Software Architecture DE- Store – 40489224- Jayveer Chall

Word Count = 3941

Contents

[**1.1 Introduction** 3](#_Toc151724863)

[**2.1 Microservices Architecture** 4](#_Toc151724864)

[**2.1.1 Components:** 4](#_Toc151724865)

[**Microservices:** 4](#_Toc151724866)

[**Databases:** 4](#_Toc151724867)

[**API Gateway:** 4](#_Toc151724868)

[**Service Registry and Discovery:** 4](#_Toc151724869)

[**2.1.2 Connectors:** 4](#_Toc151724870)

[**Protocol for Information Exchange:** 5](#_Toc151724871)

[**2.2 Three Tier Architecture Analysis** 5](#_Toc151724872)

[**2.2.1 Components**: 6](#_Toc151724873)

[**Presentation Layer :** 6](#_Toc151724874)

[**Business Logic Layer:** 6](#_Toc151724875)

[**Data Layer:** 6](#_Toc151724876)

[**2.2.2 Connectors and Protocols for Information Exchange:** 6](#_Toc151724877)

[**3.1 Justification for Selected Architecture Style:** 6](#_Toc151724878)

[**SAAM Analysis** 6](#_Toc151724879)

[**3.2 Key Reasons** 9](#_Toc151724880)

[**4.1 Design of Full System** 10](#_Toc151724881)

[**4.2 Component Diagram:** 12](#_Toc151724882)

[**4.3 Data Flow** 13](#_Toc151724883)

[**4.4 Core Security** 13](#_Toc151724884)

[**4.5 Databases** 13](#_Toc151724885)

[**5.1 Evaluation of DE-Store** 14](#_Toc151724886)

[5.2 Future Improvements 15](#_Toc151724887)

[**6.0 References** 16](#_Toc151724888)

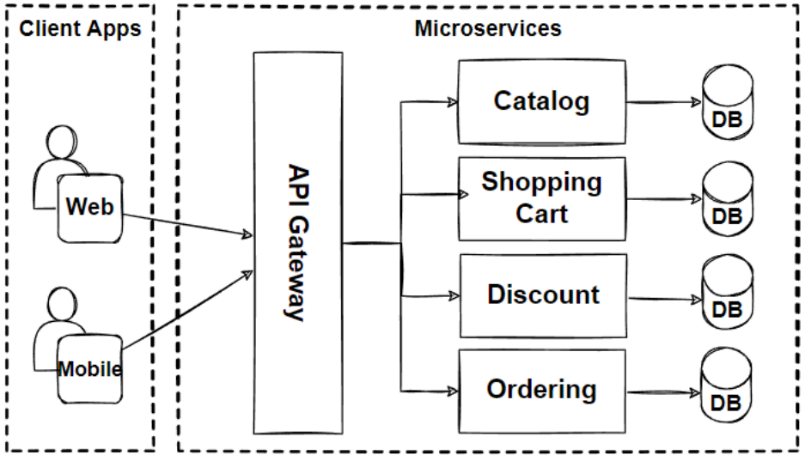
# **1.1 Introduction**

The new age of retail markets are highly competitive and constantly changing, necessitating robust and easily scalable systems that can adhere to the diverse and evolving needs of a major retail corporation such as DE-Store. The rapidly growing demand for real-time data, flawless coordination among multiple retail branches and augmented customer satisfaction drives the necessity for a distributed store management system.

The main objective of DE-Store is to choreograph a suite of store management functionalities that support the operational and co-ordinational efficiency across all retail branches. The essential functionalities that DE-Store require are: Price Control, Inventory Control, Loyalty Card Management, Finance Approval, and Reports and Analysis.

# **2.1 Microservices Architecture**

Microservices Architecture would be an excellent choice for the DE Store software as this architecture separates the application into independent services which can be deployed individually through virtual machines, physical servers or containers. This allows for easy scalability and adaptability which is important for DE Store as they may have constantly evolving business requirements.



Components are basically the building blocks of a system. They work by separating the system into modules which they all encapsulate different sets of behaviours and functionalities. Some of the components which are specific to the Microservice Architecture are as follows:

