Sri Lanka Institute of Information Technology

Procurement management System

Case Studies in Software Engineering

2022

Submitted by:

IT1913230 Lasal Sandeepa Hettiarachchi

IT19139036 Senura Vihan Jayadeva

IT19120980 Dilmi Nimasha Palliyaguruge

IT19146898 Ayodya Banuka Fernando

Table of Contents

Introduction

The following document is the implementation walkthrough of the Procurement management system. In this document , the specifics to the implementation of the system will be discussed . Along with the implementation specifics, the shortcomings of the designing phase and the alterations that were made and how the class diagram of the system was altered in order to suit the customers requirement will be discussed.

System Specification

The main objective of the system is to enable the site manager to control the procurement of goods that is necessary for their sites. Along with the site manager , there are altogether 5 stakeholders in the system.

- Site manager
- Procurement department officer
- Accounting officer
- Manager
- Supplier

The Site manager, Procurement officer, accounting officer and the Manager are internal to the company while the suppliers are external.

Site manager responsibilities

The site manager is mainly responsible for procuring the necessary goods that are required for the construction process mainly,

- Adding orders to the suppliers in the system(Which are placed by the procurement department considering whether it needs approval by the manager of not)
- Editing orders that are sent to the procurement department but not placed to the supplier
- Drafting orders to add later
- And logging deliveries

Procurement officers responsibilities

The procurement managers is responsible for handing most of the back office functions. Mainly their responsibilities are ,

- Adding sites to the system and allocating site managers to it
- Adding suppliers to the system and products relevant to the suppliers
- Creating requisition orders that does not need to be approved by the managers (Total order amt <= 100,000 and does not contain restricted items)
- Sending the orders that need approval by the managers to get approved

Accounting officers responsibilities

Accounting officers is mainly in charge of the accounting function and does not involve with other areas of the system.

- Cross tallying whether all the goods in the system are delivered and then paying the invoices that are submitted by the Supplier

Managers responsibilities

Manager monitors everything that is happening in the system . His responsibilities mainly involves

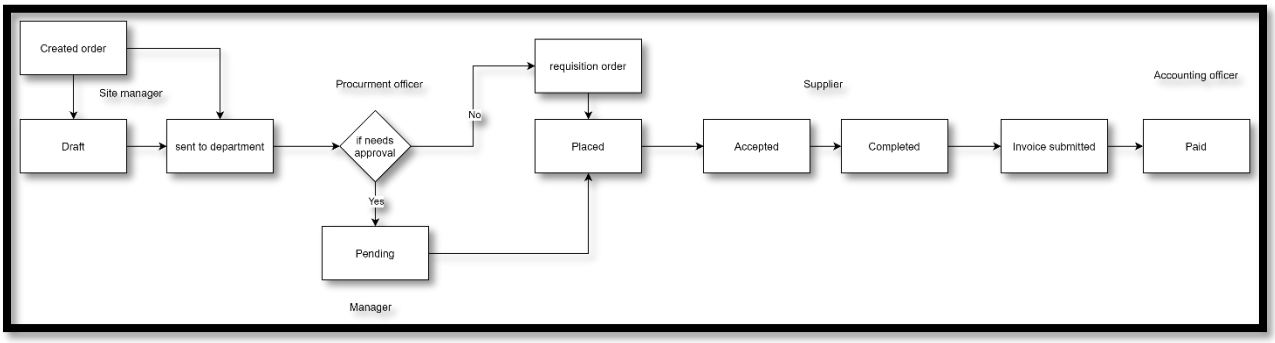
- Monitoring all the orders that are going through the system
- Accepting or declining the pending orders (if declined , give a message)

Suppliers responsibilities

Supplier is and external stakeholder to the company mainly their responsibilities are,

- Accepting orders that are placed by the company (or declining and giving a message)
- Creating deliveries
- Creating invoices after delivery completion

Considering the different stages that an order goes through in the system , a status was given to a specific order as follows



Order life cycle

- Order is created and either drafted or sent to the procurement department
- If the order doesn't contain restricted items and the total price is less than 100,000, the procurement officer can straight away make a requisition order to the supplier.
- If it does meet one of the above criteria, the manager must approve the order before it can be placed to the supplier
- After the order is placed, the supplier can either accept it or reject the order.
- If the order is accepted, the supplier has to complete the order and deliver the goods in the order either through a single delivery or multiple deliveries which adds up to the total requirement.
- After the order is completed, the Supplier can then add an invoice so that it can be paid.

Considering the above requirement, the following implementation decisions were made.

Implementation stack

Considering the distributed nature of the system , the following decisions were made in order to meet the customer requirements.

Preferred architecture : Restful API

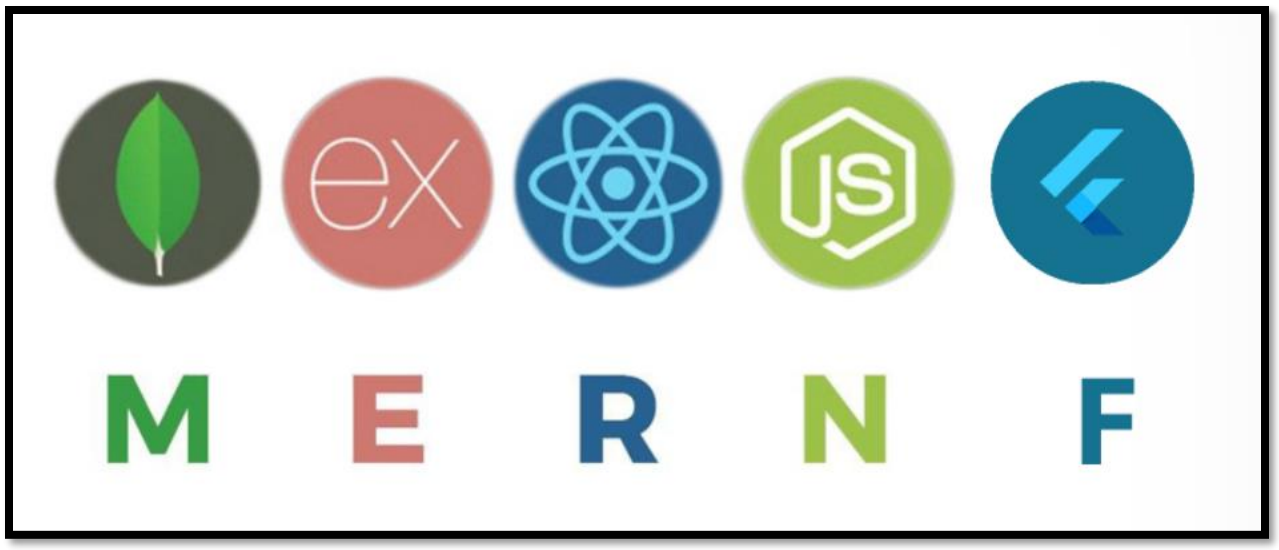
Since it was required for the site manager to access the system easily from anywhere , a flexible system which can run on mobile devices through which the manager can perform his responsibilities was required. But for handling the back office functions a web application was proposed so that it had the ability to run in any bowser

Frontend technology mobile: Flutter

Frontend technology webapp : React

Backend technology : Node JS with an express server

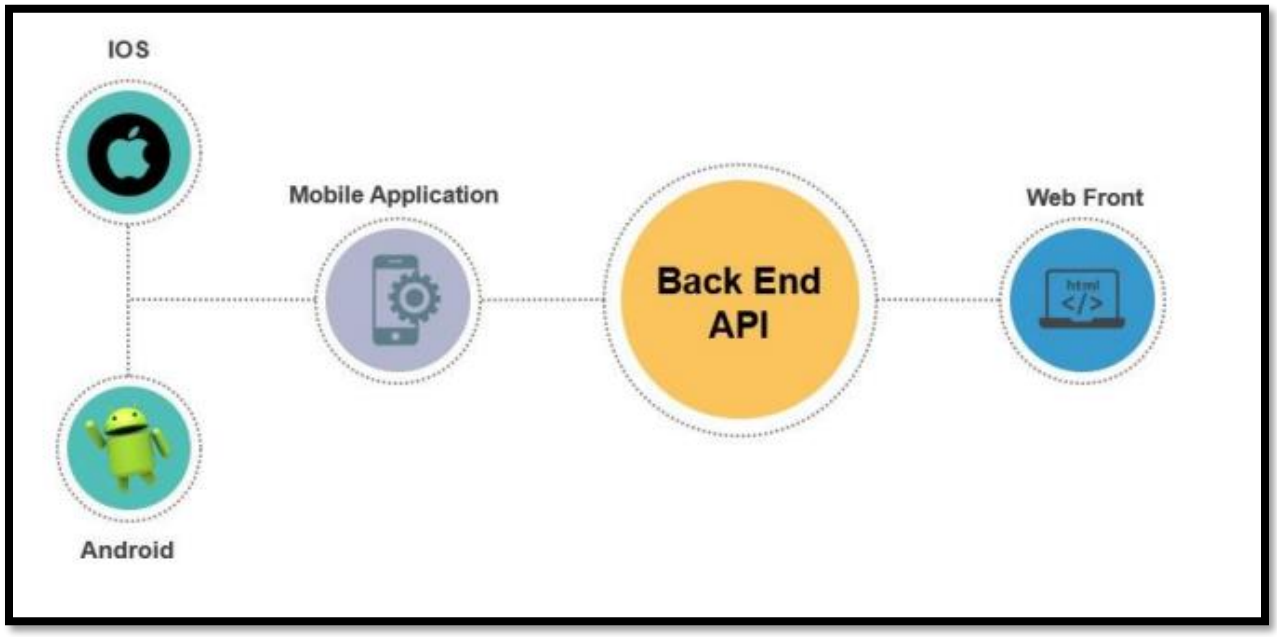
Database solution : Mongo DB



The main reason for choosing MERN stack along with flutter to implement the solution was that the relational database approach is highly scalable and the functional programming approach of JavaScript which both React and Node JS is based on allows flexible development unlike the Object oriented paradigm. Although programing principles such as inheritance is not recommended in functional programing languages , it compensates by giving the flexibility for us to implement exactly the functions that we require rather than writing a lot of boilerplate code.

Flutter was chosen as the framework to implement the mobile UI for its ability to provide fast phased development for both android and IOS using a single code base.

System architecture.

