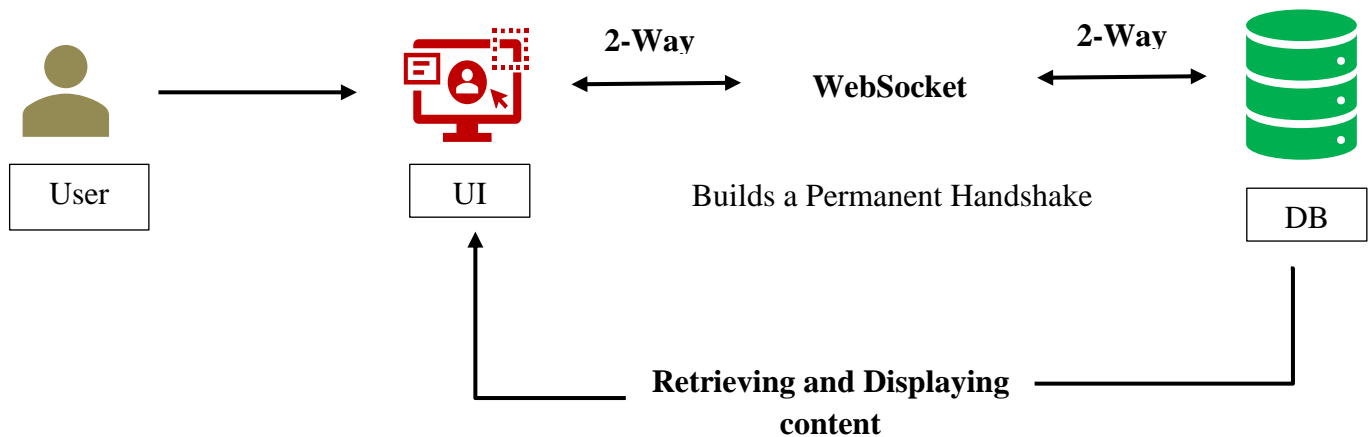
Any person with the authority of a firm can take in charge of entering data to the system

- Ideal for data-entry users
- Onsight visiting agents
- Statistics analysers
- Auditing committee members

## CHAPTER 2

### SYSTEM DESIGN

#### 2.1 System Architecture



**Explanation** : Once the user submits the details through the form application the WebSocket builds a connection between the server and the database to pass data back and forth when user requests information from the database

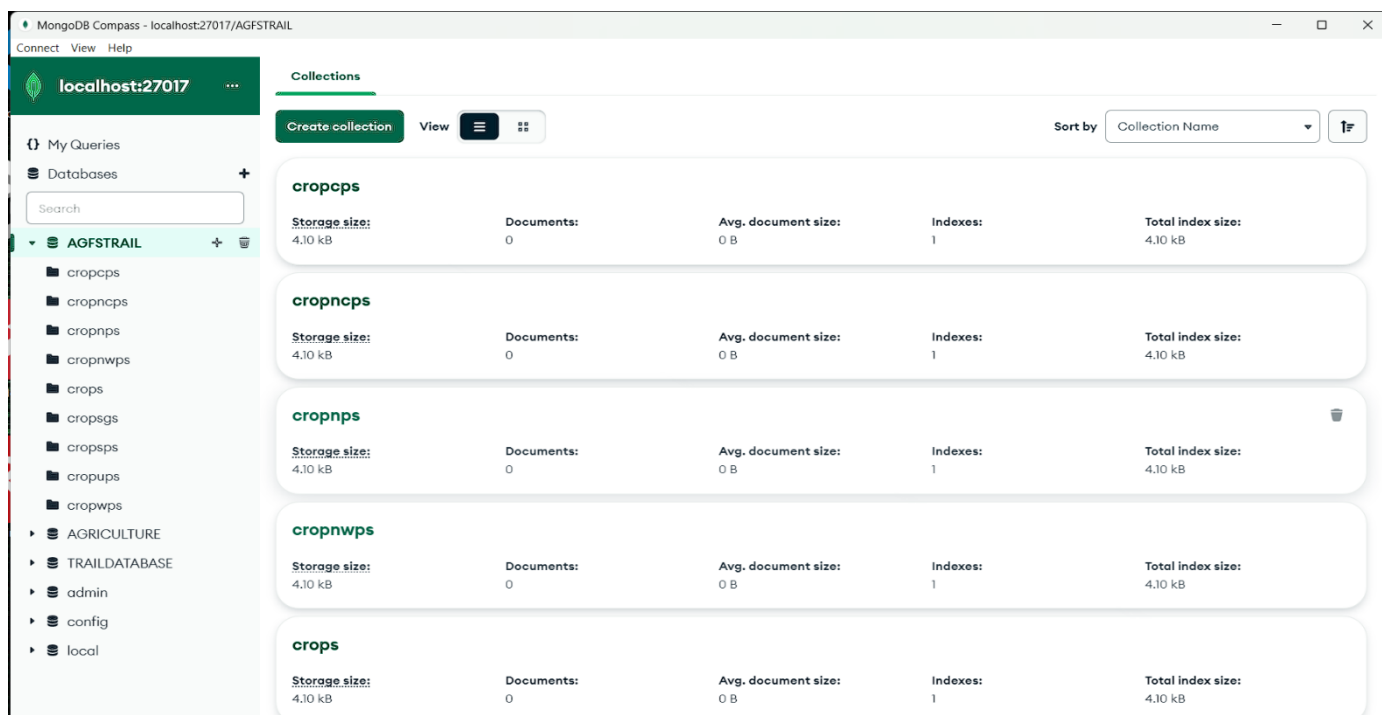
The following screenshots are from the built application documenting how the above-mentioned structure through a graphical representation

