Key Design Issues

1. Data Persistence

Introduction

In our design, we have identified three models to be stored in persistence storage: the Topics, Discussion and Transaction models. The Topics model contains the thematic topics the administrator wishes the forum to have. The Discussion model contains the forum posts and its associated comments. The Transaction model contains all information regarding past transactions that were extracted from open-source government data. The data is persistent as users are able to extract the data even after the webpage has been closed and restarted.

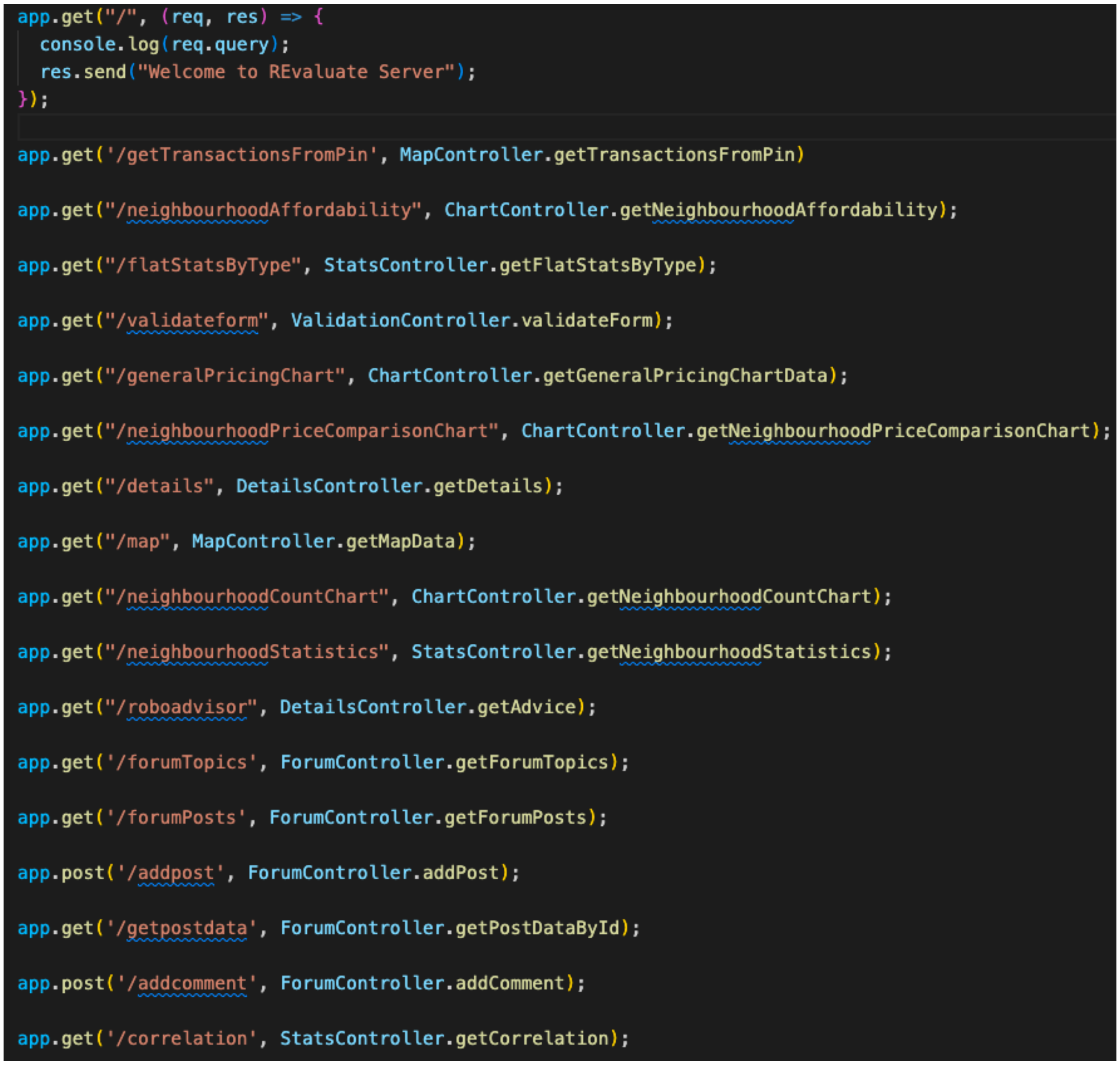
Design Issue

We have decided to use MongoDB Atlas. MongoDB is a document-oriented, noSQL database that stores data in documents similar to JSON. Due to the large data volume of our persistent data, MongoDB is suitable for handling our dataset due to its high scalability and fault-tolerant design. However, extracting and manipulating data from MongoDB Atlas can be complex as it requires knowledge of the database schema, it forms a design issue.

Design Pattern (Facade Pattern)

To abstract the complicated extraction from MongoDB, we have used the Facade design pattern to provide a layer of abstraction between the Views (client-side) and the database with our Controllers. This means that when the client makes an API call to retrieve data, the process is abstracted by a function within the Controller, reducing the need for the client to know the detailed logic behind the data extraction and manipulation.

