**Logo, company name

Description automatically generated**

**Matriculation Number**: 40397517

**Name**: Sabin Constantin Lungu

**Module Name**: Software Architecture – SET10101

**Programme of Study**: MEng Software Engineering

**Title**: Distributed DE-Store System Report

**Word Count**: 4101

Table of Contents

[List of Figures 3](#_Toc86931394)

[1.0 DE Store Software Architecture Decisions 4](#_Toc86931395)

[2.0 Architecture Selection Reasons 6](#_Toc86931396)

[3.0 DE-Store System Design – Client / Server Architecture 10](#_Toc86931397)

[3.2.1 – Design - Store Administrator MVC Diagram 12](#_Toc86931398)

[3.2.2 – Design – Products Model View Controller (MVC) Diagram 13](#_Toc86931399)

[3.3 DE Store – Database Architecture – Design 13](#_Toc86931400)

[4.0 Evaluation 14](#_Toc86931401)

# List of Figures

Figure 1.0 – Client / Server Architecture Design Diagram

Figure 1.1 – Store Manager Model View Controller Diagram

Figure 1.2 – Products Model View Controller (MVC) – Diagram

Figure 1.3 – Database Architecture - Products Database Diagram

Figure 1.4 – Database Architecture – Admin Authentication Database Diagram

Figure 1.5 – Database Schema Design

# 1.0 DE Store Software Architecture Decisions

DE-Store is a distributed store management business. DE Store allows customers to view electronic products that are currently available in stock. In addition, DE offers various system functionalities such as price control, inventory control, reports & analysis. Also, a financing service as well as a register and login system for admins. Store managers can reset their password as well by clicking on a link. The system is going to contain various users such as customers and store managers. The store managers can edit the contents of new products. They can also retrieve all the products and display them based on sale offers. By removing a product from stock, it will decrement the quantity. Once the quantity reaches below 2 for example, an e-mail is going to be sent to the store manager. Managers can view a list of orders placed by customers and this is represented in the Accounting Section of the system.

1. Client/Server Architecture Style

2. Model View Controller (MVC) Architecture Pattern

3. Database Architecture Style

The Client & Server Architecture style is one architecture that is going to be used for the distributed DE-Store system. This is because it operates independently from other architectures and works well for this distributed system. The role of the server, in this case, is to listen for client requests given an open port available over a network TCP/IP connection. Clients will request a specific resource on the server, for example, accessing all the products within the system or allowing store managers to register an account. The server will then respond with a status code. The status code will determine the outcome of the request. A 200-status code if the request was successful, 500 internal server error or a 404 resource not found if the page is not available on the server. This is a special type of software architecture. It allows multiple unknown clients on the network to request resources from various distributed servers. For the DE-Store system, the Store Manager (admin) will firstly need to create the products which will be stored in a MongoDB database, in this case, one client will send over a POST request to the server. This will create the resource on the server and store the data in the database. The server will then send back a 201 created status code if the product has successfully been created. One reason why the client/server architecture is going to be used is because it provides a central server that can be accessed by any number of clients. It is beneficial because tasks can be partitioned and generally the client/server architecture is easier to manage since there is only one single point of failure, if the server goes down then it is able to recover. Another advantage is that servers connect to networks that provide very fast bandwidth, ultimately allowing very speedy client requests.

For the DE-Store system, a list of products is going to be displayed on the webpage. Furthermore, a GET request is going to be sent over to the server with a 200 OK status code if the request to upload all warehouse products is successful after creating products. The client/server architectures components are represented by the number of active threads that are currently running. Thread runs every time the server is listening for requests and when clients send requests. The connectors of the client/server architecture use queues because each client one after the other makes requests to the server and the server listens for those requests. The protocol for information exchange used for this architecture is Hypertext Transfer Protocol Secure (HTTPS), meaning all the traffic being sent to the server is encrypted. The HTTP protocol is closely linked with its supported HTTP methods (GET, PUT, POST, DELETE) that are used by the users and the data format that is utilised within the DE-Store system is JSON. Another connector for this architecture is the Transmission Control Protocol (TCP / IP). TCP/IP is stateless, unlike UDP, this means that TCP should be used as it provides reliable connections to the server. However, with UDP, data packets are at risk of being lost when being transferred from one channel to the other. And connections may become unreliable over time. Therefore, it is recommended to not use UDP. An advantage of using the client/server architecture is that it is scalable and secure. Different security safeguards can be implemented within the architecture to prevent unauthorised access, such as authorising permissions to users.

