D424 – Software Engineering

Task 3

Capstone Proposal Project Name: StockTrack – Web-Based Inventory Management System

Student Name: Jeffrey VanMeter

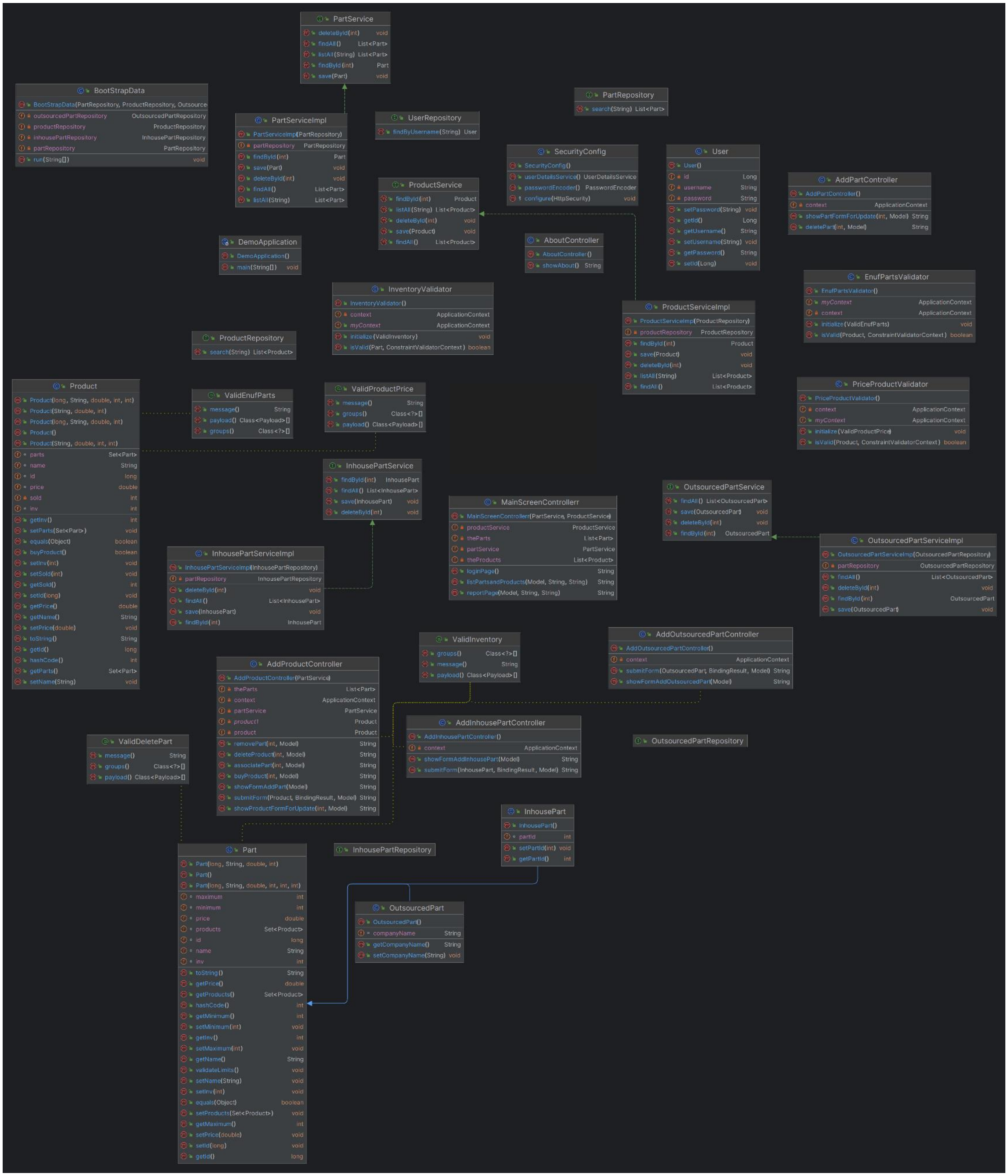
Table of Contents

Design Document

Application Design and Testing

Class Design

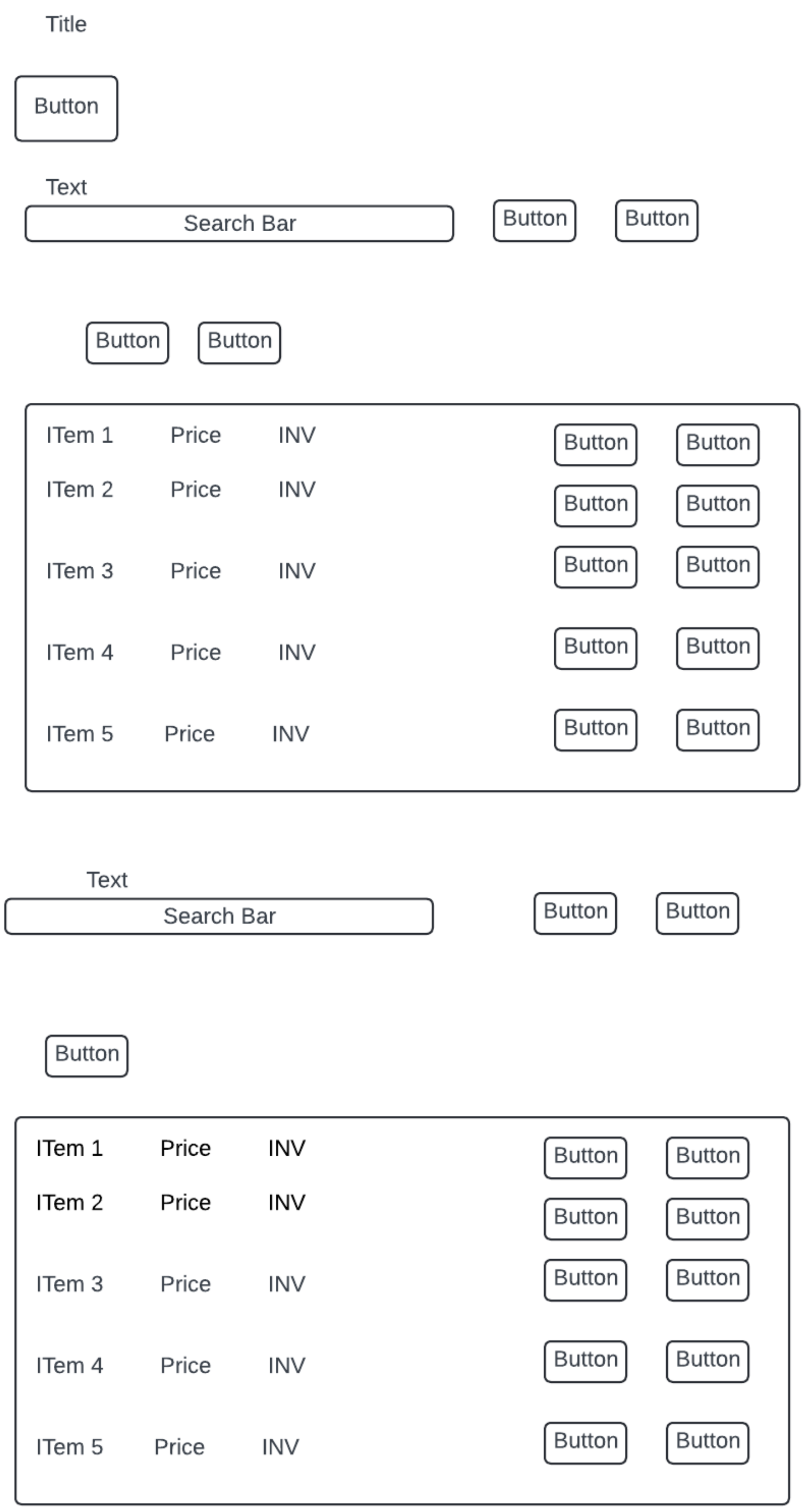
The UML class diagram for the StockTrack project provides a comprehensive overview of the software's structure, illustrating the relationships and hierarchies between all key classes within the system. This diagram captures the core classes, their attributes, methods, and associations, offering insight into the design of each class and its interactions with others. By examining these components, we can better understand the functionalities, dependencies, and modular structure of the project, which serves as a blueprint for understanding, maintaining, and expanding StockTrack's codebase.