These are the list of RESTful APIs developed and we see each API is handled by a controller function:



```
app.get("/", (req, res) => {
  console.log(req.query);
  res.send("Welcome to REvaluate Server");
});

app.get('/getTransactionsFromPin', MapController.getTransactionsFromPin)

app.get("/neighbourhoodAffordability", ChartController.getNeighbourhoodAffordability);

app.get("/flatStatsByType", StatsController.getFlatStatsByType);

app.get("/validateform", ValidationController.validateForm);

app.get("/generalPricingChart", ChartController.getGeneralPricingChartData);

app.get("/neighbourhoodPriceComparisonChart", ChartController.getNeighbourhoodPriceComparisonChart);

app.get("/details", DetailsController.getDetails);

app.get("/map", MapController.getMapData);

app.get("/neighbourhoodCountChart", ChartController.getNeighbourhoodCountChart);

app.get("/neighbourhoodStatistics", StatsController.getNeighbourhoodStatistics);

app.get("/roboadvisor", DetailsController.getAdvice);

app.get('/forumTopics', ForumController.getForumTopics);

app.get('/forumPosts', ForumController.getForumPosts);

app.post('/addpost', ForumController.addPost);

app.get('/getpostdata', ForumController.getPostDataById);

app.post('/addcomment', ForumController.addComment);

app.get('/correlation', StatsController.getCorrelation);
```

