Book Store

Final Project Status Report

Authors
Andres Hernandez - 211385325
JuYoung Kim - 213574900
Narbeh Nersisian - 212994745

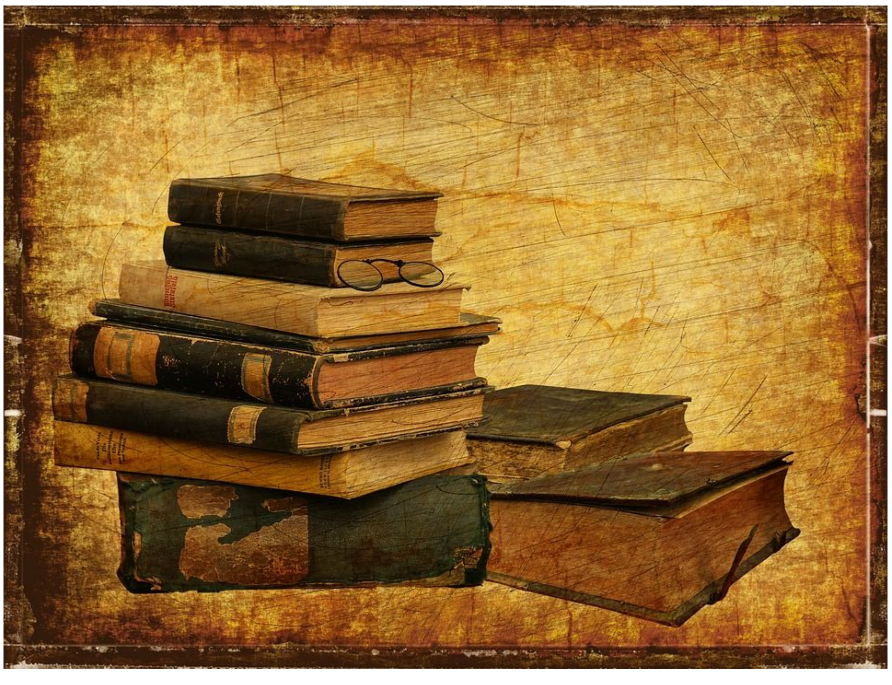


Table of contents

Architecture

UML & Class Diagrams

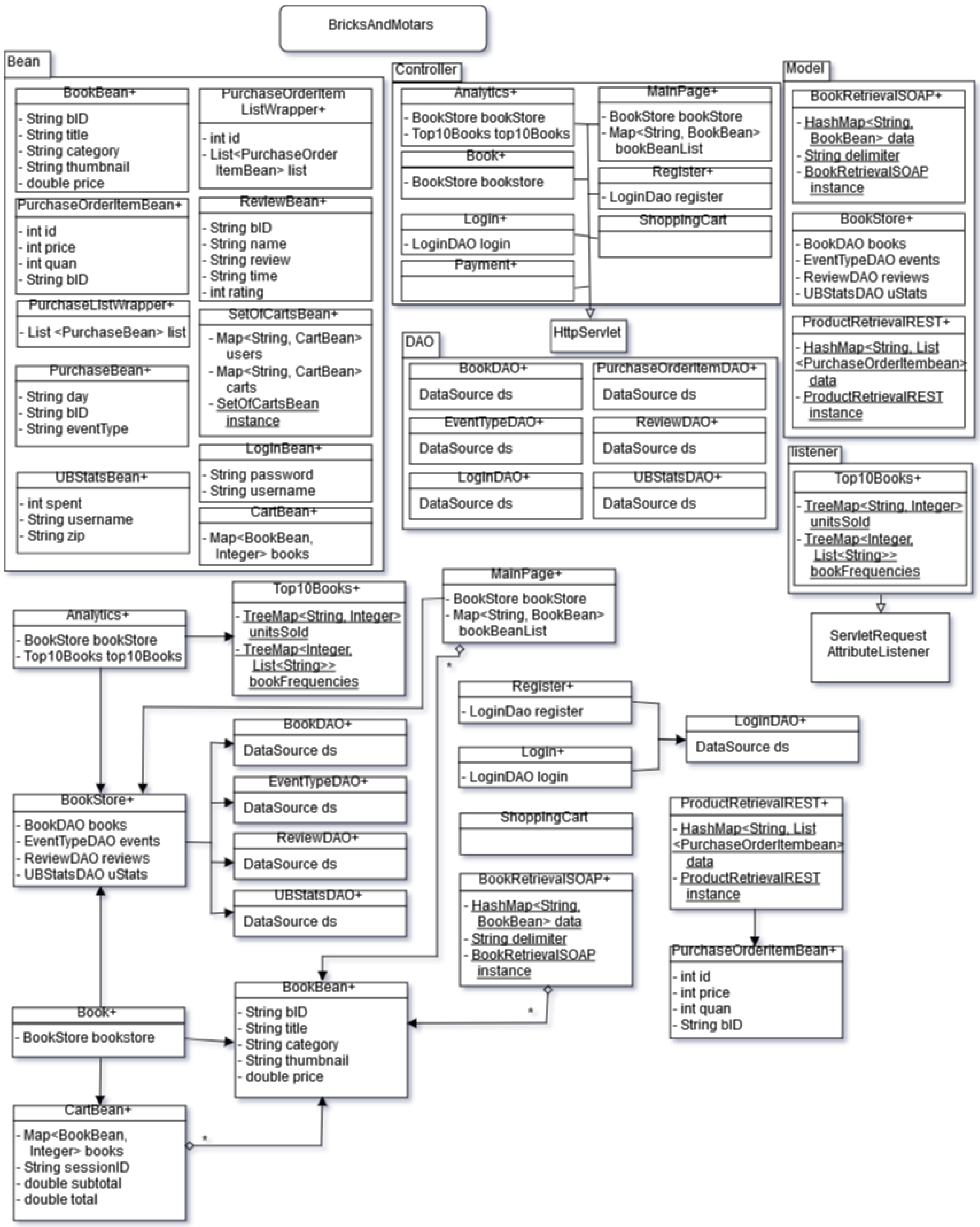


Figure 1. The UML and Class diagrams for the bookstore