PartService
deleteById(int) void
findAll() List<Part>
listAll(String) List<Part>
findById(int) Part
save(Part) void
BootStrapData
BootStrapData(PartRepository, ProductRepository, Outsource
outsourcedPartRepository OutsourcedPartRepository
productRepository ProductRepository
inhousePartRepository InhousePartRepository
partRepository PartRepository
run(String[]) void
PartRepository
search(String) List<Part>
UserRepository
findByUsername(String) User
PartServiceImpl
PartServiceImpl(PartRepository)
partRepository PartRepository
findById(int) Part
save(Part) void
deleteById(int) void
findAll() List<Part>
listAll(String) List<Part>
SecurityConfig
SecurityConfig()
userDetailsService() UserDetailsService
passwordEncoder() PasswordEncoder
configure(HttpSecurity) void
User
User()
id Long
username String
password String
setPassword(String) void
getId() Long
getUsername() String
setUsername(String) void
getPassword() String
setId(Long) void
AddPartController
AddPartController()
context ApplicationContext
showPartFormForUpdate(int, Model) String
deletePart(int, Model) String
ProductService
findById(int) Product
listAll(String) List<Product>
deleteById(int) void
save(Product) void
findAll() List<Product>
DemoApplication
DemoApplication()
main(String[]) void
AboutController
AboutController()
showAbout() String
EnufPartsValidator
EnufPartsValidator()
myContext ApplicationContext
context ApplicationContext
initialize(ValidEnufParts) void
isValid(Product, ConstraintValidatorContext) boolean
InventoryValidator
InventoryValidator()
context ApplicationContext
myContext ApplicationContext
initialize(ValidInventory) void
isValid(Part, ConstraintValidatorContext) boolean
ProductServiceImpl
ProductServiceImpl(ProductRepository)
productRepository ProductRepository
findById(int) Product
save(Product) void
deleteById(int) void
listAll(String) List<Product>
findAll() List<Product>
ProductRepository
search(String) List<Product>
Product
Product(long, String, double, int, int)
Product(String, double, int)
Product(long, String, double, int)
Product()
Product(String, double, int, int)
parts Set<Part>
name String
id long
price double
sold int
inv int
getInv() int
setParts(Set<Part>) void
equals(Object) boolean
buyProduct() boolean
setInv(int) void
setSold(int) void
getSold() int
setId(long) void
getPrice() double
getName() String
setPrice(double) void
toString() String
getId() long
hashCode() int
getParts() Set<Part>
setName(String) void
ValidEnufParts
message() String
payload() Class<Payload>[]
groups() Class<?>[]
ValidProductPrice
message() String
groups() Class<?>[]
payload() Class<Payload>[]
PriceProductValidator
PriceProductValidator()
context ApplicationContext
myContext ApplicationContext
initialize(ValidProductPrice) void
isValid(Product, ConstraintValidatorContext) boolean
InhousePartService
findById(int) InhousePart
findAll() List<InhousePart>
save(InhousePart) void
deleteById(int) void
OutsourcedPartService
findAll() List<OutsourcedPart>
save(OutsourcedPart) void
deleteById(int) void
findById(int) OutsourcedPart
MainScreenControllerr
MainScreenControllerr(PartService, ProductService)
productService ProductService
theParts List<Part>
partService PartService
theProducts List<Product>
loginPage() String
listPartsandProducts(Model, String, String) String
reportPage(Model, String, String) String
OutsourcedPartServiceImpl
OutsourcedPartServiceImpl(OutsourcedPartRepository)
partRepository OutsourcedPartRepository
findAll() List<OutsourcedPart>
deleteById(int) void
findById(int) OutsourcedPart
save(OutsourcedPart) void
InhousePartServiceImpl
InhousePartServiceImpl(InhousePartRepository)
partRepository InhousePartRepository
deleteById(int) void
findAll() List<InhousePart>
save(InhousePart) void
findById(int) InhousePart
ValidInventory
groups() Class<?>[]
message() String
payload() Class<Payload>[]
AddOutsourcedPartController
AddOutsourcedPartController()
context ApplicationContext
submitForm(OutsourcedPart, BindingResult, Model) String
showFormAddOutsourcedPart(Model) String
AddProductController
AddProductController(PartService)
theParts List<Part>
context ApplicationContext
partService PartService
product1 Product
product Product
removePart(int, Model) String
deleteProduct(int, Model) String
associatePart(int, Model) String
buyProduct(int, Model) String
showFormAddPart(Model) String
submitForm(Product, BindingResult, Model) String
showProductFormForUpdate(int, Model) String
AddInhousePartController
AddInhousePartController()
context ApplicationContext
showFormAddInhousePart(Model) String
submitForm(InhousePart, BindingResult, Model) String
OutsourcedPartRepository
ValidDeletePart
message() String
groups() Class<?>[]
payload() Class<Payload>[]
InhousePart
InhousePart()
partId int
setPartId(int) void
getPartId() int
Part
Part(long, String, double, int)
Part()
Part(long, String, double, int, int, int)
maximum int
minimum int
price double
products Set<Product>
id long
name String
inv int
toString() String
getPrice() double
getProducts() Set<Product>
hashCode() int
getMinimum() int
setMinimum(int) void
getInv() int
setMaximum(int) void
getName() String
validateLimits() void
setName(String) void
setInv(int) void
equals(Object) boolean
setProducts(Set<Product>) void
getMaximum() int
setPrice(double) void
setId(long) void
getId() long
InhousePartRepository
OutsourcedPart
OutsourcedPart()
companyName String
getCompanyName() String
setCompanyName(String) void