The Model View Controller (MVC) is an architectural pattern that is also going to be used. For the DE Store system, there is going to be a model for the store managers and the products. The Models are going to communicate with the MongoDB database architecture closely but does not provide direct access layer to it because the data layer sits underneath the model layer and has no direct access to it. It is going to allow developers to fetch data, insert data, update, and delete data. This is going to be reflected in the view layer. However, in the Model, different attributes can be specified that are stored. Each view has an associated controller linked to it. For example, the store manager is going to have their first name, last name, e-mail address and password. These are going to be used to register a new store manager in the system. Store admins are also able to login and reset their password through a forgot password link. Back-end RESTful APIs are going to be implemented using this architecture pattern. In addition to this, the MVC architecture starts up the models, then the views are initialised that renders data. Each view and controller are linked to a model. For example, within the DE-Store system, each product controller and view that renders products are linked to a product model that stores data that can be used to create product objects.

The components of the MVC pattern are:

* Model
* View
* Controller

The Model is ultimately used for handling the admin and product data. This will get stored in a non-relational MongoDB database. Furthermore, the encryption of customers passwords is going to be implemented in the Admin Model through a middleware function. The remote method invocation middleware acts as a broker between the client and the server architecture. The database centered architecture is also used indirectly to store the manager and product data. This makes it easier to import the Model within a controller file and invoke it directly within a function by querying the database. Finally, the Model is independent from the view layer and does not have any impact on it. It is only responsible for rendering the data stored in the model in within the view.

The **View** is responsible for presenting the web user interface to the users. HTML, JavaScript, and CSS code can be used together to create the user interfaces for DE-Store. For the DE-Store system, the ReactJS library is going to be used for the view layer. It will hold embedded JSX within functional components that are going to be rendered. JSX is plain HTML code that can be embedded within components with JavaScript logic. The view layer is going to have React components that will render a register page for customer registration, login, forgot password and reset password. These are extra functionalities in case the customer forgets their password.

Furthermore, the views are going to render, create, update, and delete page components. These will enable store managers of the DE-Store to coherently monitor their stock by uploading data from the warehouse database. The controller is used to interface the model layer and view layer together. The controllers contain several middleware functions that handles client HTTP requests. For example, a controller file has an exported register user middleware function that will handle a POST request to the server that registers a new admin. This way the controller can easily interface with the model layer because objects can be created.

# 2.0 Architecture Selection Reasons

In this section, there is going to be a focus on comparing two candidate architectures. The advantages and disadvantages of the two architectures are going to be discussed. Hereinafter, the Software Architecture Analysis Method SAAM analysis method is going to be used to compare the two architectures. This analysis method is going to be based on various scenarios that are going to be created based on their quality attributes. The Model View Controller (MVC) Architecture Pattern is going to be compared against the Peer to Peer (P2P) architecture. Firstly, MVC does not support and have direct access to a specific data layer and therefore MVC offers slightly more security than P2P because there is less risk of malicious users executing cyber-attacks such as NoSQL Injections, CSRF or even XSS attacks. And controllers update the model every time a new view is rendered. Each controller is associated with a view. In P2P, there are no controller functions associated with any view. MVC provides separation of concerns and therefore makes systems more performant, maintainable, modular and offers a more flexible test-driven environment to incorporate unit tests. This leads to better software quality. However, A disadvantage of P2P systems is performance. In addition to the above, MVC applications can be made even more **secure** by implementing rate limiting middleware. This is to prevent Denial of service attacks by limiting the number of requests that come from a specific IP address, this is not possible In P2P.