Sequence Diagrams

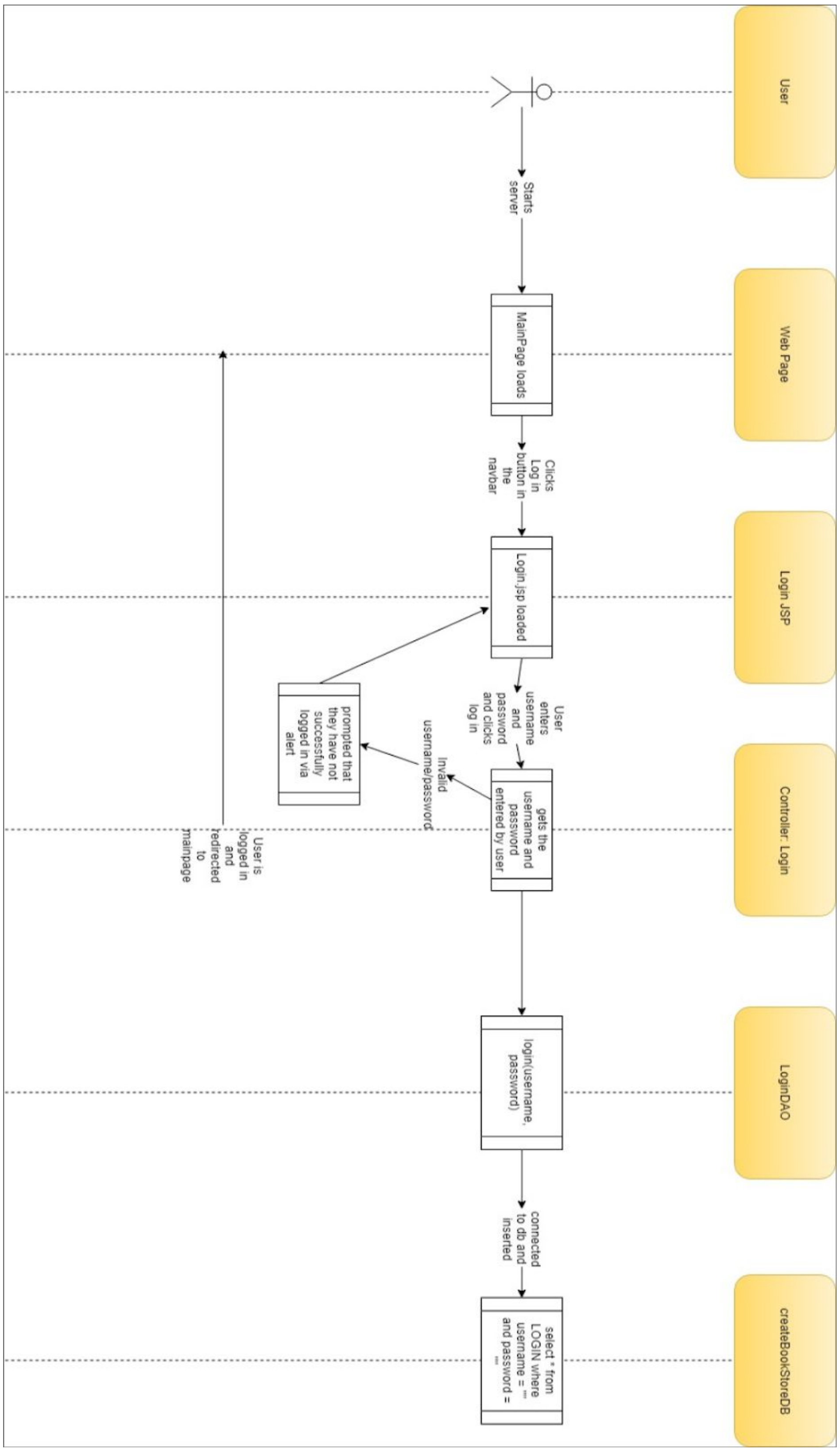

User
Web Page
Login JSP
Controller: Login
LoginDAO
createBookStoreDB
Starts server
MainPage loads
Clicks Log in button in the navbar
Login.jsp loaded
User enters username and password and clicks log in
gets the username and password entered by user
Invalid username/password
prompted that they have not successfully logged in via alert
User is logged in and redirected to mainpage
login(username, password)
connected to db and inserted
select * from LOGIN where username = '' and password = ''

Figure 1. Sequence diagram for a user logging in.