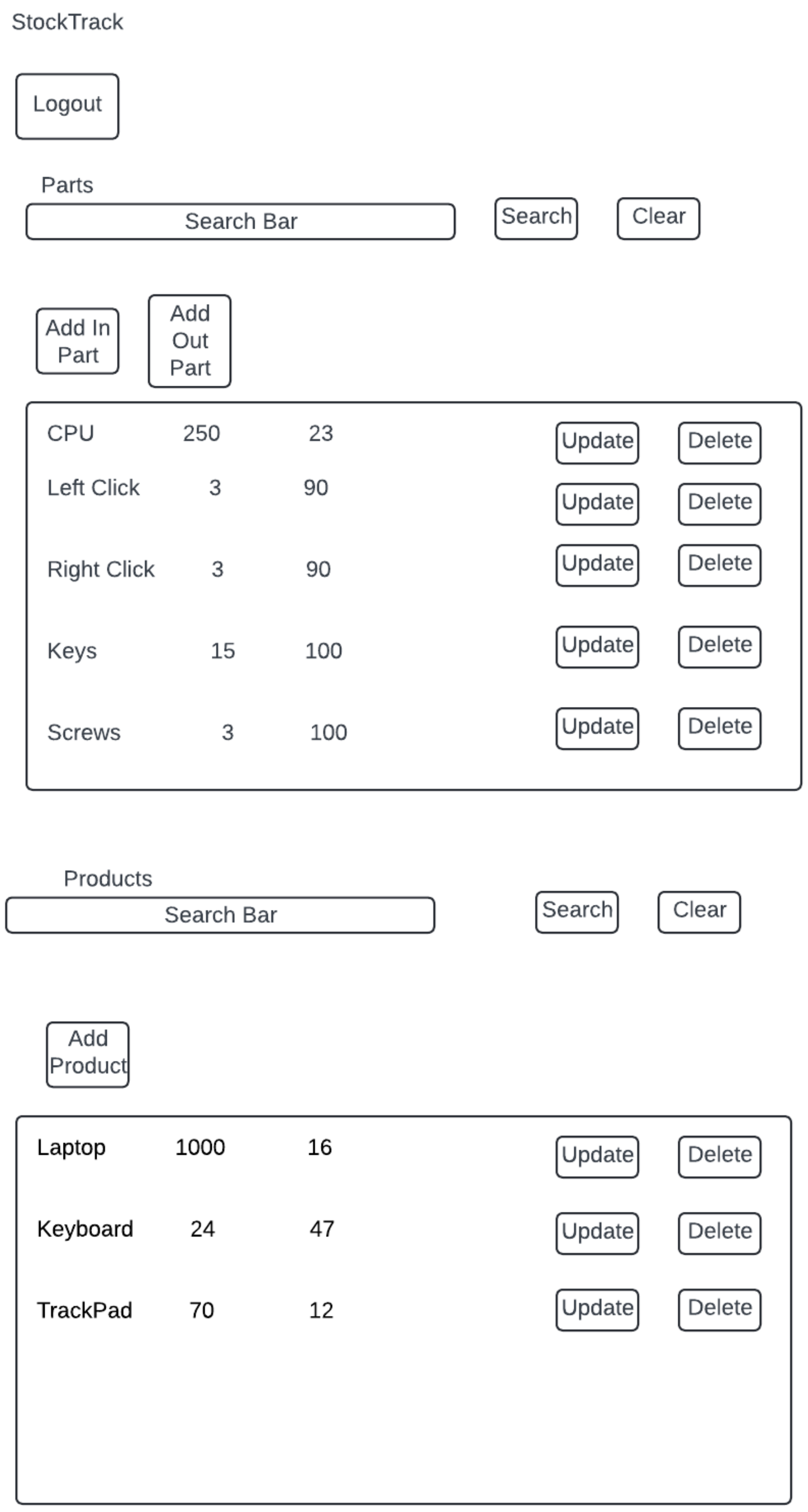
UI Design

The wireframes for StockTrack, both low-fidelity and high-fidelity, are designed to illustrate the user interface and interaction flow of the application at various stages of development. The low-fidelity wireframes provide a foundational overview of the main screens, focusing on layout, essential components, and initial navigation paths without delving into visual details. These initial drafts are intended to facilitate quick feedback and iteration. The high-fidelity wireframes, in contrast, offer a more polished representation, incorporating detailed design elements, color schemes, and typography, reflecting a near-final version of the user experience. Together, these wireframes serve as a visual roadmap guiding the project from concept to implementation, ensuring alignment on both functionality and user interaction.

Low Fidelity Wireframe



High Fidelity Wireframe



CPU	250	23	Update	Delete
Left Click	3	90	Update	Delete
Right Click	3	90	Update	Delete
Keys	15	100	Update	Delete
Screws	3	100	Update	Delete

Laptop	1000	16	Update	Delete
Keyboard	24	47	Update	Delete
TrackPad	70	12	Update	Delete

Unit Test Plan

Introduction

Purpose

The objective here is to verify that the Part class handles minimum and maximum inventory limits correctly. The scope focuses on validating the setters for minimum and maximum inventory (setMinInv, setMaxInv). The tests will run in the test package of the Java project using Junit in IntelliJ. Since the unit tests passed successfully, no code changes were required in these test cases. If any tests had failed, the resulting changes would involve fixing the logic in the setter methods (e.g., ensuring constraints on inventory limits are properly enforced).

Overview

The unit tests for the inventory minimum and maximum values in the PartTest class play a crucial role in ensuring the integrity of StockTrack's inventory management system. These tests focus on validating the functionality of setting inventory levels, which is fundamental to controlling stock flow within the application. Specifically, the tests for setting minimum and maximum inventory values ensure that the system correctly handles boundaries that prevent understocking or overstocking, vital for maintaining an efficient inventory control process. Inventory management is at the core of StockTrack, and these unit tests are part of a larger strategy to maintain data integrity and reliability throughout the application. Although the focus here is on inventory minimums and maximums, similar boundary testing principles were applied across various parts of the application.

Test Plan

Items

To successfully complete the unit tests for the inventory minimum and maximum values in the PartTest class, the following items were required:

- **JUnit 5 testing framework:** This was used for structuring and running the unit tests.
- **Part class and related setters:** The setMinInv() and setMaxInv() methods needed to be implemented and functional in the codebase.
- **Test environment:** IntelliJ IDEA with properly configured project dependencies (e.g., JDK 17, JUnit 5).
- **Mock data:** Parts (partIn, partOut) needed to be instantiated with valid and boundary-case values for the test cases.

Features

- **setMinInv():** This method sets the minimum inventory level for a part.
- **setMaxInv():** This method sets the maximum inventory level for a part.
- **Assertions:** These validate that the set inventory values match expected results under both valid and invalid scenarios.
- **Boundary testing:** Checking for cases where values are set outside acceptable ranges (e.g., negative values for minimum, excessively high values for maximum).

Deliverables

- **Test results:** Success or failure for each unit test case, determining whether the inventory methods behave as expected.
- **Documentation:** Descriptions of each test case, code comments, and error logging to highlight reasons for test failures (if any).
- **Code coverage report:** Generated by IntelliJ IDEA or a coverage tool like JaCoCo, which indicates the percentage of code exercised by unit tests.
- **Test logs:** Logs from successful or failed tests, providing insights into issues that arise during execution.

Tasks

Define Test Objectives

Determine the specific goals of each test, such as verifying minimum and maximum inventory boundaries. The outcome will be clarity on the expectations for each test, ensuring that testing aligns with project requirements.

Set Up Test Environment

Configure the testing environment with all necessary dependencies, tools, and access to the StockTrack codebase. The outcome will be a stable testing environment that mirrors the production setup, ensuring reliable results.

Develop Unit Test Cases

Write unit test scripts targeting each function or feature, particularly focusing on boundary values for the minimum and maximum inventory. The outcome will be comprehensive unit tests that validate critical StockTrack functions and edge cases.

Execute Tests

Run each test case independently and observe results, with each test focusing on the system's response to valid and invalid inputs. The outcome will be test results that confirm whether each function works as expected, catching errors early.

Analyze Test Results

Review the output of each test to confirm if the results match the expected behavior. The outcome will provide insights into functionality gaps, if any, or confirmation of expected system behavior.

Document Findings

Note any discrepancies, errors, or deviations from expected behavior and compile a report on findings. The outcome will be a detailed documentation of test results, including pass/fail outcomes and any identified bugs.

Implement Corrections for Errors (If Necessary)

Refactor code as needed to address issues revealed during testing, such as incorrectly handled input. The outcome will be corrected functionality with improved code stability and reliability for the tested features.

Retest After Changes

Run tests again to verify that any adjustments have resolved previously identified issues without introducing new ones. The outcome will be confirmed resolution of errors, ensuring each feature performs as expected under various conditions.

Finalize and Record Results

Compile a final report of the test outcomes, including successful tests, any remaining known issues, and recommendations. The outcome will be a clear record of the testing phase, providing visibility into the product's reliability and readiness for deployment.

Outcomes Identified:

Through these tasks, the testing process produced the following outcomes:

- Verification of the minimum and maximum inventory validation.
- Identification and resolution of any boundary-related issues.
- Confirmation of StockTrack's stability and accuracy in handling critical input constraints.

- Documentation of test procedures and results, enabling a clear view of system reliability and any further testing needs.

This approach ensures that StockTrack's key functionalities are robustly tested and meet the project's quality standards.

Needs

To perform the tests for StockTrack effectively, several components and support items needed to be in place. These requirements ensured the testing environment was stable and closely matched the production setup. Here are the technical requirements and necessary support items:

Environment Setup and Technical Requirements:

1. **Java Development Kit (JDK)**

   JDK 17 was required for compatibility with StockTrack's application code. It Provides the runtime and development environment necessary to execute Java applications and tests.

2. **Spring Framework and Spring Boot**

   Spring Boot 3.0 was used as the core framework for the StockTrack application. It Supports the setup of application context and dependency injection, which are critical for testing controller and service layers.

3. **Testing Frameworks and Libraries**

   JUnit 5 was used for defining and running unit tests. It allows structured testing with assertions, lifecycle methods, and robust test organization.

4. **Database Setup**

   Using a repository-based database structure allowed for realistic CRUD operations (create, read, update, delete) during testing, facilitating reliable integration and unit testing of data access and validation.

5. **IDE and Build Tools**
   - **IntelliJ IDEA**:
     - **Version**: Latest version compatible with Java 17 (e.g., IntelliJ IDEA 2023.2).
     - **Purpose**: Used for code development, test execution, and debugging. Integrated test runner and code inspection tools aid in test-driven development (TDD).
   - **Maven**:

     Maven 3.8+ was used for dependency management and build automation. It manages libraries and testing dependencies like Junit ensuring compatibility and easy setup.

Support Items Needed:

- **System Requirements**: A Windows-based machine (user's current setup) with at least 8GB of RAM to support IntelliJ IDEA and the test environment.
- **Network Access**: Required for fetching dependencies and, optionally, accessing the AWS environment if integration testing is performed.
- **Documentation**: Access to StockTrack's requirements and specifications ensured tests aligned with business and functional needs.

These elements collectively ensured that the testing environment was capable of running the tests efficiently and replicating expected production conditions, contributing to reliable, consistent test outcomes.

