PATSATS

Production and Test Servers are the Same

EECS 4413 Final Project

Group members: Saad Saeed, Matt Maceachern, Tyler Noble

Architecture

UML Diagram

Sequence Diagram 1

Sequence Diagram 2

Patterns Used

Model View Controller

This project utilized separation via the Model-View-Controller (MVC) design pattern throughout its design. Each class is organized into one of the following packages: model, controller, as well as test and bean packages. Furthermore, the bulk of the computations happen in the model package, using multiple Data Access Objects (DAOs) which interact with our database and retrieve information, which is then called by the controller. In addition to this, the view in MVC is contained in the WebContents folder, which contains all of the .jspx files along with any pictures, javascript files, and .css files.

Event Driven

In addition to above, event driven architecture was used heavily in the implementation of our system. Events are generated first through the frontend of our system and from there are carried on through the DAOs and processed to eventually become output. This is all done through the use of HTTP requests. Since we have a servlet for every function, every servlet is event driven.

Microservices

Our loosely coupled design is built around the ideas of microservices. Microservice architecture outlines the pattern of a loosely coupled set of services that structure into a single application. These microservices are highly modular and make testing and development much more resilient to bugs. Microservices frequently rely on network protocols to communicate for communication, in our case this is our HTTP requests. This is also the reason that we have many model and controller classes with a small amount of methods used as this architecture requires small and fine grained classes to perform a single function.

Decisions/Tradeoffs

The MVC is a standard design pattern for this usage scenario. However it is not without drawbacks of increased complexity in working between the front and backend, as well as knowledge of a breadth of categories required to make everything work. In particular, getting information to be put in the view or taken from the view can be a challenging part of implementing MVC architecture. With that said, the advantages are fantastic for working with a team. Team members have been able to specialize in certain areas ideally to bring their focus forward on a single part of the project. This has also made dividing work much easier between our group and allowed us to branch more into our respective roles set out.

The microservice architecture brings a lot of benefits as far as debuggability, as well as the modularity of the system. By introducing this architecture we have time and time again been able to easily locate bugs quickly and efficiently by checking only associated servlets without having to go through the whole system. However this has made us rely on HTTP requests in order to transfer data between the system, and some functionality that may tie together the system has been more complex to implement.

Implementation

Decisions

As mentioned before, the overarching principle undertaken while developing the project was that the architecture should be easily maintainable, modular, and not have any redundant components. Throughout development, we have tried to stick true to this approach by enacting the following implementation decisions:

- Each .jspx page has a distinct servlet dedicated for its functionality. A list of pages with a dedicated servlet is shown as follows:
    - Main page
    - Login / register page
    - Shopping cart page
    - Individual book page
    - Checkout page
    - Analytics page
- Use a combination of SQL queries and listeners to power the Analytics page.
- Using prepared statements to access information from the database. This will protect form SQL injection and minimize the risk of security threats.

Database

Figure 0. Database schema design

In order to implement the database for our website, we opted to use an external database manager called XAMPP in order to ease the construction and administration of the database itself. This tool allowed us to build our database using the user interface provided with

the application which allowed us to focus more of our effort on the actual website. In this section we will discuss how we connected our website to XAMPP through JDBC, as well as the design deviations we made from the original MySQL database provided in the requirements document.

In order to connect to XAMPP, we simply created a connector class called DatabaseConnector.java in our model package which uses JDBC in order to connect to our database with the url "jdbc:mysql://localhost:3306/bookstore_db" and username “root”. We decided to use MySQL as the language for our database, which is why we used port 3306 to connect to our database. In order to connect to the MySQL database, the MySQL connector jar needed to be added to our project build path. The jar file can be found on the MySQL website: https://dev.mysql.com/downloads/connector/j/8.0.html. In order to set up our XAMPP to allow Eclipse to connect to our database, we had to slightly modify the default XAMPP settings through xampp/mysql/bin/my.ini. Specifically, the bind address needed to be uncommented and set to 127.0.0.1. For details, the following tutorial was used in order to get XAMPP working properly: https://www.youtube.com/watch?v=kxIg0OUo4eI.

To import our database bookstore_db.sql, you need to start XAMPP’s Apache and MySQL server and then click the “Admin” button beside MySQL (see Figure 1 below). This will open phpMyAdmin where you will see a menu at the top of the page. From this menu you can choose to import the bookstore_db.sql file by pressing the Import button shown in Figure 2 below. After importing the database, the website will be able to connect as long as the XAMPP tool is running.