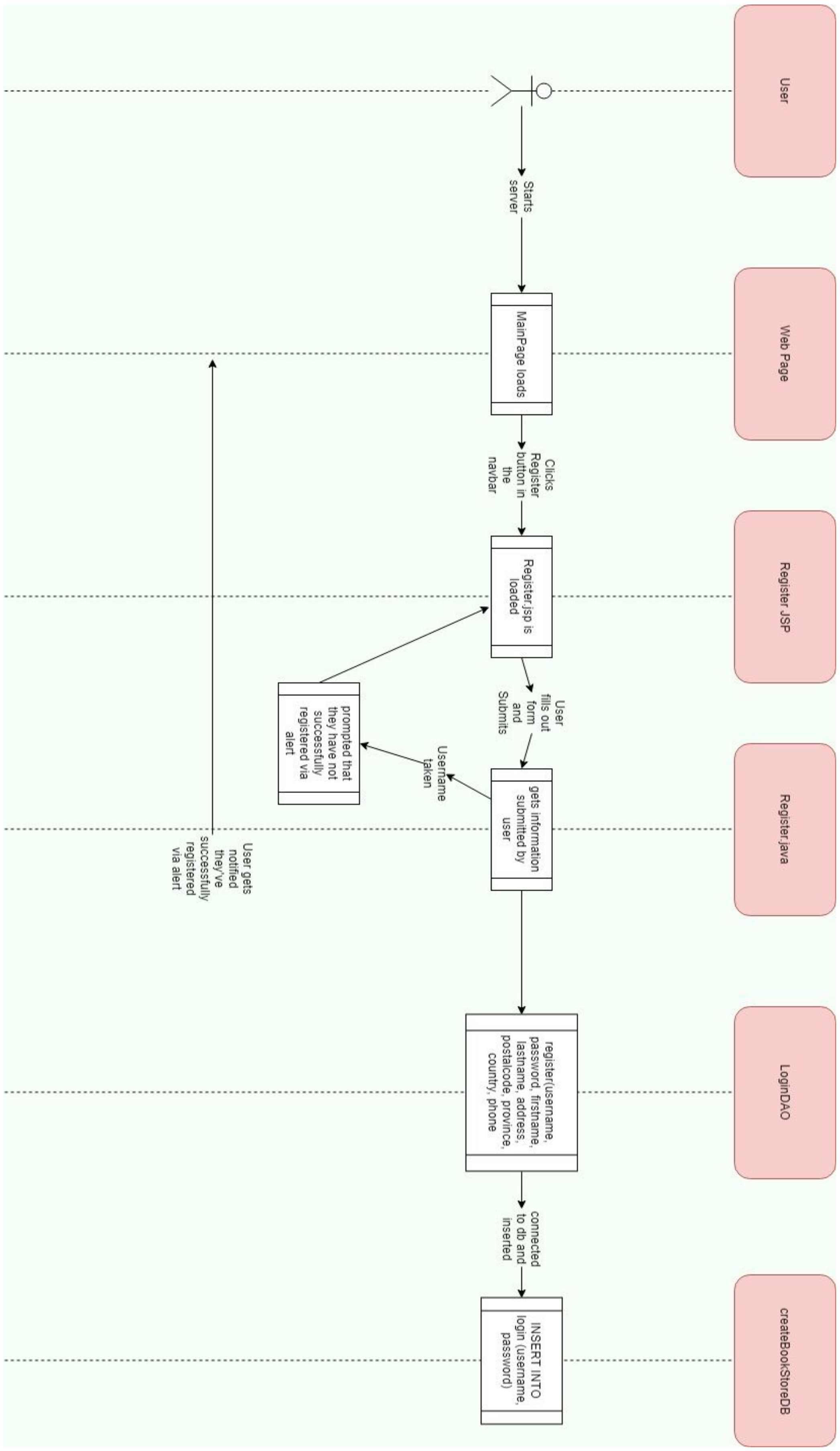

User
Web Page
Register JSP
Register.java
LoginDAO
createBookStoreDB
Starts server
MainPage loads
Clicks Register button in the navbar
Register.jsp is loaded
User fills out form and Submits
gets information submitted by user
Username taken
prompted that they have not successfully registered via alert
User gets notified they've successfully registered via alert
register(username, password, firstname, lastname, address, postalcode, province, country, phone
connected to db and inserted
INSERT INTO login (username, password)

Figure 2. Sequence diagram for a user registering.

Software Patterns

The most important software pattern is **Model-View-Controller**. The pattern exploits modularity through delegation to classes at the lower end of our model such as the Data Access Objects, the Beans, and the Wrappers. The **singleton** pattern was implemented on the setOfCarts class; a class which keeps a list of all the carts for all customers online at any given time, as it would prevent multiple instances of said class to be instantiated. The reason for this pattern is so if one servlet were to change a cart item from the list of carts, this change would be reflected upon everywhere, hence keeping the cart data consistent from any perspective. Finally, the **multiton** pattern is applied to the book bean class, the explanation for this is to assure the books accessed by the shopping cart are the same reference as the books accessed by the book page.

The database engine Derby outmatches MySQL for this project because the latter one would require additional add-ons and software to properly query the database and see the results. There were some trade offs related to this, the skeleton code provided by professor Marin Litoiu is written with MySQL syntax, so the equivalent Derby syntax was thoroughly researched with the limited documentation available on the web.