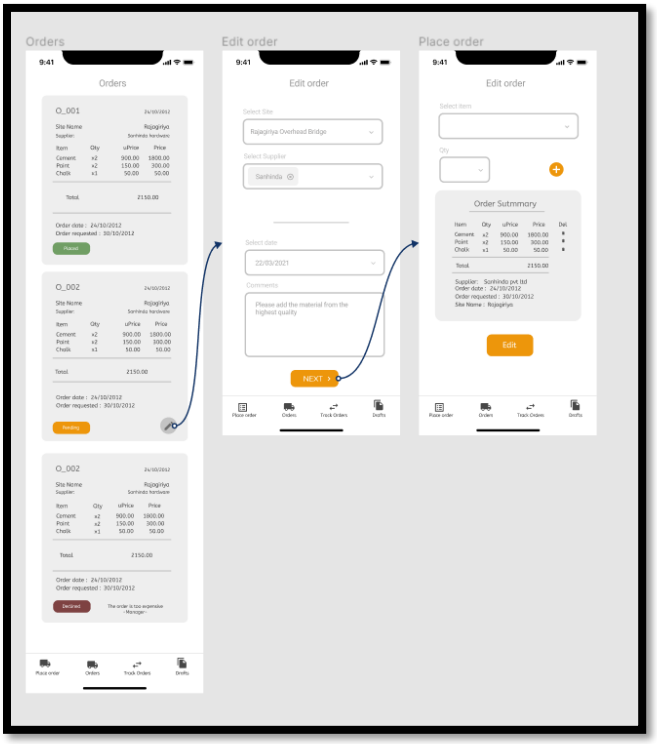


Design mockups

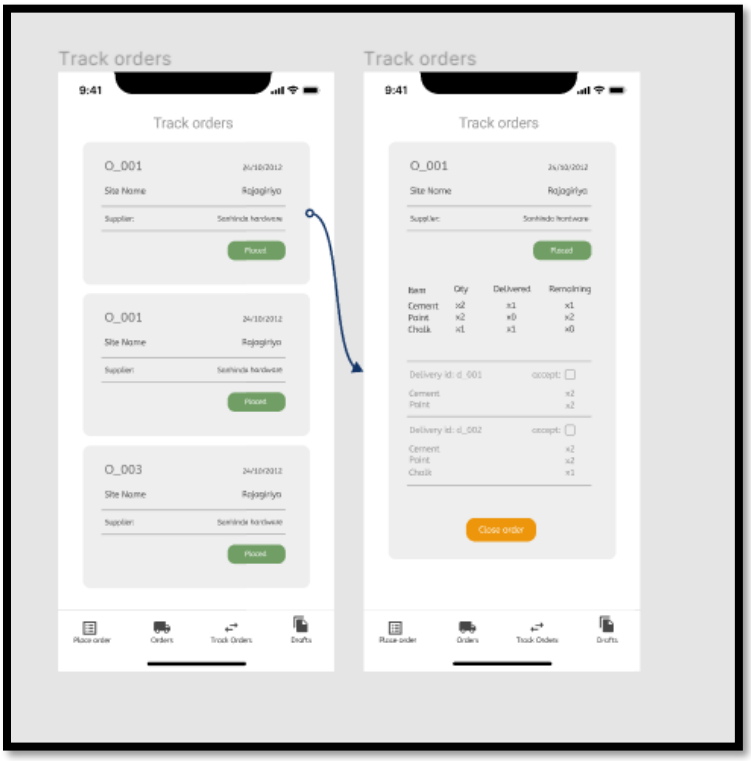
Site manager

Login / place order

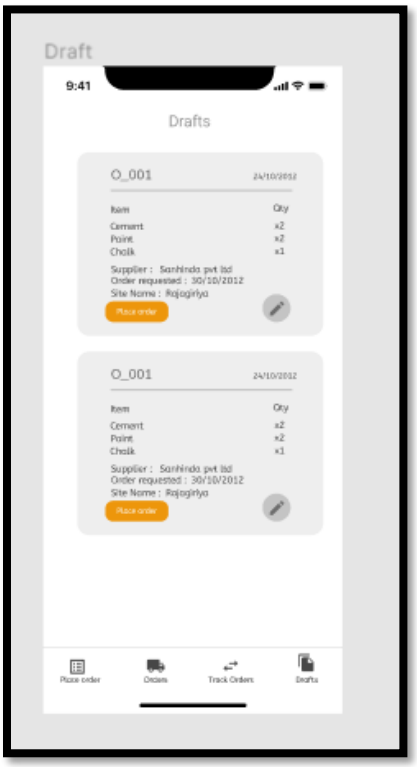
View order / edit order



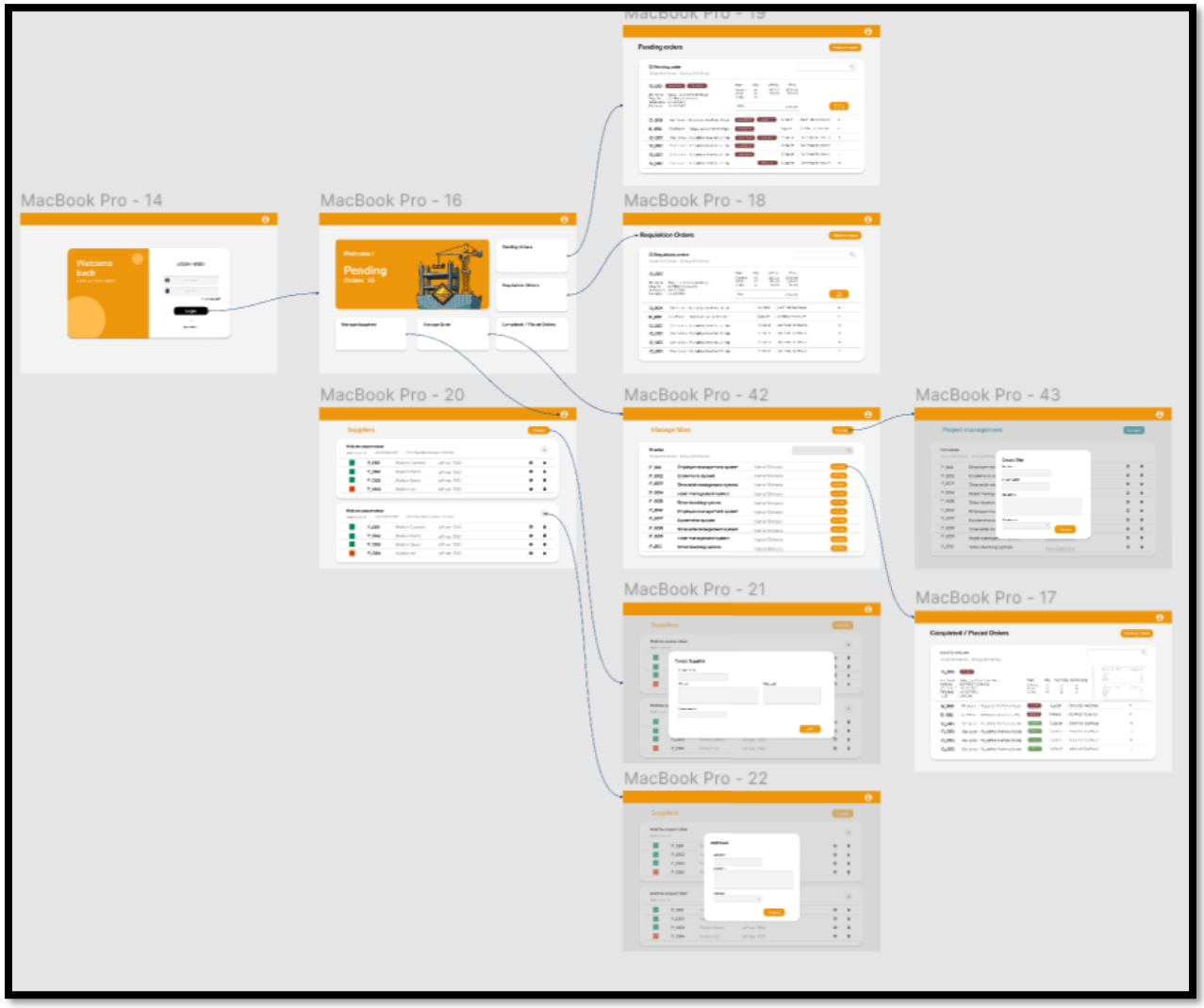
Track orders / approve deliveries



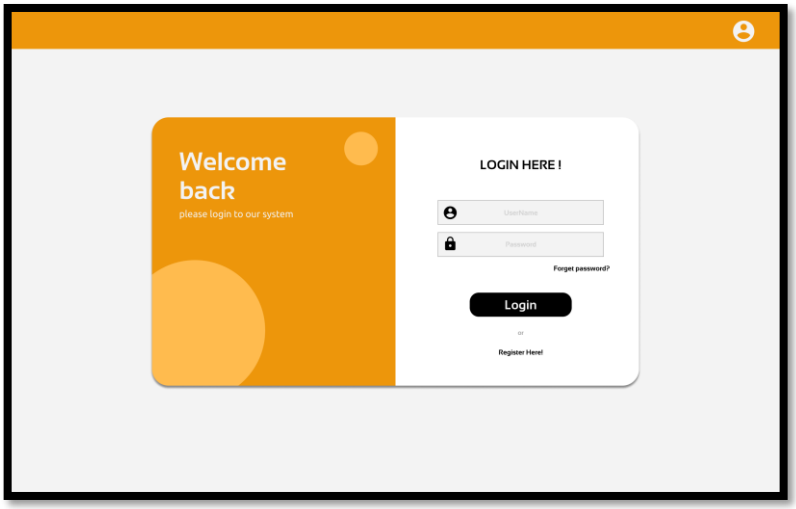
Draft deliveries



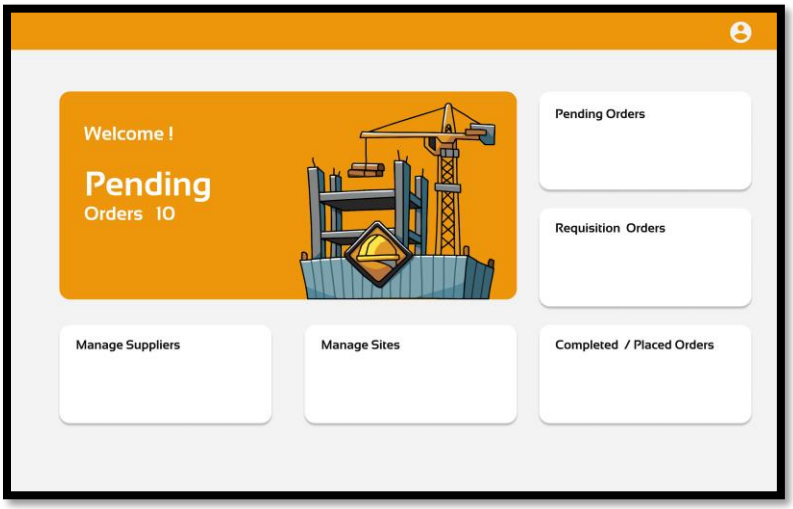
Site manager



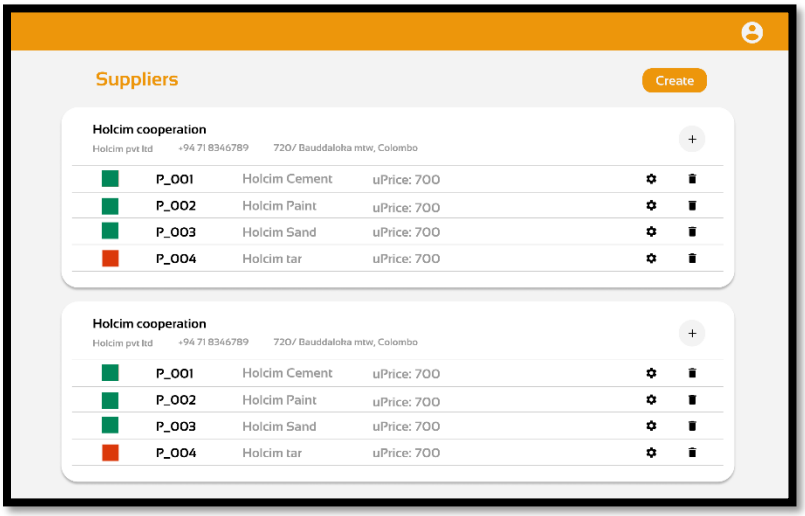
Login

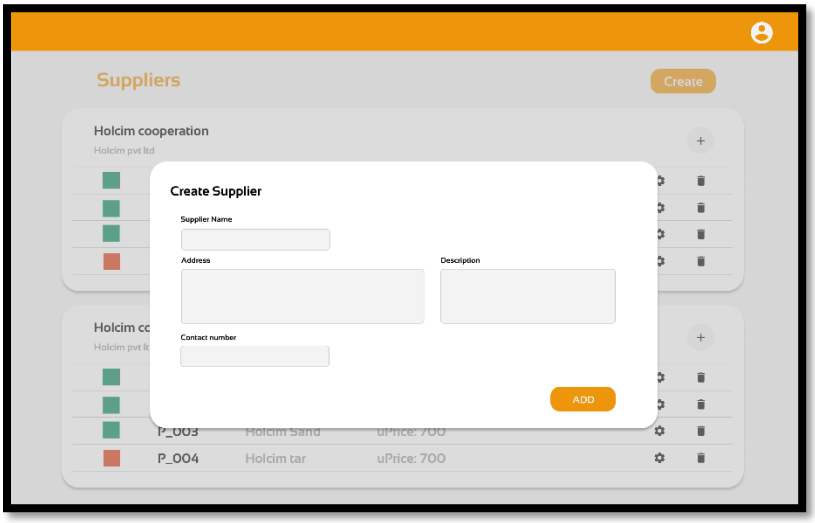


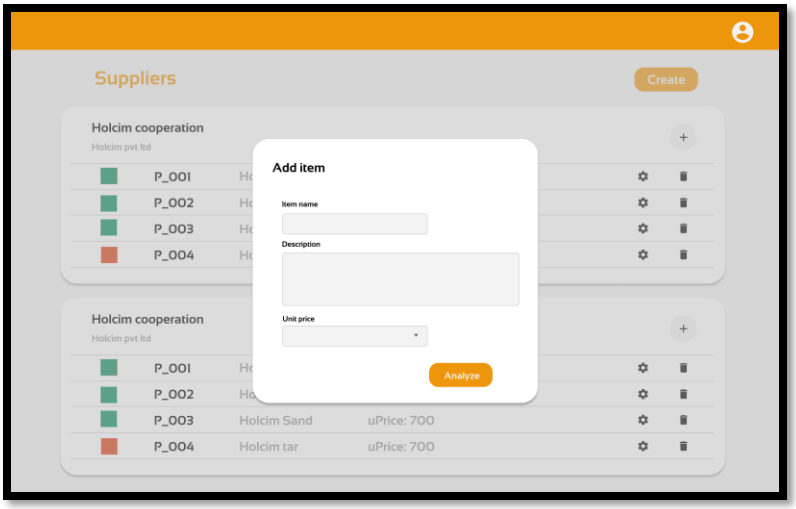
Dashboard



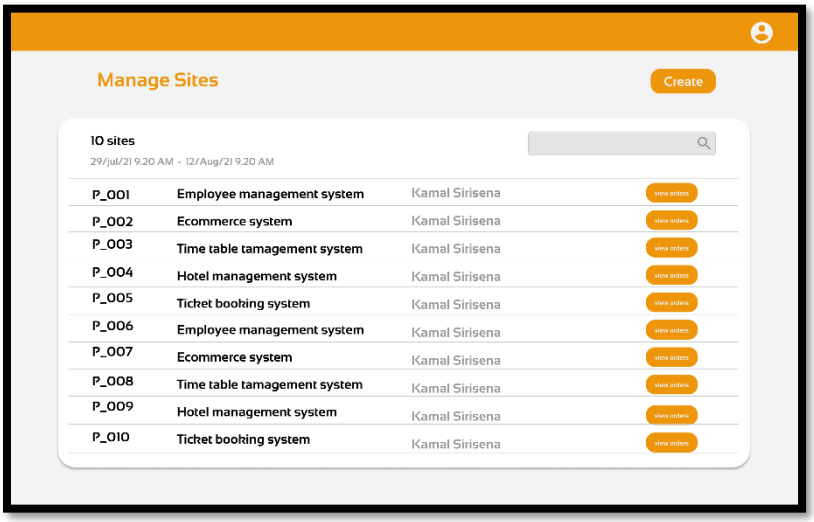
Manage suppliers

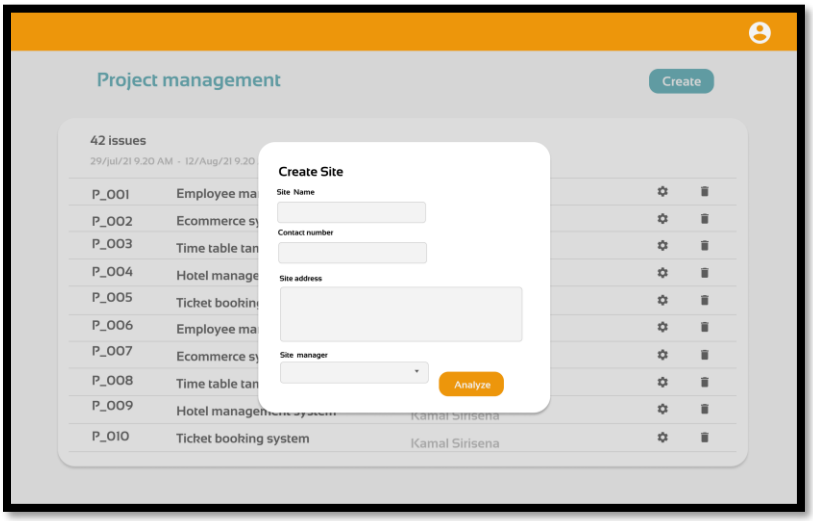


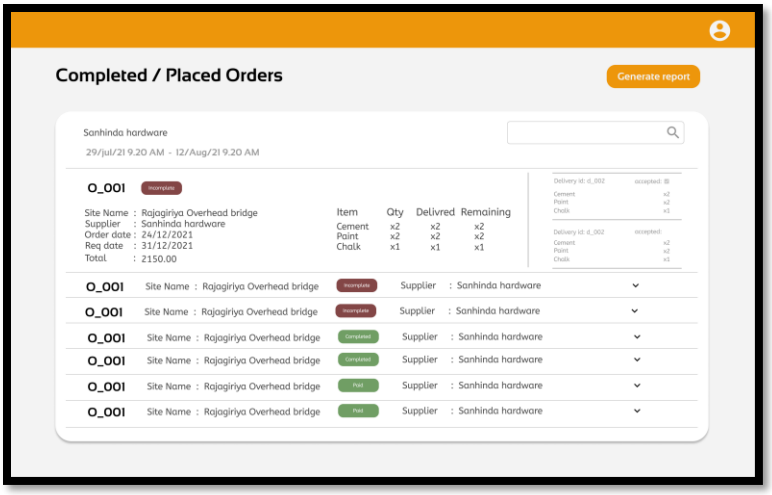




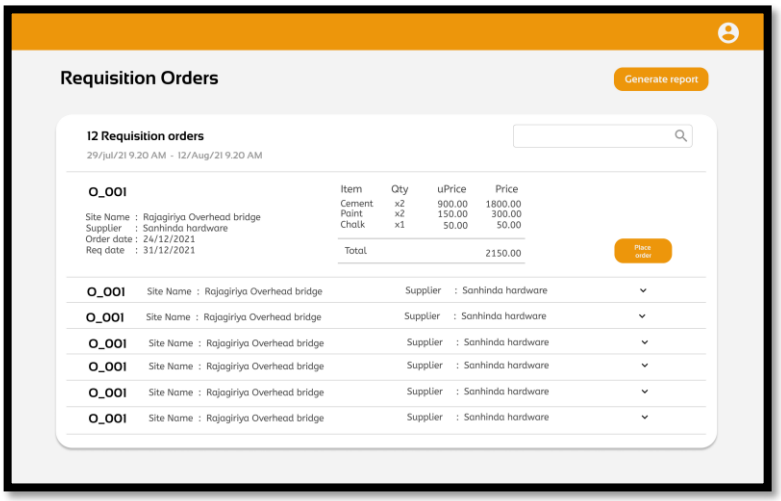
Manage sites



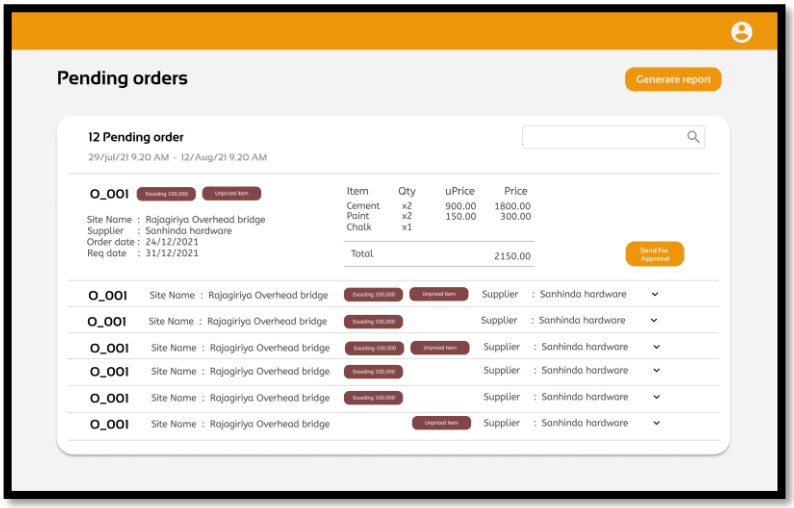




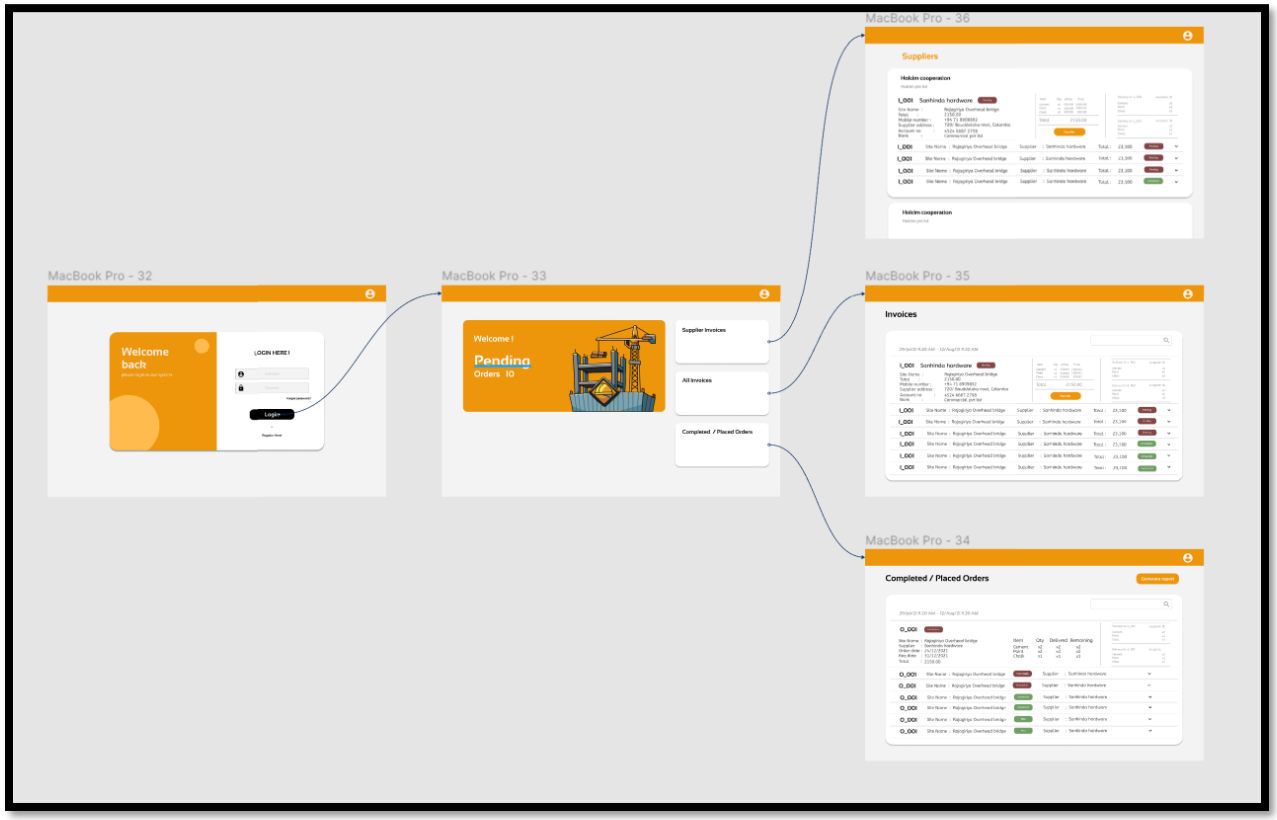
Add requisition orders



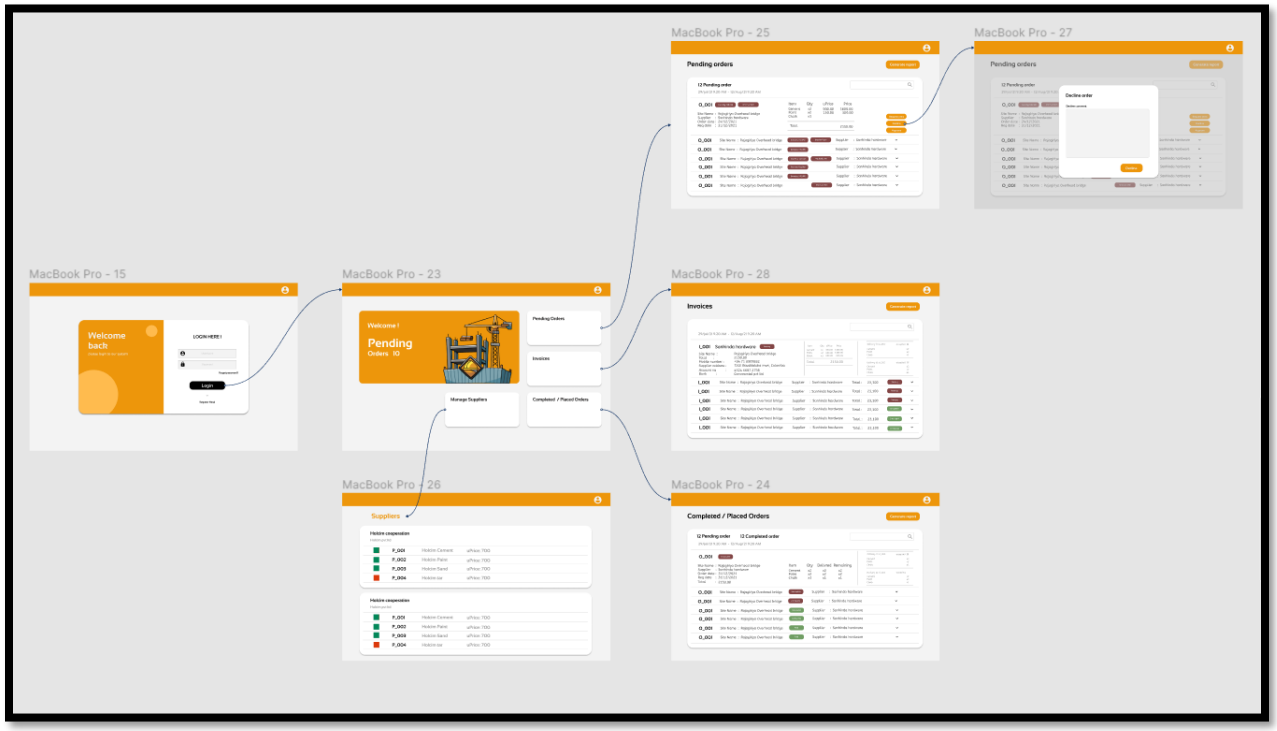
Send orders for approval



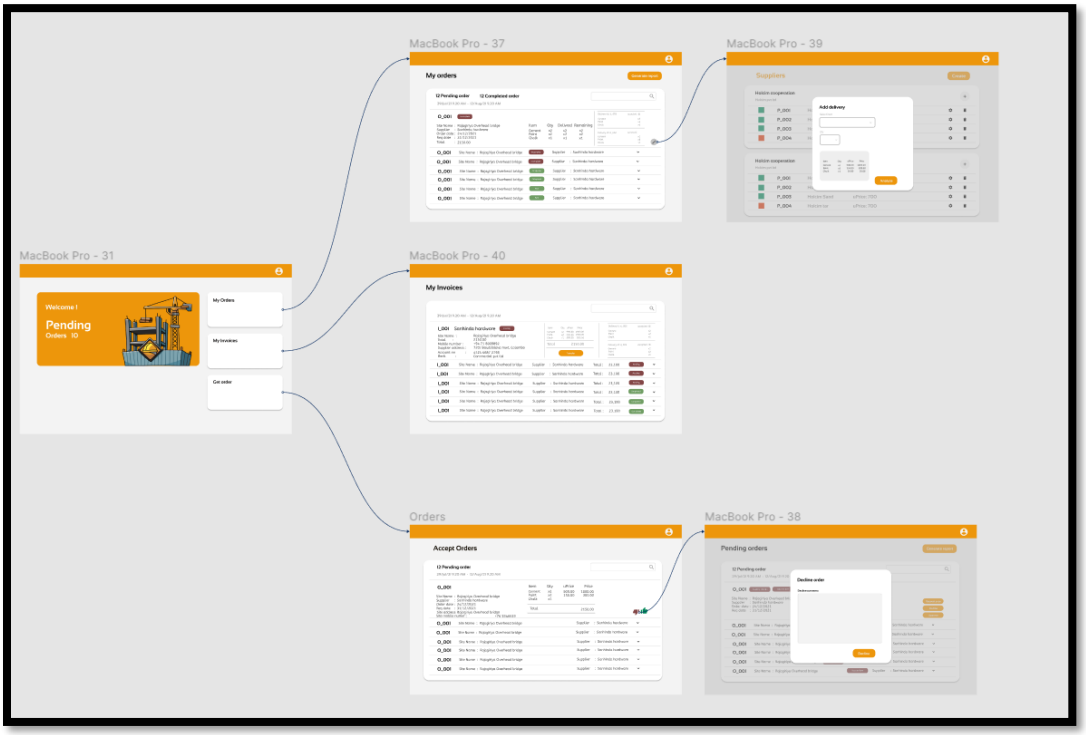
Accounting officer



Manager



Supplier



Visit the link for full figma mockups:
https://www.figma.com/file/DLt8z8KlrLy1uRxozjywMP/Untitled?node-id=0%3A1

Design alterations in UML

Several design changes were made after identifying short comings in the proposed system by the other group and alterations were done in order to meet the users requirements. In this section the design changes will be justified and a new design upon which the implementation was done will be proposed

Design changes in the class diagram

Taking to account the case study provided and the proposed class diagram for the system , the following shortcomings were identified:

- Requisition order and Order were identified as separate classes while a requisition order is a specialized case of a normal order where the order does not contain any restricted items and the total amount is less than 100,000 according to the client specification