Figure 1. XAMPP running Apache and MySQL

Figure 2. Import button in phpMyAdmin used to import the database

There were several deviations that were made in the design of our database for the purpose of our bookstore. The first deviation was in the account table, where we needed to add an extra field pass_salt, which is a randomly generated integer used as part of the hashing process for our users’ passwords. We felt that storing passwords as plain text was a huge security issue that was fairly easily fixed by hashing. The next deviation was in a new table called category. When thinking about how books would be stored early on, we decided that it

made more sense that a book would be in one or more categories rather than restricting it to a single category. This required us to make a separate table so that book IDs could be associated with many categories. Finally, an additional table we decided to include in our database was the shopping_cart table, which stored information on the contents of all of our users' shopping carts. We felt that shopping cart persistence was important, and that storing the shopping cart in a cookie or the browser's cache would not be durable enough for the purpose of our website. Therefore when a user adds an item to their cart, AJAX is used in order to add that item to the database and associate it with their username.

Trade offs

Much of the site is loosely coupled due to the decision to create every class a servlet. This makes debugging much easier. For example, say the categories page is not displaying properly. The first step would be to check the functions are called upon loading the page. In a more tightly coupled system, this could lead us down a rabbit hole spanning many classes. Due to our design decision, however, all relevant info is kept neatly in the classes pertaining to the job. However this does have its drawbacks as passing information between the system can be tricky. For instance if you wanted to load a portion of the categories into the main page you would have to call the doGet method located in the categories servlet. So upon loading the start page much of the functionality is handled by the servlet(or microservice) and is easily contained and traceable.

Performance and Testing

Testing of different classes was done in our test.java file in the test folder. The primary functions of these test was for DAOs to make sure they were querying the database properly, along with other function returns in the model.
The rest of the testing was done using Jmeter: specifically response time testing. In order to do this, a recorder was run as we logged in to the website as well as ran the category page. The response taken was done using 20 threads with a buildup time of 60 seconds and repeated 5 times. In figure 3, we can see as the buildup reaches a critical point and response time caps out as well. We noted some issues with the functions.js in the response test as well as one other function seemed to create a fairly high response time.

Figure 3. Response time testing

Team Member Contributions

Overview

As a whole, the three of us worked hard and diligently to meet all of the specifications of the project. Furthermore, we were always there to help one another in time of need and cover for a member when he had something unexpected come up in his schedule. We met in person three times throughout the duration of the development: once before development (reading and understanding the requirements) and twice during development (as a check on to see how everyone was doing. We primarily communicated on Slack, and used Google Hangouts for times when we needed to talk face-to-face without being on York campus. We used Google Hangouts once, the purpose being that the call was there to check on everyone and outline a plan moving forward into the development stages To organize individual tasks and manage deadlines, we used Asana.
For the development itself, we used GitHub to keep track of all the changes to the code. Each one of us had our very own branch, where we would work on our assigned tasks. We only merged our branches to the master branch when the code we wrote was without any bugs. This was to make sure that we were always working with the most error-free code possible to save a lot of headaches. Only critical fixes or comments were pushed directly to the master branch

Individual Contributions

Saad

Worked mostly on the frontend of the website. This included most of the CSS seen throughout the website as well as designing the shopping cart and checkout pages. Furthermore, designed the dedicated page for viewing each book and gave users the ability to add review to books they have purchased. Worked on the javascript files that validate input before being passed to the controllers. Finally, worked a bit on the backend and connected it to the frontend where clicking the username on the nav bar brings people to a page with all of the orders associated with their account.

Matt

Worked on the backend of the website. This included creating the database, as well as populating it. This also included creating all of the main DAO classes and connecting them to the front end of the website using the controller classes. This included the search, log in, register, shopping cart, category search, book, and review functionalities. Also created both the REST and SOAP services for the website.

Tyler

Worked on the analytics as well as all of the diagrams. Credits to each team mate for helping out with some parts of it. Also worked on the front end creating the start page as well as the top bar with revisions done by group mates after the initial parts where done. Lastly worked most of the descriptive documentation found throughout the code.

Member Signatures

Matthew MacEachern

Tyler Noble

Tyler Noble

Saad Saeed