The book store followed the implementation requirements provided by Professor Marin Litoiu, however there was a change in the design to keep the book review function inside the book page instead of the main page as this would give our application a more natural User Interface flow and improve our business logic as well. The majority of the implementation decisions attempted to follow a quality of service design oriented approach.

Implementation

The Bookstore is a dynamic web application, this type of application would be very demanding on the database servers if the tables within contained millions of rows, and it would also be very demanding on our application servers if the number of users increase to the tens of thousands. No limited-resourced e-commerce system like ours can afford clients to query these servers for every minor/trivial task and as a result we concluded it would be more efficient to perform the most common tasks as close to the client as possible. An example of this would be to verify inputs such as username and password syntax using javascript classes, the objective of this is to minimize the verification processes at the application/database levels, and instead delegate that task to the client.

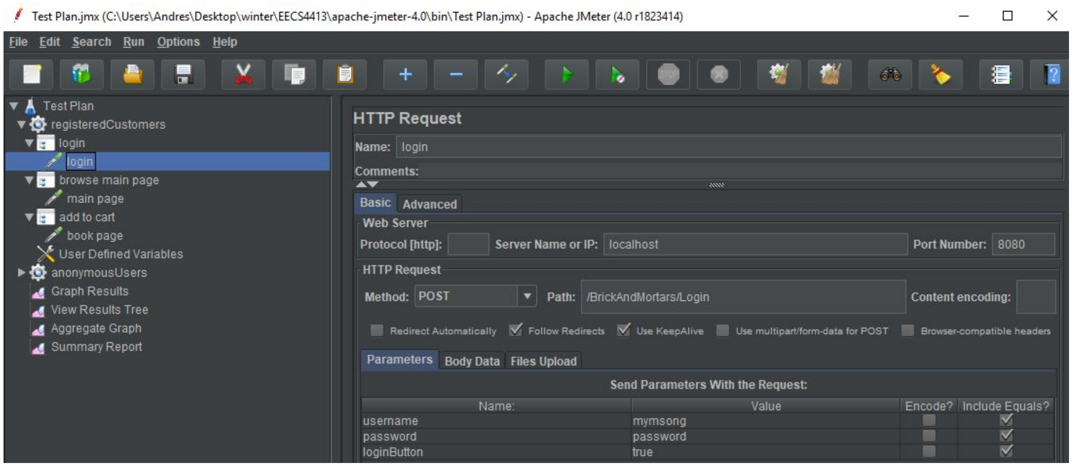
Not every task can be done at the client level. Commands that need to modify the database would effectively have to go through the entire system to reach the database and take effect. If a user were to sign up to our website, the first thing done is to verify the syntax rules using a javascript function, if that succeeds then a request is passed to the servlet, this is followed by the translation of the command into its SQL syntax equivalent, and finally the application server uses a DAO objects to issue the command to the database. A common problem that comes with commands is the need to keep data consistent throughout the system, since we keep a copy of certain data at the application level, this data needs to be updated every time there is a change to the actual database, this would effectively keep our data consistent and coherent all around. Take the top ten books as a prime example, from a high level view, there is no need to issue any commands to the database unless clients perform actions that could potentially make some books more popular than others, such as checking out. The top ten books is a complex class in our business logic that keeps and updates a list of the top ten books at any time at the application level, yet we made it so this list is consistent and coherent with the database.

The REST and SOAP communication services brought major issues to the implementation of our application. There are different software (Maven, Eclipse, etc), different versions of REST/SOAP and different ways of implementing REST/SOAP. Using the wrong combination of these produces both compiling and runtime errors difficult to debug. For instance, the Jersey used for the project is 1.19 but the version used for the REST client is 2.27. We were unable to get the SOAP client to deal with complex type xml objects and on top of this the REST had to be updated every time a change occurs in the database. We decided to make the REST and SOAP testers as war files to make the data visually easier on the client side as opposed to handing them a java console application.

There is a clear trade off between memory versus response time, however one of the most important goals for an e-commerce systems is to provide a lightning fast response and generally an outstanding quality of service. In conclusion, we effectively constructed our application by following a quality of service oriented approach which entailed keeping processes as close to the client as possible, as well as applying appropriate software patterns.