- Product class captures the quantity as an attribute. But in essence , the Product class should only handle the attributes relevant to the product such as the unit price , product name and whether it is a restricted item and the quantity and the item from which the quantity was ordered should be in a separate class(orderProduct class in the altered class diagram) according to abstraction.
- The relationship between the orderProduct class and the order class is a whole part relationship where the orderProduct cannot exist without an order class(composition).
- Similarly a composition relationship should be there between orderProduct class and the Delivery class.
- Customer bank account information such as whether the account is a savings account or a current account is captured though a separate class called bank account, with 2 specialized classes name savings account and current account. Also things such as the interest rate and the OD limit is given as attributes. But these information are irrelevant for a procurement management system and private to the user. The only information that is needed to do the invoice payment is the account number , the branch of the account holder and their bank which can be captured at invoice submission (This also allows the suppliers to change the account which they need separate invoices to be deposited to)
- Supplier is also a stake holder in the system and should be able to perform functions such as add invoices , create deliveries , therefore a role name supplier should be in the system and should be authenticated and be able to perform functions such as login. So an interface should be implemented by the supplier and the generalized class of sitemanager , manager, procuemnt officer and accounting officer which is the employee . this interface should have the login method which should be implemented by the aforementioned 2 classes.
- Storage is mentioned as a separate class , but in the case study it is mentioned that the purpose of the system is to enhance the procurement process and allow the site manager to do just in time deliveries of goods.
