Online Auctioning

Final Report

**Slippery Rock University**

Fall 2024

Griffin Vasalani – glv@sru.edu

Grant Riley – [gwr1002@sru.edu](mailto:gwr1002@sru.edu)

Jayden Williams – jsw1015@sru.edu

**References**

Johnston, J., Vioral, N., Morrow, T., Zoppelt, W., Thangiah, S. Dr. (2024). *Evaluation Manual*. Completion documentation, Slippery Rock University of Pennsylvania, Slippery Rock.

Johnston, J., Vioral, N., Morrow, T., Zoppelt, W., Thangiah, S. Dr. (2024). *Install Manual*. Installation documentation, Slippery Rock University of Pennsylvania, Slippery Rock.

Johnston, J., Vioral, N., Morrow, T., Zoppelt, W., Thangiah, S. Dr. (2024).  *Security Manual*. Security documentation, Slippery Rock University of Pennsylvania, Slippery Rock.

Johnston, J., Vioral, N., Morrow, T., Zoppelt, W., Thangiah, S. Dr. (2024).  *Technical* *Manual*. Technical documentation, Slippery Rock University of Pennsylvania, Slippery Rock.

Johnston, J., Vioral, N., Morrow, T., Zoppelt, W., Thangiah, S. Dr. (2024).  *Testing Document*. Testing documentation, Slippery Rock University of Pennsylvania, Slippery Rock.

Johnston, J., Vioral, N., Morrow, T., Zoppelt, W., Thangiah, S. Dr. (2024). *Testing Manual*. Testing documentation, Slippery Rock University of Pennsylvania, Slippery Rock.

Johnston, J., Vioral, N., Morrow, T., Zoppelt, W., Thangiah, S. Dr. (2024). *User Manual*. User documentation, Slippery Rock University of Pennsylvania, Slippery Rock.

Johnston, J., Vioral, N., Morrow, T., Zoppelt, W., Thangiah, S. Dr. (2024). *Tomcat Deployment Manual*. Tomcat Deployment documentation, Slippery Rock University of Pennsylvania, Slippery Rock

**Completion**

This table represents the tasks to be completed and the status:

|  |  |  |
| --- | --- | --- |
| **Task:** | **Status:** | **Description:** |
| Clubs | Completed | Added a new clubs page within the social page for interacting with other users |
| Groups | In Progress | Added a Groups Section Within the Clubs section for posting, commenting, and interacting with other users. |
| SubGroups | InProgress | Added a Subgroups Section Within the groups section for posting, commenting, and interacting with other users. |
| Car Parts Filter | Completed |  |
| Item Recommendation | Completed | User can choose different categories to recommend to the user |
| Search | Completed | Allows user to search items |
| Stripe Payment | Completed | Integrated Stripe payment |
| 24-Hour time window | Completed | 24 hours to complete the transaction between seller and buyer. |
| Duplicate Login Error | Completed | Fixed login requiring to be done twice. |
| Javascript functionality for Social Page | Completed |  |
| 3 offer bidding | Completed | The buyer only has 3 chances to bid on the item. |
| Sign Up Error | Completed | Error was occurring when signing up |
| Tax on purchase | Incomplete | Tax is not being updated on purchase when running on server |
| 24 hour payment window | Completed | The buyer has 24 hours to complete payment after offer has been accepted or the offer is put back on the market. |
| Notification system | Incomplete | The notification is only searching when on the view market listing page. Implement through WebSocket. |
| Counter Offer Model | Complete | The counteroffer model is able to send counter offers back and forth between the seller and the buyer. The buyer has a max of 3 offers. |

**Contributions**

Although most of the code was built and authored by the members of group 3, several external sources were utilized as references to help in accomplishing the above tasks. Some of the most valuable contributions were gathered from w3 schools, GeekforGeeks, StackOverflow, and Spring.io.