Performance Testing

Our performance testing aimed to test the limits of our e-commerce application. The jmeter testing tool was utilized here to ramp up the number of users to 1000 over the period of 50 seconds throughout 20 loops, this kept the memory of our system at a maximum of 60%. Some of the tasks included logging in, searching by category, and adding a book to the cart. These set of tasks were specifically chosen so that the database would be polled and an appropriate response analyzed in our graphs. Some of the difficulties we encountered involved performing any complex jmeter testing such as adding complex conditional samplers, as this would require more thorough research into the capabilities of jmeter. Figures 3a- 3e illustrate our settings and results



Test Plan.jmx (C:\Users\Andres\Desktop\winter\EECS4413\apache-jmeter-4.0\bin\Test Plan.jmx) - Apache JMeter (4.0 r1823414)

File Edit Search Run Options Help

```
Test Plan
  registeredCustomers
    login
      login
    browse main page
      main page
    add to cart
      book page
    User Defined Variables
  anonymousUsers
  Graph Results
  View Results Tree
  Aggregate Graph
  Summary Report
```

HTTP Request

Name: login

Comments:

Basic | Advanced

Web Server — Protocol [http]: — Server Name or IP: localhost — Port Number: 8080

HTTP Request — Method: POST — Path: /BrickAndMortars/Login — Content encoding:

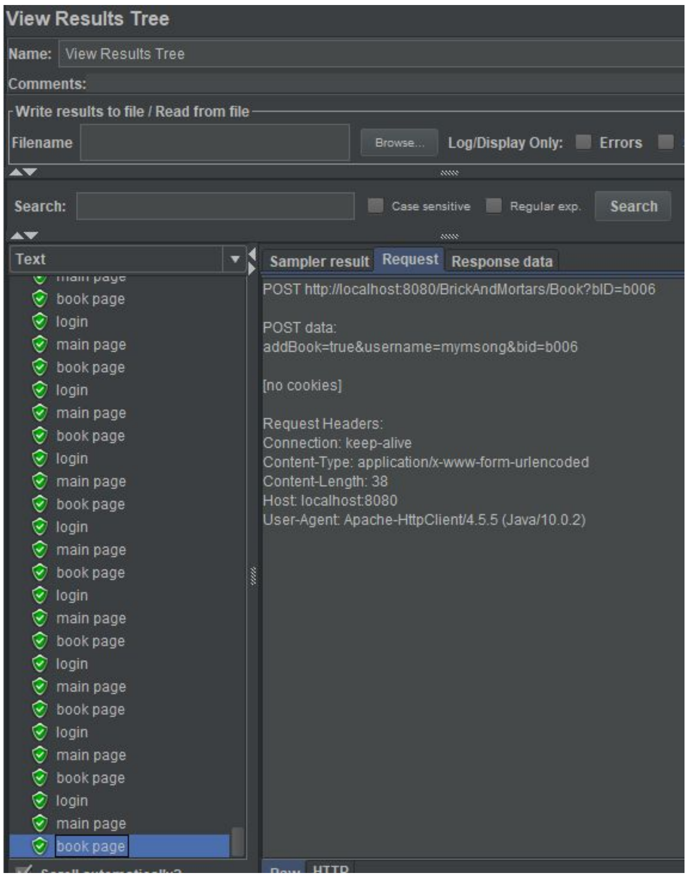
☐ Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive ☐ Use multipart/form-data for POST ☐ Browser-compatible headers

Parameters | Body Data | Files Upload

Send Parameters With the Request:

Name:	Value	Encode?	Include Equals?
username	mymsong	☐	✓
password	password	☐	✓
loginButton	true	☐	✓

Figure 3a. Jmeter Thread groups and their respective settings



View Results Tree

Name: View Results Tree

Comments:

Write results to file / Read from file — Filename — Browse... — Log/Display Only: ☐ Errors ☐

Search: ☐ Case sensitive ☐ Regular exp. Search

Text | Sampler result | Request | Response data

```
main page
book page
login
main page
book page
login
main page
book page
login
main page
book page
login
main page
book page
login
main page
book page
login
main page
book page
login
main page
book page
login
main page
book page
```