One reason for choosing MVC over P2P is maintainability. This means that if any code alterations are made to one part of the MVC architecture, it will not affect the entire web application architecture and components can be reused throughout the entire application and makes maintainability much easier. Whereas in P2P, systems are not maintainable, and peers cannot be reused. Also, security is known to be a problem within P2P systems. The P2P Architecture is not as efficient because its performance is also known to be an issue which is not good for distributed systems. There are high connection latencies, which is a disadvantage of the P2P architecture because it slows down applications as the number of peer devices that joins a network increase, the performance will get affected because multiple users are accessing the same device also P2P supports no database access because it does not have a dedicated file to access. As a result, the MVC architecture pattern is preferred and is going to be used over the P2P Architecture because it is more performant and secure. Also, Peer-to-peer systems are not considered to be very **safe.** Pure P2P systems do not have a central server to access. This way it is more difficult to keep track of potential viruses that can infect P2P systems. Multiple viruses can destroy an entire system which affects user data negatively. In P2P systems, various nodes can disconnect from one another, and this is a problem because there is no reliability between peers which is a problem in these types of systems.

Another disadvantage of the Peer-2-Peer architecture is security. P2P systems are prone to denial-of-service attacks. Any number of devices can send superfluous requests to a P2P system and make their services unavailable by taking down the devices connected to the network which indirectly affects the performance of a system. Therefore, MVC is going to be used because it offers better performance and security over P2P. Furthermore, MVC offers efficient test-driven development. This means that unit tests can easily be implemented that results in increased software quality if all the tests pass, whereas with P2P it is not possible to implement unit testing.

Below are various MVC scenarios that are SMART:

1. Smart (Reasonable)
2. Measurable
3. Achievable (Scenario that can be completed)
4. Realistic (Not complicated)
5. Testable (Ability to implement Unit Testing on)

The following quality attributes of the MVC architecture are outlined below in a list. Some quality attributes are going to be evaluated in the evaluation section of the report using the ATAM method. Each quality attribute is going to hold several scenarios which are going to be evaluated.

Model View Controller (MVC) Architecture – Quality Attributes:

1. **Performance** – Reduced JSX tags within sub-components. Number of script tags are reduced. There is no reliability on the back-end code to add new UI features using MVC. Hence, an increased response time and lower resource consumption as well as a separation of concerns. MVC also supports consistent database connections through Docker containerisation.
2. **Flexibility** – Front-end flexible with many libraries & frameworks that can be installed using Node Package Manager (NPM). MVC allows the front-end to communicate with the back-end code through API requests. Therefore, developers can either work on the front-end or back-end by using this architecture pattern. The benefit is that software can be shipped quicker as more features are developed by multiple developers.
3. **Reusability** – MVC - View layer components can be reused throughout the entire application which leads to an increase in performance as components don’t have to be declared repeatedly which results in code duplication.
4. **Modularity** – MVC View Layer components can be broken up into various sub-component modules which makes systems more modular.
5. **Testability** – MVC - allows Test Driven Development (TDD) unit tests to be incorporated within the web application architecture. Controller functions and UI can be unit tested independently within a test suite efficiently. It provides better software quality based on the feedback the unit tests give which allows developers to ship software faster.
6. **Maintainability** – MVC offers a separation of concerns. This means that the application is divided into multiple layers. The model, view, and controller. This makes it easier to maintain and keep each layer up to date without affecting the overall architecture if modifications are made to independent layers.
7. **Security** – MVC does not have a data layer to access directly. This layer sits underneath the other 3 layers. Therefore, there is an increased security. Attackers cannot carry out malicious attacks that would harm the entire application because there is no direct access to the layer. MVC supports pre middleware functions in the model layer that encrypts user passwords and packages to prevent denial of service attacks.
8. **Scalability**: MVC allows changes to be made to the overall application that can allow it to scale (grow) over time as more controllers and models are added to the system.

Candidate Architectures:

1. Model View Controller (MVC)
2. Peer-2-Peer (P2P)

The following scenarios are outlined below. Scenarios determine how suitable each architecture is based on. They are going to be determined by the number of direct scenarios each architecture possesses and not indirect ones. This is because direct scenarios do not modify the connectors and components, whereas Indirect scenarios do.

Scenario 1: Add a middleware function for updating the price of a product that is either deemed to be too expensive or too cheap.