**External Sources**

*API reference*. Stripe API Reference. (n.d.). https://docs.stripe.com/api

Chatgpt. (n.d.). https://chatgpt.com/

Claude. (n.d.). https://claude.ai/

*Documentation*. Stripe. (n.d.). https://docs.stripe.com/

*Edit in DokuwikiEdit in AsciidocEdit in Markdown🌱 PlantUML at a glance*. PlantUML.com. (n.d.). https://plantuml.com/

Gandy, D. (n.d.). *The icons*. Font Awesome Icons. https://fontawesome.com/v4/icons/

GeeksforGeeks. (2022, November 30). *Spring hibernate configuration and create a table in database*. https://www.geeksforgeeks.org/spring-hibernate-configuration-and-create-a-table-in-database/

*Getting started: Building Rest Services with spring*. Getting Started | Building REST services with Spring. (n.d.). https://spring.io/guides/tutorials/rest

*Getting started: Validating form input*. Getting Started | Validating Form Input. (n.d.). https://spring.io/guides/gs/validating-form-input

MarcusMarcus                      511 silver badge88 bronze badges, Niedermann IT-DienstleistungenNiedermann IT-Dienstleistungen                      3144 bronze badges, Ken PalmerKen Palmer                      2, & MichaelMichael                      85311 gold badge77 silver badges2525 bronze badges. (1961, December 1). *Populate HTML page with Dropdown List*. Stack Overflow. https://stackoverflow.com/questions/40218105/populate-html-page-with-dropdown-list

*Ninja AI: All-in-one AI agent for boundless productivity*. Ninja AI - Personal AI with Multi-Agent Autonomy. (n.d.). https://myninja.ai/

*Spring Blog*. Announcing ListCrudRepository & Friends for Spring Data 3.0. (n.d.). https://spring.io/blog/2022/02/22/announcing-listcrudrepository-friends-for-spring-data-3-0/

YouTube. (n.d.-a). YouTube. https://www.youtube.com/watch?v=jQ8sHz15CRg&ab\_channel=KakraDetome

YouTube. (n.d.-b). YouTube. https://www.youtube.com/watch?v=jQ8sHz15CRg&ab\_channel=KakraDetome

YouTube. (n.d.-c). YouTube. https://www.youtube.com/watch?v=6zfIxgaVkQI&t=1193s&ab\_channel=BoostMyTool

YouTube. (n.d.-d). YouTube. https://www.youtube.com/watch?v=ZBCSF3Uik0g&t=461s&ab\_channel=JavaGuides

YouTube. (n.d.-e). YouTube. https://www.youtube.com/watch?v=nf87YZ4mCQs&ab\_channel=JavaGuides

YouTube. (n.d.-f). YouTube. https://www.youtube.com/watch?v=MsNLjPKfNh8&t=12s&ab\_channel=JavaGuides

YouTube. (n.d.-g). YouTube. https://www.youtube.com/watch?v=CHVVEGRGiJU&t=520s

YouTube. (n.d.-h). YouTube. https://www.youtube.com/watch?v=kakuRkFhW3M&t=9s&ab\_channel=CodeWithBisky

Zeiniss, G. (2022, January 27). *Matches in Java Stream*. Medium. https://genezeiniss.medium.com/java-streams-predicate-methods-c1c92e06b873

**Glossary**

API, or Application Programming Interface, serves as a comprehensive set of guidelines, protocols, and tools pivotal for constructing software and applications.

A Build Tool automates the conversion of source code into executable applications (such as Maven or Gradle).

Continuous Deployment (CD) is a software release approach that leverages automated testing to verify the correctness and stability of code changes for immediate deployment to a production environment.

Continuous Integration (CI) embodies the practice of frequently merging all developers' working copies into a shared mainline.

Database Initialization entails configuring a database schema and populating it with initial data, crucial for establishing an environment conducive to database operations.

Dependency Management involves handling external libraries necessary for the proper functioning of your project.