- Department class and the specialized classes procurement department and accounting department was identified as separate classes which has an aggregation relationship with the employees were identified in the proposed system. But this information is out of the scope of a procurement management system and should be a separate employee management system.

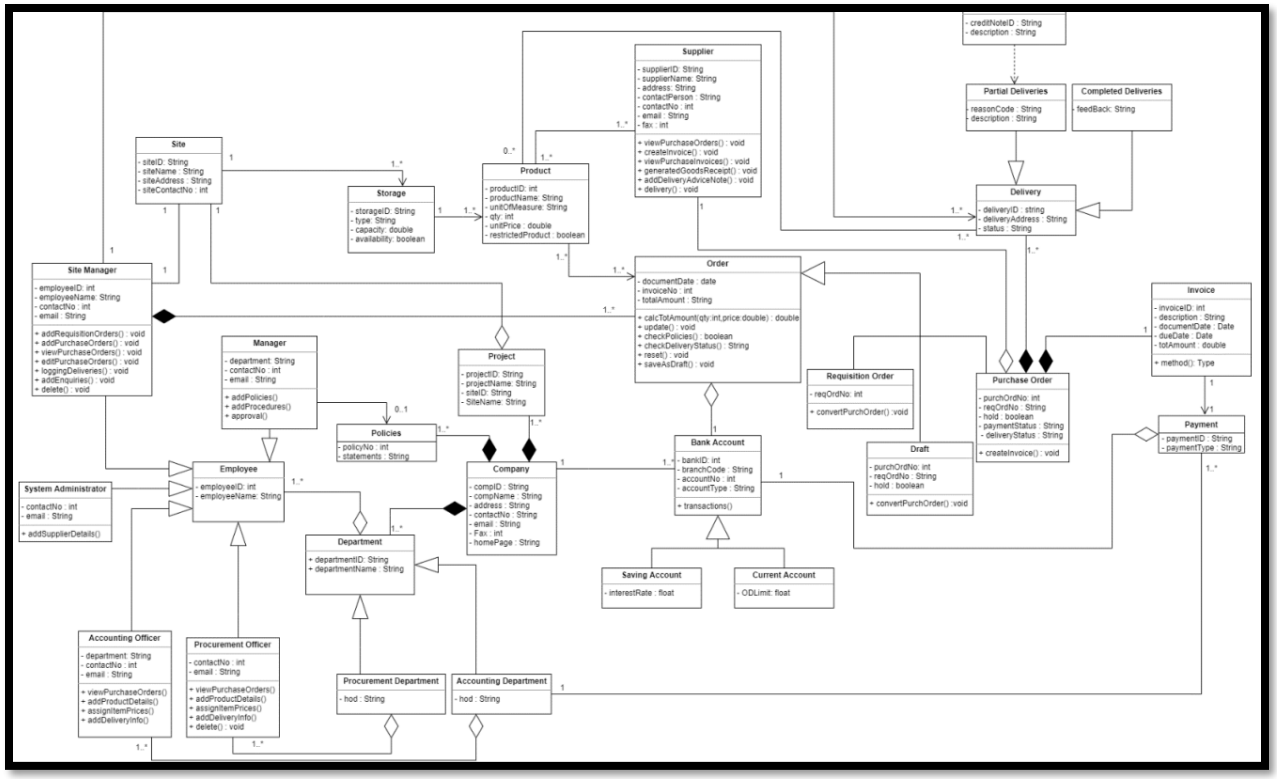


Figure 1: Proposed class diagram by the previous group

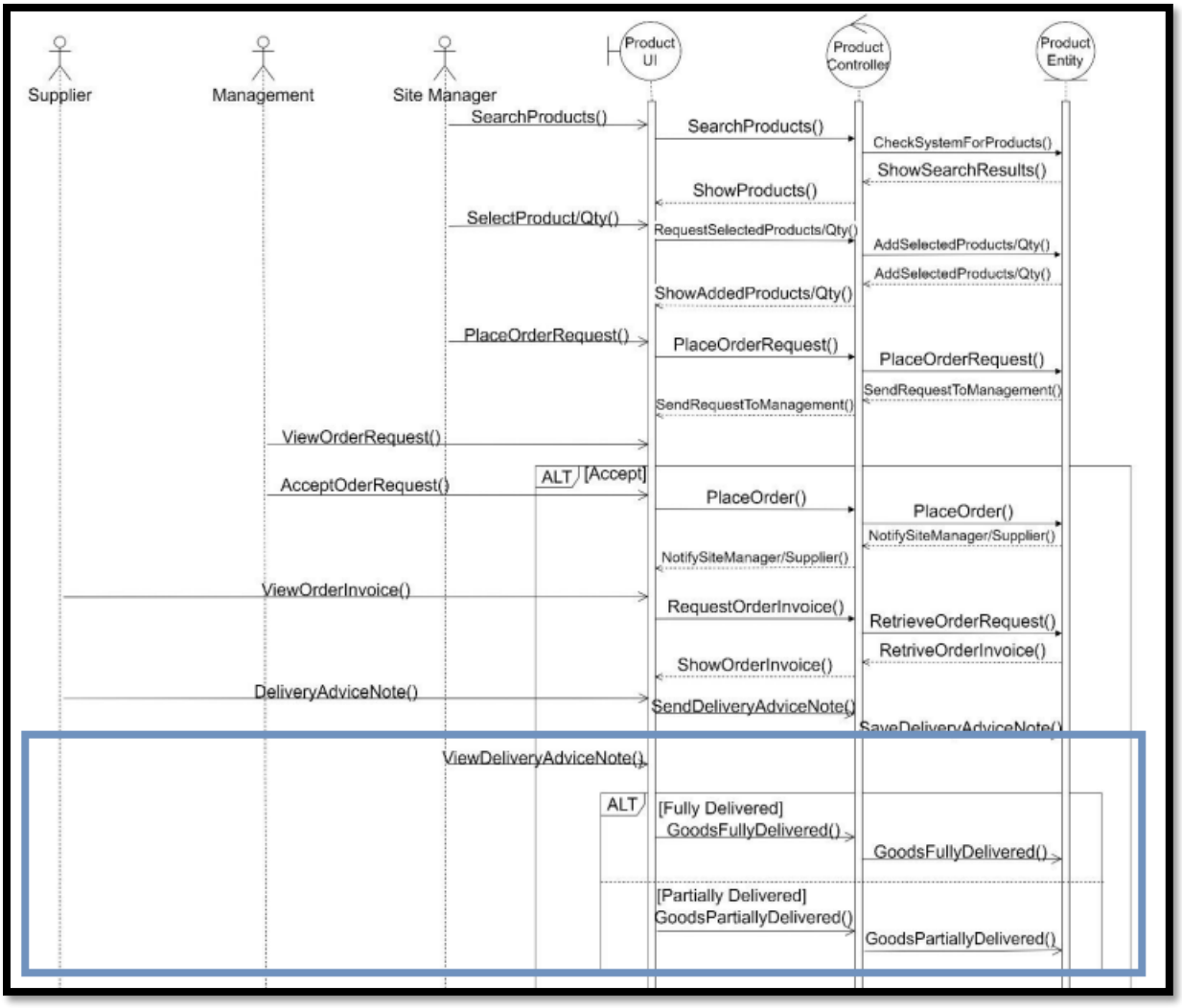
In the clients requirement specification it is required that the site manager can draft their order to add later. In order to achieve that a specialized class “Draft” was created out of the general class “Order” was created and a method was given to convert it to a “PurchaseOrder” . Another “RequisitionOrder” class is created which also has a convertToPurchaseOrder is created. This approach was taken to capture the different stages of the order life cycle. This leaves to chain object creation for each stage of the object life cycle. And looking from a document based NoSQL database standpoint, Separate collections each containing documents for each level of the order lifecycle has to be created which leaves with memory inefficiencies. Also in the product class an attribute named qty is mentioned, but a product doesn’t have a quantity , rather , an order has a quantity of products , so a method to capture that saying order product was implemented in the altered UML diagram.

Rather than this a different approach to tackle this will be proposed in the altered UML diagrams.

Since the system was proposed to be implemented using a JavaScript backend and a NoSQL database rather than relying on class level inheritance and abstraction which are not advised in functional programing languages, the UML diagram was altered focusing on function pointers , callback and other functional programing concepts in mind.