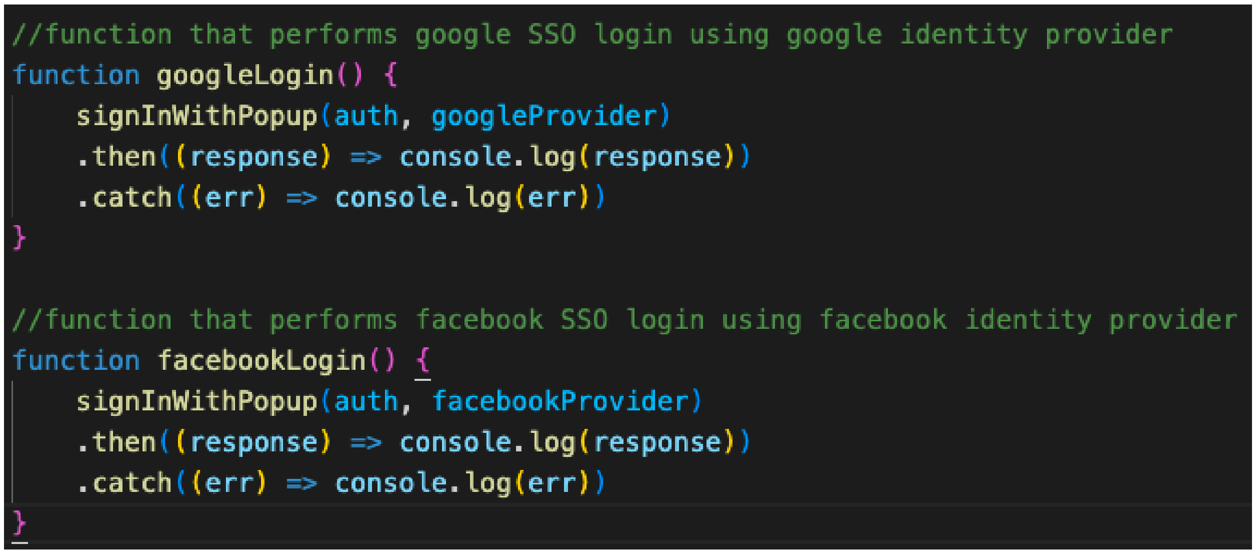
2. Access Control

Introduction

Our application uses Single-Sign-On service where users can log in using their accounts with various identity providers. This login service is handled by Firebase authentication which we pass the authenticator for each identity provider to.

Design Issue and Pattern (Facade Pattern)

This forms an implementation of the Facade pattern as it forms an abstraction from the Views (client) on the logic to authenticate using various identity providers.



```
//function that performs google SSO login using google identity provider
function googleLogin() {
    signInWithPopup(auth, googleProvider)
    .then((response) => console.log(response))
    .catch((err) => console.log(err))
}

//function that performs facebook SSO login using facebook identity provider
function facebookLogin() {
    signInWithPopup(auth, facebookProvider)
    .then((response) => console.log(response))
    .catch((err) => console.log(err))
}
```

3. Search Results

Introduction

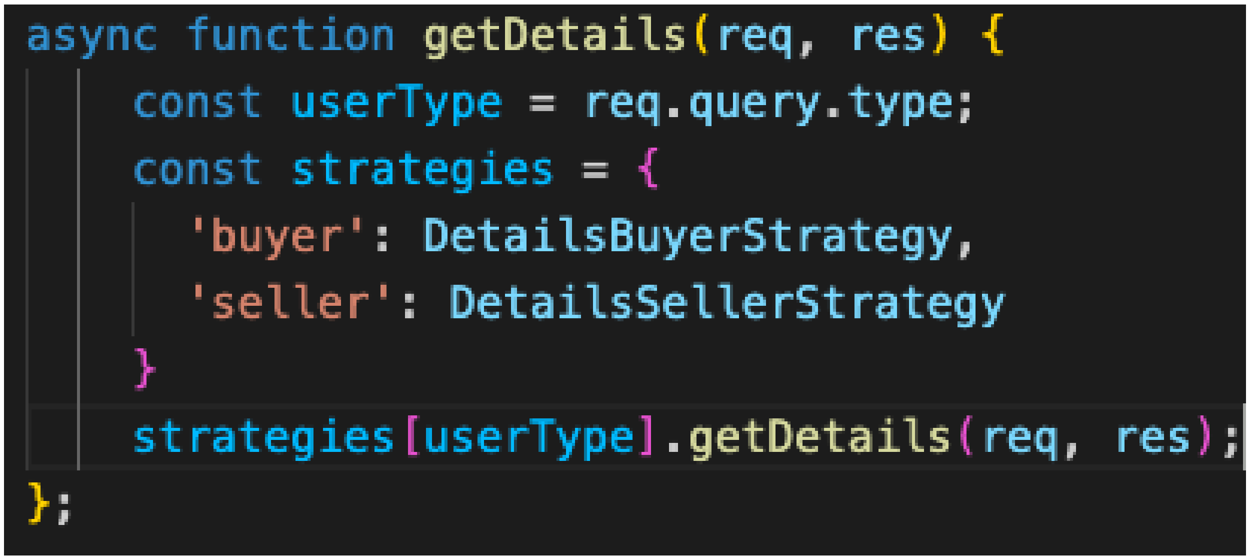
As we have different types of users: buyers and sellers, the search results for each will be different.

Design Issue

We want to get the search results based on the user input type and this will only be known at run time.