Docker comprises a suite of platform-as-a-service products utilizing OS-level virtualization to deliver software through containers.

A Framework provides a foundational platform for developing software applications tailored to a specific platform.

Git is a distributed version-control system that tracks changes in source code during software development.

GitHub Actions with WAR File Artifact Generation automates workflows for building, testing, and deploying projects. In the context of Java web applications, it involves automating the generation of a WAR file, deployable to servlet containers like Tomcat.

Integration Testing involves testing combined units to expose interaction faults, aided by test drivers and stubs.

Java Bean is a reusable software component conforming to specific conventions, encapsulating multiple objects into a single object and adhering to naming conventions, widely used in Java programming, particularly in Java Enterprise Edition and frameworks like Spring.

JDK (Java Development Kit) provides a development environment for Java applications, comprising tools such as the Java Runtime Environment, compiler, archiver, and documentation generator.

JUnit is a unit testing framework for Java.

Live Server Deployment on Tomcat involves deploying a web application to a Tomcat server for access in a production environment.

Logger is a utility for logging information during program execution, crucial for debugging and monitoring operational states.

Maven is a project management and comprehension tool, offering developers a comprehensive build lifecycle framework.

Microservices architecture structures applications as a collection of loosely coupled, independently deployable services organized around business capabilities.

Mockito is a Java-based mocking framework facilitating test-driven and behavior-driven development by creating and configuring mock objects.

REST API establishes rules for constructing web services allowing interaction with RESTful web services.

Spring Boot simplifies the creation of stand-alone, production-grade Spring applications.

Spring Security is an authentication and access-control framework highly customizable for Spring applications.

UAT Testing (User Acceptance Testing) is the final phase of software testing, where end-users ensure the software meets specifications in real-world scenarios.

Unit Testing tests individual software components separately to verify correct functionality, typically automated and integral to test-driven development.

WebSocket is a communication protocol enabling full-duplex communication over a single TCP connection, often used for real-time data transfer between client and server.

Payment Intent is an object that tracks the lifecycle of a payment, from creation to confirmation.

Stripe Dashboard is a web-based interface where you can view and manage all Stripe operations, such as transactions, customers, and disputes.

**Problem Explanation**

This semester's goal was to update Offerly and transition it into an online auctioning site for all people, and not just car enthusiasts. We introduced functionalities such as a preferences system, more items, a way bigger social section, and even a working offer system with time windows. Additionally, we updated the process of online shopping with a more social atmosphere, and significant improvements. Moreover, significant updates and changes were made to the website's overall appearance to increase usability.

One of the major updates made was expanding the range of products beyond just car parts, which now includes accessories, tools, and other automotive-related items. While this has certainly enhanced the platform’s appeal, managing this broader catalog has created challenges in terms of inventory management and product categorization. Ensuring that the expanded range of products is easily navigable and searchable for users requires continuous effort to keep everything up-to-date, especially as new products are frequently added, and old ones are discontinued or updated.

Another critical feature added was a filter search system, which allows users to refine their search results based on various product attributes, such as car make, model, year, price range, and product type. While the filter system improves the user experience, it requires constant refinement to ensure that all products are accurately tagged with the appropriate attributes. Missing or incorrect product attributes can result in frustrating user experiences.

In addition, the platform introduced a preferences page where customers can personalize their experience, as well as a club’s page that allows users to create and join groups based on shared interests in specific products or automotive categories. This feature aims to build a sense of community among users, but it introduces the challenge of moderating user-generated content and ensuring the system runs smoothly without issues such as spam or inappropriate posts.

A working offer system was also integrated, allowing for promotions, discounts, and time-sensitive offers. While this feature has the potential to drive sales, it comes with its own set of challenges. The system requires constant updating to reflect new offers, time windows, and pricing strategies. Managing these offers effectively while ensuring they are visible to the right users at the right time requires ongoing attention.

