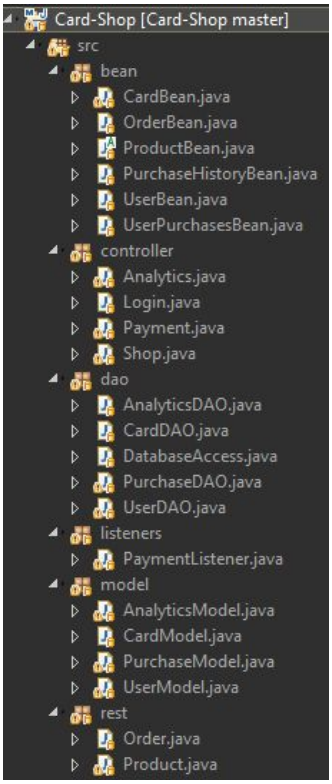
Card Shop Software Design Document

First Name	Last Name	Student #	email	Signature
Allen	Kaplan	215494925	allenkap@my.yorku.ca	AK
Jeremy	Winkler	214915854	jeremyw@my.yorku.ca	JW
Connor	Ahearn	215141245	ahearnc@my.yorku.ca	CA
Damanveer	Bharaj	214946701	damanb19@my.yorku.ca	DB

Table of Contents

Project Overview

About the Card Shop



The Card Shop is an interactive e-commerce platform based on the fictional store from the television Series *Hunter x Hunter*. The Card Shop is an interpretation of the card shop from the series with all cards available for purchase. Cards are represented by an id, a class rating (SS, S, A...F), type category and limit (on the number of cards that can be purchased). Users are able to query the store through a search box or by card categories such as class rating or type. Further, the platform provides REST endpoints for users who are looking to get card and order information directly. Through REST endpoints, partners can integrate the Card Shop platform into their services.

Architecture

Overview

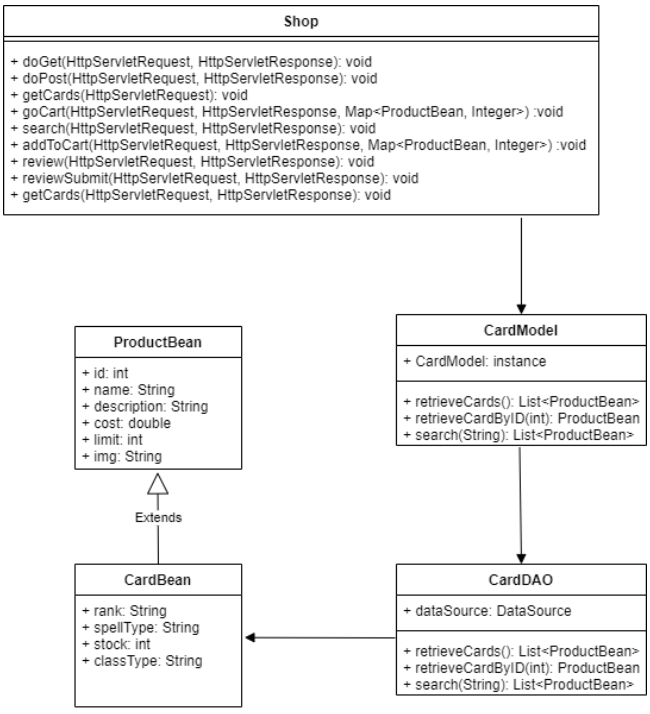
The technical design of the Card Shop was based on the Model-View-Controller (MVC) architecture as taught by Professor Marin Litou as a part of EECS 4413, E-Commerce Systems. The design through MVC was led through creating Plain Old Java Objects (POJOs) to represent the business logic of the products. T

Model

The POJOs are represented by beans within the beans package. Once the beans were made, there were models implemented to perform the functions needed by a controller.

View

A view was implemented through data access objects (DAOs) and the JDBC connector. Through a DatabaseAccess class in the DAO package, the architecture abstracts the database implementation allowing for local and cloud deployment with JDBC compatible SQL RDBMSs.