Design Pattern (Strategy Pattern)

To return different search results at run time, we adopted the Strategy pattern where we have a strategy for buyer and seller each. We can then choose which strategy to adopt depending on the user input type. The code snippet is seen below:



```
async function getDetails(req, res) {
    const userType = req.query.type;
    const strategies = {
      'buyer': DetailsBuyerStrategy,
      'seller': DetailsSellerStrategy
    }
    strategies[userType].getDetails(req, res);
};
```

4. Charts

Introduction

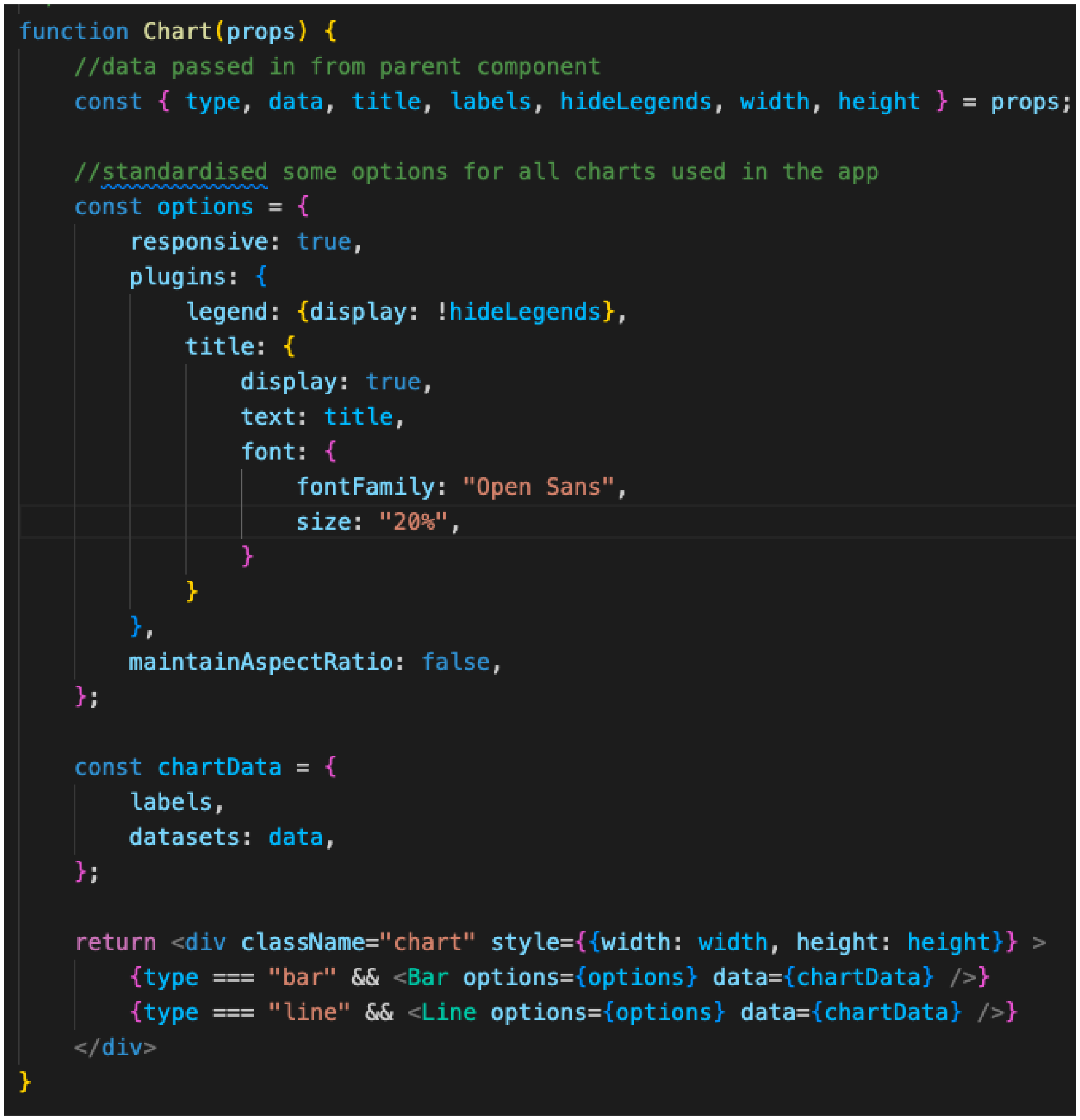
As our application involves data visualisation using charts, we imported an external library Chart.js which offers different types of chart templates.

