EECS 4413 Final Project Report

EECS 4413 - Building E-Commerce Systems

Section B

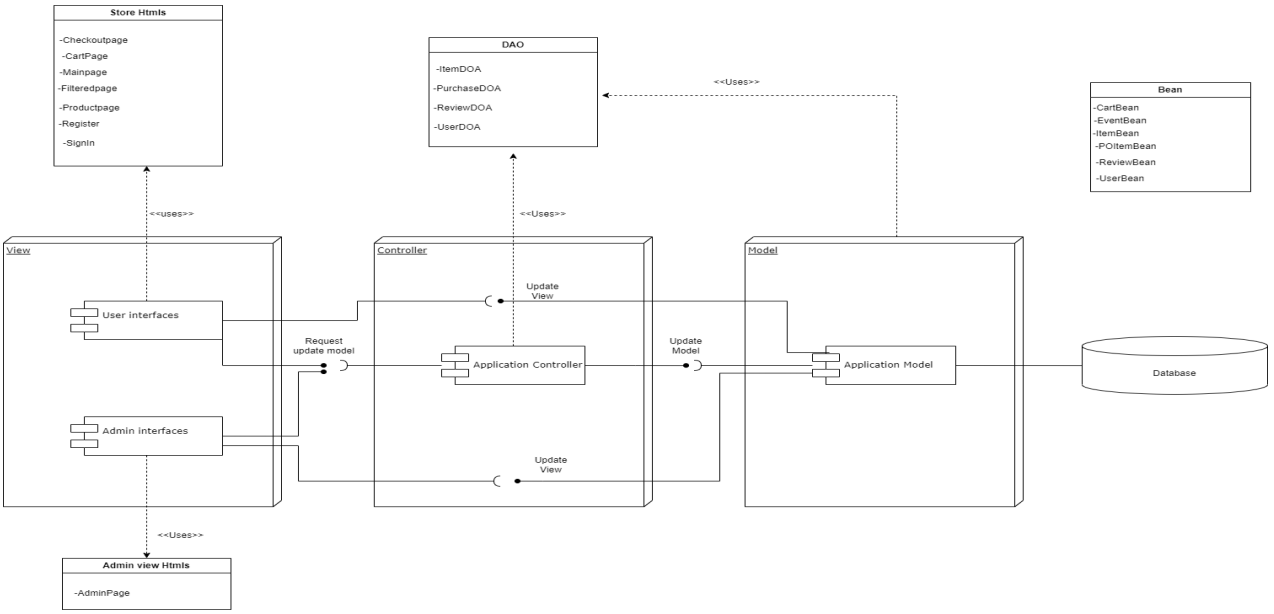
Team D
Ronny Blostein (215621915)
Anthony Iafrate (216743742)
Yudthesvar Raj (216846396)
Zohair Ahmed (215867633)