#### **1.Step : Entering User Information**

**Crop Type**  
[Rice](#)  
[Condiments](#)  
[Coarse Grains](#)  
[Legumes](#)  
[Root Crops](#)  
[Oil Crops](#)  
[Vegetables](#)  
[Fruits](#)  
[Mushrooms](#)  
[Unutilized Fruit Crops](#)

**Crop Information**  
Crop Name  Crop Quantity  Quantity Unit  Payment No   
  
Paid Amount  Deficit Amount  Time  Date   
  
**Verification Information**  
Cultivation Season  District No   
  
Fertilizers Used   
  
**Transaction Information**  
Market Location(Buyer)  Crop Cultivation Location   
  
**Crop Vendor**  
Name  Age  NIC No  Trade No   
  
**Crop Buyer**  
Name  Age  NIC No  Trade No

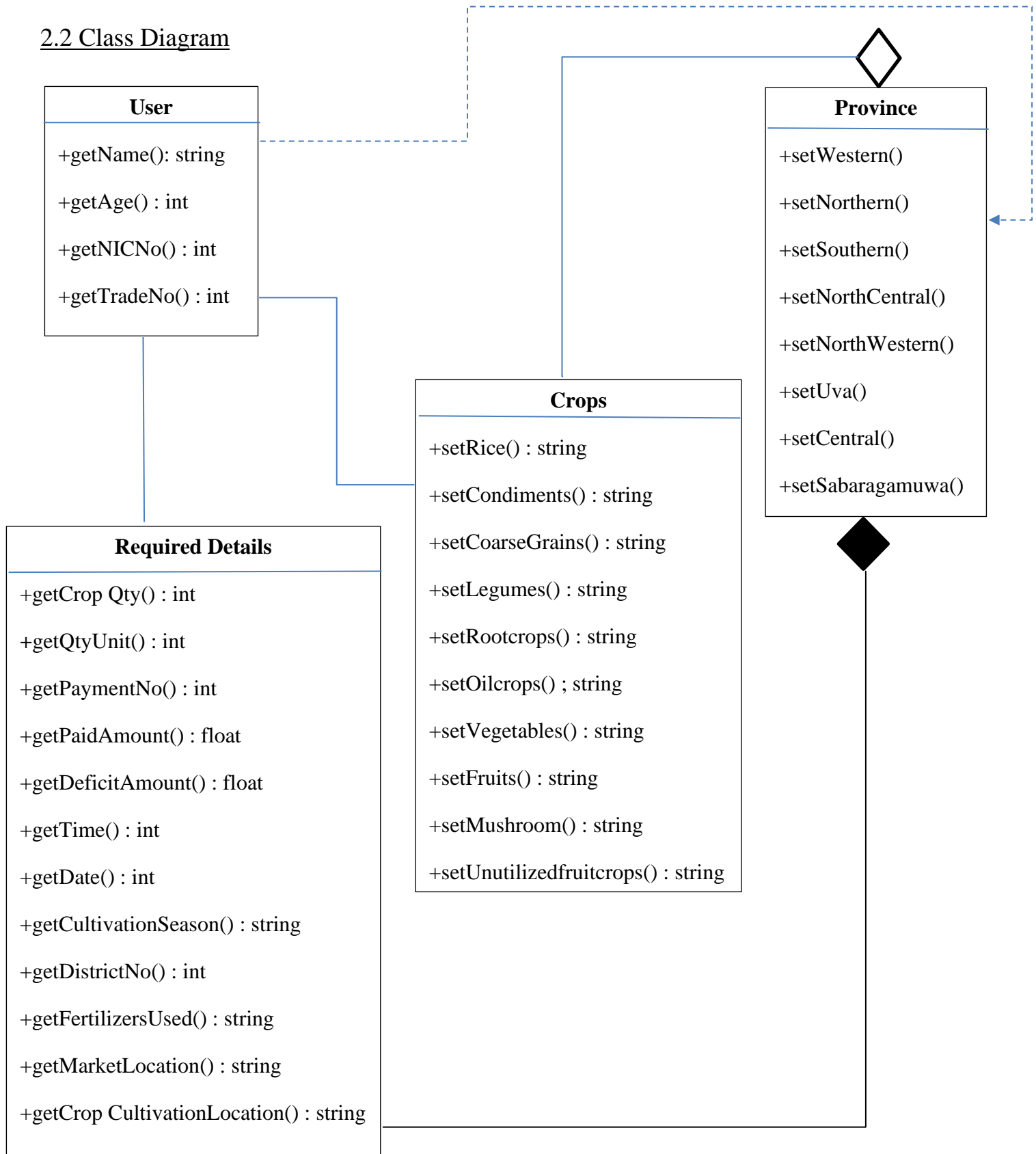
**2.Step :** Once the data is inserted the server will transfer the data into the relevant collection in the database