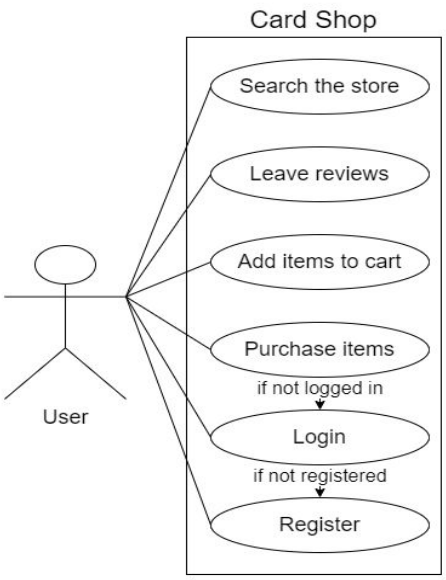
Controller

Controllers were implemented within the controller package. There are four controllers that are used to manage the shop, payment, user login/authentication and data analytics.

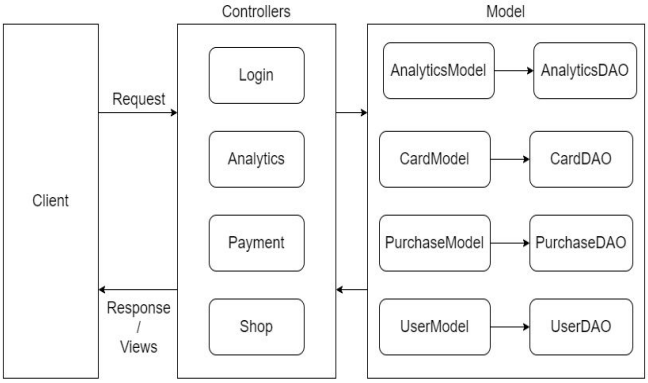
UML use cases

General User's Use Cases

As a user, they have many options that they can choose from when first entering the store. They can start searching, leaving reviews, and adding items. From then on, they can either leave the site or continue with the purchasing by logging in or creating an account. Only once logged in can the user then proceed to purchasing the items which they added to the cart.



Class Diagrams

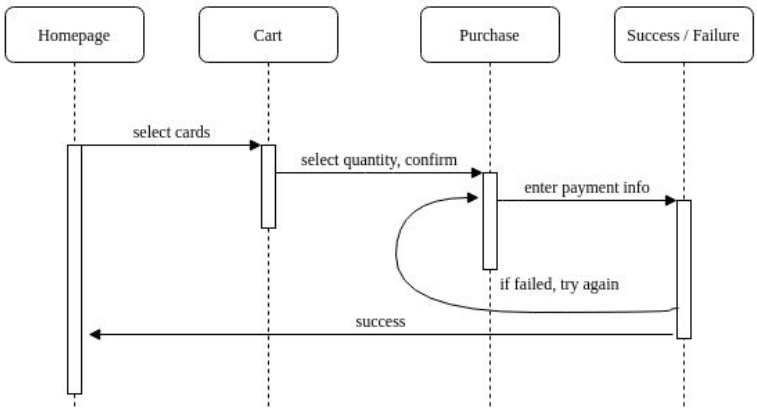


For a more general diagram to show how our system communicated with each other, we have the diagram above. This shows our architecture of having the users interface with the controllers through requests, then the controllers were able to communicate to the models which each had the singleton pattern implemented. Only the models were allowed to have direct access to the DAOs. Once the controller had the necessary data from the model, it returned the views. We decided to go with this structure since we believe that MVC is a good way to ensure we have separation of concerns throughout our code.

Sequence Diagram

Making a Purchase (Logged in user)

For this diagram, login was omitted due to the application's flexibility. There are more than 1 point in the above sequence where a user could either register or login to their existing account. This level of branching that adding login to the sequence would bring would take away from the readability of the diagram.