Design Issue

As we have a lot of charts, we do not want to have to import multiple charts for each file as this will reduce code readability and offer less flexibility.

Design Pattern (Factory Pattern)

We adopted the Factory pattern by creating a Chart component. The component will take in a type parameter and create a chart of that type. Hence, for every file where we need to have charts, we only need one import of the Chart component instead of importing multiple charts.



```
function Chart(props) {
    //data passed in from parent component
    const { type, data, title, labels, hideLegends, width, height } = props;

    //standardised some options for all charts used in the app
    const options = {
        responsive: true,
        plugins: {
            legend: {display: !hideLegends},
            title: {
                display: true,
                text: title,
                font: {
                    fontFamily: "Open Sans",
                    size: "20%",
                }
            }
        },
        maintainAspectRatio: false,
    };

    const chartData = {
        labels,
        datasets: data,
    };

    return <div className="chart" style={{width: width, height: height}} >
        {type === "bar" && <Bar options={options} data={chartData} />}
        {type === "line" && <Line options={options} data={chartData} />}
    </div>
}
```

5. Model-View Interactions

Introduction

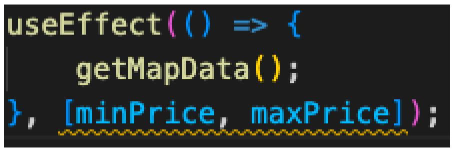
Whenever our model has an update in information, our views will need to update so it will display the latest information.

Design Issue

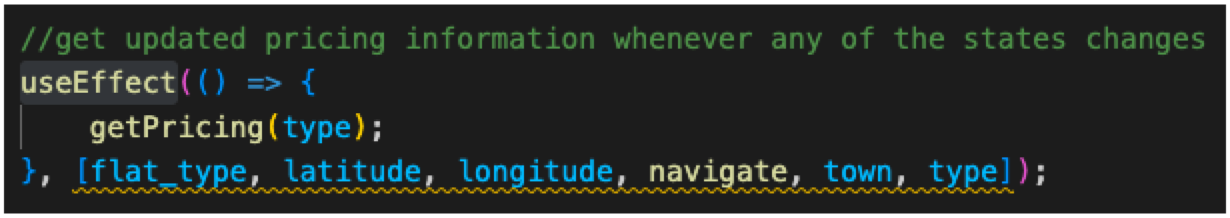
This requires the views to listen for changes and updates of the information. Hence, we need a way for the views to be automatically updated whenever there are changes to the information.

Design Pattern (Observer Pattern)

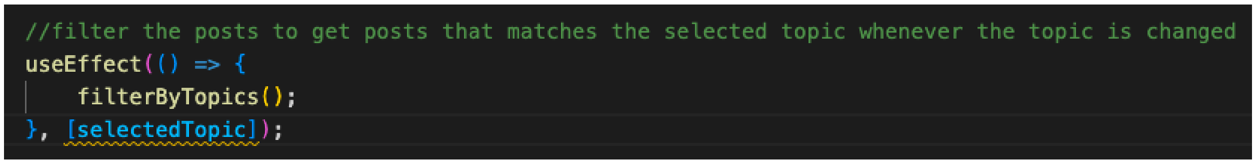
We can adopt the Observer pattern for this. As we use React as our frontend library, we can use one of its hooks, useEffect. The useEffect hook takes in a function and a list of dependencies and will call the function whenever any of the dependencies in the dependencies list changes. We have adopted the use of useEffect extensively and have included some code snippets below:



```
useEffect(() => {
    getMapData();
}, [minPrice, maxPrice]);
```



```
//get updated pricing information whenever any of the states changes
useEffect(() => {
    getPricing(type);
}, [flat_type, latitude, longitude, navigate, town, type]);
```



```
//filter the posts to get posts that matches the selected topic whenever the topic is changed
useEffect(() => {
    filterByTopics();
}, [selectedTopic]);
```