# **2.1.1 Components:**

## **Microservices:**

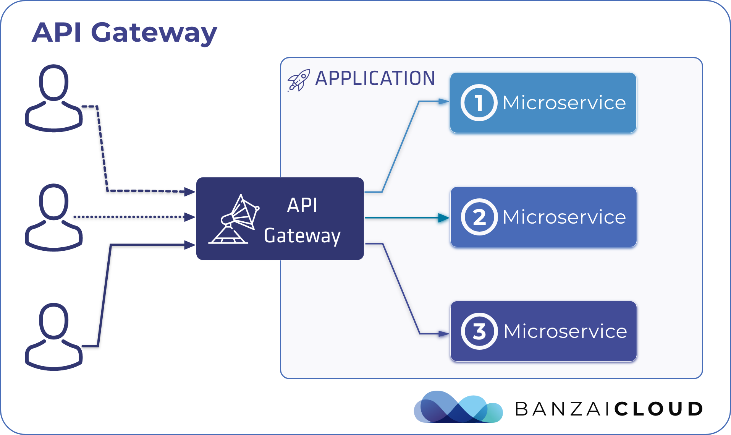
With this type of architecture all the software functionality for the business is split into their own microservice and functions as its own service, e.g. the Inventory control and Price Control are two completely separate services which run independently from one another. This means each service can be developed using completely different technology stacks which allows for a more robust and scalable software.

## **Databases:**

Instead of using one large relational database in the microservice architecture each service has its own individual database which in turn promotes microservice autonomy.

## **API Gateway:**

The API gateway acts as the control point for the software, this is where requests from other users or other systems are first received and then the API gateway decides which request should be handled by the corresponding service. This being said the API gateway shouldn’t be running complex processes or making any big decisions, each microservice should be fully self-contained and not require any decision making or work to be done outside that microservice.



## **Service Registry and Discovery:**

The Service Registry’s main goal is to keep track of where all the services are running and when they are running, since it is a distributed system and there are different servers and/or containers the service registry must take note of what is running. It does this by having each service register itself before it starts running.

The Service Discovery is called when two individual services need to interact with eachother, this is done by querying the service registry to locate its current location and status.

# **2.1.2 Connectors:**

RESTful APIs : For synchronous communication between microservices. REST is the most commonly used for web based interactions, REST api’s work by sending messages over the internet normally in the form of XML or JSON.

Message Brokers: Message brokers allow for asynchronous communication between microservices, they provide security, automated management and most importantly simplify the communications between microservices.

## **Protocol for Information Exchange:**

HTTP/2 or HTTP/1.1: Are used for RESTful API interactions this supports stateless communication. HTTP/2 is a binary protocol this means it uses less bandwidth and are less error prone than the textual based protocols such as HTTP/1.1.

Advanced Message Queuing Protocol : AMQP is a protocol for message orientated software. Its best used when secure messaging is required between distributed systems. This aids in creating systems that are robust and scalable since it allows for asynchronous communication.

# **2.2 Three Tier Architecture Analysis**

Three Tier is another architectural style we can use in software design, it consists of 3 different as suggested by the name. The data layer which is most commonly referred to as the “Bottom Tier”, this is where all the information and data for the software is stored commonly in the form of a database. The “Middle Tier” is where the business logic is handled such as the methods for updating prices or inventory and amending the database, allowing the software to have proper functionality. The middle tier acts a link and passes requests on to the database. Finally, the client tier or often referred to as the “Top Tier” is where all the visual aspects of the software are located (user interface), this is where someone who was using the software would be able to physically see and access.



The components of Three Tier Architecture are the building blocks that make up the software. For Three Tier Architecture the components are as follows :

# **2.2.1 Components**:

## **Presentation Layer :**

The user Interface of the software such as the text boxes and buttons which allow users to update and query the databases.

## **Business Logic Layer:**

This is where all the logic and functionality for the software is processed such as the price control and inventory control e.g., The system can do scheduled checks on the inventory which will send updates on the items low on stock to the users email.

## **Data Layer:**

This layer handles all the data for the software so we can add and retrieve data from the database in this specific case the H2 database was used.

# **2.2.2 Connectors and Protocols for Information Exchange:**

HTTPS and HTTP requests are used to allow communication between the frontend and backend, in other words the “Top layer” and “Middle layer”, especially when making RESTful API calls.

To connect to the “Bottom layer” Java Database Connectivity (JDBC) will be used facilitated by JPA. This allows for an extra layer of separation for database interactions, it gives an object oriented style of data handling by changing database entries to JAVA objects and vice versa.

# **3.1 Justification for Selected Architecture Style:**

## **SAAM Analysis**

Scenarios for DE-Store:

1. DE-Store has launched a new promotional product over the Christmas holidays, this must be added to the Inventory Control System.
2. There has been too many customer complaints about not being able to return unwanted items so DE-Store is introducing a brand new module/service for handling return shipments.
3. As the business needs grow so does the demand for multiple payment methods so DE-Store has decided to incorporate a brand new payment method for PayPal.

**Candidate Architectures:**

DE-Store system uses a Three-Tiered heterogenous architecture made up of three different layers and will be compared against the microservices architecture:

Three Tiered:

* Presentation Layer: User Interface components.
* Business Layer: All business logic and core functionalities.
* Data Layer: Where all the data is managed and stored for the system.