Design changes in the sequence diagram

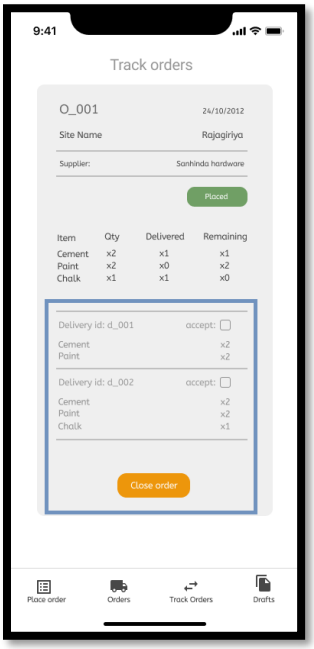
Considering the design flows discussed under the class diagram section and several user requirements such as accepting deliveries from the supplier by the site manager which were not captured in the design process, several alterations were made in the sequence diagrams that were proposed.



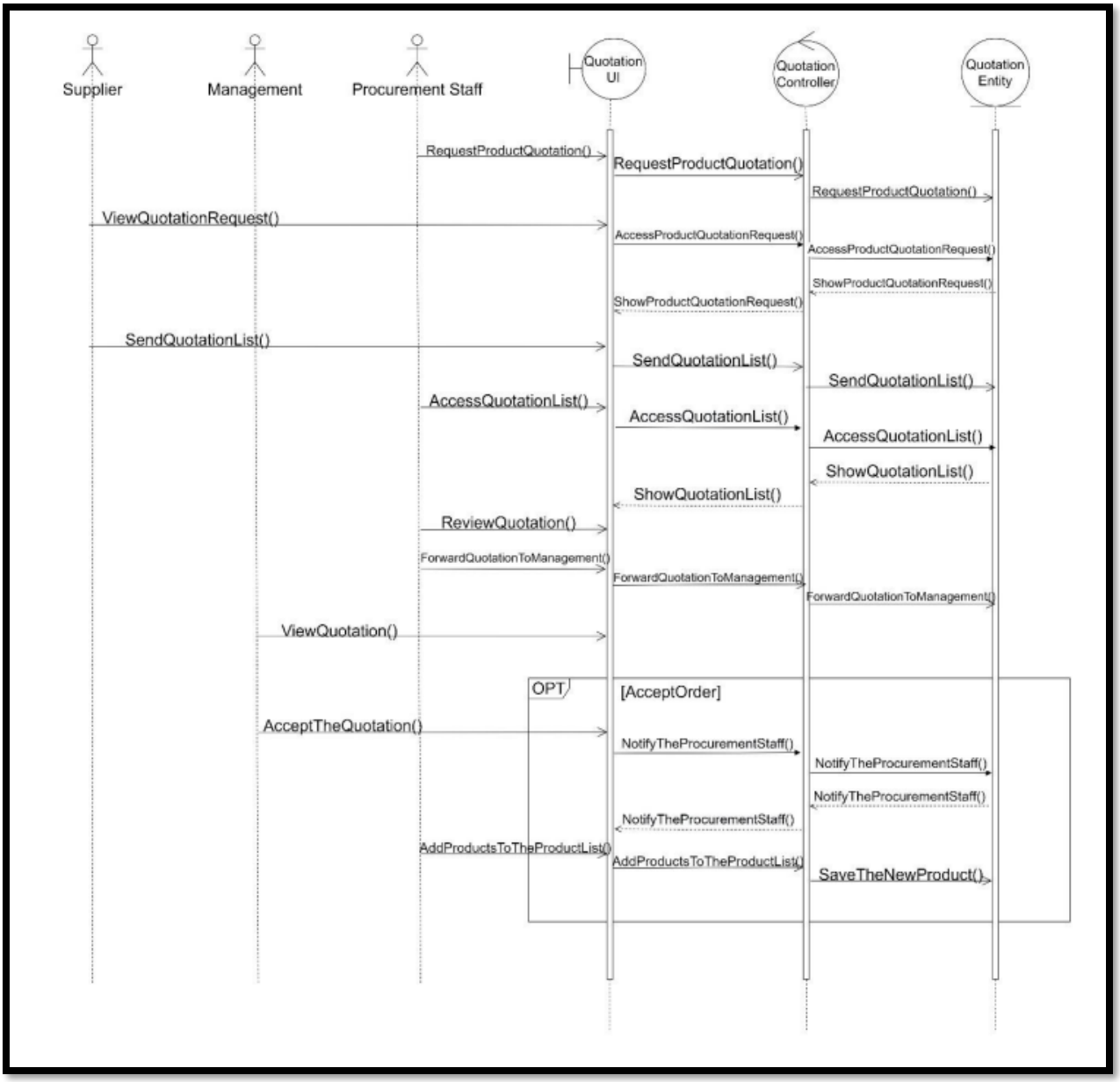
In the highlighted section of the proposed sequence diagram 1 , it is discussed how the supplier sends a delivery advice for the deliveries that they make in order to complete the order (An order can comprise multiple deliveries or all the goods that were ordered are delivered all at once). And based on whether it is a partial delivery or a full delivery (alt clause) the site manager sends whether it is a partial delivery or a total delivery.

But this approach was changed since it didn't capture the dynamic nature of the partial deliveries properly. Instead of this , when ever a delivery is logged in the system by the supplier by adding a delivery, A goods receipt will be generated in the system. There is a composition relationship (A whole part relationship in which the part cannot exist without the whole) between the Goodsreciept and the

delivery. The goods receipt has the **OrderReference** so that the association between the delivery and the order can be made easily. Apart from that the price of the delivery is also included. The site manager can view all the logged deliveries to the system and they can accept the deliveries. From the mobile UI.

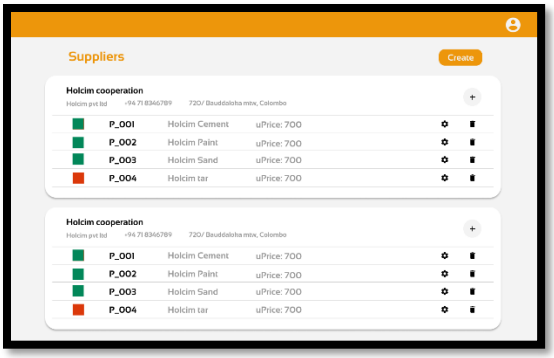


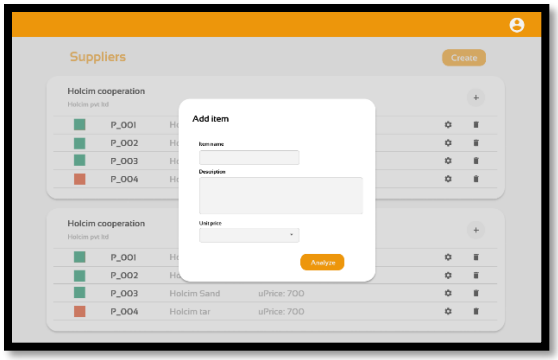
When all the deliveries are accepted, and deliveries match the ordered quantity (There is no remaining amount) the site manager can close the order.



In the sequence diagram 2 , the process of sending quotations from the supplier is given. But the case study, it specifies that pre agreed prices are in the system so that the accounting functionality is simplified. Therefore another approach which doesn't rely upon quotations is proposed which is much more efficient to handle the pre agreed prices was proposed.

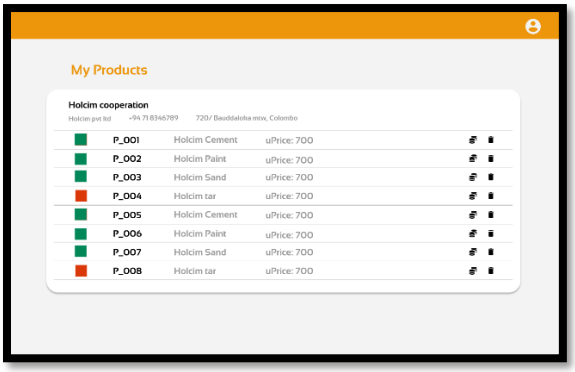
In this approach , the procurement officer adds the trusted suppliers to the system and adds products under each supplier that they might expect to purchase from the supplier. (This information should be added manually considering the existing system or by asking for a report from the suppliers about the products that they provide and their prices).

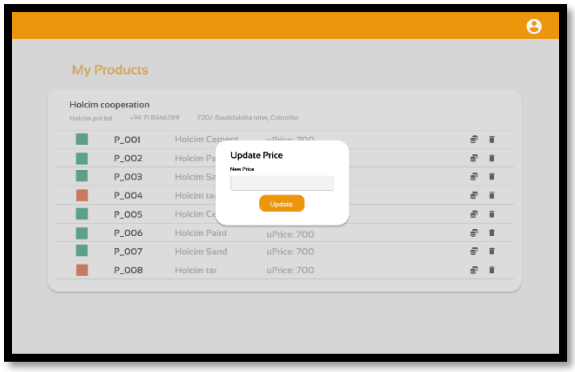




The procurement officer has the freedom to add the unit price if there is a pre agreed price is provided by the supplier in the manual report.

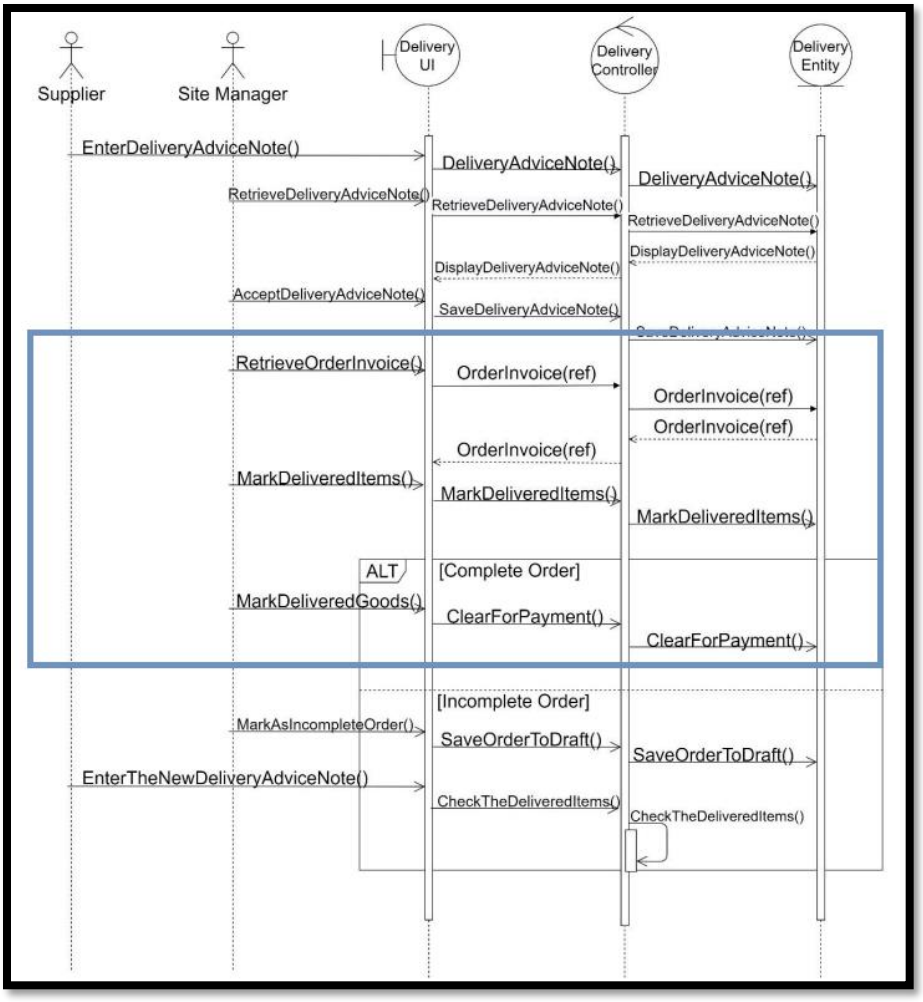
After adding the items to the trusted suppliers, the suppliers themselves can view the items under my products. If the price is not provided , they can provide a price. They are also able to update their prices for relavant items





The quotation approach that was proposed only allows the customer to add a unit price for products only once. But the price of items in the market is a volatile factor and the supplier should be able to constantly update their prices for products.

The approach which gives the supplier to provide their prices and update the prices in the system is a much more viable approach to tackle pre agreed prices in the system.