**3.Step :** The inserted data will be retrieved and shown in a different page in order to rectify if any mistake has been done

e Crop Back Home About Us Statistics																			
Western Province																			
Crop Type	Crop Quantity	Quantity Unit	Payment No	Paid Amount	Deficit Amount	Time	Date	Cultivation Season	Fertilizers Used	District No	Market Location	Crop Cultivation Location	CV Name	CV Age	CV NIC No	CV Trade No	CB Name	CB Age	CB NIC No
Uva Province Details																			
Crop Type	Crop Quantity	Quantity Unit	Payment No	Paid Amount	Deficit Amount	Time	Date	Cultivation Season	Fertilizers Used	District No	Market Location	Crop Cultivation Location	CV Name	CV Age	CV NIC No	CV Trade No	CB Name	CB Age	CB NIC No
Southern Province Details																			
Crop Type	Crop Quantity	Quantity Unit	Payment No	Paid Amount	Deficit Amount	Time	Date	Cultivation Season	Fertilizers Used	District No	Market Location	Crop Cultivation Location	CV Name	CV Age	CV NIC No	CV Trade No	CB Name	CB Age	CB NIC No
Sabaragamuwa Province Details																			
Crop Type	Crop Quantity	Quantity Unit	Payment No	Paid Amount	Deficit Amount	Time	Date	Cultivation Season	Fertilizers Used	District No	Market Location	Crop Cultivation Location	CV Name	CV Age	CV NIC No	CV Trade No	CB Name	CB Age	CB NIC No

## 2.2 Class Diagram



## 2.3 Implementation Structure

### Creating the backend server (app.js)

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

const app = express();

const wss = require('./websockets');
const WebSocket = require('ws');

wss.listen(app, { port: 8080 }, () => {
  console.log('WebSocket server is connected to port 8080');
});

const port = 3000;
app.listen(port, () => {
  console.log(`Express server listening on port ${port}`);
});
```

Express libraries were installed and imported to create the web server and store it in the app variable and WebSocket server is set up to listen for connection in the app server, where the connections are using the port 8080.

### Integrating MongoDB

```
const WebSocket = require('ws');

module.exports.listen = function (app) {
  const port = 8080;
  const wss = new WebSocket.Server({ server: app,
    port: port });

  wss.on('connection', function connection(ws) {
    ws.on('message', function incoming(message) {
      const data = JSON.parse(message);

      let collection;
      if (data.type === 'cropwp') {
        collection = cropwp;
      } else if (data.type === 'croppcp') {
        collection = croppcp;
      } else if (data.type === 'croppsp') {
        collection = croppsp;
      } else if (data.type === 'cropup') {
```