Microservices:

* Individual Services: All business logic is split into its own services.
* Separate Databases: Each Service has its own Database.
* API Gateway: acts as a central point for routing users to the correct service.

Both the Three Tiered and Microservices architecture are described using Architecture Description Language to make sure there is no uncertainty and to make sure the descriptions are clear.

**Classified Scenarios:**

Direct Scenarios:

Adding a new product into the inventory management system in the Three-Tiered Architecture, this would require no architectural modification.

Indirect Scenarios:

Adding a fully new returns service for unhappy customers as well as integrating PayPal transactions, this would require modifications for both architectures.

**Scenario Evaluations:**

Three Tiered Heterogenous Architecture:

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Impact** | **Cost** **Weighting (Days)** |
| Adding a new product | Low | 1 |
| Creating a new returns service | Moderate | 3 |
| Integrating PayPal transactions | Moderate | 3 |

Microservices Architecture:

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Impact** | **Cost Weighting (Days)** |
| Adding a new product | Low | 1 |
| Creating a new returns service | High | 5 |
| Integrating PayPal transactions | High | 5 |

**Scenario Interaction:**

Three-Tiered Heterogenous Architecture

Adding a new returns service:

To integrate the new returns service many changes would need to be made such as adding new rules and processes for the handling of returns, such as updating inventory back when items are returned back, managing refunds and payment details as well as a quality check system to make sure the items are returned in the way they were sent out. The data layer would also need to be adjusted to support the new data such as dates of returns and repayment data including sensitive credit card information for customers. This highlights the importance of modular separation within a system so it is easy to add new services and make modifications along the way in turn this would enhance the maintainability and scalability of the system.

Microservices Architecture:

By adding the new returns service or even the PayPal integration within the microservices architecture could cause many problems. Due to the nature of microservices each service has its own individual dependencies and needs careful management, when working with different dependencies for different services this can cause problems when specific dependencies don’t cooperate. This is crucial for something like returns as there would be multiple services involved e.g. inventory, price and finance. While Microservices offer the potential for scalability at the same time this could cause the system to run slowly as well as increasing the management needs and costs.

**Overall Evaluation**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Architecture** | **New product** | **Returns Service** | **PayPal Integration** | **Summary** |
| Three Tiered | 0 | + | + | + |
| Microservices | 0 | - | - | - |

New Product – There is no better option in this scenario for adding a new product as both architectures are more than capable of adding and handling a new product being added to the store

Returns Service – The Three Tiered Heterogenous Architecture is the winner for the returns service, this is due to its cohesive structure which allows the system to add a new service without a huge amount of work. For the Microservices architecture, whist it could handle a new service being added due to the nature of microservices it loses out here as integration could be a real challenge since microservices are very complex.

PayPal Integration – The Microservices architecture misses out here again due to complexity and dependency issues, since the PayPal integration would need to span across multiple different services this could cause dependency issues and potentially damage the rest of the system.

## **3.2 Key Reasons**

Ease of Implementation – One of the main reasons that the 3-tiered approach was used is due to the fact it’s a lot more straight forward and easier to build rather than the microservices architecture. This was important for project as due to time constraints it allows for a more complete prototype with more functionality and quality features. Since microservices architecture uses its own independently deployable services it means each service requires its own unique dependencies as well as its own stack, with a larger team and less time constraints this is more manageable. To ensure all the services are able to work with another without constantly crashing requires massive amounts of coordination and time which was not available for this project.

Networks – Since Microservices communicate with one another over a network it means they have to make frequent network calls so the services can communicate with each other. Every time there is a network call there is also latency. This is different to method calls as the network calls massively delay response time even more so when multiple services are attempting to communicate with each other. This can be a big problem where this a scenario with high volumes of traffic as all the network calls can seriously degrade the systems response time.

API Gateway – An API gateway has the potential to simplify interactions since it serves as a single entry point however this can also cause problems especially in situations with high load as it creates a bottleneck. Within a three tiered approach the business layer controls the routing and load distribution within the system, this is not only far less complex but also decreases the chances of any bottlenecks. Not only this but with a API gateway it must be constantly managed and optimised even more so as the system grows and scales larger which requires constant resources and attention. In this case the three tiered approach is superior as it allows for an easily manageable and maintainable system that can scale accordingly with the business needs.

Thin Client Architecture – This would be especially beneficial for DE-Store as It means various hardware devices can access the system without the need for high performance machines, this means it’s cheaper for the company as they are able to setup the software on basic machines or even tablets. This is highly important when there are multiple different stores all accessing the same software from different locations. As well as this it allows for a simple and easy to use user interface in turn reducing training costs and any potential software malfunctions.