Database Design Decisions

The database schema was broken down into 3 sections, the Cards, Users, and Purchases. Each section (only one schema was allowed on db2) contained its own respective tables for its respective definition. For example, Cards contained a table for storing all the cards that would be potentially sold, followed by more tables such as CardMarket that would hold business information such as price and stock. The Users section would hold user information such as logins, accounts, addresses, etc. Finally, the Purchases section stored information about orders being made for cards.

Cards can be indexed using their card number as the primary key. This is because the card number is a unique value that will never change. Users are indexed by their username, which is designed to be unique. A user's orders are associated based on their username, which can then be grouped by date purchased, card numbers, or other order specifications.

In order to handle SQL injections, the database was accessed through our java code using PreparedStatments. The use of the PreparedStatments allows us to make clear distinctions in our SQL queries on what portions of the query are intended SQL and what are parameterized arguments. This is done through the use of the '?' character in sql characters. When generating the PreparedStatments, we specify the index of the '?' character and which argument will take its place. Using this, the JDBC will handle the PreparedStatement in a way that could prevent many types of SQL injections.

Development Operations

Version Control: Github

The Card Shop application was developed using a Github repository. The repository is cloned onto our local IDEs. IDE tools along with Git Bash were utilized to access the repo. By using version control, the team was able to revert bad commits and work concurrently.

Deployment: IBM Cloud Foundry

The Card Shop was deployed on the IBM Cloud Foundry as a Java Liberty application. The approach to accomplish replicates that of the notes provided in class. The application is developed using Eclipse IDE. When we wanted to deploy on IBM Cloud, I pushed the changes by running the application as a server on the cloud foundry.

One limitation of deploying on cloud was the time it took to fully deploy the application. It would take roughly 5-10 minutes per delivery. Due to this, it was not feasible to continuously run our source code changes on cloud to test. As a result, we developed and tested our application using the local tomcat server and when a major milestone was met, the application was deployed and observed.

The IBM cloud site provided me with important resources for managing this foundry application. There is a log which allows me to see errors and java exceptions that occur. This way my team can understand what went wrong on the cloud when unexpected behaviour occurred.

Database: Cloud DB2

The DB2 Lite database service on the IBM cloud was used to complement this application. This database provided a nice console to easily develop tables and run sql statements. This service is connected to the cloud foundry app.

Given that this is a free lite version of the DB2 database, one of the major limitations for developing and deploying with this service was the max number of instance connections allowed to the database. Only 5 instances of this database are allowed to be active at a given time. This included local tomcat server connections, browsers with the website open, and connections with the ibm cloud console. As a result, we found ourselves continually pausing our work requesting for someone to finish with their connections so that we may continue our work. It should also be noted that if someone wishes to access the website, if too many connections are in use, the website will not be able to retrieve anything from the database and will essentially not work.

Implementation

Implementation Decisions

Overall, the design of our web application closely follows the MVC design pattern taught through the course material. This can be inferred by the folder & project structure of the project, as seen on the right.

Bean classes are used as data types throughout the system, their attributes are private and are accessed through getters and setters.

For controllers, we took a slightly different approach than we did in class. As a group, we found that the Java platform was better suited to have multiple controllers than having a limited number of them like we did in the labs. This led to us having less nested if statements in our doGet() methods, which made the code much more readable and easier to work with.

The Card Shop team stuck true to the DAO / Model design pattern taught in class. The only way that we swayed from the prof's examples was through the Singleton Design pattern. We ensured that all DAOs as well as Models were stored in memory as a Singleton, in order to ensure there is only ever one instance of any such objects.

The REST interface was designed using Jackson as well as Jersey, again following closely with what we learned in class.

Limitations

Design Limitations

By splitting up the controller design from class into multiple controllers, we learned about the advantages of limiting your controller count vs having many. We thought in any case that having more controllers would lead to cleaner and simpler code, but in reality there's a balancing act of selecting the right amount of controllers in order to keep a clean design.