```

    collection = cropup;
  } else if (data.type === 'croppnp') {
    collection = croppnp;
  } else if (data.type === 'croppnwp') {
    collection = croppnwp;
  } else if (data.type === 'croppncp') {
    collection = croppncp;
  } else if (data.type === 'croppsg') {
    collection = croppsg;
  } else { /* empty */ }

const doc = new collection({
  CropType: data.CropType,
  CropQuantity: data.CropQuantity,
  QuantityUnit: data.QuantityUnit,
  PaymentNo: data.PaymentNo,
  PaidAmount: data.PaidAmount,
  DeficitAmount: data.DeficitAmount,
  Time: data.Time,
  date: data.date,
  CultivationSeason: data.CultivationSeason,
  FertilizersUsed: data.FertilizersUsed,
  DistrictNo: data.DistrictNo,
  MarketLocation: data.MarketLocation,
  CropCultivationLocation: data.CropCultivationLocation,
  CVName: data.CVName,
  CVAge: data.CVAge,
  CVNICNo: data.CVNICNo,
  CVTradeNo: data.CVTradeNo,
  CBName: data.CBName,
  CBAge: data.CBAge,
  CBNICNo: data.CBNICNo,
  CBTradeNo: data.CBTradeNo
});
doc.save(function (error) {
  if (error) {
    console.log(error);
  } else {
    console.log('Crop Information saved successfully!');
  }
});
});
});

```

The structure implements the code to export the WebSocket connection server which contain the event listener for incoming messages. 'Type field' is used to receive the details given by the message object to determine to which specific collection in MongoDB to save the messages obtained. The goal of the code is to create a new

document and pass the relevant information to the collection created, using 'json parse' where the data is passed as JavaScript objects.

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

mongoose.connect('mongodb://localhost:27017/AGFSTRAIL');

const {
  cropwpModel,
  cropcpModel,
  cropspModel,
  cropupModel,
  cropnpModel,
  cropnwpModel,
  cropnsgModel,
  cropncpModel
} = require('./models/crop');
```

Finally, the exported models are linked with the main file app.js to connect to the server using the port and importing the 'mongoose library' to link the connection string.

Below states the codes written to create the relevant schemas as mentioned above

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

const cropSchema = new mongoose.Schema({
  CropType: String,
  CropQuantity: Number,
  QuantityUnit: String,
  PaymentNo: Number,
  PaidAmount: mongoose.Types.Decimal128,
  DeficitAmount: mongoose.Types.Decimal128,
  Time: String,
  date: Number,
  CultivationSeason: String,
  FertilizersUsed: String,
  DistrictNo: Number,
  MarketLocation: String,
  CropCultivationLocation: String,
  CVName: String,
  CVAge: Number,
  CVNICNo: Number,
  CVTradeNo: Number,
  CBName: String,
  CBAge: Number,
  CBNICNo: Number,
  CBTradeNo: Number,
});

const cropwpModel = mongoose.model('cropwp', cropSchema);
```

```

const cropcpModel = mongoose.model('cropcp', cropSchema);
const cropspModel = mongoose.model('cropsp', cropSchema);
const cropupModel = mongoose.model('cropup', cropSchema);
const cropnpModel = mongoose.model('cropnp', cropSchema);
const cropnwpModel = mongoose.model('cropnwp', cropSchema);
const cropncpModel = mongoose.model('cropncp', cropSchema);
const cropnsgModel = mongoose.model('cropsg', cropSchema);

module.exports = {
  cropwpModel,
  cropcpModel,
  cropspModel,
  cropupModel,
  cropnpModel,
  cropnwpModel,
  cropncpModel,
  cropnsgModel
};

```

## Routing in Angular application

```

import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { WPDataComponent } from '../WPData/WPData.component';
import { UPDataComponent } from '../UPData/UPData.component';
import { SPDataComponent } from '../SPData/SPData.component';
import { SGDataComponent } from '../SGData/SGData.component';
import { NWPDataComponent } from '../NWPData/NWPData.component';
import { NPDataComponent } from '../NPData/NPData.component';
import { NCPDataComponent } from '../NCPData/NCPData.component';
import { CPDataComponent } from '../CPData/CPData.component';
import { AppComponent } from '../app.component';