Scenario 2: Add a middleware function to encrypt store manager’s password when registering a new account by communicating with the database and manager model.

Scenario 3: Change the CSS User Interface Code.

Scenario 4: Add a function that displays all DE-Store products.

Scenario 5: Adding a method to maintain a constant connection to the database.

Scenario 6: Add unit tests to test the back-end RESTFUL APIs.

Scenario 7: Implement security mechanism to prevent denial of service attacks.

Based on the scenarios above. A SAAM scenario evaluation table is outlined below that determines if the scenario is an indirect or direct one. The cost of each scenario is also going to be outlined.

This will determine how long it could potentially take to achieve this scenario. Thereafter, an overall SAAM evaluation table is going to be outlined. This table is going to assign weighting values of (0, -/+) so the two architectures are ranked accordingly.

|  |  |  |  |
| --- | --- | --- | --- |
| Architecture | Scenario | Indirect / Direct | Cost |
| Model View Controller | 1 - Add a function to Update a product price. | Direct | Couple of minutes |
|  | 2 – Add function to encrypt store manager password | Direct | Couple of minutes |
|  | 3 – Change the CSS User Interface Code | Direct | Couple of minutes |
|  | 4 – Add function to display all warehouse products | Direct | Couple of developer minutes |
|  | 5 – Add method in a file to maintain a connection to the database. | Direct | Couple of developer minutes |
|  | 6 – TDD - Add Unit tests to the back-end APIs. | Direct | Couple of developer minutes |
|  | 7 – Implement security mechanism to prevent denial of service attacks | Direct | Can be achieved through a middleware function. |
| Peer-2-Peer (P2P) | 1 – Add a function to update product price | Indirect | Very difficult and almost not possible due to high connection latencies. |
|  | 2 – Add function to encrypt store manager passwords | Indirect | Very difficult because P2P has security flaws |
|  | 3 – Add a change the CSS code. | Indirect | n/a |
|  | 4 – Add function to display all products. | Indirect | Several minutes due to high latency |
|  | 5 – Add method to maintain a consistent connection to the database | Indirect | Several minutes / hours |
|  | 6 – Add Unit Tests to the back-end APIs. | Indirect | Couple of hours |
|  | 7 – Implement security mechanism to prevent denial of service attacks | Indirect | Not possible because there is no central server. |

Based on the table above, a SAAM overall evaluation table is outlined below.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Architecture | Scenario 1 | Scenario 2 | Scenario 3 | Scenario 4 | Scenario 5 | Scenario 6 | Scenario 7 | Overall |
| MVC | + | + | + | + | + | + | + | + |
| Peer-2-Peer (P2P) | - | - | - | - | - | - | - | - |

Overall, in conclusion, the Model View Controller Architecture pattern is going to be used due to the number of advantages that it comes with. MVC is going to be used alongside the client/server and data-centred architecture to develop the DE-Store distributed system. Firstly, any modifications made to any of the layers in MVC does not affect the entire architecture. Furthermore, MVC offers reusability, modifiability, testability, flexibility, and performance. Secondly, because it is triangular, this means that data flows in 3 directions without affecting independent layers in a negative way. In P2P systems, it is much harder to maintain consistent connections to a database because there are high connection latencies which affects the overall performance.

# DE-Store System Design – Client / Server Architecture

In this section of the report a design of the entire DE-Store system is going to be created. Firstly, the client/server architecture is going to be designed. This is going to show how clients send requests over to the DE-Store server. The client/server architecture uses TCP/IP as its connector. Figure 1.0 below shows the way in which clients sends requests over to the server.

Diagram

Description automatically generated

Figure 1.0 – Client / Server Architecture DE Store Design

The figure above represents the client/server architecture design. A docker compose file that allows multiple containers to run simultaneously builds a docker image that runs the back end. Once the back end is running, the server is going to listen on a specific port for clients to send requests to. Axios is a node package manager library that allows HTTP requests to be made on the front-end that communicates to the back end to fetch data from the database and that data is rendered on the front-end. Also, the server constantly listens for client requests.