1. Introduction

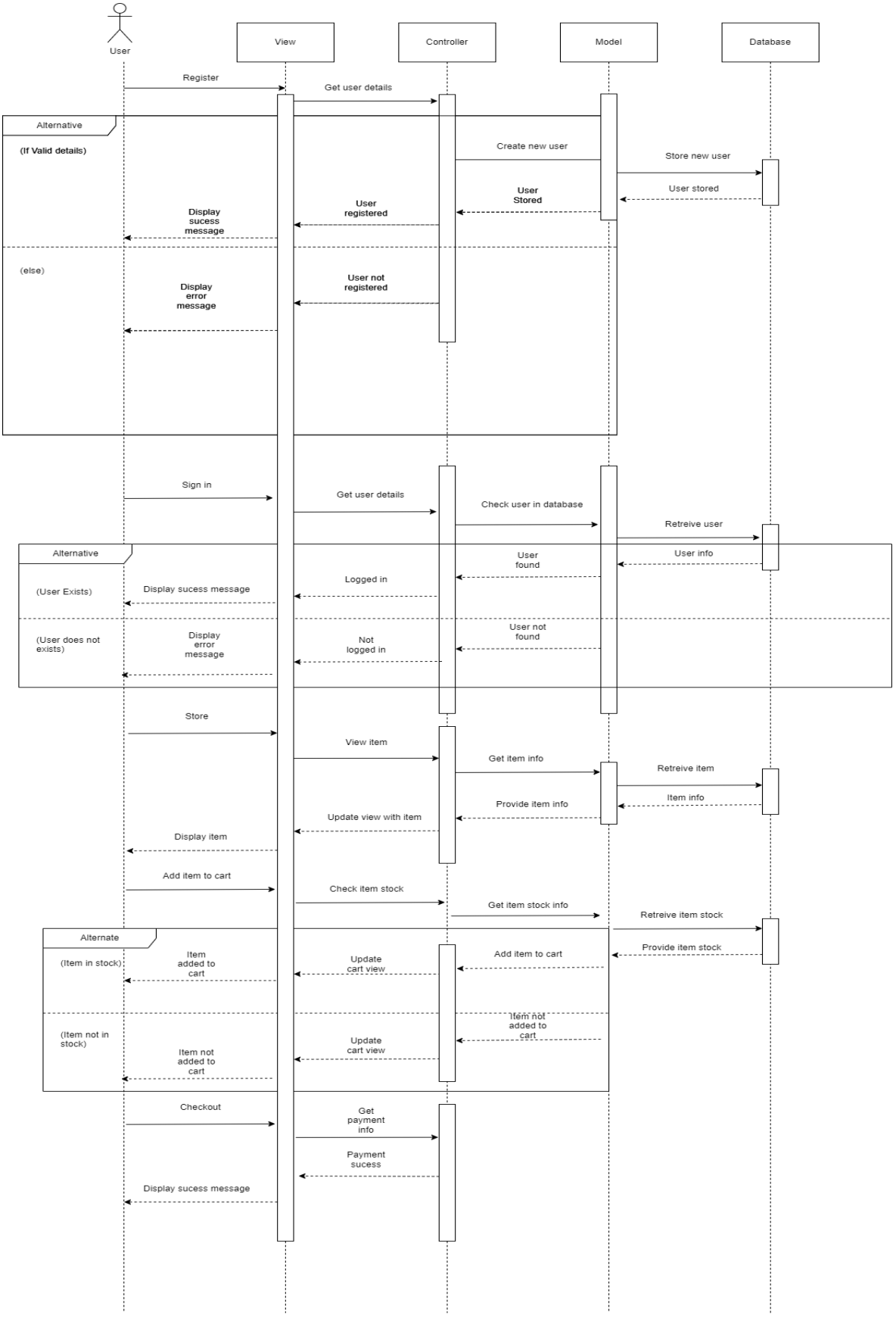
In this project, we developed a web application for an e-store. This application allows customers to purchase products conveniently at any time they desire. This application consists of many features. Customers will be able to register and sign in to their own accounts, filter products based on different criterias, review products that they have purchased and have full control of their online shopping cart and check out using their credit card we also added a unique use case which is where customers can filter products based on the prices of products. Administrators are also given access to tools where they are able to run reports on the number of products sold and the application usage.
This application was developed using the Eclipse IDE for Java Developers. Requests from the clients are handled using java servlets to generate a response. The web server and servlet container that we used was Apache Tomcat. In order to have clear separation of the front end and the back end, we used the model-view-controller architectural pattern. Using this pattern we were able to develop a multi-tier web application. To store the data that we needed such as the items available for purchase and the users information we used the MySQL database. In order to isolate the business layer from the application layer we used DAO packages. Our e-commerce application design is user friendly, as we have multiple different features and a user friendly navigation system. The application is also secure as we have implemented the necessary tools needed to deal with SQL injections.
The drawbacks of our design was that it consists of tight coupling, lack of scalability and a complex framework.