const routes: Routes = [
  { path: '', component: AppComponent },
  { path: 'WPData', component: WPDataComponent },
  { path: 'UPData', component: UPDataComponent },
  { path: 'SPData', component: SPDataComponent },
  { path: 'SGData', component: SGDataComponent },
  { path: 'NWPData', component: NWPDataComponent },
  { path: 'NPData', component: NPDataComponent },
  { path: 'NCPData', component: NCPDataComponent },
  { path: 'CPData', component: CPDataComponent }
];

```

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Once after populating the '.ts' files and html files with the relevant codes, navigation through the web application is necessary. The code defines an array of route configuration objects, routes, using the 'Routes' type. 'RouterModule' is imported, and routes are defined to the 'forRoot' method to generate a set of routes for the application

## 2.4 Technology Strategies

### **Use of WebSockets**

Compared to http, WebSockets are faster and creates persistent connection between client and a server under a low latency level, since it would only require a single connection to access all the communication. Therefore, it would be ideal to use WebSockets for database integration in an Angular Node project because they allow for **real-time updates** to be pushed from the server to the client, making it easy to keep the UI in sync with the database.

### **Angular Frontend Development**

Html codes are written to design the framework outline, while importing '**Formsmodule**' to create the form application within the framework. All the input fields are connected using '**ngmodel**' with the database collections to pass the information upon submit. Every input field is defined under the typescript file to link with the html file. Also, necessary retrieval mechanism is established to display and create instances to edit and delete record fields in a tabular format.

### **Code Analyses**

'**Eslint**' static analysis tool was used to inspect the code to identify for possible syntax error and to enforce specific coding standards, making it easier to maintain a consistent codebase and to write code that is consistent. The below example would clarify the stated details

WPCropType: string=""; can be also written as WPCropType="" which minimizes all unnecessary lines and definitions which helps to sustains the lines of codes.

### **Version Control**

Git is being used to commit the changes and add it to the GitHub repository  
<https://github.com/MahinduBandaranayake/Full-StackDEVPROJECT.git>

## Dependencies and Libraries

Necessary dependencies were installed and imported to relevant files making it available to support WebSockets and other technologies mentioned

```
"dependencies": {
  "@angular/animations": "^15.0.0",
  "@angular/common": "^15.0.0",
  "@angular/compiler": "^15.0.0",
  "@angular/core": "^15.0.0",
  "@angular/forms": "^15.0.0",
  "@angular/platform-browser": "^15.0.0",
  "@angular/platform-browser-dynamic": "^15.0.0",
  "@angular/router": "^15.0.0",
  "bufferutil": "^4.0.7",
  "crypto-browserify": "^3.12.0",
  "http-browserify": "^1.7.0",
  "https": "^1.0.0",
  "rxjs": "~7.5.0",
  "stream-browserify": "^3.0.0",
  "stream-http": "^3.2.0",
  "tls-browserify": "^0.2.2",
  "tslib": "^2.3.0",
  "utf-8-validate": "^5.0.10",
  "ws": "^8.11.0",
  "zone.js": "~0.12.0"
}
```

Also, for testing, other relevant dependencies were installed to the project environment

```
"devDependencies": {
  "@angular-devkit/build-angular": "^15.0.4",
  "@angular/cli": "~15.0.4",
  "@angular/compiler-cli": "^15.0.0",
  "@types/chai": "^4.3.4",
  "@types/jasmine": "~4.3.0",
  "@types/mocha": "^10.0.1",
  "chai": "^4.3.7",
  "jasmine": "^4.5.0",
  "jasmine-core": "~4.5.0",
  "karma": "~6.4.0",
  "karma-chrome-launcher": "~3.1.0",
  "karma-coverage": "~2.2.0",
  "karma-jasmine": "~5.1.0",
  "karma-jasmine-html-reporter": "~2.0.0",
  "mocha": "^10.2.0",
  "typescript": "~4.8.2"
}
```

## CHAPTER 3

### TESTING CRITERIA

#### 3.1 Unit Testing

The goal of unit testing is to validate each component of the application to ensure it is working as intended and meets the specified requirements. Each component in the application is run against this test scans to identify the parts with implications to justify them and rescutinise the issues of the code, where '**Karma**' test runner is used to execute the following unit testing cases in the '**spec.ts**' file until requirements are met by using the '**jasmin**' framework.

Each additional component was tested. The components are listed below

- WPData
- CPData
- NWpdata
- NPData
- SGData
- SPData
- NCPData
- UPData
- Statistics

The same procedure is carried out for each component. The framework of testing is explained below using a component above as an example

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { DebugElement } from '@angular/core';

import { WPDataComponent } from './WPData.component';

describe('WPDataComponent', () => {
  let component: WPDataComponent;
  let fixture: ComponentFixture<WPDataComponent>;
  let backLink: DebugElement;
  let homeLink: DebugElement;
  let aboutUsLink: DebugElement;
  let statisticsLink: DebugElement;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
```

```

imports: [ FormsModule, ReactiveFormsModule ],
  declarations: [ WPDataComponent ]
})
.compileComponents();
}));

beforeEach(() => {
  fixture = TestBed.createComponent(WPDataComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();

  backLink = fixture.debugElement.query(By.css('.bt4'));
  homeLink = fixture.debugElement.query(By.css('.bt1'));
  aboutUsLink = fixture.debugElement.query(By.css('.bt2'));
  statisticsLink = fixture.debugElement.query(By.css('.bt3'));

});

it('should create', () => {
  expect(component.onSubmit).toEqual(true);
});
it('should have the correct routerLink for the back link', () => {
  expect(backLink.properties['routerLink']).toEqual('/');
});

it('should have the correct text for the home link', () => {
  expect(homeLink.nativeElement.textContent).toContain('Home');
});

it('should have the correct text for the about us link', () => {
  expect(aboutUsLink.nativeElement.textContent).toContain('About Us');
});

it('should have the correct text for the statistics link', () => {
  expect(statisticsLink.nativeElement.textContent).toContain('Statistics');
});
it('should submit the form and handle the data', () => {
  // Find the form element
  const formDebugElement = fixture.debugElement.query(By.css('form'));

  // Set the form input values
  component.WPCropType = 'Rice';
component.WPCultivationSeason = 'Yala';
  component.WPDistrictNo = parseInt('1',10);
  component.WPFertilizersUsed = 'Urea, DAP';
  component.WPMarketLocation = 'Matale';
  component.WPCropCultivationLocation='Nawala';
  component.WPCVName='Nimal';
  component.WPCVAge = parseInt('23', 10);