Pass/Fail Criteria

Each test in the StockTrack application had specific criteria for determining success based on the functional and performance requirements of the application. The criteria included the following:

Assertion Matching:

Each test's primary criterion was that all assertions should pass without error. This means that actual outputs from the code matched the expected outputs precisely. For example, in tests related to minimum and maximum inventory values, assertions confirmed that values were set and retrieved accurately in accordance with business rules.

Error-Free Execution:

The test should execute without any runtime exceptions, such as NullPointerException or IllegalArgumentException, which would indicate flaws in handling invalid inputs or null checks.

Code Performance:

Although not extensively measured in unit tests, tests that executed in minimal time (under milliseconds) were deemed efficient, which confirmed that the methods tested were lightweight and optimized.

Mock Validation:

For components that relied on external dependencies, mock objects were validated to confirm that expected methods were called and the interactions behaved as designed. This ensured correctness without requiring a live connection to the production environment.

Edge Case Handling:

Tests were run against typical, boundary, and edge cases (such as zero or negative inventory) to confirm the application's robustness. Success was achieved if these cases produced correct, expected behavior without errors.

Protocol for a Positive Result

Upon meeting the success criteria, the test results were reviewed and documented. The protocol for a positive result included the following steps:

Mark Test as Passed:

- Each passing test was marked as passed within IntelliJ IDEA or the test reporting tool used (Maven for test reports).

Documentation of Results:

- Successful test results were documented, with screenshots taken for future reference or inclusion in project documentation. Logs were reviewed to verify expected behavior and performance, and any relevant information was added to the test records.

Git Commit:

- A commit was made to the code repository with a note indicating successful test results, ensuring that any changes made during testing were preserved.

Protocol for Failed Tests

If a test failed to meet the criteria, a structured protocol was followed to identify the cause and correct the issue. The protocol included:

Error Analysis and Debugging:

- The cause of failure was immediately investigated by examining the test output, exception messages, and logs. IntelliJ's debugging tools were employed to step through the code and pinpoint the source of the failure.

Remediation Strategies:

- **Code Review**: The test and associated code were reviewed to check for any errors in logic, data handling, or assumptions in the test script.
- **Mock Adjustment**: If mocks were used, the mock configurations were reviewed to ensure they accurately represented real-world scenarios.
- **Re-run of Tests**: Once issues were addressed, the test was re-run to verify that the issue had been resolved.

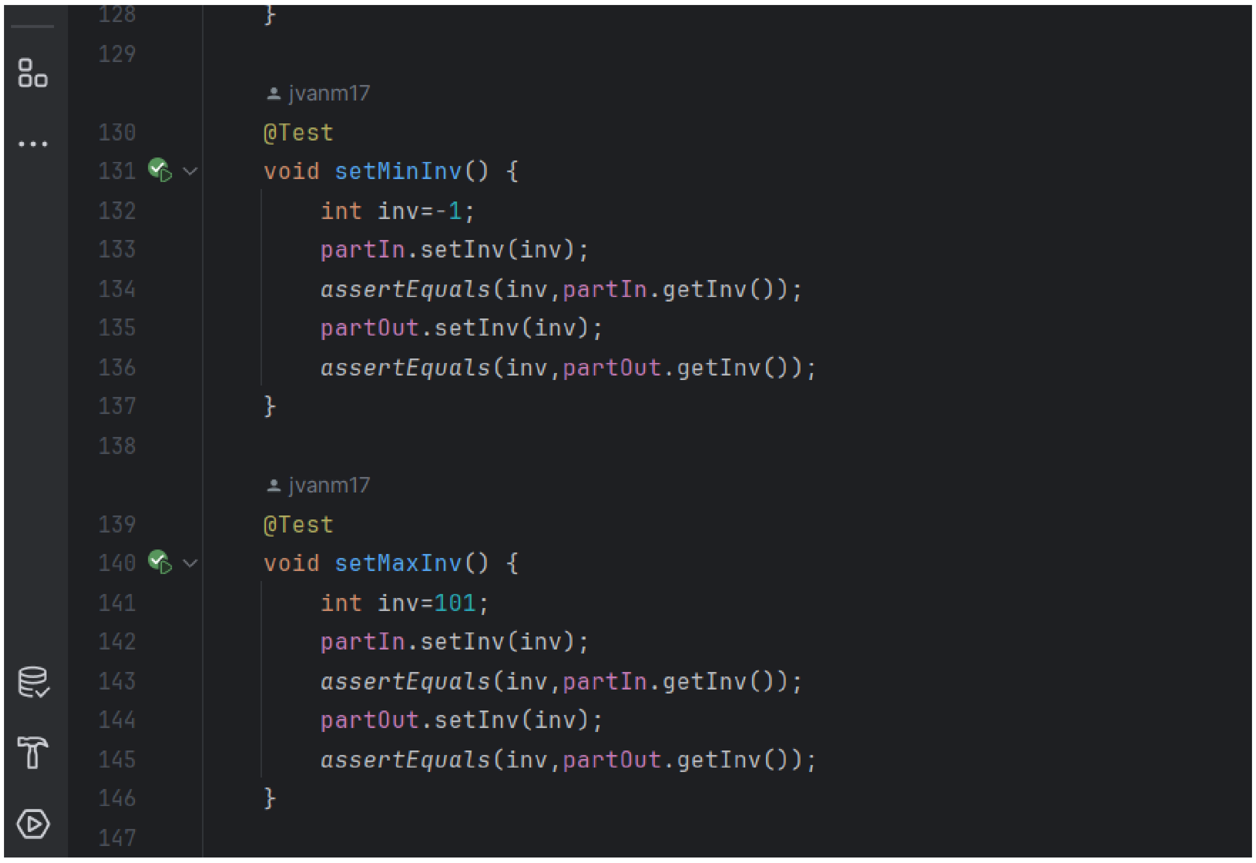
Documentation Requirements:

- **Issue Tracking**: Failures and their causes were documented in an issue tracking system (such as Jira) if the failure could not be resolved immediately. This allowed for assigning priorities and tracking progress.
- **Test Log Updates**: Detailed notes were added to the test documentation, including the initial failure reason, the changes made to resolve it, and screenshots of the final passing results.

- **Commit with Reference to Fix**: If code modifications were needed, they were committed to the repository with a reference to the failed test case and the solution implemented.

This approach ensured each test was rigorously validated, with any failed cases thoroughly investigated, remediated, and documented to improve the robustness of StockTrack and maintain a reliable testing history.

Specifications



```
    }

    jvanm17
    @Test
    void setMinInv() {
        int inv=-1;
        partIn.setInv(inv);
        assertEquals(inv,partIn.getInv());
        partOut.setInv(inv);
        assertEquals(inv,partOut.getInv());
    }

    jvanm17
    @Test
    void setMaxInv() {
        int inv=101;
        partIn.setInv(inv);
        assertEquals(inv,partIn.getInv());
        partOut.setInv(inv);
        assertEquals(inv,partOut.getInv());
    }
```

Procedures

Here is a detailed list of steps used to complete the testing process for StockTrack, outlining the iterative nature of the process and specifying when pass/fail results were recorded and analyzed.

1. Initial Test Planning

Define test objectives and identify specific components, methods, or functions of the StockTrack system to be tested, focusing on critical functionalities like inventory management, part addition, and validation. Create test cases and develop individual test cases for each functionality, focusing on both normal operation and edge cases (e.g., setting minimum/maximum inventory limits, handling invalid input). Select testing tools and set up the testing framework, including JUnit for Java-based unit tests, and configure test execution within IntelliJ IDEA. Set pass/fail criteria for test success, which includes correct output (assertions passing), handling of invalid inputs, and proper logging of error messages.

2. Prepare the Testing Environment

Configure IntelliJ IDEA for testing and Ensure the test package structure is set up, and confirm that test classes are recognized by the IDE and executable within the test framework. If applicable, configure mock data or mock services for test cases that involve dependencies like databases or external services. Take snapshots or baseline references of the current code to track changes and identify potential sources of errors.

3. Run Initial Test Iterations

Run the first round of test cases to verify that basic functionality works as expected and to establish a baseline for identifying bugs or failures. After each test, capture pass/fail results immediately. If a test passes, log the success. If it fails, note the failure along with error details or output discrepancies. For any failed tests, review error messages, debug the code, and identify any issues in the logic or input handling. Adjust the code as necessary and prepare for re-testing.