We found that with fewer controllers (like in the labs) shared data between different pages and functionalities was very simple to implement, and took very little bookkeeping. Storing information in the class field of a "supercontroller" ensures that any page has access to that information.

As you can imagine, as we added more controllers, the sharing of data became more complicated, but our logic became much simpler. There were more interactions between controllers with the Session and ServletContext, but in turn, there was much less indentation and global variables to keep track of when looking at a specific block of code.

Technology Limitations

Overall, the JavaEE framework is a tried and tested framework that has been around for years. We're aware that many people use it and some people prefer it over other web infrastructures, and we're fine with that. However, we found that this choice of web framework caused more problems than it did help us.

Working on a JavaEE project is like a research project. You can't write a single sentence (or in our case, line of code) without citing some other source where someone explains the topic at length. Even for experienced Java

developers like ourselves, the libraries used for this project were found to be very unintuitive and not user friendly at all.

Some examples of the frustrations of the required platform:

The console output of your server, regardless of its complexity, will almost always be shooting out exceptions throughout its runtime. Sometimes they crash your server, sometimes they don't. It's up to you to try your luck, and settle your anxiety as you deploy your final project and hope for the best!

The abstraction layers of Tomcat, JSP, JSTL are a mess of build path errors in the best case scenario, and commonly lead to a bunch of unhandled exceptions (the server crashing kind) for tiny HTML mistakes, which in any other backend framework aren't even mistakes. This suite of tools that we assume were designed to abstract the system and make developer's jobs easier have evolved into these buggy, undocumented fossils that rarely have a recently updated source for help.

Another platform we had a lot of trouble with was the IBM DB2 free database service recommended by the professor. The problem with the "free" service was that it would only allow up to 5 devices / services to access the database at a given time. At face value, this sounds fine. There's only 4 people in this group, and 4 < 5, so they shouldn't reach that limit realistically. *WRONG!*

During our last week of working on the project, we were constantly being locked out of the database due to too many people accessing it. 4 people were accessing it through more than 5 servlets at a time. Why? Because JavaEE / Tomcat seem to run our servlets in the background on more than one port at times, without any indication or an explanation as to why that's happening. Many times it was found that students had to just restart their computer to ensure they weren't the one running the extra service that was putting us over the limit.

Oh and did I mention the cloud deployment also takes one of those slots? Because it does. So if there was ever a single extra servlet instance running, we all had to terminate Tomcat together, rerun the server, and hope that someone didn't have an extra process of our server running, otherwise we would have to all restart our computers.

Database: JDBC

The Java Card Shop application connected to the DB2 database using a JDBC driver provided by IBM. The approach for setting up and installing the JDBC is similar to that provided by class notes for connecting to the DB2 database. The DB2 database on IBM has its own console for executing SQL code. This made testing our queries and updates easy. However, these queries would then have to be implemented into our java application's DAOs via JDBC.

The Java application required two different JDBC data source connections. One connection that would allow a local tomcat server to connect to the db2 and another so that the cloud foundry can connect to the db2. This was handled by creating a singleton class called DatabaseAccess. This class contained an instance that specified which datasource the JDBC should use to connect. All DAOs referenced the value of the DatabaseAccess instance. For example, whenever we wanted to deploy, we simply changed the value of this instance to DEPLOYMENT_ACCESS. All DAOs would then be configured for cloud deployment access. As a result, we did not need to change each individual DAO class datasource everytime we deployed.