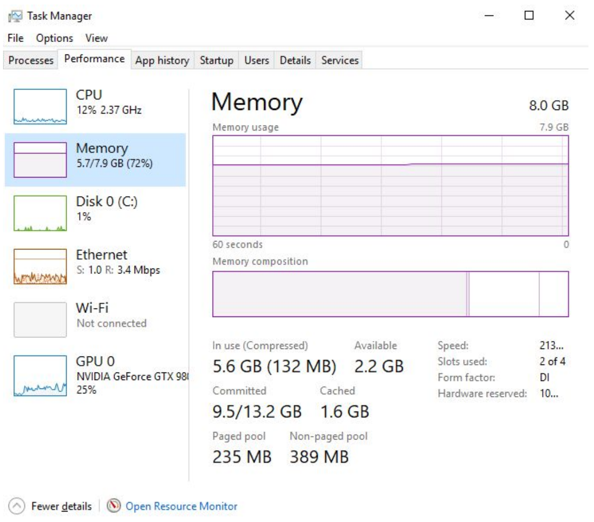
```
POST http://localhost:8080/BrickAndMortars/Book?bID=b006

POST data:
addBook=true&username=mymsong&bid=b006

[no cookies]

Request Headers:
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 38
Host: localhost:8080
User-Agent: Apache-HttpClient/4.5.5 (Java/10.0.2)
```

Figure 3b. The HTTP request of a bookpage leaf of the result tree


Task Manager

File Options View

Processes | Performance | App history | Startup | Users | Details | Services

CPU 12% 2.37 GHz

Memory 5.7/7.9 GB (72%)

Disk 0 (C:) 1%

Ethernet S: 1.0 R: 3.4 Mbps

Wi-Fi Not connected

GPU 0 NVIDIA GeForce GTX 98 25%

Memory 8.0 GB

Memory usage 7.9 GB

60 seconds 0

Memory composition

In use (Compressed) 5.6 GB (132 MB)

Available 2.2 GB

Committed 9.5/13.2 GB

Cached 1.6 GB

Paged pool 235 MB

Non-paged pool 389 MB

Speed: 213...

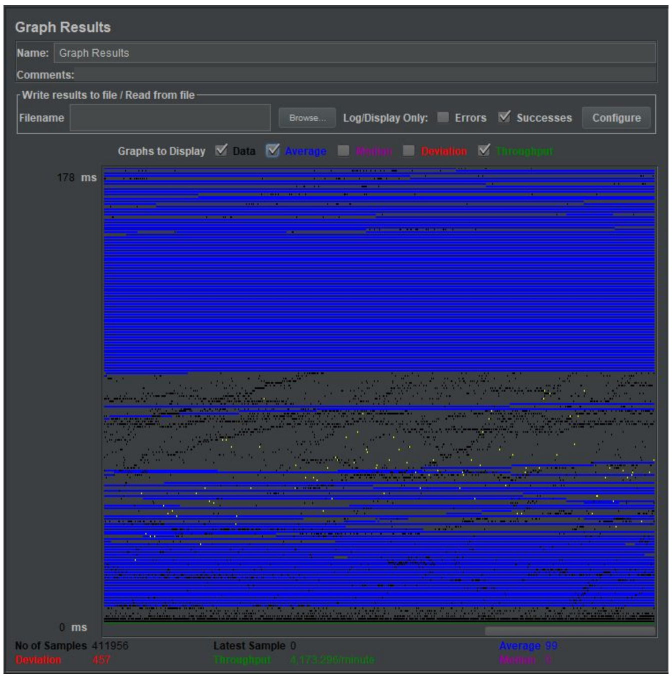
Slots used: 2 of 4

Form factor: DI

Hardware reserved: 10...