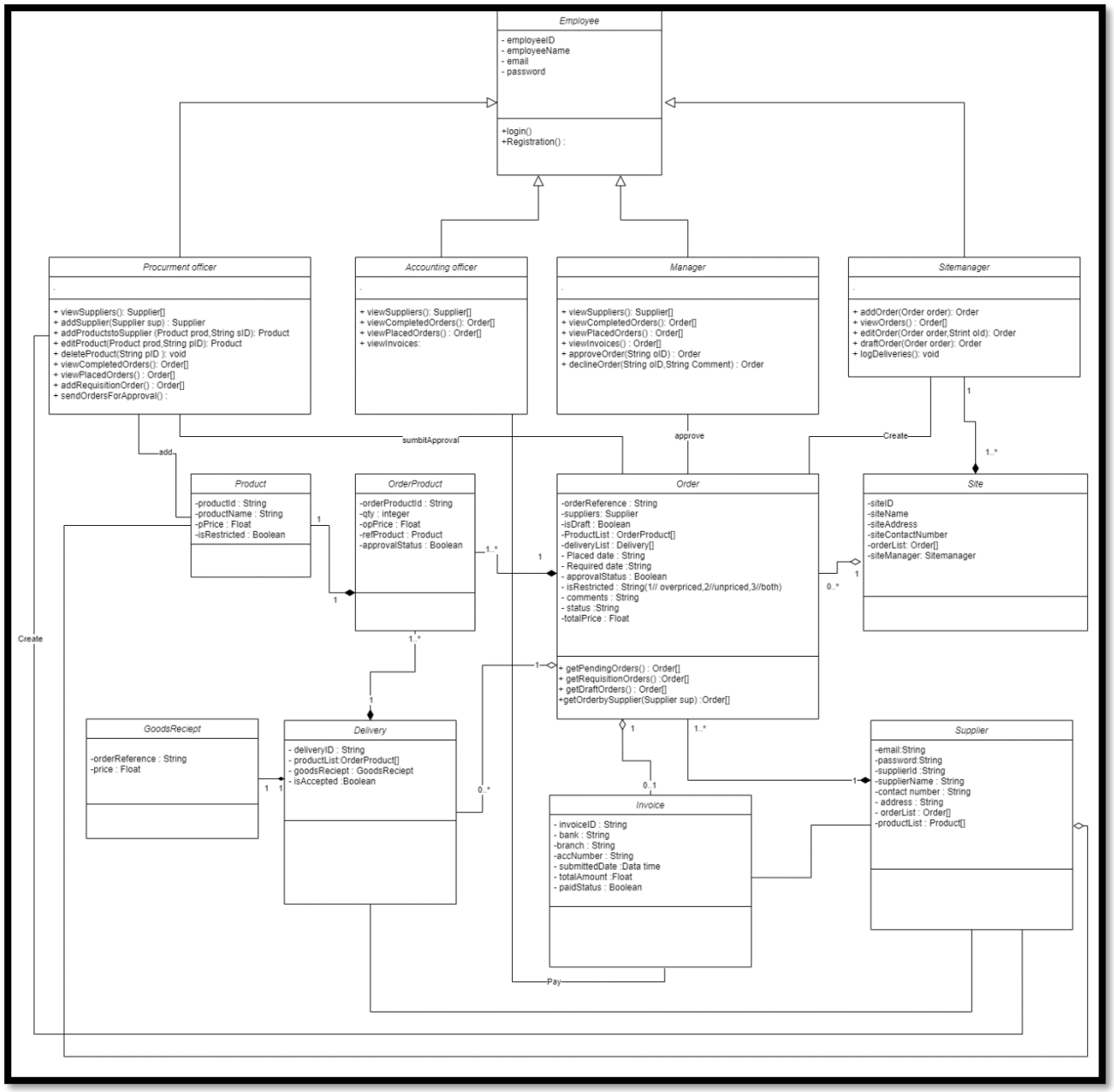


The 3rd sequence diagram breaks down the sequence of accepting the deliveries and the invoice functionality. In there , the Site manager retrieves all the invoices and marks the delivery of goods in the invoice . But the invoice management and accounting functionality falls under the umbrella of the accounting officer and should be handle by him rather than the site manager

In the altered approach, the supplier will add the deliveries to orders that they have accepted , the site manager will accept the deliveries that are associated with the order instead of marking it in the invoice. When all the requested items in the order are delivered , the site manager is able to close the order. The supplier is then able to add and invoice to that specific order, which goes to the accounting officer. The accounting officer tallies whether the information in the invoice and the order tallies together and pay the invoice. The payment mechanism is out of the scope of a procurement management system and might call an external API or done manually. Therefore that functionality doesn't need to be addressed.

Redesigned class diagram

Due to the reasons justified in the above section , the class diagram was altered. In this section the redesigned class diagram and how it handles the procurement system and the design flows in the previously proposed class diagram will be discussed



The main objective of the system is to provide the site manager with the ability to procure the necessary goods for the construction site just in time so that the there is no hinderance in the construction process. In order to achieve this and the other client requirements , the following approach was taken.

- The main class in the system is the order class.
- An order in the system goes through many stages in the order lifecycle(Explained earlier) such as draft , placed, pending etc.

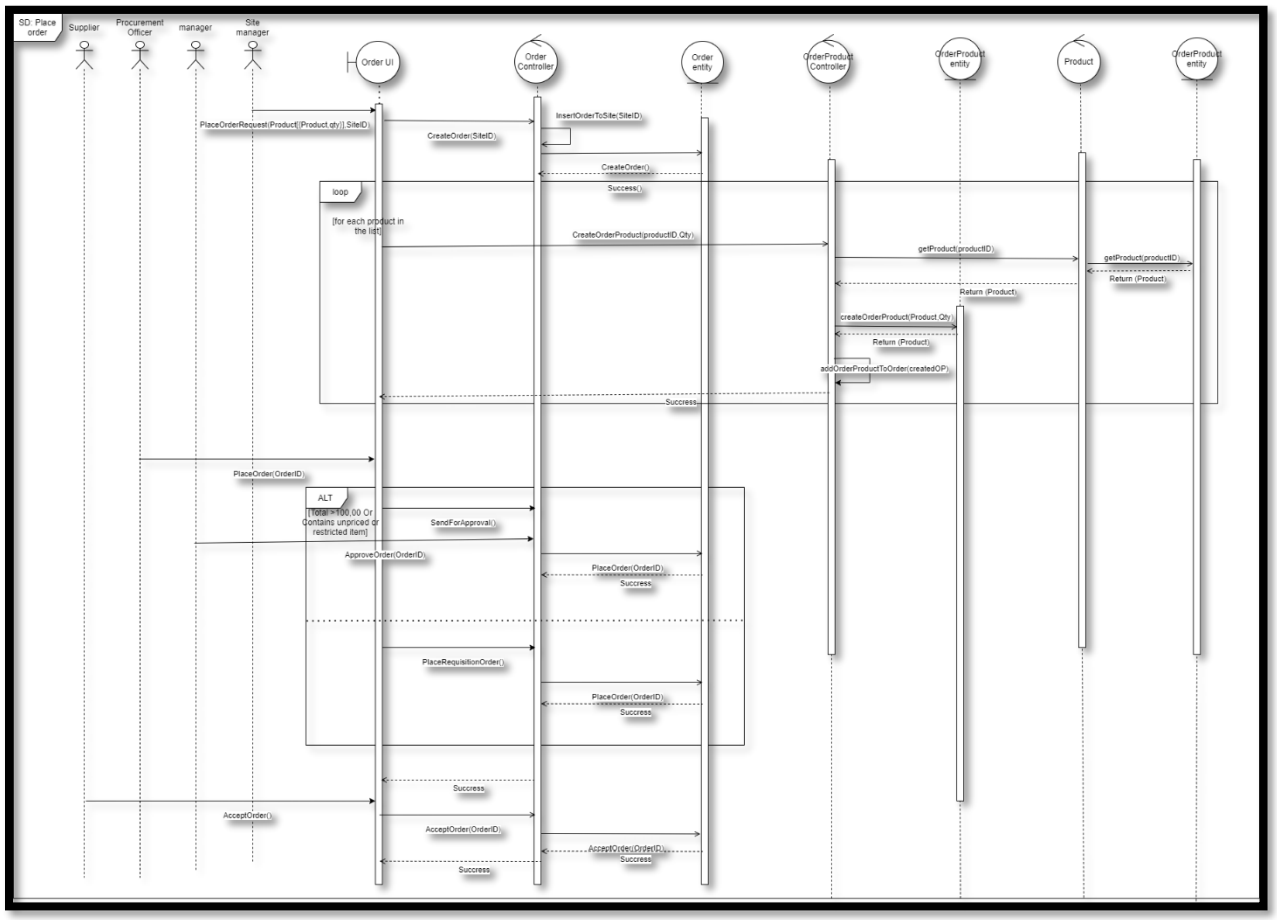
- A site and a site manager has a composition relationship. A site manager must be specified at the creation of a site.
- One site manager can have many sites.
- The site manager can place zero or many orders under the site.
- Therefore there is an aggregation relationship between the order class and the site class.
- This ensures that a list of orders are maintained under each site but when the site is deleted , the orders does not necessarily have to be deleted.
- The Order has a variable called approval status. This manages whether the order needs managers approval or not (this is automatically updated depending on the total order amount and whether there are restricted or unpriced items).
- The status variable carries the status of the order.
- An order contains a list of Order products
- There is a composition relationship between order and the order product class. A single order must contain one or many order products.
- Order product class contains a reference to a product and the quantity of the product.
- In the previously proposed class diagram , it was proposed to keep the quantity in the product class. But according to abstraction, only the product specific information such as the unit price and the product name should be in the product class.
- The quantity of items in each order must be taken into a separate class(order product in this case) .
    - An order for an example might take a form like this
    - O_01:
        - Cement x1
        - Paint x2
        - Chalk x3
    - In this case , 3 order product objects , each having references to cement , pain and chalk respectively must be created and added to the productList array of the order class. The order product object also contains the quantity of each item along with the reference to the product
        - Ex : Cement (ref) , x1 (qty)
- When an order is accepted by the supplier , the supplier can then add deliveries to the order list
- A delivery (Similar to the order) contains a list of order product items. This list of products contains the items that are delivered in that particular order (According to the clients specification , the supplier must be able to complete the order with a single delivery or a series of deliveries). Therefore a list of deliveries is maintained in the order class (Aggregation , since an order doesn't necessarily need to have deliveries)
- When a delivery is created , a goodsReciept is created along with it . This contains a reference to the order class and also the price of the delivered goods in that particular delivery.
- After all the deliveries are completed and accepted , the site manager can change the order status to completed , at which point an invoice can be created by the supplier.
- The invoice class handles the payment related activities for that given order.
- The supplier is able to provide information such as account number , bank , branch and other payment related details.(This allows he supplier to ask for a particular order to be credited to a given account rather than all the orders being credited to one provided account).

- An order can have only a single Invoice.
- This approach allows direct access for the supplier for information such as the site number and address and directly contact the site manager for deliveries instead of going through the procurement department.

Redesigned sequence diagrams

Following up on the justified reasons above under the previously proposed sequence diagram and how the functionalities of the system was addressed, the sequence diagrams were redesigned to fit the new approach. In this section , the sequence if the diagrams and how each functionality achieve the short comings of the previous sequence diagrams will be discussed.

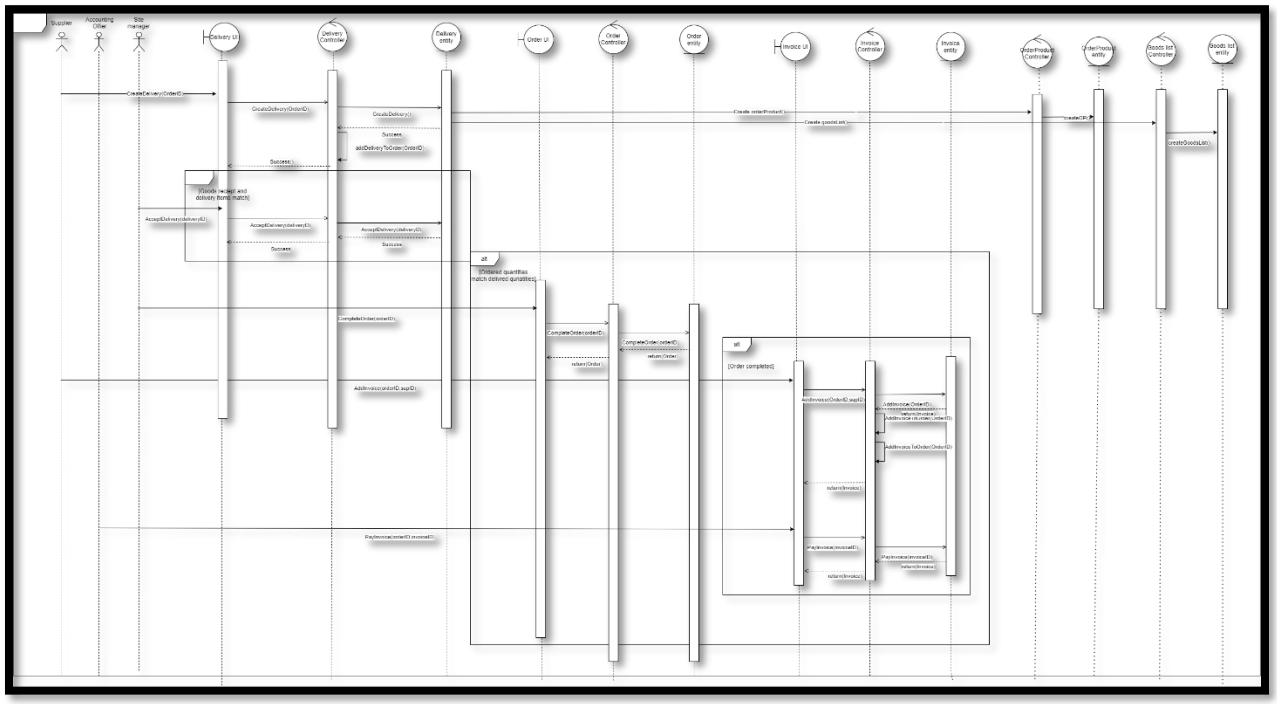
SD01: Place order



- When the site manager places an order, it is unshifted to an orderList in site while also being created.
- For each of the products and quantities passed in the order request , the create OrderProduct() method is called and the order is built.
- The created order products are added to the orderproduct list in the order.
- After that , based on whether there are any restricted products or if the price is more than 100,000. The approval of the manager must be taken before the order is placed to the supplier.