```

```

component.WPCVNICNo = parseInt('1267358901', 10);
component.WPCVTradeNo = parseInt('1207', 10);
component.WPCBName='Siripala';
component.WPCBAge = parseInt('45', 10);
component.WPCBNICNo = parseInt('3412456781', 10);
component.WPCBTradeNo = parseInt('2017', 10);
// Trigger the form submission
formDebugElement.triggerEventHandler('ngSubmit', null);

// Verify that the onSubmit() method was called and handled the form data correctly
expect(component.onSubmit).toEqual(true);
expect(component.WPCropType).toEqual('Rice');
expect(component.WPCropQuantity).toEqual(50);
expect(component.WPQuantityUnit).toEqual('kg');
expect(component.WPPaymentNo).toEqual(123456);
expect(component.WPPaidAmount).toEqual('10000');
expect(component.WPDeficitAmount).toEqual('0');
expect(component.WPTime).toEqual('14:00');
expect(component.WPdate).toEqual(2022-12-28);
expect(component.WPCultivationSeason).toEqual('Yala');
expect(component.WPDistrictNo).toEqual(1);
expect(component.WPFertilizersUsed).toEqual('Urea, DAP');
expect(component.WPMarketLocation).toEqual('Nawala');
expect(component.WPCropCultivationLocation).toEqual('Matale');
expect(component.WPCVName).toEqual('Nimal');
expect(component.WPCVAge).toEqual(23);
expect(component.WPCVNICNo).toEqual(126735890);
expect(component.WPCBName).toEqual('Siripala');
expect(component.WPCBAge).toEqual(45);
expect(component.WPCBNICNo ).toEqual(3412456781);
expect(component.WPCBTradeNo).toEqual(2017);

});

});

```

The above suite contains testing framework to test the component's template behaviour, form submission and other properties. The 'it' block contains the test case and verifies that the component's **'onSubmit'** method returns the expected value. The remaining test cases verify various aspects of the component's template, such as the presence and content of certain elements, and the values of certain component properties after form submission



## 3.2 Integration Testing

This software testing type is used to test the compatibility nature of different components working together. Integration testing can be done in many ways but for this development environment it is being used to test and identify the issues when integrating with the Mongo database.

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

const expect = chai.expect;
const WebSocket = require('ws');
const listen = require('./websockets/index');

describe('Integration tests', function() {
  let server;
  let ws;

  beforeEach(function(done) {

    server = listen.listen(8080);
    ws = new WebSocket('ws://localhost:8080');
    ws.on('open', function open() {
      done();
    });
  });

  afterEach(function(done) {

    ws.close();
    server.close(done);
  });

  it('Should save data with the "cropwp" type to the "cropwp" collection',
function(done) {
    ws.send(JSON.stringify({
      type: 'cropwp',
      CropType: 'Rice',
      CropQuantity: 100,
      QuantityUnit: 'kg',
      PaymentNo: 123456,
      PaidAmount: 1000,
      DeficitAmount: 0,
      Time: '10:00',
      date: '2022-01-01',
      CultivationSeason: 'Yala',
      FertilizersUsed: 'Agrochemicals',
      DistrictNo: 13,
      MarketLocation: 'Nawala',
      CropCultivationLocation: 'Matale',
      CVName: 'Maithrapala',
      CVAge: 35,
      CVNICNo: '1234567890123',
    }));
  });
});
```

