EECS 4413: Mom & Pop Design Document

Friday, 30 March 2016

Team Members

- Drew Noel (212513784)
- Skyler Layne (212166906)
- Siraj Rauff (212592192)

Architecture

Implementation

Performance Testing

Contributions

Signature

System Architecture

UML Use Case Diagram

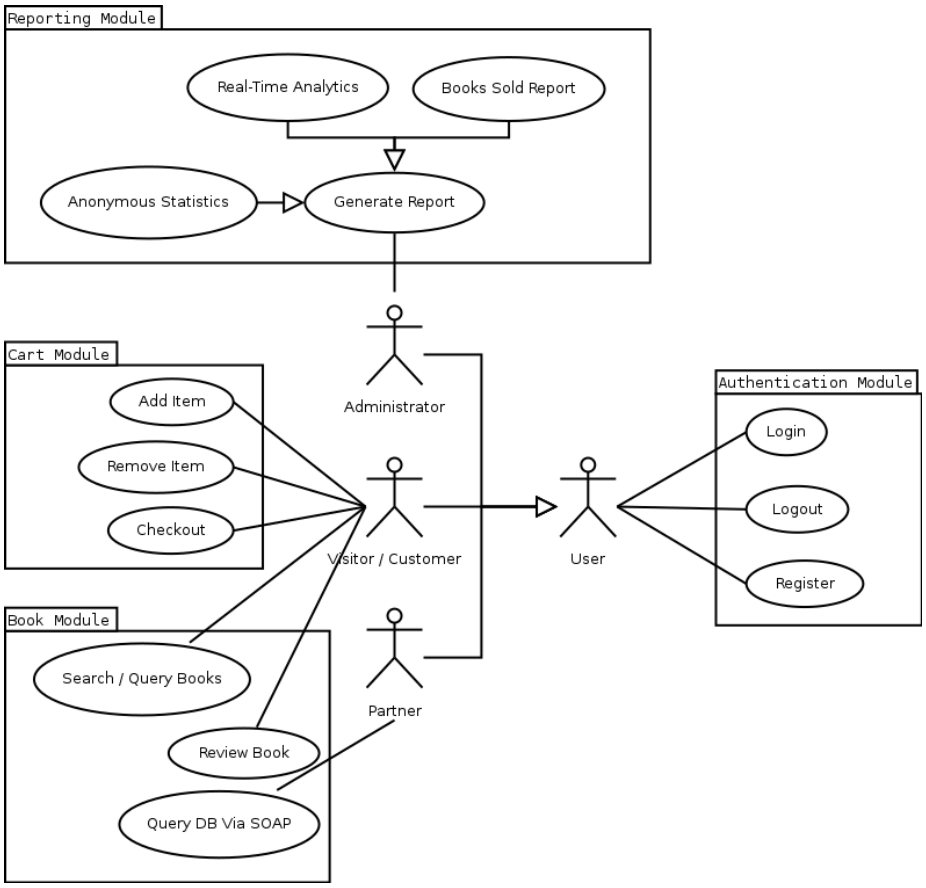

Figure 1: UML Case Diagram

The above use diagram was extracted from the requirement descriptions provided. All people visiting the web application are allowed to log in, log out, or register. Visitors or customers are then also permitted to interact with the cart by adding or removing items from the card, and finally may place an order. Administrators are able to generate reports, and partners are able to interact with the SOAP API.