Centralised Maintenance – Since all the business logic is in one centralised place on the application server this means that any updates for the software that are required are all made in one singular place and rolled out to all stores together. This is important as it means there is consistency across all stores as well as business operations.

Load Balancing Between Database and Logic Functions – By separating the database functions (such as transaction processing) it allows the system to handle massive amounts of transactions efficiently. During busy retail periods this is highly important as it means the system can keep up with all the sales transactions and inventory updates.

Scalability – Whilst the business expands it is easy to add more modules if there is more functionality required or if there is new stock it is easier to update the database add support for new items that may be required. This means that the system can grow with the business and the original software can just be regularly updated for the business needs rather than having to create a new software when the business expands.

# **4.1 Design of Full System**

The DE-Store system prototype was constructed using a Three-Tiered heterogenous approach, the system uses a modular take on the three tiered style of architecture. This architecture was chosen due to simplicity and scalability. Since it was only a prototype design of the system it was mostly focused around having key functionality and showing off the architecture rather than a complete implementation. The system has 3 distinct layers, Presentation, Business and Data.

**Presentation Layer:** The presentation layer was created using JavaFX for the user interface and FXML for defining the layout in an XML format.

**Key Features:**

UI for Inventory Management: Provides a live list of what stock is still available in the store.

UI for Pricing Management: There are multiple buttons with different functionalities such as updating price, adding new prices and deleting prices.

UI for Finance Approval: Allows for users to apply for finance instead of using a regular price.

UI for Loyalty Card: Each user it rewarded a certain number of “points” based on how much money is spent, the system tracks which users have the most points and offers them discounts on promotional items.

**Business Layer:** The business layer is where most of the core functionality required for DE-Store is stored such as price control, inventory control etc. For the business layer I used SpringBoot as it provided me with a good framework to build the backend for the DE-Store project. I also used Spring Data JPA for the object relational mapping and the database functionality. Finally I used Lombok which is an annotation processor that fills in boiler plate code at compilation time.

**Key Features:**

Inventory Control Service: Contains most of the business logic for managing stock, including CRUD operations and an inventory scanner that checks stock levels and sends out updates to a user’s gmail of which items are running low on stock. I used the ”@Schedule” annotation within Lombok so the system scans the inventory every 1000 seconds and provides updates.

Price Control Service: Controls all the pricing logic such as removing and applying sale offers as well as any price adjustments that may potentially be made.

Finance Approval Service: This allows users to submit finance requests as well as update them, within this class there is also the ID system which makes sure there is no duplicate user Id’s.

Loyalty Card Service: Adds points to a user’s profile for every purchase they make, this is where the logic for sending and receiving discounts and offers for regular customers is also stored.

Report and Analysis: The programme queries all the databases and provides summaries for each service for analysing and tracking.

**Implementation:**

Model: Created an entity class for each service e.g., Inventory to represent items which are in stock, featuring fields like itemId, itemName, quantity, etc.