2. Architecture

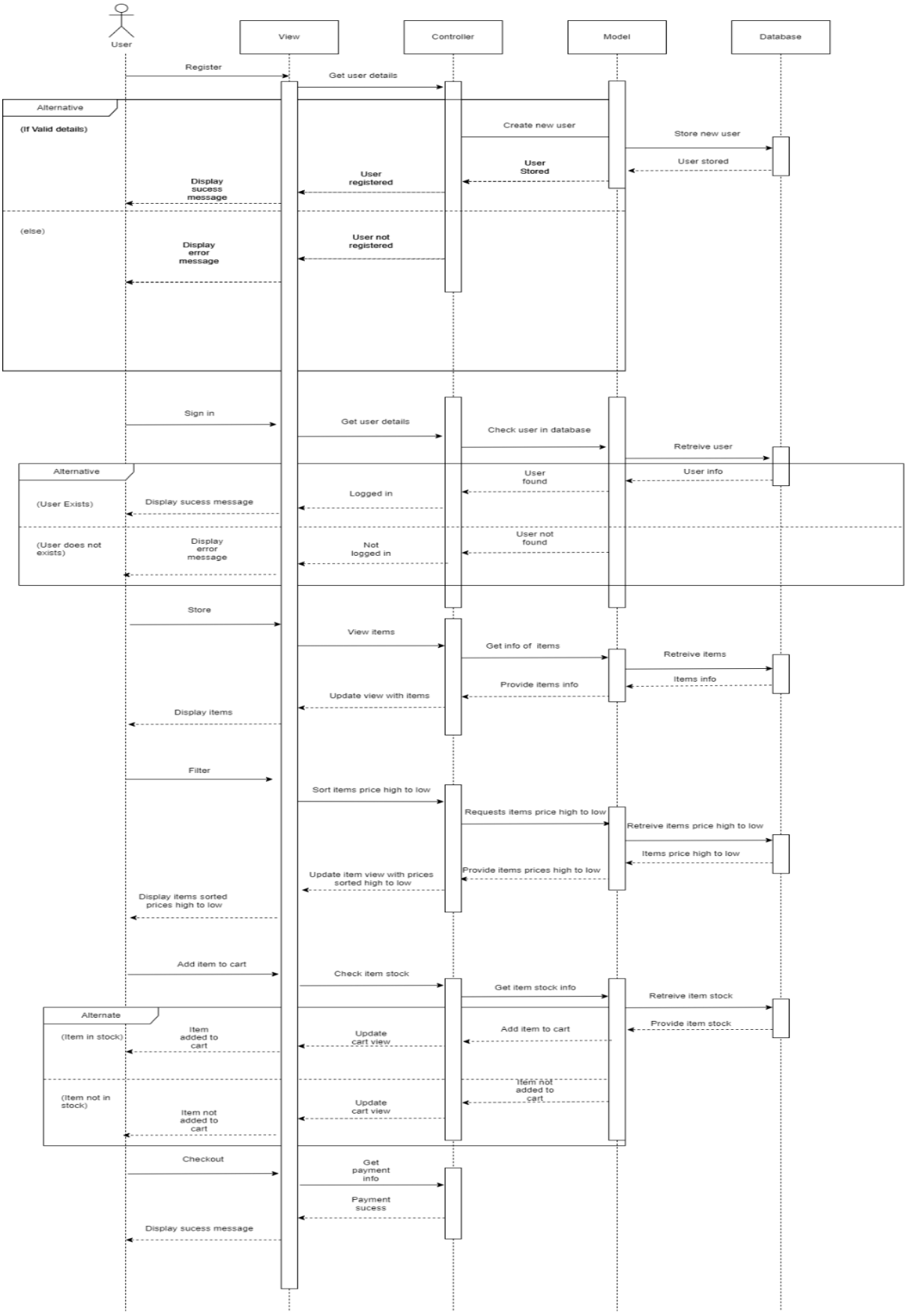
Component Diagram



Sequential Diagrams

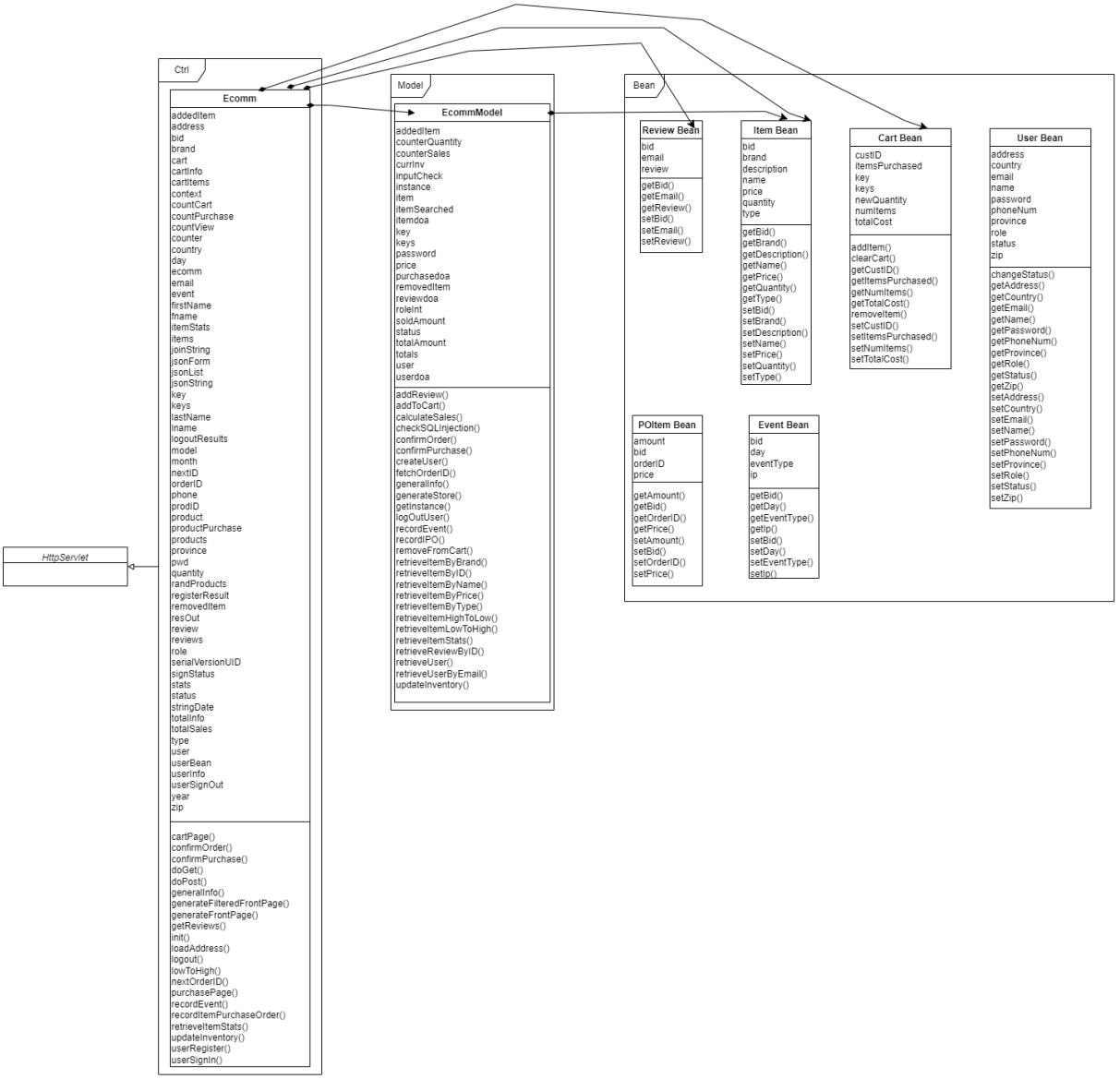


Sequential 1



Sequential 2 - Unique Use case

Class Diagram Of Application



3. Design Description

Model-View-Controller

The first pattern that we used is the model-view-controller architectural pattern, which is an architectural pattern that separates an application into three main logical components (Kaalel,2022 October 13). The three logical components that our application was separated into are the model, the view and the controller. The view section of our application consists of what the user will see and interact with. Multiple html files were created in order to display different web pages and javascript was used to allow the user to interact with the web pages. The model section of our application handles all of the data related logic that the user may work with. Here, several different functions were implemented in order to add or retrieve and make changes to the necessary information stored in the applications database. The controller section of our application connects the model and the view of our application, it