Ultimately, the platform has made significant strides in enhancing its functionality, but the sheer volume of updates needed to maintain these features, add new ones, and ensure everything works together seamlessly presents a substantial challenge. Regular maintenance, bug fixes, and feature enhancements are required to ensure the platform remains competitive, user-friendly, and capable of meeting the ever-evolving demands of its customer base.

**System Requirements**

Specifications:

* Windows OS
* MySQL Workbench 8.x
* JDK: 17
* Tomcat 10.0.x
* Spring Boot 3.1.x

**UML Diagrams**

The project's documentation folder contains a comprehensive set of pertinent UML diagrams. Within this document, you'll find examples of UML diagrams illustrating some of the platform's key features.

**Class Diagram:**

A screenshot of a computer

Description automatically generated

The CarClubController is designed to handle all the operations for actions within the club’s page of the social page. Users are able to join and leave groups, and subgroups that are within the groups. Users are able to post too, and comment on those posts.

**Use Case Diagram:**

A diagram of a user

Description automatically generated

The above image shows a Use Case Diagram displaying all the different options/actions available to a User and Seller on a Listing Page. The Buyer can send offers, buy the product at the asking price, view/watchlist an item and send a message to the Seller. The Seller can list an item, view offers, send counter offers, and respond to messages. The buyer can only send up to 3 offers per listing, and once the first offer is sent, the transaction must be completed within 24 hours. Also, payment must be completed within 24 hours once the payment is accepted by the buyer. The Customer Service Representative can Notify users and offer help on listing/product information.

**State Chart Diagram:**

A diagram of a user interface

Description automatically generated

This State Chart diagram outlines how interests are saved within the site for user profiles. Users can select interests, and once those interests are saved, the users will only see the stuff that they are interested in.

**Sequence Diagram:**

A screenshot of a computer

Description automatically generated

This sequence diagram shows the operations related to buying and selling an item. The buyer can make an initial offer or buy the product for the full asking price. Once the buyer makes the initial offer to the seller. The seller is then notified of a new offer and a 24- time window begins before the offer expires. The seller has the option to send a counter offer back or accept the offer. The buyer is then notified upon the seller’s decision and can either send another offer back to the seller or pay for the product. This cycle is limited to three offers being sent to the seller from the buyer. Once the offer is accepted the buyer is then prompted to pay for the product within 24 hours or the product is put back on the market.

**Data Flow Diagrams**

A diagram with text and arrows

Description automatically generated with medium confidence

The diagram shows how the system handles user preferences and product recommendations. First, when a user accesses their preferences, the system retrieves or creates their interest data by communicating between the interface, controller, and database. Next, when the user submits their interests, the system updates the data and saves it to the database. Finally, when the user views recommended products, the system fetches the recommendations by using the user interests.

A diagram of a company

Description automatically generated with medium confidence

The diagram shows how the UserInterestService manages user preferences and generates product recommendations. It saves and retrieves user interests from the database and uses this data to suggest products. The service can filter recommendations by price or find popular items based on categories and page views. It also interacts with the MarketListing to get product details like categories and prices.

**Caveats/Minefields**

A diagram of a computer

Description automatically generated with medium confidence

The diagram shows how MarketListing connects to other parts of an online marketplace. MarketListing represents a product for sale, linked to a widget and an auction for bidding. Each listing has one seller or user. Sales are tracked through Transactions, it records details like the quantity sold, price, and the buyer and seller. Shows how products, users, auctions, and sales work together in the marketplace.

**Caveats/Minefields**

While working on this product we have run into a lot of issues. One is learning a new tool like Springboot and thymeleaf. Having to understand the syntax and notations for springboot and thymeleaf caused a lot of delay in the development of this program. There were a lot of issues with deploying the tomcat server which also caused delays in our development. When running on the Tomcat server the URL is different which caused a lot of issues with redirects and mapping. This occurred while working on the clubs and payment sections. In the clubs section it was fixed but for the payment section on the server the tax is not being updated in the database.