4. Iterative Testing and Debugging

Adjust test cases if any assumptions change or if additional cases are identified (such as edge cases discovered during initial testing). For any code changes made during debugging, re-run the affected tests and ensure all related functionality is retested. This iterative approach allows for refining both code and test accuracy. Continue running and refining tests until each test case consistently produces a pass result under all expected conditions.

5. Final Test Execution and Validation

Once individual tests are consistently passing, run the entire test suite to confirm that all parts of the application work in tandem without errors. Verify that each test has passed and no unintended interactions or bugs have been introduced by recent changes. Record the final pass/fail results of each test, along with any notable findings or adjustments made during testing. Take screenshots of test results as required.

6. Post-Test Documentation and Cleanup

Create a summary of the testing process, including test objectives, final results, any changes to the code or tests, and any issues encountered along the way. Based on test outcomes, make final refactoring to improve code clarity, efficiency, or maintainability. Once all tests are validated as passing, commit the final code changes to the repository with appropriate commit messages and documentation.

Iteration and Pass/Fail Results

The process of running tests, analyzing failures, and modifying code was iterative. After each iteration, pass/fail results were recorded, and any code changes were documented and re-tested. Each pass/fail result was documented immediately after testing. In cases of failure, steps to address the issue were implemented and tested again to ensure a successful outcome in subsequent runs.

Results

The testing process for StockTrack involved multiple test cases focused on validating core functionalities, such as inventory control through minimum and maximum values, part creation, and field validation. Each test case was designed to validate individual units of functionality and ensure consistent and reliable operation across the application. Below are examples and descriptions of key tests and their results.

1. Test Case: Setting Inventory Minimums and Maximums

- **Objective**: Ensure that the setMinInv and setMaxInv methods in the Part class correctly set inventory minimum and maximum values and handle invalid inputs appropriately.
- **Test Details**:
  - **Input**: Set inventory to a negative value for setMinInv and above a specified limit for setMaxInv.
  - **Expected Output**: The system should set the inventory to the specified minimum or maximum value and throw an error or log a message if the value is out of range.
- **Results**:

- **Pass**: The test passed, showing that when a value below zero was input for setMinInv, it was successfully handled, and the assertion validated the expected outcome.

2. Test Case: Validating Part Creation and Data Integrity

- **Objective**: Ensure that the AddInhousePartController and AddOutsourcedPartController can create parts with the correct attributes, validate data fields, and prevent invalid data entry.
- **Test Details**:
  - **Input**: Populate required fields (name, inventory, price) with valid and invalid data (e.g., leaving the price field blank or entering a non-numeric value).
  - **Expected Output**: The test should validate that all fields are correctly populated. If a required field is missing or invalid, the application should not allow part creation.
- **Results**:
  - **Pass**: The test confirmed that data integrity checks were in place and prevented the creation of parts when mandatory fields were left blank.

4. Test Case: System Boundary Conditions for Inventory

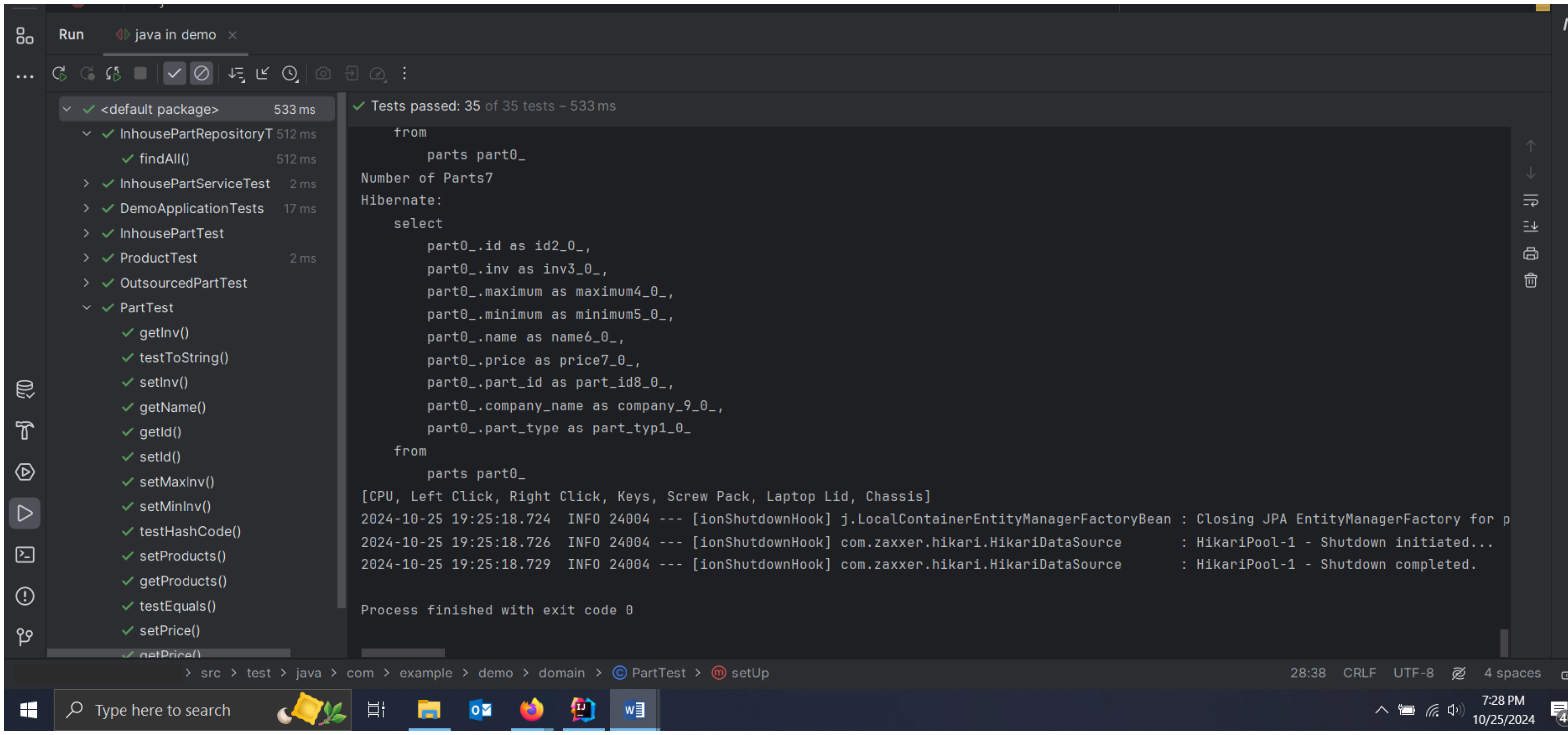
- **Objective**: Ensure that the inventory limits work within the defined boundary conditions and prevent operations outside allowed inventory ranges.
- **Test Details**:
  - **Input**: Set inventory to the exact minimum, maximum, and slightly above/below these limits.

  - **Expected Output**: Operations within limits should succeed, and attempts to exceed limits should throw an error or be prevented.

- **Results**:
  - **Pass**: The boundary conditions held true, confirming that inventory management behaves as expected within specified minimum and maximum values.
  - **Example**:

Summary of Testing Outcomes

Overall, the testing process confirmed that StockTrack's primary functionalities were working as expected. Each test case resulted in clear pass/fail outcomes, which helped refine both the code and validation logic in the application.