mainly tells the model section of our application what to do and display on the users view. The main reason for using the mvc architecture was due to the numerous benefits that it provides. Using this architecture it would be easier to maintain and extend the code of our application, testing would also be easier because components can be tested individually as they are independent of each other. The complexity of the application is also reduced due to the fact that the application is divided into three different logical components, this also makes it easier for the application to be split up and worked on by multiple people. The tradeoffs of using this architecture was that navigating through the framework was complex at first as each group member needed to adapt and learn about each layer of abstraction and parts of the code that were implemented by another member.

Singleton

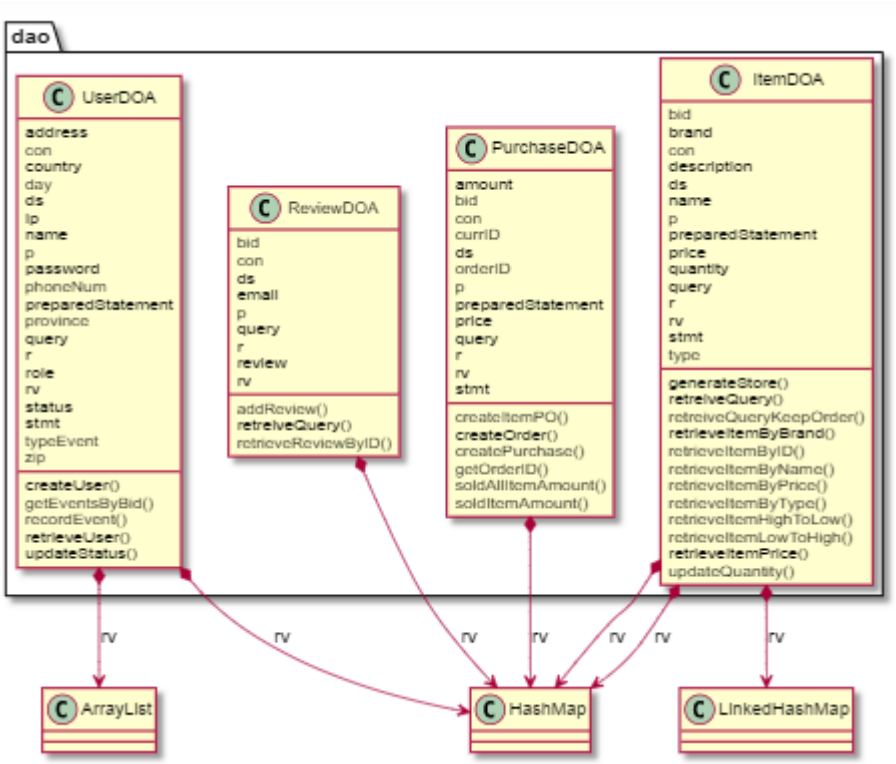
The next pattern that we used is the singleton pattern, this pattern was used when constructing the model section of our application. The singleton pattern is a creational design pattern (Zhart, 2022), that allowed us to make sure our model class had only one instance with a global access point. Using the singleton pattern we were able to make sure that there was only one instance of the model class, if a model was not created a new one would be created or if one already existed it would be used as the model and only requires one time initialization.However using this pattern had some drawbacks the first being the introduction of dependencies,as the singleton class is promptly accessible all through the code base, it tends to be overused,thus this makes it hard to track the class due to its reference not being completely transparent. The next drawback is due to the tight coupling between the objects and the methods. In the future there may be complications when it comes to trying to test the code using unit testing and also with the scaling.