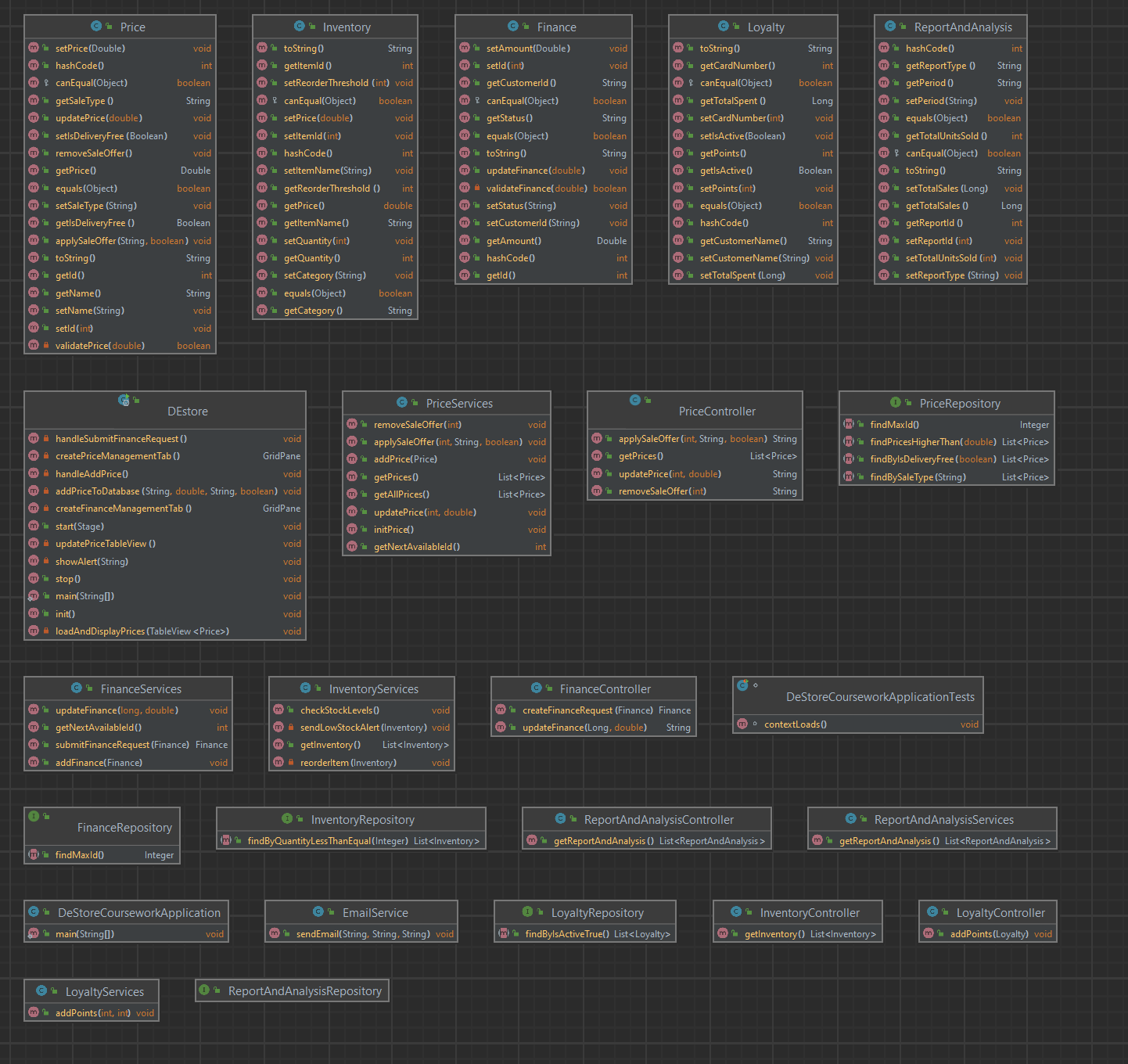
Repository: Used the Spring Data JPA repository (InventoryRepository) for CRUD operations on the data for store inventory.

Service: InventoryService is where the actual logic for the inventory control is stored such as sending out gmail alerts if stock is low and putting items on reorder if they are out of stock.

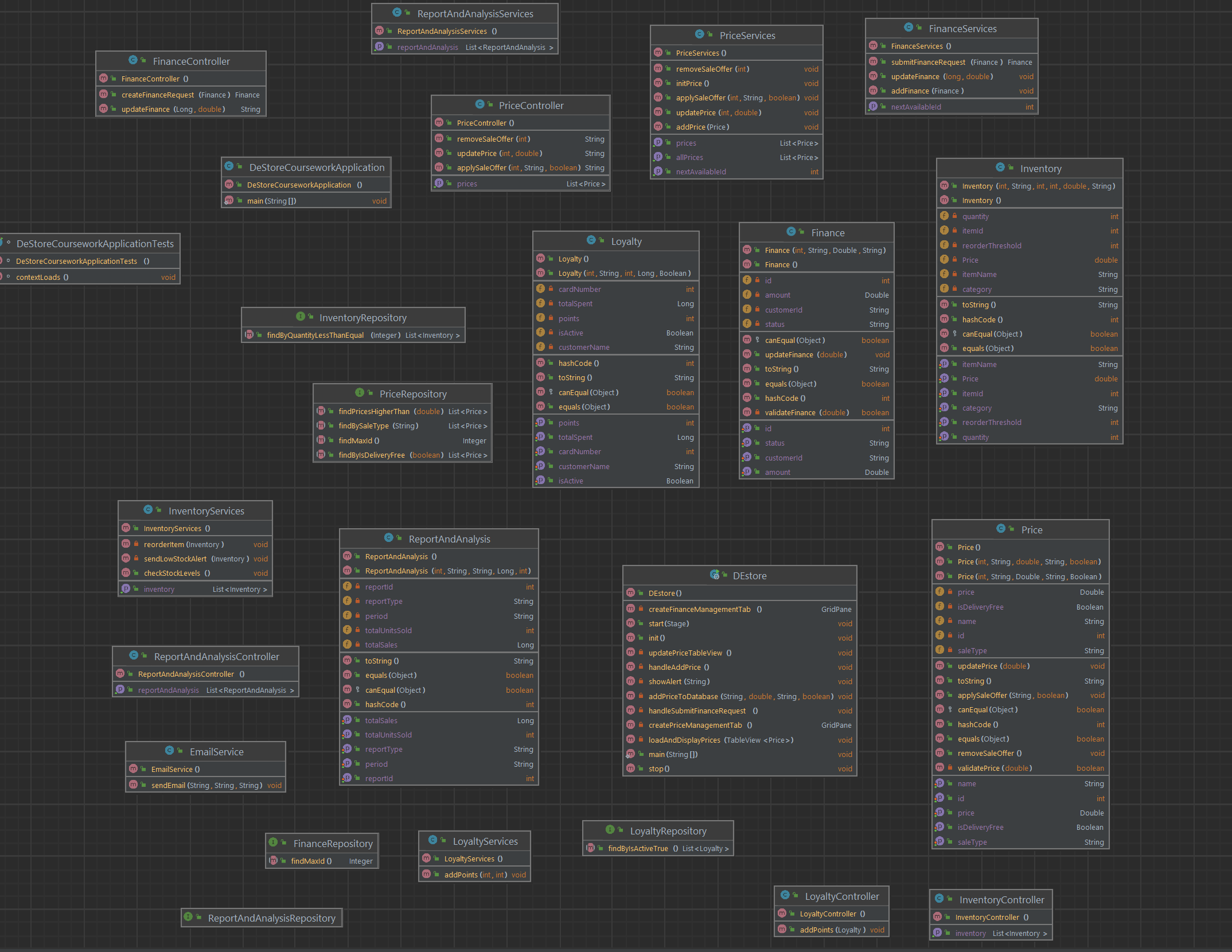
Controller: InventoryController uses the “@GetMapping” annotation to handle http GET requests this provides RESTful endpoints for the InventoryService.