UML Class Diagram

Full UML Class Diagram

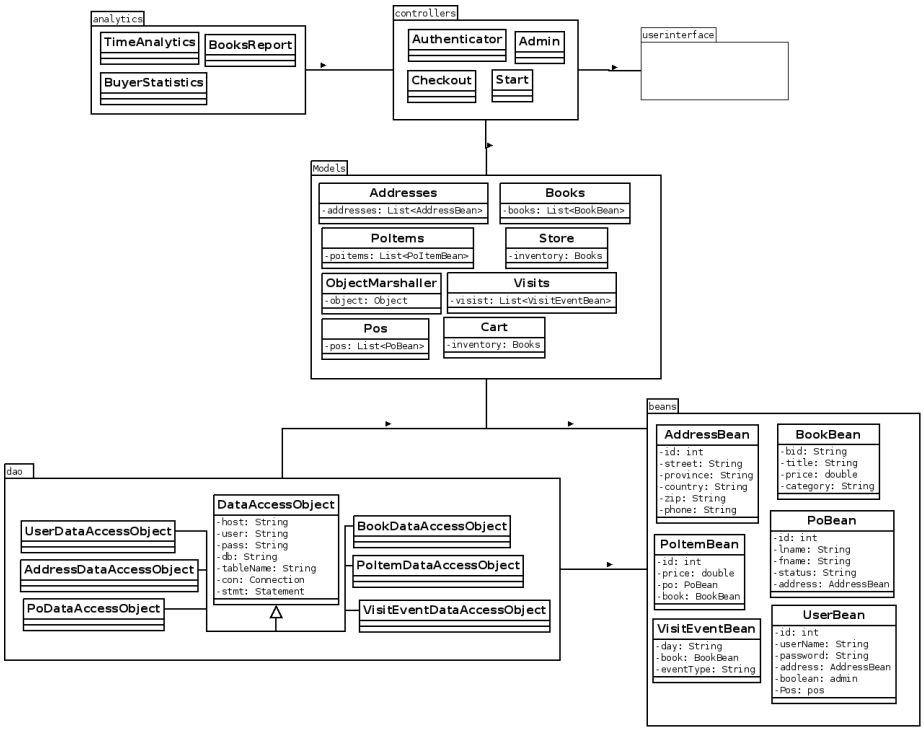

Figure 2: UML Class Diagram

Database Scheme

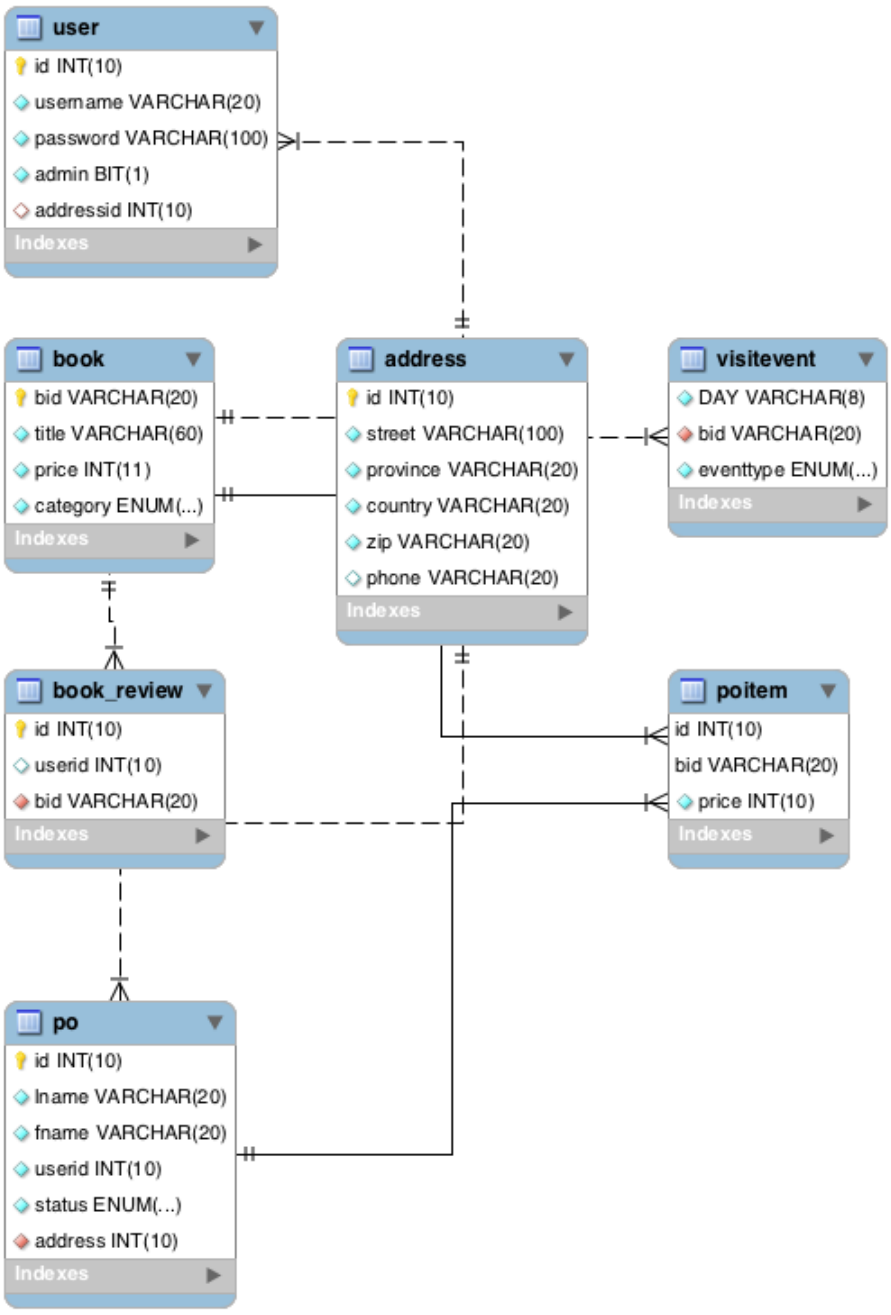

Figure 3: Database Schema

Java Beans

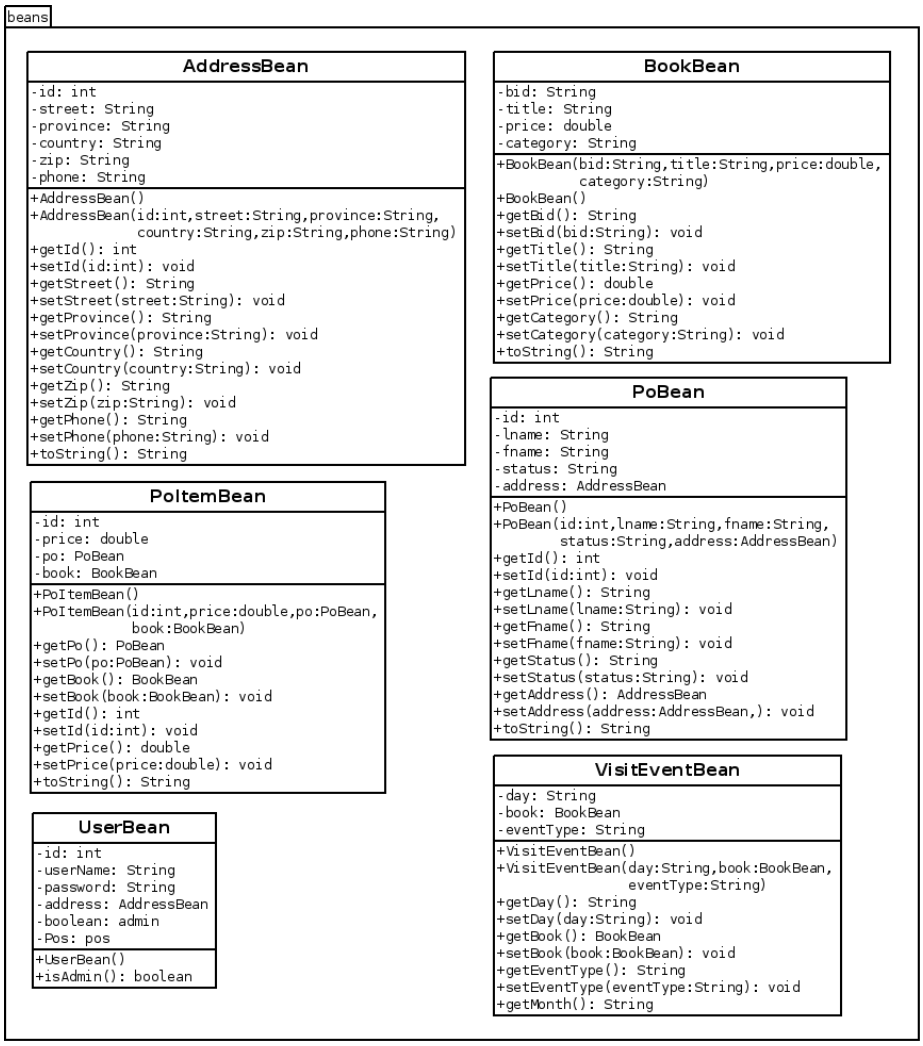

Figure 4: JavaBean UML Diagrams

Data Access Objects

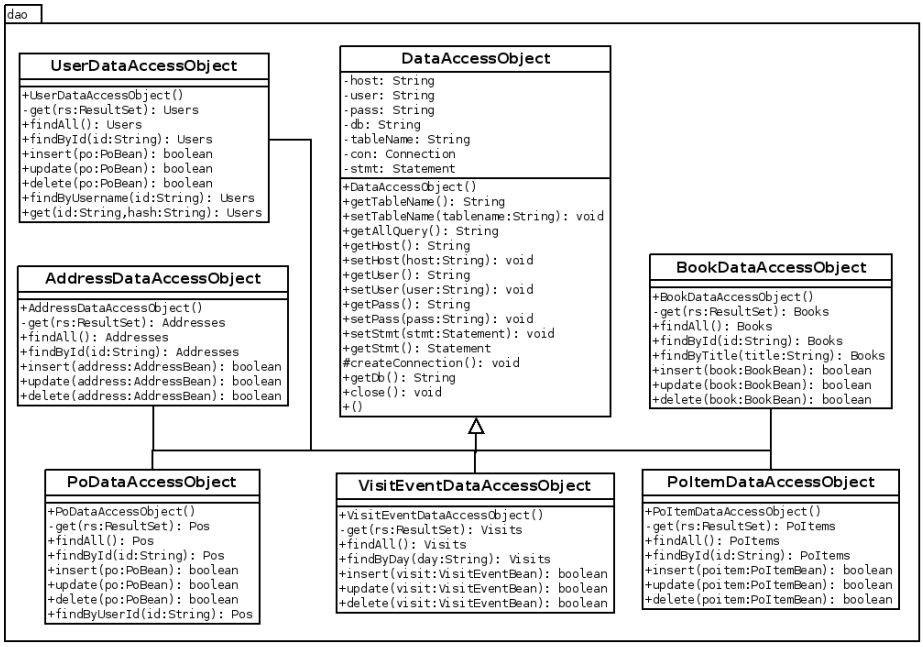

Figure 5: Data Access Object UML Diagrams

Models

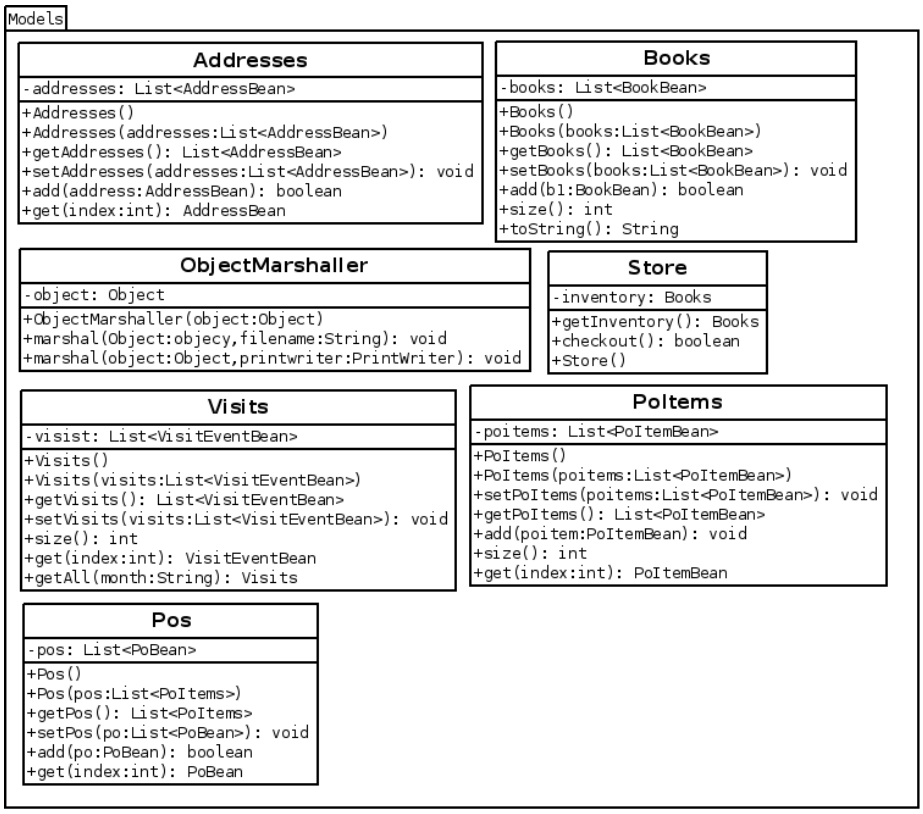

Figure 6: Models UML Diagrams

Controllers

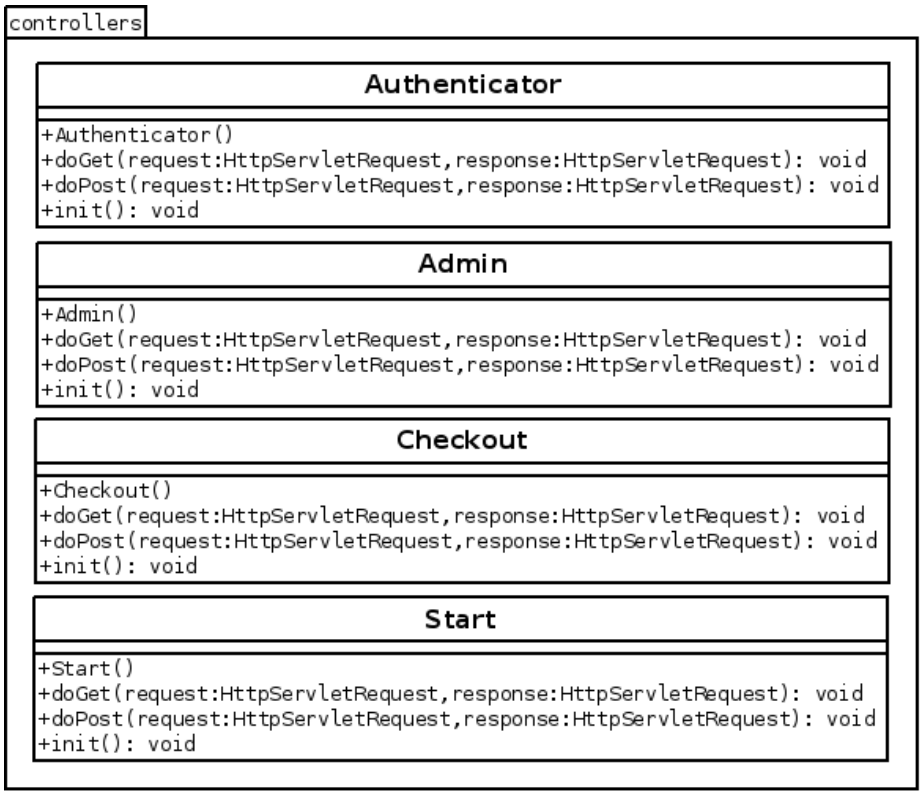

Figure 7: Data Access Object UML Diagrams

Implementation

In the implementation portion of the design document, we discuss the design decision which we have chosen. This is broken up into the following sections User Interface, Data Layer (corresponds to the Data Access Object and beans packages), Models (models package), Controllers (controllers), and Analytics (analytics).

User Interface