DAO Packages

We wanted to isolate the application layer from the business layer as best as possible when it came to creating the model for our project because we knew we had to deal with many queries and data storage but we had relatively simple business logic, so we went with a Data Access Object (DAO) structural pattern (as opposed to the repository pattern). The difference between the two in terms of definition is the DAO pattern is “closer” to the database (abstraction of data persistence, data-centric) while the repository pattern is more closer to the business logic (abstraction of a collection of objects, business-centric).

In choosing the DAO pattern, we were able to hide complex queries however we lost the benefit of the repository pattern which is to hide the complexity of combining and collecting data. In the DAO package where the program communicates with the database and performs different functions for objects such as users (add or get user), items (get or sort items), review (get or add item reviews) and purchase (create purchase).

Class Diagram for DAO pattern



4. Implementation

Monolithic Architecture

We went with a **monolithic** approach. There were several pros with this approach. It is easier **deployment**: we would have only one executable file, making it easier to deploy on the cloud. It is easier **to develop**: everything is on one codebase making connections easier, this would also mean that we only need one API rather than numerous. It is easier to **test**: Since everything is in one centralised repository, it is easier to run end-to-end tests. This also made it easier to debug since there weren't multiple systems to work with. It is **cheaper**: Since we knew this application had to be deployed to the cloud, we knew we would need cloud resources. Rather than paying for multiple hosting infrastructure, we only need one. There are cons to this approach as well. It takes **longer to build** the project, which is troubling during times of quick changes. **Scalability becomes difficult**: You can't scale individual components. It is also less **reliable**: If one part of the program does not work, the entire system collapses.

Servlets and JSP

Java Servlets and JSPs are tools used to generate dynamic web pages. Java Servlet and JSPs are server-side technologies to extend the capability of web servers by providing support for dynamic response and data persistence which is good when we generate dynamic web pages, save into databases, etc.

Model Package

The model package is used to create the business logic for the application (such as retrieving users and creating events) but the data storage mechanism has been abstracted

due to the use of the DAO structural pattern. We felt this would make the Model package easier to read and develop due to the “separation of concerns”

Bean Package

The bean package where the fundamental entities of objects are stored. This is what would be used by the controller to notify the model of changes. For example, the DAO objects would return a mapping of the specified parameter to the Bean objects. For example, if the query specified to the information on a user with a specific email, it would return a map of the email to the UserBean

Why we chose Bean over POJO: POJOs do not follow any real convention for constructing, accessing, or modifying the class's state which causes 2 problems: harder learning curve, and harder to understand how to use the class. With Bean, we still have a POJO but there are more strict implementation rules such as method names, access levels and default constructors, which can help us predict the API of objects making them easier to communicate between the subsystems of the MVC architecture

The limitations of Bean is the potential of boilerplate code since you need accessors and mutators for all parameters and the need for code of concurrency (since objects are mutable). However, due to the simplicity of our objects and the separations of concerns we had created with the above patterns, we felt like the pros of JavaBeans outweighed the cons.

Controller Package

The controller is responsible for sequence of interactions with the user and notifying views of changes in the model, essentially where we bring everything together. Using the URL endpoints and queries, we adjust the EcommModel.java file to comply with the request of the user. For example, the EcommModel is able to obtain the list of items from the database, but if the user wants the list sorted by price (which is identified using the query parameters), the controller lets the EcommModel know to return the list of items as you are able to, but this time in sorted order.

Database

For the database, we opted to go for MySQL. The pros are that it is a relational database, this means that we can normalise the data and not worry about boilerplate or repeated data like in non-relational databases (we felt this would suit our needs since there was a lot of connectivity between our objects). Also, AWS has RDS, meaning we would still be able to deploy it as a cloud database. Thirdly, MySQL has a lot of online information making it easier to learn. Last but not least, It deals well with encryption.

The cons of using MySQL are that it is not good for large amounts of data (which can hurt scalability of the project) and the robust scheme definitions are difficult and time consuming.