While adding the recommendation section a reoccurring issue was that the users’ preferences were not being saved to the database due to mapping issues. So having known Springboot methods would have greatly helped.

When adding notifications the system would check if there was a notification every second, but we found that the system’s performance, so we increased the time.

When implementing the stripe payment and changing the old payment system saw many issues occurring when adding Stripe’s API to the program. One being able to save the user’s payment details in both Stripe’s and our database.

**File Path Names**

* Application Configuration File (application.properties):

Purpose: Serves as a central repository for application-level configurations, encompassing database connections, server settings, and other Spring Boot-specific configurations.

* CI/CD Configuration Files (GitHub Actions for WAR Generation):

Purpose: Define workflows for GitHub Actions, specifically tailored for building and deploying the application as a WAR (Web Application Archive) file, automating the continuous integration and delivery processes.

* Configuration Directory (.configuration):

Purpose: Houses classes and resources pertinent to the application's configuration, such as database configurations, connections to external services, and application-specific settings.

* Controller Directory (.controller):

Purpose: Manages MVC controllers responsible for handling incoming HTTP requests, preparing model data, and rendering views.

* Database Initialization File (import.sql):

Purpose: Facilitates the initialization of the database with necessary data upon startup, typically containing SQL statements for data insertion.

* Data File (StateTaxes.xlsx):

Purpose: Likely contains specific data relevant to the application, such as state tax information, utilized for data import, analysis, or reference.

* Domain Directory (.domain):

Purpose: Hosts domain models embodying the application's core business logic, including entities, enums, and data transfer objects (DTOs).

* Project Object Model File (pom.xml):

Purpose: A pivotal file in Maven-based projects, defining the project's structure, dependencies, build configurations, and other essential details.

* Repository Directory (.repository):

Purpose: Focuses on data persistence, housing interfaces for data access layers, typically implementing Spring Data JPA for database interactions.

* Security Directory (.secure):

Purpose: Devoted to the security configurations of the application, encompassing authentication, authorization, and other security aspects.

* Service Directory (.service):

Purpose: Contains the service layer, encapsulating business logic, executing operations on domain objects, and interfacing with repositories.

* Utility Directory (.util):

Purpose: Stores utility or helper classes furnishing common functionalities utilized across the application, such as date/time utilities and string manipulation tools.

**Code Reusability**

Our development team focused on building a scalable system by focusing on code reusability. Object-oriented programming and the MVC pattern were central to the project, ensuring that almost all Java classes in the system were designed with reuse in mind. These design principles facilitated the quick development of new features without redundant effort.

One instance of this scenario is the enum status within the Market Listing Status. These status settings have the flexibility to serve various purposes. Similarly, the clubs section layout has potential beyond merely posting. Additionally, there could be coding solutions addressing challenges related to Thymeleaf.

Throughout the project, complex features were built upon straightforward, reusable methods, demonstrating a consistent emphasis on reducing redundancy. This approach not only accelerated development but also optimized resource utilization.

**Testing**

**Black Box Testing**

Throughout the project lifecycle, our team conducted systematic black box testing to evaluate functionality from an end-user perspective. A notable example of this approach was our comprehensive testing of the offer system. Initially, users had unlimited offer-sending capabilities per listing. We implemented new restrictions allowing only one active offer per listing with a maximum of three total offers. Our testing protocol involved attempting to send offers on multiple listings simultaneously, typically five or more, as well as multiple offers per individual item. The system successfully enforced these restrictions by displaying appropriate notification messages when users attempted to exceed the established limits.

**White Box Testing**

White box testing focused on verifying the intricate relationships between frontend displays and backend database operations. One significant test case involved the car club membership system. When users joined a car club, we verified that the frontend HTML correctly displayed member information, chat functionality, and associated club data. Simultaneously, backend testing confirmed proper database population with user information, club membership details, role assignments, and all necessary metadata. This dual-focused approach ensured complete system integrity across all architectural layers.