The user interface was created by using Twitter's Bootstrap as a base, along with jQuery. These two components removed the burden of the "heavy-lifting" from the project. Basic scaffolding and grid layout is predefined by Bootstrap, along with a few very basic styling elements. This allowed the team to focus on making the interface functional.

Data Layer

In the *Data Layer* we explore our Database schema, Java Beans, and Data Access Objects. The importance of this implementation is that it creates a base for our implementation.

Database Schema

We have chosen to use a *MySQL* database. This decision was made in part because it provides more functionality than *Derby* but, mostly due to developer familiarity.

Our database consists of 6 tables. These tables include `book`, `address`, `po`, `poitem`, `visitevent`, and `user`. Each of these database tables has a corresponding Data Access Object and Java Bean to convert these flat database rows into Java Objects.

Java Beans

Each of our database tables has a corresponding Java class called `Bean`. The bean pattern maps rows within a database table to Java objects. In our implementation we use beans to represent a singular database row. For this reason, we need to create a bean class for each of our database tables. In practice, each table row from a database query instantiates a new instance of the corresponding bean class (see Data Access Objects section).

As the beans keep track of the singular class, we treat them as a specific type of model. If we need to preform any operations on a singular model instance, here is where the logic lay. E.g. if we would like to get a user's full name, we add the method `getFullName()` to the UserBean which concatenates the user's first name and last name. This is to avoid clouding our database with redundant data.

Within these beans we use the XML annotation library to allow XML to be converted to, and from beans (see figure below). The XML annotation library aids in marshalling/unmarshalling data to/from XML files (see Object Marshaller in the Models section).

Data Access Objects

The Data access objects directly interface with `JDBC` for generic CRUD (create, read, update, destroy) operations. The specific implementation chosen includes many `DataAccessObject` classes which inherit from a `DataAccessObject` class. Within the parent `DataAccessObject` class contains common information such as database and database connection information as well as generic queries. Each of the children of the `DataAccessObject` class provide implementation of various ways to select information from the database e.g. `findById(String id)`, `findByUserName(String username)`. This is achieved by querying, and passing the received `ResultSet` to a private method `get(ResultSet queryResults)` which creates an instance of the corresponding model class.