In our case, since we did not have a large number of objects to store in a database and we had the time to create normalised schema definitions, it was easy to choose SQL as our database.

Front-End

The appearance of the web pages were all written in HTML and were structured based on the requirements and or purpose of each page, and the functionality of each web page has a respective JavaScript script. We were able to remove boilerplate code where pages had repetitive functionality. We feel like this helps readability of the code, and makes it easier to maintain as the site grows in scale.

Cloud

In terms of cloud deployment, we had to deploy the app itself as well as its database. To deploy the cloud database, we used AWS Relational Database (RDS) since we had opted to use SQL. We first created an instance of AWS RDS that is deployed for public use. Then we established a connection to this public cloud database via its generated URL in MySQLWorkbench. We ran our queries on MySQLWorkbench which was populated in AWS RDS. For proof, *AWS RDS Monitoring* in the *6. Performance Report* section. Once we had the cloud database populated, we had to connect our app to it. This was done by changing the URL in the `context.xml` file to the one of AWS RDS. Here was the generate link: `eecs4413-db-instance.cj7m5thvogjc.us-east-1.rds.amazonaws.com`

The portion to deploy the app itself was not able to be done. The first route taken was deploying AWS Elastic BeanStalk, however, this did not work because for Tomcat projects, only Java 8 and 11 are compatible, but the version the project was compiled in was Java 17. There were several attempts to compile the project in JRE 8 and 11 but to no success. The next option was to deploy the app using Heroku, however, Heroku only accepts Maven applications, which our application was not. There were several attempts to convert the project into a Maven project but to no avail.

Testing

In terms of testing we did find a few limitations. We noticed that some test cases should not be tested through curl because they depend on previous actions and validations that are not done in the one bit of a curl command. For example, you cannot add to cart (and have it produce a meaningful output) without being first signed in, viewing or adding a product and purchasing. To complete a transaction you need a cart that is assigned to an account, which is unique per account per session and cannot be completed through a curl command without doing the other actions that are checked in the monolithic app as a whole.

We also noticed that curl commands are good for unit testing. However when it comes to other tests such as UI, end-to-end, integration testing and load testing, curl command testing is not a suitable option.

5. Security report

Authentication
We set a role option (0 = admin, 1 = user) during the time of user creation, the appropriate role would be checked before certain actions to ensure only the admin had access to capabilities such as when the administration link in MainPage.html through the mainpage.js script or the ability to access different pages as specified in signinregister.js.

Authorization
We have a handler method in signinregister.js to ensure those who are users who are signing in have already authenticated their email. If they are authorised, they will be able to sign in and continue their specified functionality for their role, else there will be an error message prompt.

Confidentiality
We check for role access before (using the above methods) to ensure that users do not have access to removing or adding products to the main database. They also do not have access to the certain statistics of the product.

Data Integrity
We checked for SQLInjections to prevent database manipulation. For example, in EcommModel.java, before any database operation to the user was going to take place, the method SQLInjection is called, if an SQL injection was found, we would return -2 (to let us know the error is specifically an SQLInjection) and revoke the request. Lastly, we also ran a security scan using HCL AppScan available in VS Code to run a security scan on all our files. This gave us some security vulnerabilities but we tried to remedy them to the best of our ability. We also selected the encryption feature for AWS RDS, for further safety measures.