Scheduled Task: Uses the “@Schedule” annotation which allows the system to check every 1000 seconds if there has been any update to the stock level.

**Data Layer**: For the data layer I used H2 database as it worked well and was sufficient for a prototype. JPA repositories was also used for abstracting data access. This is where all the data for the system is stored, instead of using a relational database each service was given its own database to ensure system reliability. This way if one of the databases isn’t working correctly then the system can function without that service however if I used a relational database the full system would collapse if there was an error in the database.



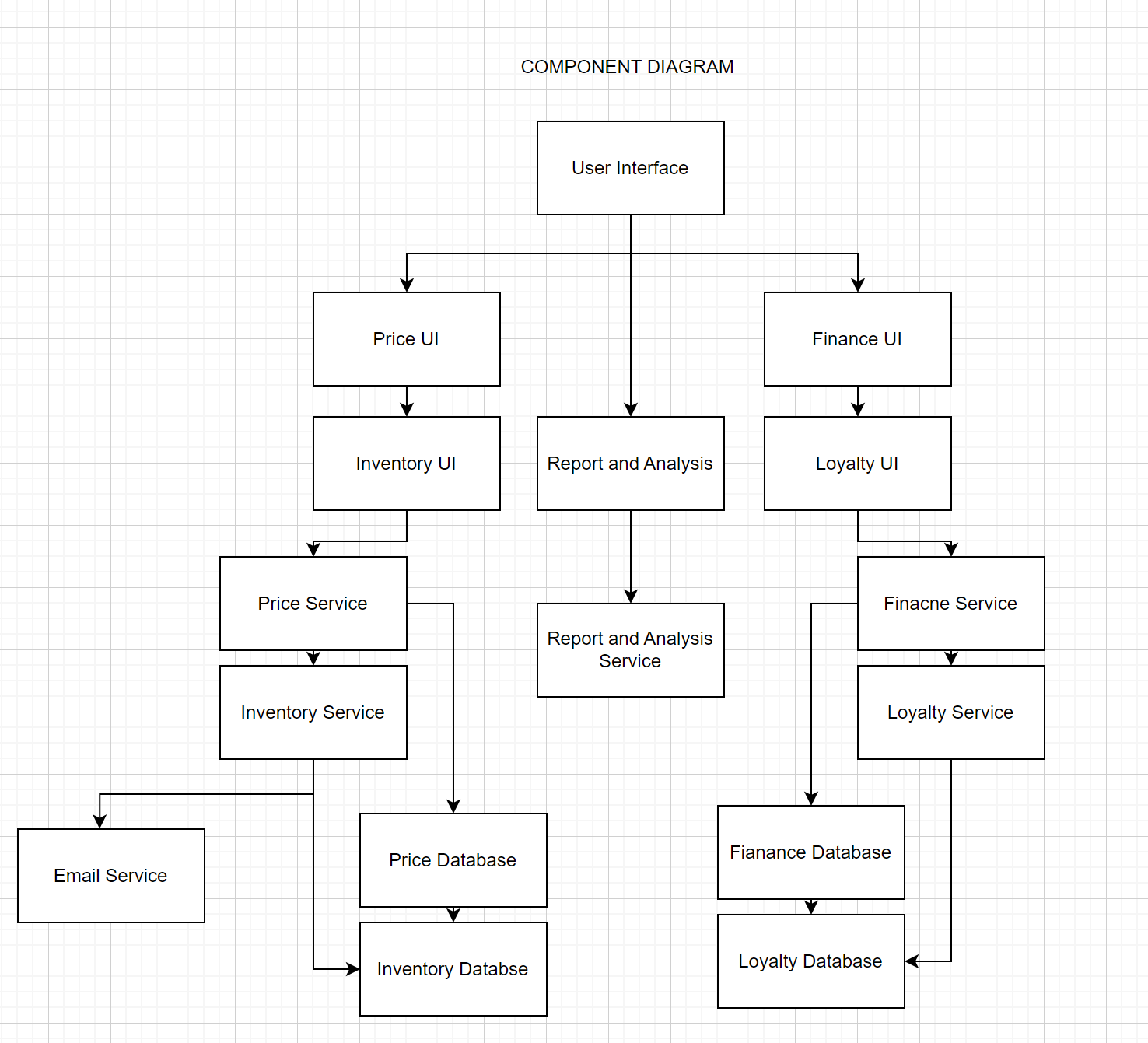
This is the class diagram for the DE-Store prototype it highlights all the classes and their respective methods.