Models

The model package contains classes which are essentially wrappers for `ArrayList<G>`, where the `G` argument is a Java bean representation of that particular class. This pluralized pattern was chosen due to the fact that each singular class already has an associated bean. E.g. The plural `Addresses` class has a singular `AddressBean` class. Each of these plurals can be seen in the Figure below (their singular, bean counter parts have already been discussed in the Java Beans section).

In most cases, these models are fairly straight forward, as they are extensions of the singular `Beans`. However, there are 3 cases which are not, these include `ObjectMarshaller`, `Store`, and `Cart`. We will explain these in more detail.

Store

The `Store` class holds all of the information of the store. It holds reference to the stores *users*, *books*, *addresses*, *purchase orders*, *items*, and *visit events*. The store is global, it is created in the `init()` method and stored in the session. Most

controller operations are forwarded through the `Store` class, with the exception of user authentication

Object Marshaller

The `ObjectMarshaller` facilitates converting a correctly annotated generic class into XML (marshalling) and vice versa (unmarshalling). In this case, the annotated classes are both the models, and the beans.

Cart

The Cart Model represents a subset of the items in the store; similar to a shopping cart of other ecommerce applications. It achieves this by storing reference to a specific instance of the `Books` model.

Controllers

Another part of the MVC stack is the controller. The controllers package facilitates the communication between the user interface and model layer. The classes in the controllers can be seen in the figure below. The controllers package also includes our filters and listeners.

Each of these controllers has a very specific role. The `Authenticator` controller is a filter which filters certain routes to determine if a user is properly authenticated, is an administrator, if the route permits, and is stored in the session. The `Admin` controller facilitates the routing of the admin panel. The checkout controller of the checkout, the `Index` of the main, `Soap` of the SOAP requests.

Analytics

We have chosen to implement the analytics as a separate component. The analytics page is essentially a Java Listener.

The primary concern of the performance of the system is due to the decision to use a single connection for the database. This severely limits the amount of concurrency possible with the system. In an attempt to discover the true scalability of the design, the following stress test was used. All tests were run using the Apache HTTP server benchmarking tool.

In this series of tests, 1000 index queries were performed against the web application. Each time the number of simultaneous connections increased, the response time dramatically climbed. Monitoring the system resources, the CPU utilization was approximately 100% for the majority of the testing. This suggests that there may be a serious computational bottleneck. This also indicates that

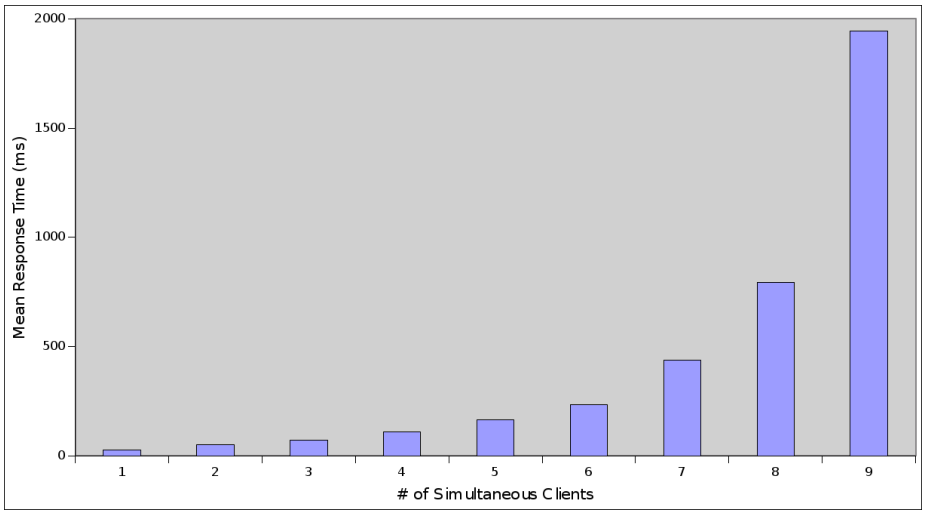

Figure 8: Test Results

the single database connector may not be as major a concern as previously believed.