Testing

REST Testing

For testing of REST endpoints, the Card Shop team used Postman. Postman allows for easy testing of local REST endpoints during development. To ensure that the endpoints are consistent on the live, deployed Card Shop, the client.jar tester was made. The client.jar is located in the test-clients folder. By executing `java -jar client.jar [cardID]`, two endpoints will be queried for that cardID, `rest/product?productID=[cardID]` and `order/?productID=[cardID]`. Example output is attached below.

```
$ java -jar client.jar 3
Sending Request to https://eecs4413-cardshop.mybluemix.net/rest/product?productID=3
Status 200
{"id":3,"name":"Pitcher of Eternal Water (Pot of Spring Water)","description":"A jar
 from which pure, clean water (1440 L per day) continually flows.","cost":5.88000011
4440918,"limit":17,"img":"https://vignette.wikia.nocookie.net/hunterxhunter/images/3
/3d/Pitcher_of_Eternal_Water_%28G.I_card%29_%3Dscan%3D.png/revision/latest?cb=201312
02205733","rank":"A","spellType":"null","stock":17,"price":5.880000114440918,"classT
ype":null}
Sending Request to https://eecs4413-cardshop.mybluemix.net/rest/order?productID=3
Status 200
[{"cardNumber":3,"cardName":"Pitcher of Eternal Water (Pot of Spring Water)","purcha
sePrice":5.0,"quantitySold":1,"image":null}]
```

Security Testing

As explained by the professor, SSL is taken care of by IBM when deploying on the cloud. However, this is not the only security issue.

mlit oiu
when deploying on cloud, SSL is taken care of.

Let's say someone tries to inject some SQL into our program as shown below.
The user would be met with a friendly message indicating that users cannot add SQL commands.

Team Member Contributions

Overview

This team consisted of four members. The workload was broken down such that each person worked on completing a different milestone of the project. Given the circumstances of quarantine, most collaboration was done online. The team communicated regularly using online text chats and made video calls once a week to discuss progress and goals for the next week.

Individual Contributions

Connor Ahearn

For this project, I was in charge of the Cart functionality as well as the Analytics functionality. How our group worked was by assigning a set of use cases to each group member, and these were assigned from a “full-stack” perspective, meaning we would work on both the front-end and the back-end.

For me, this project was really cool to see a complicated database “in action”. As someone taking EECS 3241 this semester, I learned a lot about databases this semester, but did very little implementation leveraging an actual database. It actually makes these kinds of projects much more straightforward in a lot of aspects. It also introduces an aspect of teamwork to the project that I enjoyed!

Allen Kaplan

Throughout the project Allen worked as the project manager and lead developer of the shop and reviews. As project manager, Allen worked to schedule consistent team meetings and set project goals. Further, Allen developed the initial platform for MVC and the outline for the Software Design Document. In regards to the technical implementation, Allen managed the implementation for the central Shop controller which allows users to browse, search, review and add-to-cart cards. To do this, Allen developed the MVC structure for the cards, allowing for CRUD operations to power the platform. Through working with others, Connor and Jeremy were able to ensure the payment and user authentication worked with the shop respectively.

Although a verteran cloud developer, many tools were new to Allen. Through working with Daman, Allen was able to learn about how to deploy to the IBM Cloud, how to use the cloud instance of DB2 and how to manage development operations on a team of developers. Further, Jeremy and Connor’s skill in full-stack development allowed Allen to learn ways to implement a user friendly front end.

Damanveer Bharaj

My role in this project was database and deployment. I created a DB2 database service on the ibm cloud and connected it to a cloud foundry app where our application is deployed. I developed and implemented the schema for the database. My team members were provided access to the DB2 so they can see the schema as well. Essentially, I handled all requests that needed use of the database and deployed changes to the cloud as needed.

I communicated with my team to understand what database elements were needed and created the respective DAO query/updates for them. By doing this I got a better idea on what elements my teammates were working on

and how I can help them accomplish the goal. Overall, I found managing the database and deployment to be a fun task that I learned alot from.

Jeremy

For this project I mainly worked on login, registration, and payment. Working on these three components was a good idea since it is mainly when people go to pay that they have to login. The payment page had to be integrated with the cart page to know what was being purchased. The analytics page also had the requirement that only administrators could access it. Other than this, I helped out with some other things such as some work on the cart page, and some database work as well.

To learn about what other members did, we had our weekly meetings and we often communicated with messenger to update each other on our progress.