Run
java in demo
src > test > java > com > example > demo > domain > PartTest > setUp
28:38 CRLF UTF-8 4 spaces
Type here to search
7:28 PM
10/25/2024

```
✓ <default package>            533 ms
  ✓ InhousePartRepositoryT     512 ms
    ✓ findAll()                512 ms
  ✓ InhousePartServiceTest       2 ms
  ✓ DemoApplicationTests        17 ms
  ✓ InhousePartTest
  ✓ ProductTest                  2 ms
  ✓ OutsourcedPartTest
  ✓ PartTest
    ✓ getInv()
    ✓ testToString()
    ✓ setInv()
    ✓ getName()
    ✓ getId()
    ✓ setId()
    ✓ setMaxInv()
    ✓ setMinInv()
    ✓ testHashCode()
    ✓ setProducts()
    ✓ getProducts()
    ✓ testEquals()
    ✓ setPrice()
    ✓ getPrice()
```

```
✓ Tests passed: 35 of 35 tests – 533 ms
    from
        parts part0_
Number of Parts7
Hibernate:
    select
        part0_.id as id2_0_,
        part0_.inv as inv3_0_,
        part0_.maximum as maximum4_0_,
        part0_.minimum as minimum5_0_,
        part0_.name as name6_0_,
        part0_.price as price7_0_,
        part0_.part_id as part_id8_0_,
        part0_.company_name as company_9_0_,
        part0_.part_type as part_typ1_0_
    from
        parts part0_
[CPU, Left Click, Right Click, Keys, Screw Pack, Laptop Lid, Chassis]
2024-10-25 19:25:18.724  INFO 24004 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory for p
2024-10-25 19:25:18.726  INFO 24004 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
2024-10-25 19:25:18.729  INFO 24004 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.

Process finished with exit code 0
```

C2. http://ec2-44-204-44-100.compute-1.amazonaws.com/

C3. https://gitlab.com/wgu-gitlab-environment/student-repos/jvanm17/d424-software-engineering-capstone/-/tree/work_branch?ref_type=heads

Maintenance User Guide

Introduction

This guide is designed for developers who wish to modify the application to meet their specific use cases or to conduct maintenance on the application's code. Ongoing maintenance is generally minimal, primarily involving updates to dependencies as new versions become available. For information on using the application from a user's perspective, please refer to the User Guide (User) located below this Maintenance User Guide. This document will help you build and run the application within IntelliJ, enabling you to enhance its functionality.

Prerequisites

• Intellij Idea or community edition

• Java Development Kit 17

• spring-boot-maven-plugin version 2.6.6

Steps to build and run in Intellij to perform application maintenance

1. Open IntelliJ and start a new project from version control.

2. Paste in the link the repository link and click clone:

https://gitlab.com/wgu-gitlab-environment/student-repos/jvanm17/d424-software-engineering-capstone.git

3. Using the git tool inside IntelliJ, right click on work_branch and checkout

4. A pop-up asking to load maven may appear, click load

5. Click the run button at the top shaped like a triangle to run the current version just pulled from Git.

User Guide

Introduction

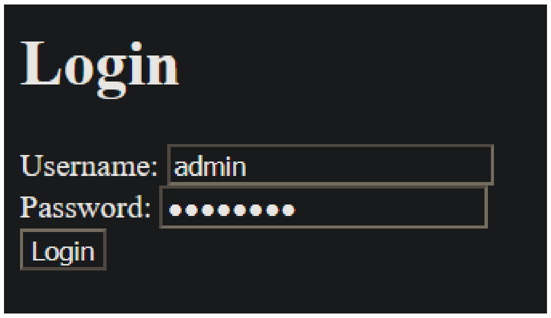
This guide provides step-by-step instructions for using StockTrack, a comprehensive web-based inventory management system designed to help you easily manage, track, and optimize your inventory. Whether you're adding new items, updating stock levels, or generating reports, this guide will walk you through all essential features to make the most of StockTrack in your daily operations.

Installation and Using the Application

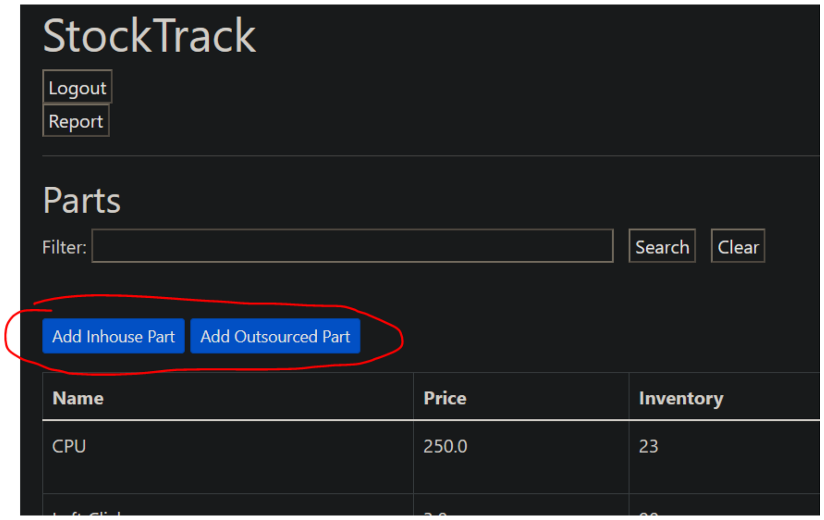
1.Log into StockTrack using either of the testing credentials the click the login button.
(If using user & user123 for username and password your experience will be limited to the customer view and you will only be able to view and purchase products from the site.)

Usernames: admin, user

Passwords: admin123, user123



2.(ADMIN ONLY) Create either Inhouse Part or Outsourced Part (both are similar in function) by clicking one of the two buttons.


StockTrack

Logout
Report

Parts

Filter: Search Clear

Add Inhouse Part Add Outsourced Part

Name	Price	Inventory
CPU	250.0	23

3.Fill out the details listed on the page then click 'Submit'.