This is the class diagram for the DE-Store however in this one I’ve included, methods, properties, constructors and fields to provide a better understanding of how the system is built.

## **4.2 Component Diagram:**

This helps to visualise the structure of the system and how the architecture is structured, you can clearly see the three different layers, UI components, Service Classes and the respective database for each class which makes up the 3 tiered heterogenous architecture style.



## **4.3 Data Flow**

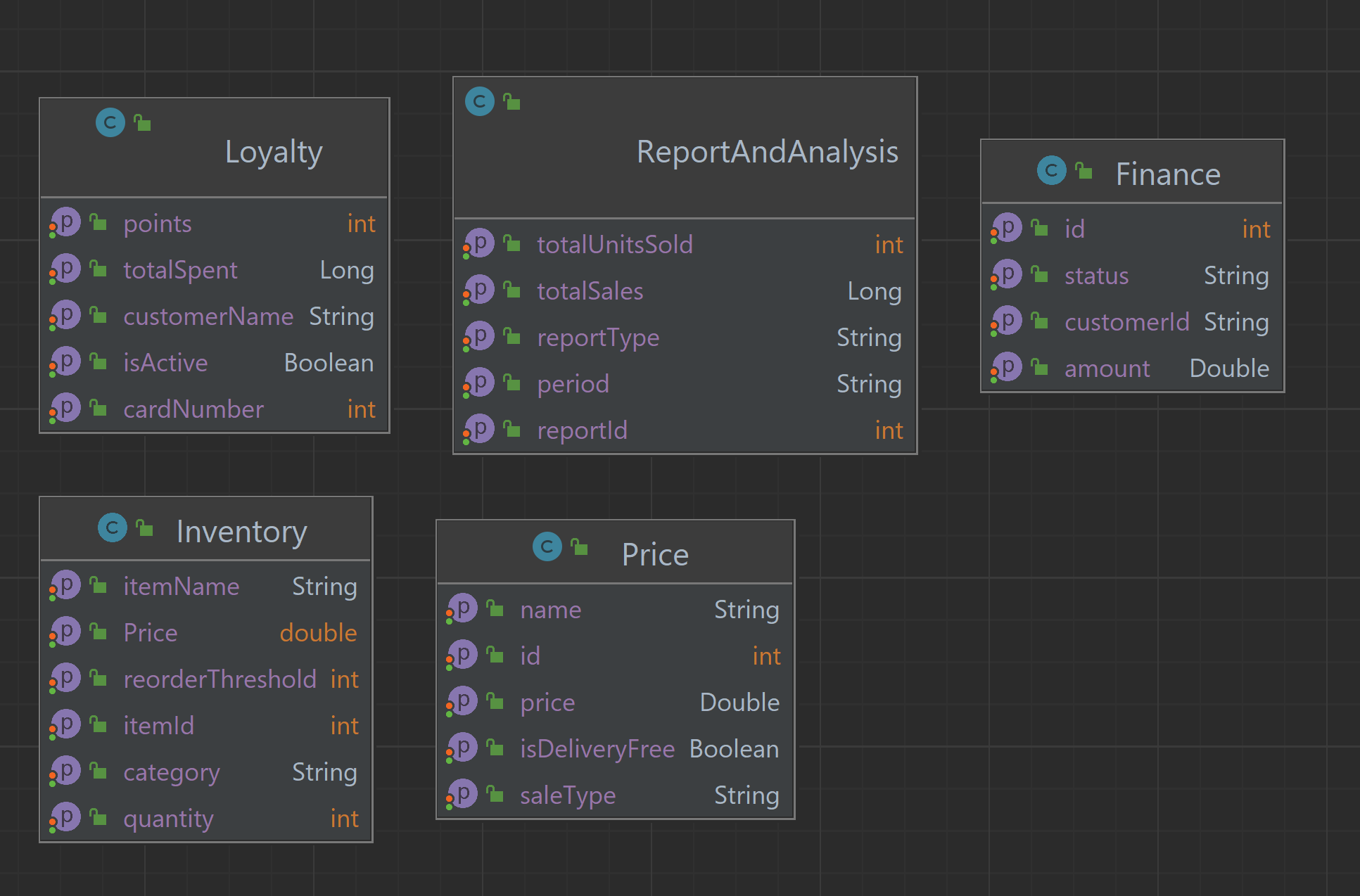
When a user interacts with the user interface requests are sent from the JavaFX frontend to the Spring controllers.