```

CVTradeNo: '12345',
  CBName: 'Kumaradasa',
  CBAge: 40,
  CBNICNo: '2345678901234',
  CBTradeNo: '23456'
}));
ws.on('message', function incoming(message) {
  expect(message).to.equal('Crop Information saved successfully to cropwp
collection!');
  done();
});
});

```

The same procedure is used for the rest of the collections in the DB as well. In here it uses the ‘it’ function to define an individual test, which consists of sending a message to the server and verifying the response. Overall, this code is testing the integration between a WebSocket client and server, and it is using the **Mocha and Chai** libraries to structure the tests and make assertions about the expected behaviour of the system under test.

### 3.3 Usability Testing

The software was tested by a non-developer to get the raw feedback from the user. The following are thoughts reciprocated

	Excellent	Very Good	Satisfactory Level	Improvements Needed
Navigation				
User Instructions				
Accessibility towards functionalities				
Functional Options				
Extra Details				
Overall Feedback on the Application				

### 3.4 Reasoning

**‘JavaScript unit testing framework’** is used for testing JavaScript code. Due to its availability of good documentation and simpleness the components in the application are inspected using these methods. While integration testing was carried out to determine the tensile strength between the connection of database with the frontend application using mocha and chai, since facilitates testing the end-to-end functionality of the system and for variety of other elements in the application.



## CHAPTER 4

### DEVOPS IMPORTANCE

#### 4.1 Pipeline usage

Piping is the methodology used to attach the output stream to the input of another stream, making the necessary pathway to chain multiple streams together for manipulation of data. This method is useful to transform data which is read and write on a different stream.

The following example **(Note: not a part of the application)** demonstrated how to carry out a pipe activity in a component inside the application

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

import { Employee } from './employee.model';
import { EmployeeService } from './employee.service';

@Component({
  selector: 'app-employee-list',
  templateUrl: './employee-list.component.html',
  styleUrls: ['./employee-list.component.css']
})
export class EmployeeListComponent {
  employees$: Observable<Employee[]>;
  filteredEmployees$: Observable<Employee[]>;

  constructor(private employeeService: EmployeeService) {
    this.employees$ = this.employeeService.getEmployees();
  }

  filterEmployees(searchTerm: string) {
    this.filteredEmployees$ = this.employees$.pipe(
      map(employees => employees.filter(employee =>
employee.name.includes(searchTerm)))
    );
  }
}
```

The code performs an action where observable gets a list of employee objects from the EmployeeService and the filteredEmployees\$ observable is implemented. Using the map operator, it is possible to filter the list of employees, and which then will be transferred to observable stream in pipeline.

## CHAPTER 5

### PERSONAL REFLECTION

#### 5.1 Issues Encountered

- Was unable to implement the feature of depicting relevant graphs for the information provided. Initially the idea was to establish charts with the help of chart.js library and implement the feature of generating pdf. Due the fact that it was giving many errors the relevant parts were forced to remove from the application
- Implementation of **pipe** was difficult, but the following attempt was made but many errors were shown

```
import { Injectable } from '@angular/core';
import * as mongoose from 'mongoose';
import { Subject } from 'rxjs';
import { Decimal128 } from 'bson';
export type CropData = {
  CropType: string,
  CropQuantity: number,
  QuantityUnit: string,
  PaymentNo: number,
  PaidAmount: mongoose.Types.Decimal128,
  DeficitAmount: mongoose.Types.Decimal128,
  Time: string,
  date: number,
  CultivationSeason: string,
  FertilizersUsed: string,
  DistrictNo: number,
  MarketLocation: string,
  CropCultivationLocation: string,
  CVName: string,
  CVAge: number,
  CVNICNo: number,
  CVTradeNo: number,
  CBName: string,
  CBAge: number,
  CBNICNo: number,
  CBTradeNo: number,
};

@Injectable({
  providedIn: 'root'
})
export class CropFormDataService {

  socket: WebSocket;
  PaidAmount: mongoose.Types.Decimal128 = new Decimal128('0');
  DeficitAmount: mongoose.Types.Decimal128 = new Decimal128('0');

  data = new Subject<any[]>();

  updateData(newData: any) {
    // Add the new data to the existing data array
    this.data.next([...this.data.getValue(), newData]);
  }
}
```