- If none of the above criteria is met, the procurement officer can make a requisition order without approval.
- After the order is placed , the supplier can accept the order.

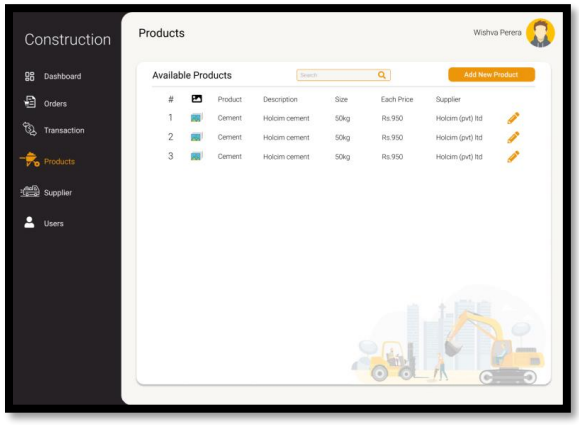
SD02:Complete order



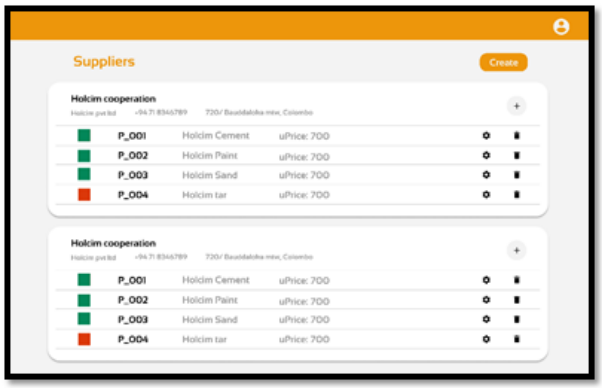
- After the order is accepted by the supplier that was suggested, the supplier can then deliveries for each order(partial or total).
- When a delivery is made, based on the products and the quantities of products in the delivery , a goods notice will be created carrying the reference of the order.
- After the delivery is made , the site manager can view deliveries under trackOrder, and accept the delivery comparing the goods receipt and the delivered items.
- After all the items in the order are delivered , the site manager can complete the order .
- After the order is completed , then the supplier can add an invoice to that particular order at which point it is both unshifted to the suppliers invoiceList and added to the order.
- After the invoices are added , the accounting officer will cross check the order again and pay the order.
- **This approach is much more suitable than the previously proposed approach since it handles partial deliveries better than the previous method. The site manager only has to accept deliveries when the supplier logs deliveries.**

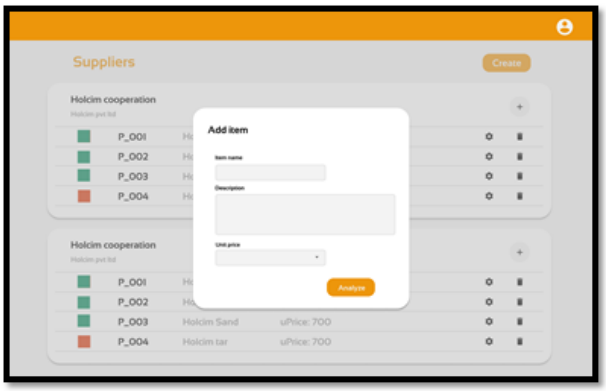
Design changes in the prototypes

Add products

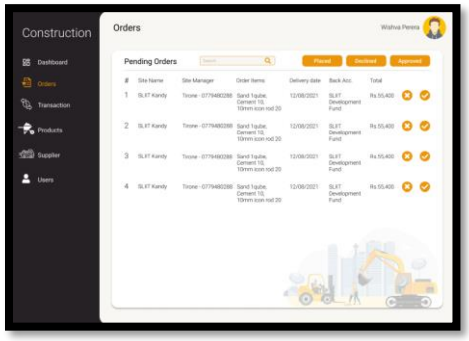


In the proposed UI to add products to the system , All products of all the suppliers are displayed in one table . The delete functionality for products is also not provided. But instead of this approach, in the redesigned interfaces the procurement officer must first add suppliers to the system and under each supplier they can add items. In this way it is much easier to filter the different products that each supplier provides and notice changes in their pricing.

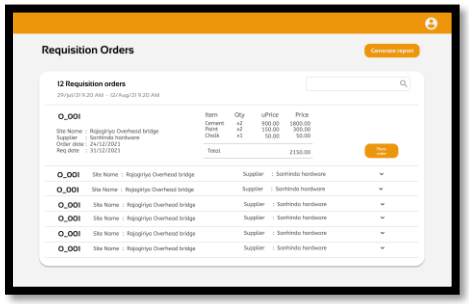




Pending orders



In the proposed interface, the pending orders , that does not need approval are shown(requisition orders that can be made by the procurement officer are shown).Rather than this interface , a detailed interface is shown in the altered interfaces to easily have requisition orders.



Implementation walkthrough

The application was done using JavaScript and the document based data solution mongo DB. The frontend code was written using the react library and backend code was implemented using NodeJs with the help of express package to handle the http requests. Several other libraries were also used to help with the implementation

Main frontend dependencies

- Redux: State management library
- Axios : to handle http requests easily
- Bootstrap : styling

Main backend dependencies

- Bcrypt : password encryption
- Cors : cross origin request handling
- Jsonwebtoken : authorization and session management
- Mongoose : mongoDB

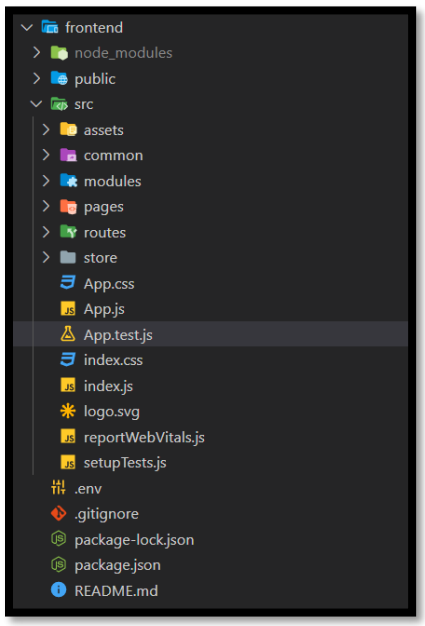
Dev dependencies

- Jest : Unit / integration testing
- Supertest : Testing

Frontend implementation

Maintaining a proper folder structure

It was made sure that a proper folder structure was maintained within the frontend folder such that modularity can be easily ensured



- src / routes : contains all the routes to the pages . The react router logic is abstracted out from the rest of the code and implemented here
- src/store : All the logic related to redux state management is handled here from the actions , the reducers and the apis
- src/pages : This folder contains all the pages that are in the system
- src/common : All the common components such as navigation bar is implemented within the folder.
- src/modules : All reusable modules such as login forms and order tables are implemented here so that they can be reused throughout the pages

Using Redux for state management

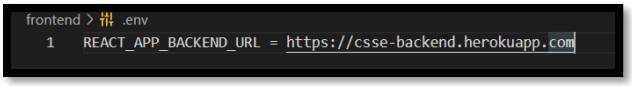
Redux is a state management library that helps with the process of managing internal state of components and reduces the number of API calls that are sent to the backend. This also means that the responsiveness of the application is greatly increased since redux is managing state in a local store and does not need to fetch data from the back end each time.

```
import { ACTION_TYPES } from "../actions/ProductActions";

const initialState = {
  productList: [],
  singleProduct: null,
};

export const productReducer = (state = initialState, action) => {
  switch (action.type) {
    case ACTION_TYPES.ADD_PRODUCT:
      return {
        ...state,
        productList: [...state.projectlist, action.payload],
      };
    case ACTION_TYPES.GET_SINGLE_PRODUCT:
      return {
        ...state,
        singleProduct: action.payload,
      };
    case ACTION_TYPES.GET_ALL_PRODUCTS:
      return {
        ...state,
        productList: [...action.payload],
      };
    case ACTION_TYPES.DELETE_PRODUCT:
      return {
        ...state,
        productList: state.productList.filter((x) => x._id !== action.payload),
      };
    case ACTION_TYPES.UPDATE_PRODUCT:
      return {
        ...state,
        productList: state.productList.map((x) =>
          x._id === action.payload._id ? action.payload : x
        ),
      };
    default:
```

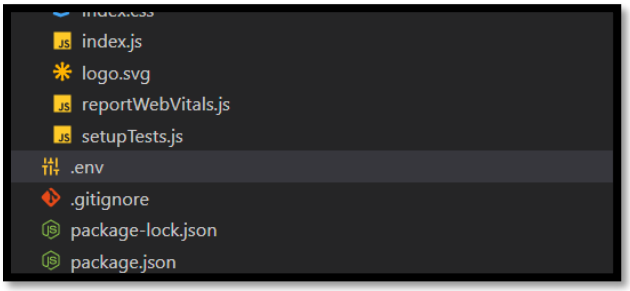
```
import ProductAPI from "../apis/ProductAPI";

export const ACTION_TYPES = {
  ADD_PRODUCT: "ADD_PRODUCT",
  GET_SINGLE_PRODUCT: "GET_SINGLE_PRODUCT",
  GET_ALL_PRODUCTS: "GET_ALL_PRODUCTS",
  DELETE_PRODUCT: "DELETE_PRODUCT",
  UPDATE_PRODUCT: "UPDATE_PRODUCT",
};

export const addProduct = (data, OnSuccess, OnFailure) => (dispatch) => {
  ProductAPI.apiProduct()
    .addProductByProcurementOfficer(data)
    .then((response) => {
      dispatch({
        type: ACTION_TYPES.ADD_PRODUCT,
        payload: response.data,
      });
      OnSuccess();
    })
    .catch(() => {
      OnSuccess();
    });
};

export const fetchProduct = (productId) => (dispatch) => {
  ProductAPI.apiProduct()
    .getProduct(productId)
    .then((response) => {
      dispatch({
        type: ACTION_TYPES.GET_SINGLE_PRODUCT,
        payload: response.data,
      });
    })
    .catch(() => {});
};
```

Usage of environment variables

Environment variables were used to handle external variables such as the backend deployment URL which changes from the development environment to a staging environment



```
frontend > .env
1   REACT_APP_BACKEND_URL = https://csse-backend.herokuapp.com
```



Modularizing and abstracting out the commonly used code

Using the capabilities of the react libraries it was made sure that modular , reusable code was written as much as possible and the components received props through which they . In the following example , since the pending order table was used in multiple user interfaces , it was modularized into a separate component and props were passed to it and depending on the props passed to it , the component was rendered differently .

```
export default class PendingOrderTablePage extends Component {
  render() {
    return (
      <div>
        <div className="container">
          <div className="row mt-5">
            <div className="col-md-12">
              <h1>Pending Orders</h1>
            </div>
            <div className="col-md-12 mt-2">
              <PendingOrderTable orderList={this.state.orderlist}/>
            </div>
          </div>
        </div>
      </div>
    );
  }
}
```

Also the common logics such as calling the backend end point was abstracted out into sperate components. According to the **“Single responsibility”** principle , this class only handles the responsibility of calling the api.

```
import axios from "axios";
import authHeader from "./authHeader";

const baseUrl = process.env.REACT_APP_BACKEND_URL;

const config = {
    headers: authHeader(),
  };

const authSupplierApi = {
    authSupplier() {
        return {
            fetchPlacedOrdersForSupplier: (supplierId) => axios.get(`${baseUrl}/api/supplier/${supplierId}/pla
            fetchAcceptedorCompletedOrdersSupplier: (supplierId) => axios.get(`${baseUrl}/api/supplier/${suppl
            fetchAllSuppliersList: () => axios.get(`${baseUrl}/api/supplier/all`),
            getSupplierDetails: () => axios.get(`${baseUrl}/api/supplier`,config),
            register: (newSupplier) =>
                axios.post(`${baseUrl}/api/supplier/register`, newSupplier),
            login: (loginSupplier) => axios.post(`${baseUrl}/api/supplier/login`, loginSupplier),
        };
    },
};

export default authSupplierApi;
```

Using comments and properly structuring the code.