**Regression Testing**

Regression testing formed a crucial part of our quality assurance process, focusing on core system functionality. We consistently verified essential features including the authentication system, offer management, messaging functionality, listing operations, payment processing, and navigation systems. This ongoing testing regime helped maintain system stability throughout the development cycle.

**Boundary Analysis**

Our boundary analysis testing concentrated on critical system limitations, particularly in the offer system. We implemented and verified strict boundaries including the three-offer maximum per item and single active offer requirement. Testing included edge cases in offer submission timing and validation of offer amounts within acceptable ranges.

**Volume Testing**

Volume testing evaluated system performance under high-load conditions. We conducted extensive tests with large numbers of simultaneous offers, high notification volumes, and multiple concurrent user sessions. This testing phase also included evaluation of bulk listing management and large-scale message thread handling capabilities.

**Resource Management**

Initial testing of the notification system revealed important performance considerations. We began with a 30-second notification check interval and systematically tested different polling frequencies. Through careful analysis, we identified performance degradation at sub-15-second intervals, where the system showed increased memory consumption and image processing errors. This led to establishing an optimal polling interval of 15 seconds or greater to maintain system stability while ensuring timely notifications.

**Missing Resource Handling**

We implemented robust error handling mechanisms, particularly for image processing failures. The system now automatically implements placeholder images when encoding issues occur, with automatic replacement upon page reload. Testing confirmed successful error recovery and proper image replacement functionality.

**Crash Recovery**

The system demonstrated exceptional crash recovery capabilities during our testing phase. In the event of unexpected termination or system crashes, the database-maintained data integrity and proper state restoration. The system successfully preserved session data, protected user information, and implemented transaction rollback procedures when necessary. Upon restart, all HTML elements rendered correctly, and the system restored to its pre-crash state with all critical data intact.

**Unit Testing: Test Environment Configuration**: We implemented our unit testing framework utilizing the Eclipse JUnit plugin, incorporating Mockito for mock object generation. This architecture enabled isolated component testing through effective dependency simulation.

**Test Suite Implementation**: Our comprehensive unit testing approach involved developing test cases that validated core functionality across domain entities, including property accessors and mutators. The testing scope encompassed both expected behavior and edge cases, extending across domain objects, form structures, and data transfer objects. We conducted thorough unit testing of service layer components to verify method behavior and outcomes aligned with specifications.

**Integration Testing**: Integration Strategy: Our integration testing framework focused on validating the seamless interaction between architectural layers - specifically the controller, service, and entity components - to ensure system cohesion under realistic operating conditions. The Test Cases we developed and executed integration test scenarios centered on core business functions, including product listing workflows, authentication processes, and transaction handling. These tests were designed to verify robust component interoperability and maintain system stability across integrated operations.

**Conclusion**:

During the initial project handover, we received a codebase containing 380 test cases. The initial test execution revealed 38 errors, 2 failures, and 340 successful tests. Through extensive development and enhancement efforts, we have significantly expanded the test coverage to 528 total test cases. Current test execution results show remarkable improvement with only 32 errors, 2 failures, and 494 successful tests.

The substantial increase in test coverage was necessary due to implementation of several key features and modifications to existing functionality. New test cases were added specifically for the preferences page, car clubs page, notification system, and modifications to the offer system. The test results demonstrate that the newly implemented features are successfully integrated with the existing codebase and are functioning according to specifications. The remaining errors are the result of the test base running out of memory during the initial boot phase and should be looked at closer. However the functionality of the program and the features that the tests are checking are in proper working order.

**Deployment/Maintenance**

**Overview:**

Our Spring Boot application deployment uses GitHub Actions for automated builds, producing a WAR file. This ensures consistent builds with precise control during deployment and monitoring.