Part Name

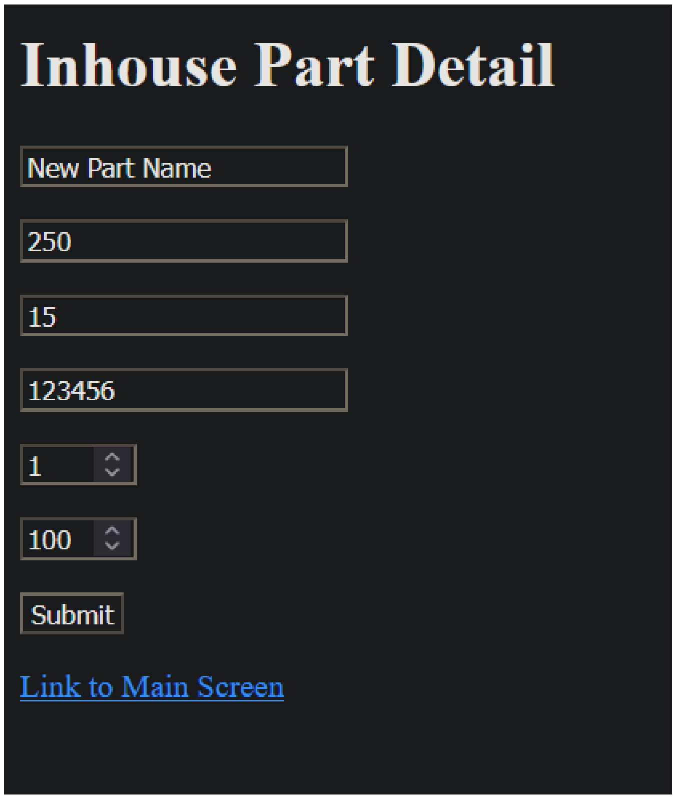
Part Price

Number in Inventory

PartID

Minimum Inventory Allowed

Maximum Inventory Allowed


Inhouse Part Detail

New Part Name

250

15

123456

1

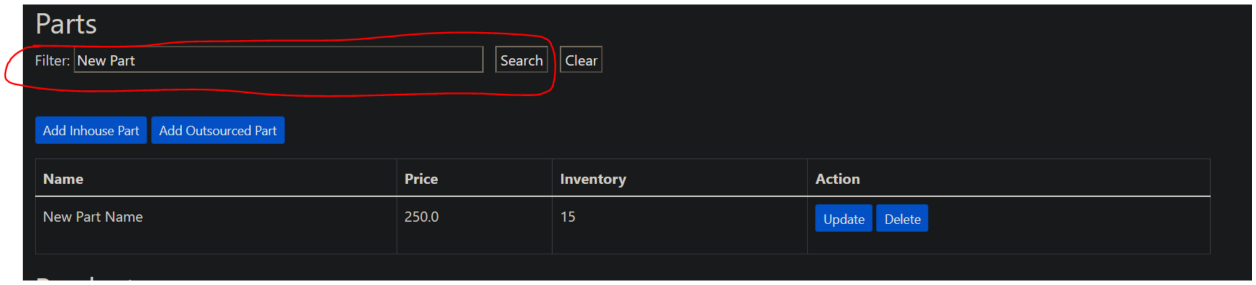
100

Submit

Link to Main Screen

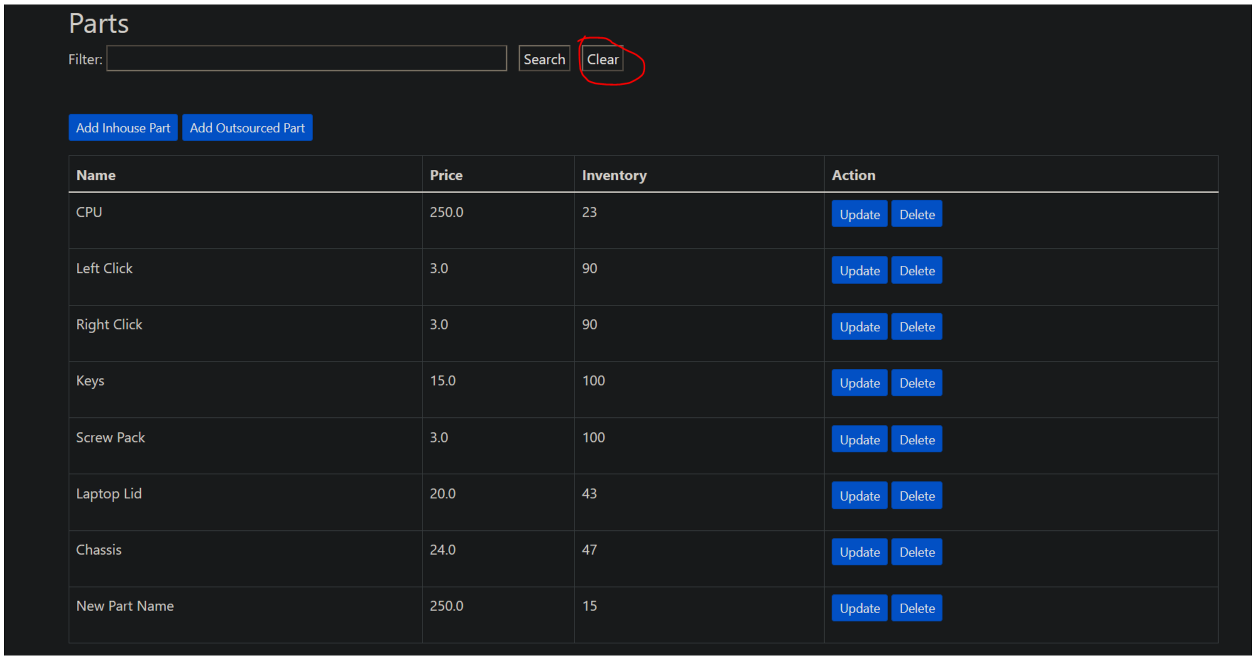
You will then be brought back to the main page where you will see your newly added part in the list of parts displayed. Here you can update and delete these parts as needed with relevant buttons next to each product.

4.You are able to search for any part in the list by clicking in the search bar, typing in your keyword(s) then click search to submit.



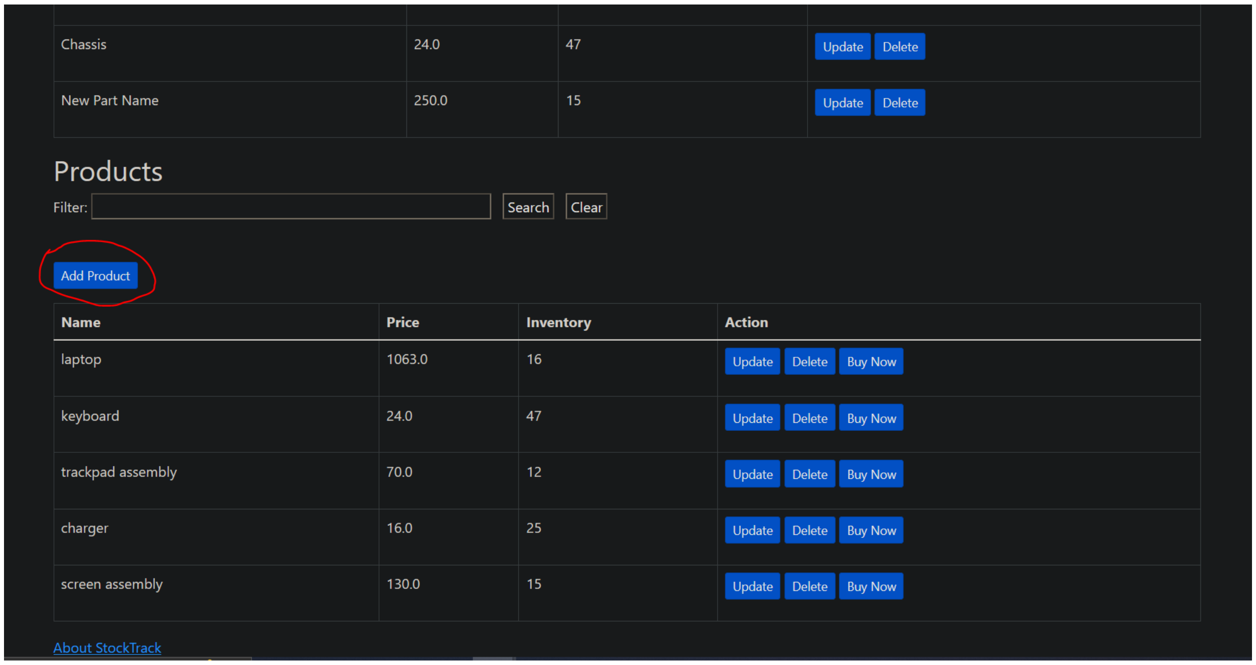
Name	Price	Inventory	Action
New Part Name	250.0	15	Update Delete

5.Clear can be clicked to view the entire list again.



Name	Price	Inventory	Action
CPU	250.0	23	Update Delete
Left Click	3.0	90	Update Delete
Right Click	3.0	90	Update Delete
Keys	15.0	100	Update Delete
Screw Pack	3.0	100	Update Delete
Laptop Lid	20.0	43	Update Delete
Chassis	24.0	47	Update Delete
New Part Name	250.0	15	Update Delete

6.Products can be added by clicking the Add Product button below the list of Parts.



Chassis	24.0	47	Update Delete
New Part Name	250.0	15	Update Delete

Products

Filter: Search Clear

Add Product

Name	Price	Inventory	Action
laptop	1063.0	16	Update Delete Buy Now
keyboard	24.0	47	Update Delete Buy Now
trackpad assembly	70.0	12	Update Delete Buy Now
charger	16.0	25	Update Delete Buy Now
screen assembly	130.0	15	Update Delete Buy Now

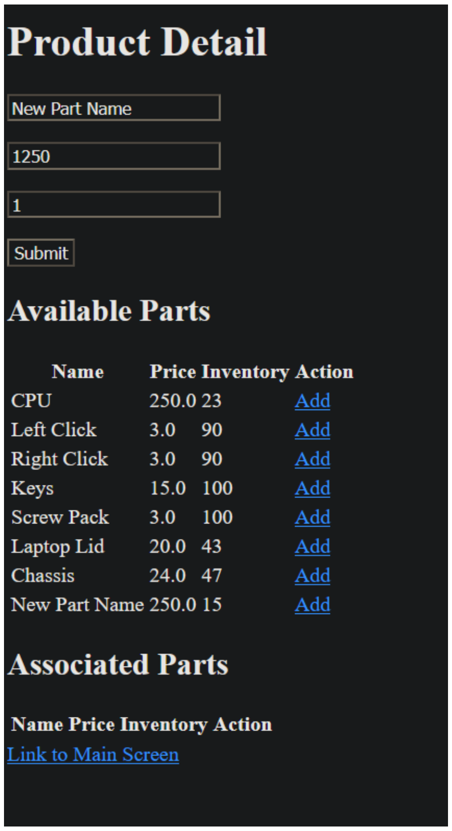
About StockTrack

7.Fill out the product details then click submit. Parts can be associated to the product after it is saved (you will receive an error messages stating this).