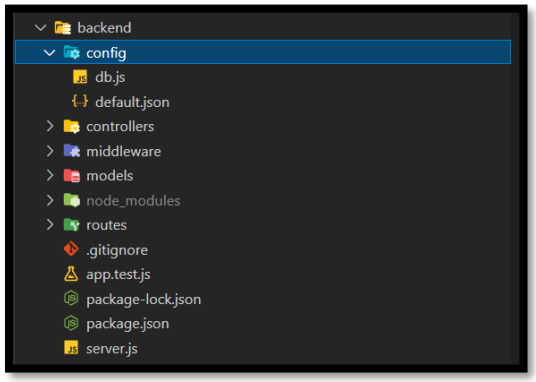
Comments were used to help with properly structuring the code and for better readability

```
<Router>

  {/* supplier routes */}
  <Route path="/addsupplier" exact component={AddSupplierPage} />
  <Route path="/supplierregister" exact component={AddSupplierPage} />
  <Route path="/supplierlogin" exact component={LoginSupplierPage} />
  <Route path="/supplier/myProducts" exact component={SupplierMyProducts} />
  <Route path="/supplier/myinvoice" exact component={SupplierInvoiceTablePage}
  <Route path="/supplier/acceptorders" exact component={AcceptOrdersPage} />
  <Route path="/supplier/myorders" exact component={MyOrdersPage} />

  {/* manager routes */}
  <Route path="/managerregister" exact component={RegisterManagerPage} />
  <Route path="/managerlogin" exact component={LoginManagerPage} />
  <Route path="/manager/invoice" exact component={InvoiceTablePage} />
  <Route path="/manager/completedorplacedorders" exact component={ManagerCompl
  <Route path="/manager/pendingorders" exact component={PendingOrderTablePage}
  <Route path="/ManagerViewSupplierPage" exact component={ManagerViewSupplierP

  {/* accounting officer routes */}
  <Route path="/accountingofficerregister" exact component={RegisterAccounting
  <Route path="/accountingofficelogin" exact component={loginAccountingOfficer
  <Route path="/accountingofficerDashboard" exact component={AccountingOfficer
  <Route path="/accountingofficer/invoice" exact component={AccountingOfficerI
  <Route path="/accountingofficer/completedorplacedorders" exact component={Ac

  {/* procurement officer route */}
```

Backend implementation

Maintaining a proper folder structure

It was made sure that a proper folder structure was maintained within the backend folder such that modularity can be easily ensured



- controller : Contains the controller classes
- routes : Externally accessible routes
- models : mongoose schema models

Usage of JWT session tokens

Jason web tokens were used to handle user authentication and session management . When the user logs in , based on the user role(“Supplier”,”Manager”, ect). a session token will be created in the frontend and stored in the local storage of the browser , this helps to manage the session from him after the login till he logs out and deletes the token. Also , some methods can be called by only authorized users . Therefore the appropriate token should be passed through the request header.

Exception handling and usage of promises

Using the capabilities of react , it was ensured that the exceptions were properly handled and the asynchronous capabilities of java script and promises(**.then()** , **.catch()**) were properly used. Also the status codes and descriptive error messages were given in the responses.

```
//add Product By Procurement Officer
const addProductByProcurementOfficer = async (req, res) => {
  const { ProductName, pPrice, isRestricted, supplierId } = req.body;

  let productPrice = 0;
  if(pPrice){
    productPrice = pPrice
  }

  try {
    //create a product instance
    const product = new Product({
      ProductName,
      pPrice:productPrice,
      isRestricted,
      deleteStatus: false,
    });

    //save product to the database
    await product
      .save()
      .then(async (createdProduct) => {
        const supplier = await Supplier.findById(supplierId);
        supplier.productList.unshift(createdProduct);
        await supplier.save();
        res.json(createdProduct);
      })
      .catch((err) => res.status(400).json("Error: " + err));
  } catch (err) {
    //Something wrong with the server
    console.log(err.message);
    return res.status(500).send("Server Error");
  }
};
```

Testing modular code.

API end point testing were done in order to ensure that the endpoints worked as intended. The jest package was used to run the testing, In here several assertions were run to ensure that the endpoints were running properly . Both negative and positive assertions were added.

```
// IT19132310 - Lasal Hettiarachchi
describe("Testing the Site  API", () => {
  const site = {
    id: "61571f3b0a910a199d125e12",
    siteName: "Site 1",
    siteAddress: "Colombo",
  };

  it("GET/Site -> get an array of sites", async () => {
    const response = await axios.get(
      "https://csse-backend.herokuapp.com/api/site"
    );
    expect(response.status).toBe(200);
  });
  it("GET/Site/id -> get a specific site by id", async () => {
    const response = await axios.get(
      "https://csse-backend.herokuapp.com/api/site/61571f3b0a910a199d125e12"
    );
    expect(response.status).toBe(200);
    expect(response.data._id).toBe(site.id);
    expect(response.data.siteName).toBe(site.siteName);
    expect(response.data.siteAddress).toBe(site.siteAddress);
  });
  it("POST/Site/ -> add a site", async () => {
    const response = await axios.post(
      "https://csse-backend.herokuapp.com/api/site",
      {
        siteName: "Site 2",
        siteAddress: "Colombo",
        siteContactNumber: "0779142664",
        siteManager: {
          _id: "61571d99d6bfb9e562e40d95",
        },
      }
    );
    expect(response.status).toBe(200);
    expect(response.data.siteName).toBe("Site 2");
    expect(response.data.siteAddress).toBe("Colombo");
    expect(response.data.siteContactNumber).toBe("0779142664");
  });
});
```

```
// IT19139036 - Senura Jayadeva
describe("Testing the Product  API", () => {
  const product = {
    id: "61571dc0d6bfb9e562e40d99",
    ProductName: "Product 1",
    pPrice: 2500,
    isRestricted: false,
  };
  it("GET/Product -> get an array of products", async () => {
    const response = await axios.get(
      "https://csse-backend.herokuapp.com/api/product"
    );
    expect(response.status).toBe(200);
  });
  it("GET/Product/id -> get a specific product by id", async () => {
    const response = await axios.get(
      "https://csse-backend.herokuapp.com/api/product/61571dc0d6bfb9e562e40d99"
    );
    expect(response.status).toBe(200);
    expect(response.data._id).toBe(product.id);
    expect(response.data.ProductName).toBe(product.ProductName);
    expect(response.data.pPrice).toBe(product.pPrice);
    expect(response.data.isRestricted).toBe(product.isRestricted);
  });
  it("POST/Product/ -> add a product", async () => {
    const response = await axios.post(
      "https://csse-backend.herokuapp.com/api/product",
      {
        ProductName: "Product 3",
        pPrice: 3000,
        isRestricted: false,
        supplierId: "61571d48d6bfb9e562e40d8d",
      }
    );
    expect(response.status).toBe(200);

    expect(response.data.ProductName).toBe("Product 3");
    expect(response.data.pPrice).toBe(3000);
    expect(response.data.isRestricted).toBe(false);
  });
});
```

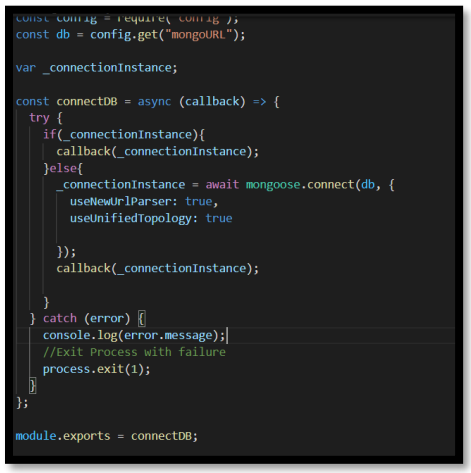
```
// IT19146898 - Ayodya Fernando
describe("Testing the Order API", () => {
  const product = {
    id: "61571dc0d6bfb9e562e40d99",
    ProductName: "Product 1",
    pPrice: 2500,
    isRestricted: false,
  };
  it("GET/Order -> get an array of orders", async () => {
    const response = await axios.get(
      "https://csse-backend.herokuapp.com/api/order"
    );
    expect(response.status).toBe(200);
  });
  it("POST/Order/sitemanager/site/:siteId -> add an Order", async () => {
    const response = await axios.post(
      "https://csse-backend.herokuapp.com/api/order/sitemanager/site/61571f3b0a910a199d125e12",
      {
        placedDate: "2021-09-29",
        requiredDate: "2021-10-02",
        supplier: {
          _id: "61571d48d6bfb9e562e40d8d",
        },
      }
    );
    expect(response.status).toBe(200);
  });
});
```

```
// IT19128036 - Dilei Palliyaguruge
describe("Testing the Manager  API", () => {
  const managerLog = {
    email: "senurajayadeva@gmail.com",
    password: "senura123",
  };
  const managerReg = {
    email: "senurajayadeva2@gmail.com",
    password: "senura123",
    name: "Senura Jayadeva",
  };
  it("POST/Manager/login-> login the manager", async () => {
    const response = await axios.post(
      "https://csse-backend.herokuapp.com/api/manager/login",
      {
        email: managerLog.email,
        password: managerLog.password,
      }
    );
    expect(response.status).toBe(200);
  });
  it("POST/Manager/register-> login the manager", async () => {
    const response = await axios.post(
      "https://csse-backend.herokuapp.com/api/manager/register",
      {
        email: managerReg.email,
        password: managerReg.password,
        name: managerReg.name,
      }
    );
    expect(response.status).toBe(200);
  });

  it("POST/supplier/all-> list of suppliers", async () => {
    const response = await axios.get(
      "https://csse-backend.herokuapp.com/api/supplier/all"
    );
    expect(response.status).toBe(200);
  });
});
```

Implementing design patterns

Since javascript is a procedural language , and programming concepts such as inheritance is ill advised , it is hard to properly implement traditional design patterns designed to cater object oriented programing languages. But for better solution of code , Some design patterns such as singleton was implemented with a javascript flavor

Singleton pattern

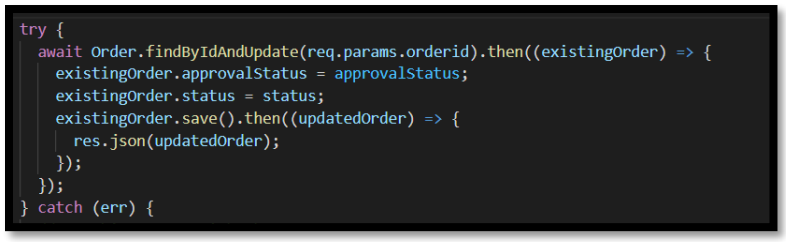


```
const config = require('config');
const db = config.get("mongoURL");

var _connectionInstance;

const connectDB = async (callback) => {
  try {
    if(_connectionInstance){
      callback(_connectionInstance);
    }else{
      _connectionInstance = await mongoose.connect(db, {
        useNewUrlParser: true,
        useUnifiedTopology: true

      });
      callback(_connectionInstance);

    }
  } catch (error) {
    console.log(error.message);
    //Exit Process with failure
    process.exit(1);
  }
};

module.exports = connectDB;
```

The database connection should be ideally a single object . Singleton patten ensures that only a single object of the database is created and for each subsequent creation of the database, the existing object will be returned.

Control flow patter.

Since JavaScript is an asynchronous programing language , callbacks are used to ensure the sequence of code that is running , But this can lead to callback hell problem when multiple nested levels of callbacks occur. In order to avoid this callback hell , control flow pattern and JavaScript promises are used .



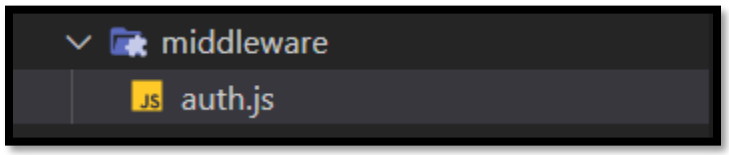
```
try {
  await Order.findByIdAndUpdate(req.params.orderid).then((existingOrder) => {
    existingOrder.approvalStatus = approvalStatus;
    existingOrder.status = status;
    existingOrder.save().then((updatedOrder) => {
      res.json(updatedOrder);
    });
  });
} catch (err) {
```

In the above code , the database query method findByIdAndUpdate returns a promise. If that process is successful , then the code inside the then block will be implemented .Inside the then block there is another database function which also returns a promise.

Middleware patten

“When an incoming request or data needs to be augmented/modified before it actually is consumed by the target object, a series of intermediary functions can be designed. The middleware pattern helps us implement such a pipeline of functions easily.”

In the above mentioned system , a middleware name auth.js is running and all the requests are running through that pipeline to authenticate the user.




```
const jwt = require("jsonwebtoken");
const config = require("config");

module.exports = function (req, res, next) {
  //Get the token from the header
  const token = req.header("x-auth-token");

  //Check if no token
  if (!token) {
    return res.status(401).json({ msg: "No token, authorization denied" });
  }

  //Verify token

  try {
    //valid token

    //decoding token through jwt verify method
    const decoded = jwt.verify(token, config.get("jwtSecret"));

    req.user = decoded.user;

    next();
  } catch (err) {
    //There is a token, but its not valid
    res.status(400).json({ msg: "Token is not valid" });
  }
};
```