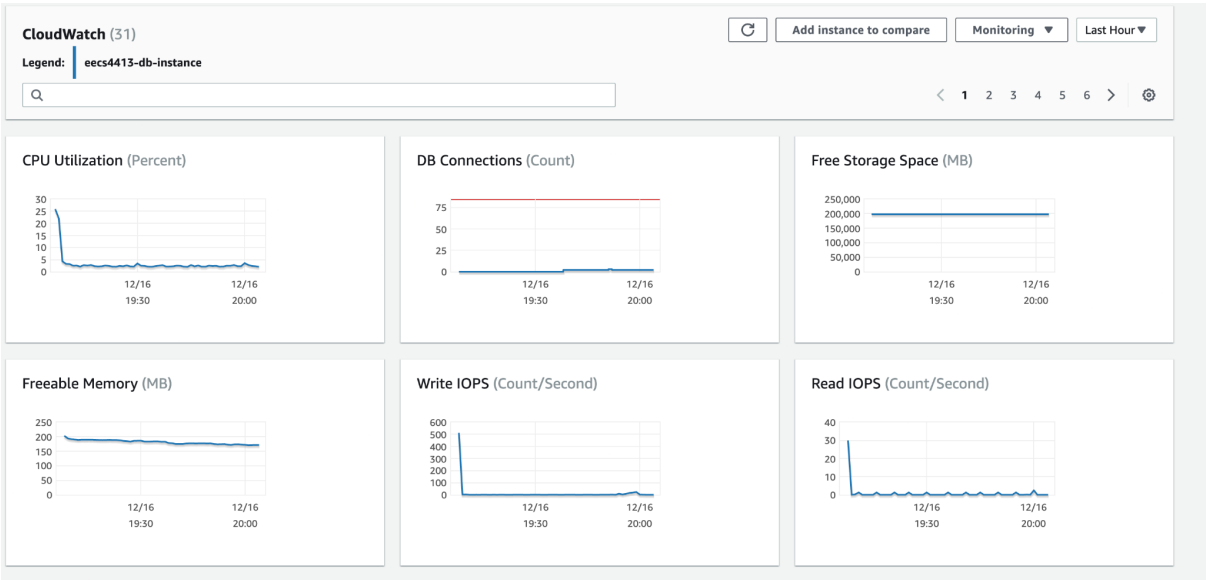
Quality of Service
We checked for Cross-Site Scripting by verifying URLs to check for malicious scripts in Ecomm.java.

Non-Repudiation
In our database, we would store the statistics of purchases, such as who it came from and the time of purchase to prove that a user made a transaction. This can be seen in UserDAO.java.

Auditing
This is something that we did not do and would be an area of improvement for the future.

6. Performance Report

AWS RDS Monitoring



Space Usage of AWS RDS

7. Contributions

Ronny Blostein (215621915)
This member contributed to the website. They developed and designed both back end and front end components as well as setup the database for the project.

RB

Anthony Iafrate (216743742)
This member contributed to the website. They developed parts of both the front and back end of the website.

Yudthesvar Raj (216846396)
This member contributed to the report. They wrote the introduction,design description,conclusion and made the uml diagrams for the architecture section.

Zohair Ahmed (215867633)
The main task of this member was to work on the cloud portion of the project. They were able to successfully connect the application to the AWS RDS cloud database. They were unable to deploy the project on Cloud however. They also wrote the sections on implementation, security and performance for the report.

8. Conclusion

In conclusion we managed to develop a working e-commerce system that consists of multiple different features. Customers of the e-store are able to make their own unique accounts and purchase, filter and leave reviews on products that they desire using this system. Communication between each member of the team went well, we were able to discuss and convey our ideas to one another smoothly. Even though most of the project went smoothly we did have some drawbacks when it came to using GitHub but we managed to resolve the issue in the end. Through this group project we have learnt a lot, we learnt how to work together as a group, how to convey our ideas to one another. We also gained more knowledge on the different e-commerce architectures and how to apply them. Working in a group presented us with the opportunity to split up different tasks among the group members which saved us time and enabled us to complete the project in a more efficient way. Some of the drawbacks of working in a team were the conflicting schedules that each member had, it was a challenge to set up meetings where everyone could be present.

References

Kaalel. (2022, October 13). *MVC Framework introduction*. GeeksforGeeks. Retrieved December 11, 2022, from https://www.geeksforgeeks.org/mvc-framework-introduction/#:~:text=The%20Model%2DView%2DController%20(,development%20aspects%20of%20an%20application.

Zhart, D. (n.d.). *Singleton*. Refactoring.Guru. Retrieved December 16, 2022, from https://refactoring.guru/design-patterns/singleton