## **4.4 Core Security**

Since this was a prototype there wasn’t much need for security as the system is designed mostly for a demonstration rather than the actual working complete components, however there were still some security features that were required. For the Email service which sends out low stock notifications requires the username and password for the gmail account you are trying to connect, rather than having a real password on the application properties a passkey from google was used instead. This meant there was a reduced amount of sensitive data on the application properties just in case. Another place where security was implemented was for the H2 database, instead of using the default settings which automatically logs you in a password and username was added so only people with the correct level of access could access this potentially sensitive information.

## **4.5 Databases**

This is the database structure for each service within the DE-Store application.



# **5.1 Evaluation of DE-Store**

**Ease of Implementation:**

Due to the time constraints provided for this project the Three-Tiered Approach provides a far more complete prototype with a lot of quality features. This was possible as the Three Tiered approach provides a much more straightforward approach over Microservices architecture because it distinctly splits up each role into its own tier (Presentation, Business, Data). This is more suitable for a business as it decreases the cost of maintenance and client devices.

The Three Tiered architecture also uses a thin client this means it can be accessed easily from lots of different devices, such as tablets, smartphones and old/less powerful computers. This is important for DE-Store because it allows for a much wider variety in the hardware that can be used in turn reducing costs.

**Centralised Maintenance:**

With the business logic centralised within the application server, all bugs, updates and new features that need implemented can be done in a singular place, even if there was an error or bug that needed to be fixed this is a lot easier with the 3 Tiered Architecture as you can debug a singular location rather than many different services. By centralising the business logic for DE-Store it reduces the overall maintenance and management required further reducing the costs for DE-Store.

**Optimised Load Balancing:**

Since we can split the load into the different layers of business and data operations this allows the system to be far more optimised for handling a large number of transactions through the system. This is critical for DE-Store, if they cannot handle large amounts of transactions during busy and peak times, they will lose lots of potential sales and business which in turn would harm the company financially.

**Scalability:**

The Three Tiered Architecture is very easily scalable for any potential future business needs, since the system uses a modular approach this is even easier as modules can be added at any given time and provide whatever functionality is required for the system all whilst the original system can run normally and uninterrupted. The database structure also allows for future proofing as it uses separate databases for each individual service which allows it to be more efficient. Using individual databases also ensures there aren’t any single point failures this means if there is a problem/error in one of the databases it doesn’t affect the full system and only that service.

## **5.2 Future Improvements**

**Database Robustness**

Whilst the H2 database served its purpose for the needs of a prototype a more robust and persistent database such as MariaDB or MySQL would be better. By using a more robust database it would have improved scalability, data persistence and allowed the opportunity for more complex queries as well as a relational database.

**Real time Data**

At the moment the system uses a set of fake test data I inputted to give the illusion that the system has real customers and users however updating this so the database could work from a real dataset would have massively improved functionality.

**Functionality**

It would have been nice to have a fully functioning prototype however due to time constraints this wasn’t possible, features such as the update price and apply sale offer are only designed to show alerts with messages rather than actually updating price or applying any sale offers.

Report and Analysis needed a lot of work as well, currently there was a basic class setup to support this but no real methods or functionality which could have been very useful. For the Finance management there is a page created however it once again has minimal functionality due to the time constraints as well the loyalty card.

# **6.0 References**

<https://medium.com/design-microservices-architecture-with-patterns/microservices-architecture-for-enterprise-large-scaled-application-825436c9a78a>

https://techblog.cisco.com/blog/backyards-api-gateway