Fewer details | Open Resource Monitor

Figure 3c. The CPU resources throughout the test


Graph Results

Name: Graph Results

Comments:

Write results to file / Read from file

Filename

Log/Display Only: ☐ Errors ☑ Successes

Graphs to Display ☑ Data ☑ Average ☐ Median ☐ Deviation ☑ Throughput

No of Samples 411956

Deviation 457

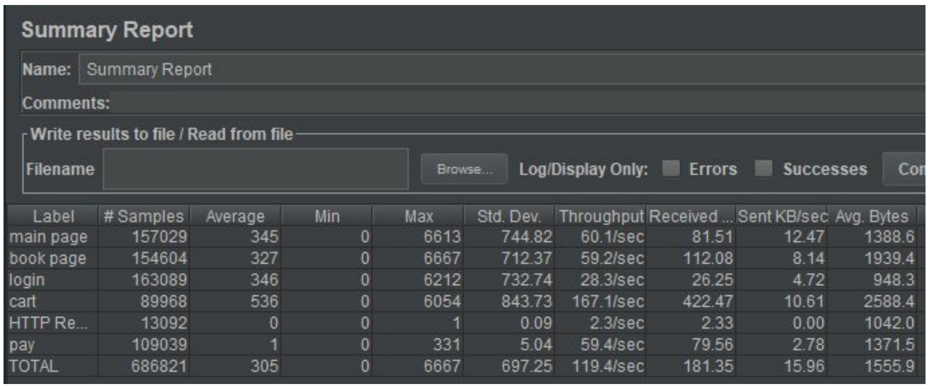
Latest Sample 0

Throughput 4,173.296/minute

Average 99

Median 0

Figure 3d. The graph results with number of samples and average response time (~ 99)


Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename

Log/Display Only: ☐ Errors ☐ Successes

Label	# Samples	Average	Min	Max	Std. Dev.	Throughput	Received ...	Sent KB/sec	Avg. Bytes
main page	157029	345	0	6613	744.82	60.1/sec	81.51	12.47	1388.6
book page	154604	327	0	6667	712.37	59.2/sec	112.08	8.14	1939.4
login	163089	346	0	6212	732.74	28.3/sec	26.25	4.72	948.3
cart	89968	536	0	6054	843.73	167.1/sec	422.47	10.61	2588.4
HTTP Re...	13092	0	0	1	0.09	2.3/sec	2.33	0.00	1042.0
pay	109039	1	0	331	5.04	59.4/sec	79.56	2.78	1371.5
TOTAL	686821	305	0	6667	697.25	119.4/sec	181.35	15.96	1555.9

Figure 3e. A summary report for our tests

Team

The team worked by delegating work and helping each other when possible. We collaborated using github, with a master branch made up of crucial checkpoints throughout our project, as well as three branches for each member where constant commits would be made every day. There was a weekly meeting held to update each other on the progress of each individual part, bring up issues, redesign our business logic when necessary and merge to the master repository.

Contributions

Andres Hernandez

- Implemented the Model-View-Controller with modularity which exploits delegation to classes at the lower end of our model such as the Data Access Objects, the Beans, and the Wrappers
- Implemented the Analytics server-side algorithms which allowed us to query the top 10 books and keep them in memory while staying consistent with the actual data, as well as the monthly report XML schema
- Modified and acclimated the database data and metadata as the project specifications were met and new issues arised.
- Implemented the search by text function and helped implement the search by category function.
- Performed Load and throughput testing using the Jmeter tool.
- Managed progress and kept tabs on group meetings as often as needed.
- Heavily contributed to this report and its edits

Juyoung Kim

- Implemented the REST system and a simple client to see if it runs correctly.
- Attempted to implement soap but was not successful on making a client for the SOAP.
- Created some bean objects to use for the REST and SOAP implementations.
- Helped out with errors in servlet and javascript codes along with getting the project to work on the different machines the group used.
- Fixed bugs and issues in the java code when errors occurred.

Narbeh Nersisian

- Implemented the front-end of the project including the main page, payment page, book page, etc.
- Worked on implementing search function, namely the search by category
- Built the table to display the results of a search after fetching them from the database with the help of a group member.
- Populated the database.

Signatures

Narbeh

Narbeh Nersisian