# 3.2.1 – DE-Store Administrator MVC Diagram

Figure 1.1 – Store Manager Model View Controller Diagram

Diagram

Description automatically generated

**Figure 1.1** above shows the different views that are to be implemented within the web application. The admin model stores data such as the username, email, and password. There are various methods that can be accessed within the authentication controller that allows managers to register, login and reset their password. The authentication controller manipulates the admin model directly to create a store manager object to hold data. The functions within the model such as comparing passwords is available to access within the authentication controller. The function is used to compare the inputted password against the one in the database and should be checked before logging an administrator in. If they do not match then the admin is not granted access in the system, otherwise they are granted access if they match. Therefore, there is this separation of concerns between layers. The Admin Products List View sends a request from the front-end to the back end to fetch all the products that are being stored in the database. These products are only available for administrators to view, create, update, and delete. However, users can only view the products and not perform other actions.

# 3.2.2 – Design – Products Model View Controller (MVC) Diagram

Figure 1.2 – Products Model View Controller (MVC) – Diagram

Diagram

Description automatically generated

The diagram above shows how the products controller interacts with the model and products list view. Each product has specific attributes as defined in the model. These are used to create unique product objects. The CRUD operations are also implemented within the products controller file. They are responsible for creating, reading, updating, and deleting products on the back end. JSON is the usual response message sent back. Finally, the products can then be retrieved on the front-end using an API library called axios that sends requests to the back end to retrieve the products. It renders all the products in stock on the web application.

# 3.3 DE Store – Database Architecture – Design

MongoDB database is used within the DE-Store system which is a non-relational database. MongoDB stores data in collections, not tables. For the DE-Store, there are various databases that have their own collections. Store Manager data is stored in a collection named “admin” and the products in their own collection called products. Connections to the database are made through a connection protocol on port 27017. In this case the connector that makes a connection to the database is the protocol itself and the components of the database architecture is the database itself and the clients (agents) that reads / writes data to and from the database to manipulate JSON data. Figure 1.3 below shows how clients interact with. **Figure 1.5** in an appendix depicts a MongoDB schema design that shows how the admin data is linked to the products data. Each document in the database is linked with a unique Object ID. Figure 1.5 represents the JSON data that is stored in different collections and how the schemas are connected to each other.

Figure 1.3 – Database Architecture - Products Database Diagram

Diagram

Description automatically generated

Figure 1.4 – Admin Database Diagram

# Diagram Description automatically generated4.0 Evaluation - ATAM

This section is going to evaluate the MVC architecture pattern against various quality attributes that has been specified in the architecture comparisons. Furthermore, the attributes will have a priority pair of low, medium, and high next to them. These are used to represent how important the scenarios are to achieve and how difficult they are to achieve. To achieve full success of the system, there must be a high and/or medium importance for more than one scenario. If a scenario is classified as high or medium, then it has been simple to achieve. In this section, a table is going to be outlined which evaluates the system on the following quality attributes for the MVC pattern.

* Performance
* Security
* Modifiability
* Testability

|  |  |  |
| --- | --- | --- |
| Quality Attribute | Scenario | Priority Pairing |
| Performance | Latency | (H,M) – Maintain a consistent database connection 99.9% of the time.  (L,M) – Minimise 50 % of storage in database  (H,M) – Maintain Database Connections in < 2 seconds |
| Security | Data Confidentiality & Data Integrity | (H,M) – Admin Login Works 90% of the time.  (H,M) – Admin Register Account Queries are secure 99.9% of the time  (H,L) – Encrypting Admin Passwords < 2 seconds  (H,L) – Log Out Admin In < 1 second.  (M,L) – Validating 99.99% of data. |
| Modifiability | Modify Products  Change Web User Interface | (H,L) – Edit price of a product in < 1 second.  (H, L) – Create a new product in < 1 second  (M,L) – Change Store Manager Dashboard User Interface < 2 Developer Weeks |
| Testability | Adding Unit Tests | (H,L) – Unit Tests run in <= 5 seconds |

A picture containing text, indoor, public, tiled

Description automatically generated

# Appendix

Figure 1.5 – Database Schema Design

Diagram

Description automatically generated