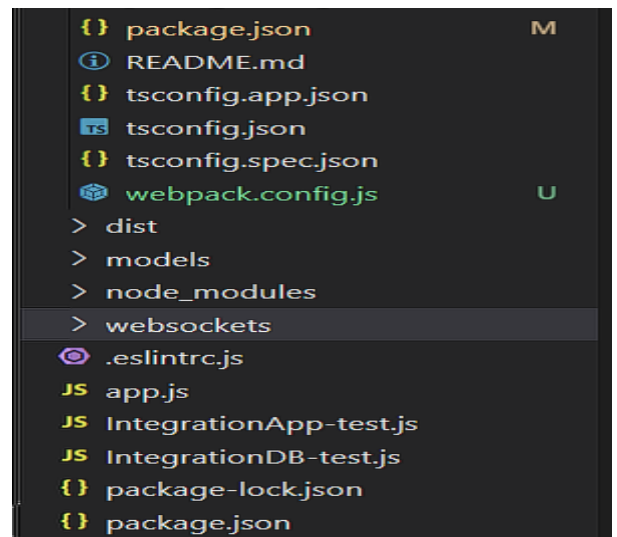
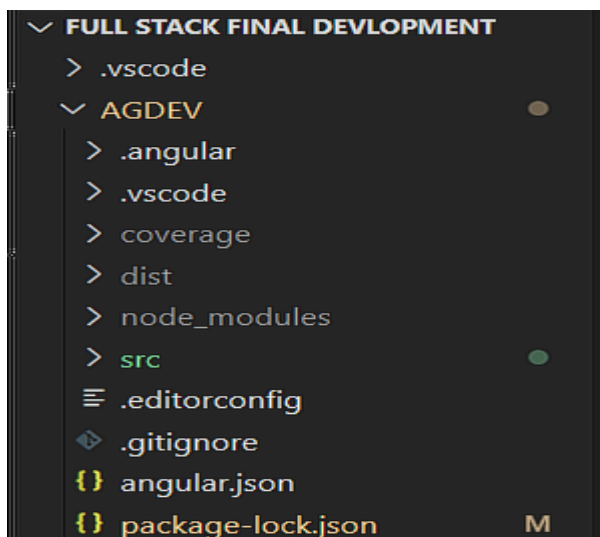
```

•   getData() {
•       return this.data.asObservable();
•   }
•   constructor() {
•       this.socket = new WebSocket('ws://localhost:8080');
•   }
•
•   sendCropWP(cropwp: CropData) {
•       this.socket.send(JSON.stringify(cropwp));
•   }
•
•   sendCropUP(cropup: CropData) {
•       this.socket.send(JSON.stringify(cropup));
•   }
•
•   sendCropSP (crobsp: CropData) {
•       this.socket.send(JSON.stringify(crobsp));
•   }
•
•   sendCropSG (cropsg: CropData) {
•       this.socket.send(JSON.stringify(cropsg));
•   }
•
•   sendCropNWP (cropnwp: CropData) {
•       this.socket.send(JSON.stringify(cropnwp));
•   }
•   sendCropNP (cronpn: CropData) {
•       this.socket.send(JSON.stringify(cronpn));
•   }
•
•   sendCropNCP (cropncp: CropData) {
•       this.socket.send(JSON.stringify(cropncp));
•   }
•
•   sendCropCP (cropcp: CropData) {
•       this.socket.send(JSON.stringify(cropcp));
•   }
•   }
•

```

## 5.2 Conclusion

Overall, CRUD operation is integrated in the application and user is able to enter and view details after form submission and make necessary adjustments. Run time errors were found when test runs were performed, was able to fix most of the issues but some were left out. Node modules(Angular + Node) from the application were removed before submission since it was taking up disk place. Code coverage was performed, and servers were cross checked.



### **D3- VIDEO (PRESENTATION LINK)**

YouTube Link : Unlisted Video

<https://youtu.be/-HQ6TkYvnAY>

## **REFERENCE**

Ranathunga, L.N. *et al.* (2018) ‘Agriculture in Sri Lanka: The Current Snapshot’, *International Journal of Environment, Agriculture and Biotechnology*, 3(1), pp. 118–125. Available at: <https://doi.org/10.22161/ijeab/3.1.15>.