All members participated in reviewing code commits from other members. In addition, all members were involved general code upkeep and acting as a rubber-duck for the other members.

- Drew Noel
  Created base code for the front-end, such as the index page, the base navigation bar. Created the UML Use Case Diagram. Additionally created and pushed fixes for the `docker` related issues.
- Skyler Layne
  Wrote the classes for beans, Create cart, UML class diagrams, data access objects, SOAP API, and all the unit tests.
- Siraj Rauff
  Constructed the login/register forms, the administrator reports, and enforced the SSL requirement.

Signatures

Skyler Layne

```
-----BEGIN RSA PUBLIC KEY-----
MIIBCgKCAQEAsa34mZsuqMglX3p03tT2uHuQf/mnC2n0wiUKdrVL0qDO5MbDA2+F
b+l+XQEy2c+7M88zkjhPZb7BydqVw4uY/yasjGdD/EJODek0S2768M0EpaSj7fFJ
gy1G1QkjNh2H2xq0ceFhzsNCgMAnQYc2B3S7pok+CvWq/O/bxqarKDJItdq0Tvzn
```

```
5YtEhfAKt5YL8OsGk+vCdl17QsROmNScnGZK6trQgADCAR7B32U1Yo4qOO5HMjak
Pd4kvFJ8JTzkFikPZlNvHbbLa7xBOlsR1drZ9ZlD5AXobucKbfw9/i1/kuwmRiAL
gUqssbYDOca2mK/fc+czC48Gw6NcrfHm2QIDAQAB
-----END RSA PUBLIC KEY-----
```

Drew Noel

```
-----BEGIN RSA PUBLIC KEY-----
MIICCgKCAgEA2VqJSJ+ocqtEiJ7iyoglX1D4HdbgGlnLNKR9ev0tHpVZqk76Mep/
TrUmWfjWSmj79ZGXv6+HLZ7jnn6jjXEMhyvaRhTex/5PFE4odSGLYw+LR7kEjnm0
4t7dIPEYcUjHEFYheKV6JzmtPQ42XH5PKZ6X8T0ODh2wfmZJjwTee4Oudqlew+G2
l5AilaSvz0vIMLk9y9VF116cVOa3nx1V/aSmsEQEwI2A7C/JJ2ILIVEkGwzimCci
YlnyZ294A0GyCd91s3K1JQySY/dHd69/oKQOXaRCsprbuwXk1RDaIa8Eom/NbgUK
x83weSjuq40IqDVRSpKVeRM3jeXSlNsJ1G0fJ7Z/U5WzW2vqybp/uwLXY3hqs0Dv
GJ6n5CmlrGpYWyWe3polVrHflnziEqTroayPe+R5fhz6LaSO40NJ6usdRsHIyPk8
41rqLqeIQlZaEDxZjM4FEXD8ZcIPWw13qaz2ZMJebW5CnYxV1QUziDe4sPL3Eb+u
ae5E3CZkQzBNR7uTnqthka5ZGybp5pmTpKeAr7EZsJsTrbwge2cutxQewsBCBDah
4ahBFJ1EUb7PLRTcWz6Pv748WHWwk/NJFNjZiDRdAukyVPKdTosE7GeBEZoibAzL
58RvIeJ8bHgqWmzV+Mcc+tvkD4QhMOXKGEN1ItNbmU2zk/xbgtAyB6ECAwEAAQ==
-----END RSA PUBLIC KEY-----
```

Siraj Rauff

```
-----BEGIN RSA PUBLIC KEY-----
AAAAB3NzaC1yc2EAAAADAQABAAABAQCxXvcbNrRA/Yyzu6mChNgB4vEAU6fC88Zq
9vZxoNrImWWeZK1kJRBiHoHIyqECnzXpbmzLgtv0wgcPW1Hi0XlS6ACyH+/inoHe
q6jPOXlbpLH7pKX/O6c3WZCpSOUK6yA6x91RodbxCE8N8Iwz7fOZ1FpWlEPScbvx
LPyYaK5By1rUvGeolrk5EDX3UHlVVI//sy9R3xWhShytqmae1mjxZW+RKW57ALsx
0pDwt/g98EwZ2lWaoWzsP8RR7enMr8VRGoVRrHvjSa0etsnXNW6g60WIgWsRU2Ts
zOc2yp4ymwLKapn3oJ8miqUJxryZNOoeFWGusoE7lhVvtCx5mtx7
-----END RSA PUBLIC KEY-----
```