**Deployment Procedure:**

Automated Building via GitHub Actions:

* Trigger: Builds are triggered on commits to the main branch or merged pull requests.
* Workflow: Actions include code checkout, dependency installation, testing, and WAR file generation.
* Artifact: The WAR file is downloadable from the GitHub Actions run.

**Maintenance:**

Monitoring and Logging:

* Application health is monitored like CPU, memory, response times.
* Logs provide runtime insights and are reviewed for error resolution.

Updates and Patches:

* Regular updates and bug fixes use the GitHub Actions pipeline for consistent deployment.
* Critical patches follow a hotfix process to minimize downtime.

Backup and Recovery:

* Source control in GitHub ensures regular backups.
* Restores and rollbacks are supported for version control.

**Port-Mortem Analysis**

While working on the project we ran into many issues. We implemented new features fully to completion. Some issues in the beginning were learning new tools like Springboot, Thymeleaf, and using MVC.

When implementing the recommendation section, we wanted to use a more complex system using recommendation systems similar to Netflix and Amazon using content based and collaborative filtering methods but ended up using a simpler method with just having the user choose their own preferences rather it being based on their search and previous purchased items. Saving the users’ interests into the database was an issue that was spent on for weeks.

While working on the payment we implemented Stripe payment system. We ran into issues with the capabilities of Stripe and what it allows. Like being able to save the card information into our database and inside Stripe.

Looking back on the way we enter this program there are some things we wished we would have done to make working on it easier. Before coding we think that starting with the UMLs would have made the coding and mapping of everything easier. Also running on the server, the URL is different adding /OnlineAuction this causes many errors with mapping and redirect links.

**Future Works**

**Issues:**

This is a list of features that need adjusted and attended to.

* There is a method to mark messages as read, though it is never called.
* Admin view of the discussion board needs to be completed; comments associated with a post should appear in a lower table when the button “show comments” is clicked.
* The discussion board has the option for the admin to delete posts that haven't had activity in a defined amount of time.
* The Discussion Board Tab in the nav bar locks you into the page and forces you to use the back arrow in the browser to leave.
* Inside the home page in the nav bar that shows all the listings, the image behind the title “Your Perfect Products” is not there and in place is a grey division.
* Sometimes when a user leaves a group and is the owner the transfer of ownership does not happen immediately.
* Whenever running on the server with two users logged in, the users must refresh the page to see either a new group, or a new post. The same applies to subgroups.
* When clicking the message button in the user’s profile, it takes you to the message page, but if you do not have the user added as a friend, it doesn’t show it at all.
* There is a red button with a trashcan under the user’s profile that take you to the social page, and it does nothing.
* State tax when running on server
* Users should have to login if they want to add a new shipping address

**Cleanup:**

This is a list of suggestions and possible solutions to the issues stated above.

* A method to retrieve all messages should be implemented and conversations should start as they happen.
* Read Messages Method needs to be called once a conversation between users is opened.
* A timer to automatically archive and then delete after a certain amount of time would be a much-needed improvement.
* Unsure why the page is locked in and other buttons do not work when in discussion board tab, would have to look at backend.
* Possible file path is wrong, as tomcat is strict with the pathways, and could be why the image for home page is not displaying.
* Possibly the membership set is not being read correctly.
* Have the page dynamically updates the post model and same for the group model.
* Change functionality of buttons and their redirects.
* Map the method for tax to include the server URL
* For adding new shipping details have the button go to login page and once logged in it should redirect you to the purchase page with the shipping model open. Also make sure the server URL is accounted for.

**Expansion and Improvement:**

This is a list of possible future implementations that can be added and enhanced.

* Make notifications appear instantly with expansion and WebSocket.
* Also update posts and group status dynamically with WebSocket.
* Include images in the creation and in the ability to post within the groups and subgroups in the club page of the social page.
* Include a way for the usernames in the members list within the groups to be linked back to their profile to add as a friend and so on.
* Include a quantity on the item being sold so buyers know how many are left