Product Name

Product Price

Number in Inventory



Product Detail

New Part Name

1250

1

Submit

Available Parts

Name	Price	Inventory	Action
CPU	250.0	23	Add
Left Click	3.0	90	Add
Right Click	3.0	90	Add
Keys	15.0	100	Add
Screw Pack	3.0	100	Add
Laptop Lid	20.0	43	Add
Chassis	24.0	47	Add
New Part Name	250.0	15	Add

Associated Parts

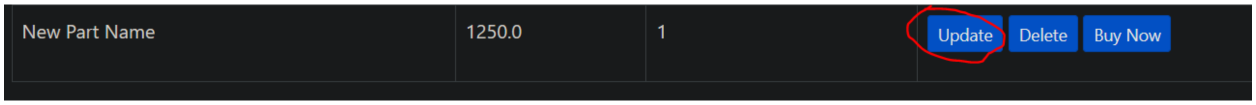
Name	Price	Inventory	Action

Link to Main Screen

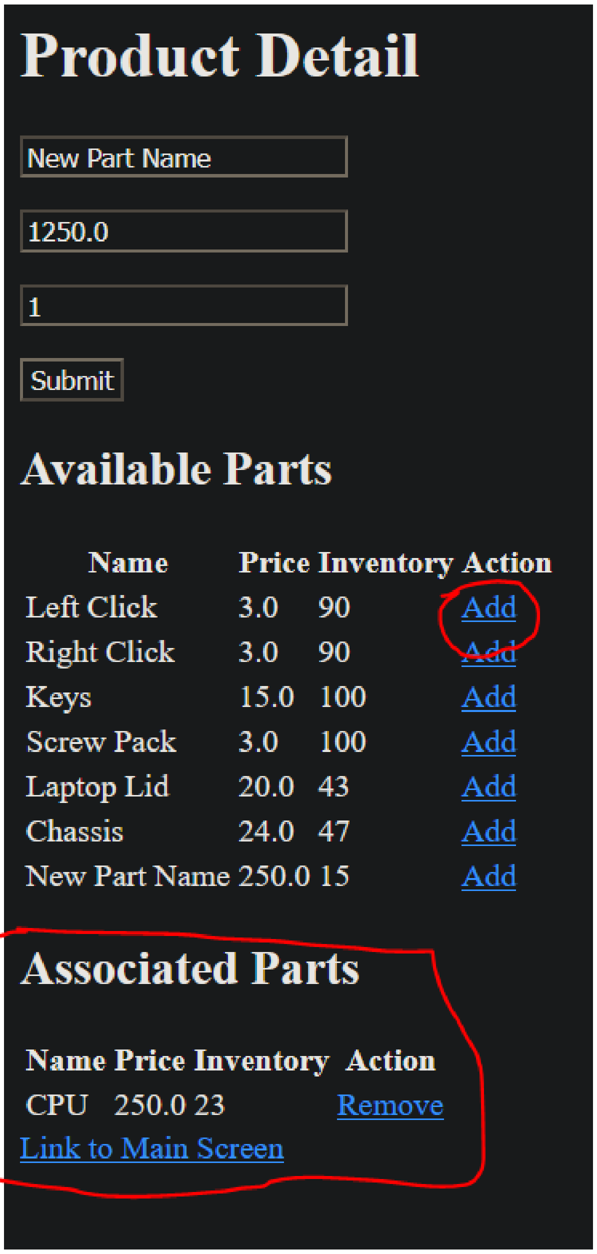
You will then be brought back to the main page again where you will see your newly add product at the bottom. The same search function is available for product as well and works in the same manner. Products also have the same update and delete buttons as the parts do and work in the same manner.

8.Parts can be associated to products by updating them after they are saved by clicking the update button. Once on the Product Details page, parts can be associated by clicking Add and will show under Associated Parts at the bottom. Finalize by clicking Submit

Side note – Increasing a product's inventory that has associated parts will decrease that part's inventory based on the amount of product being added.



New Part Name	1250.0	1	Update Delete Buy Now



Product Detail

New Part Name

1250.0

1

Submit

Available Parts

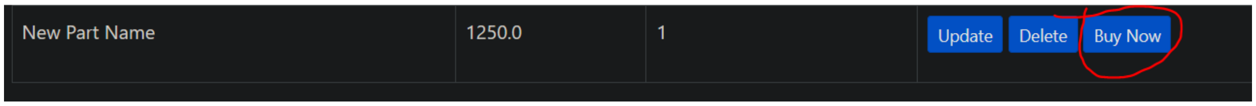
Name	Price	Inventory	Action
Left Click	3.0	90	Add
Right Click	3.0	90	Add
Keys	15.0	100	Add
Screw Pack	3.0	100	Add
Laptop Lid	20.0	43	Add
Chassis	24.0	47	Add
New Part Name	250.0	15	Add

Associated Parts

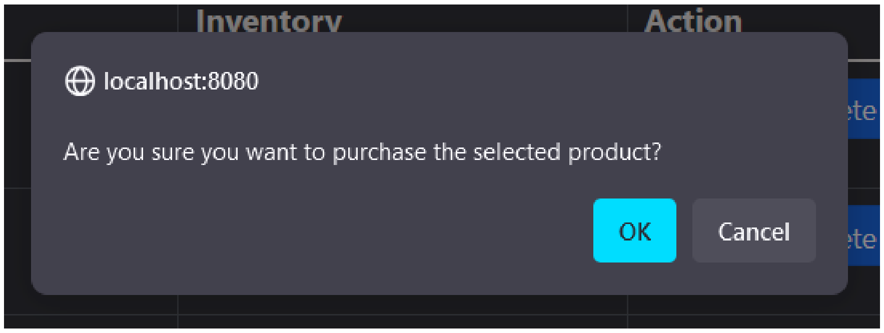
Name	Price	Inventory	Action
CPU	250.0	23	Remove

Link to Main Screen

9.Products are what the company sells so these are the only items that have the “Buy Now” button available. You will get a pop-up asking for confirmation of purchase and clicking OK will confirm the transaction. After which you will get either a confirmation or relevant error screen. After purchasing a product inventory will update as needed.

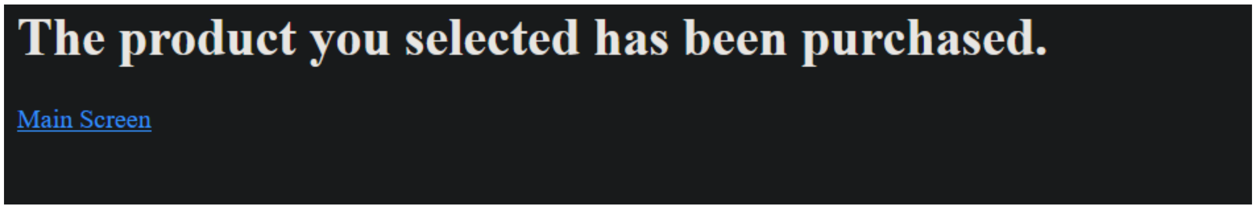


New Part Name	1250.0	1	Update Delete Buy Now



localhost:8080

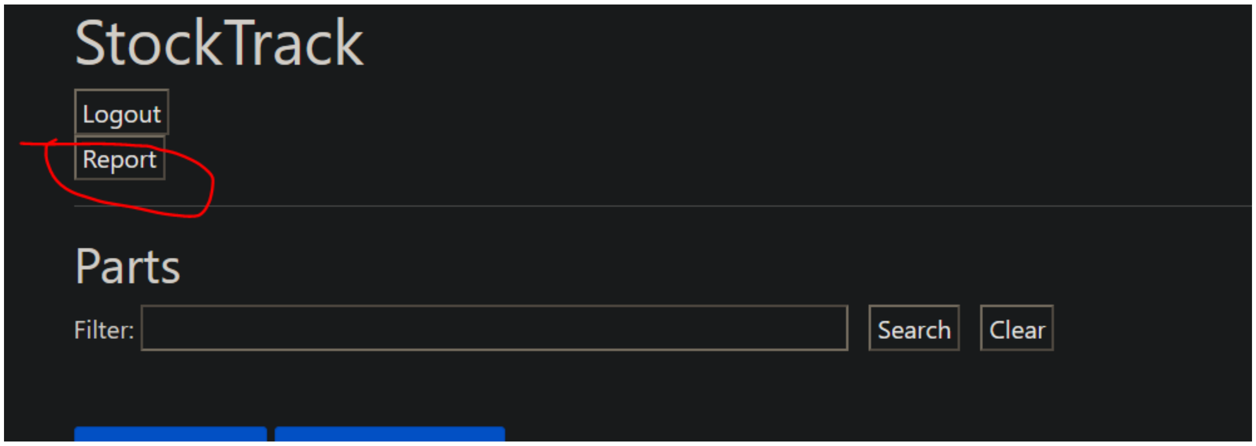
Are you sure you want to purchase the selected product?

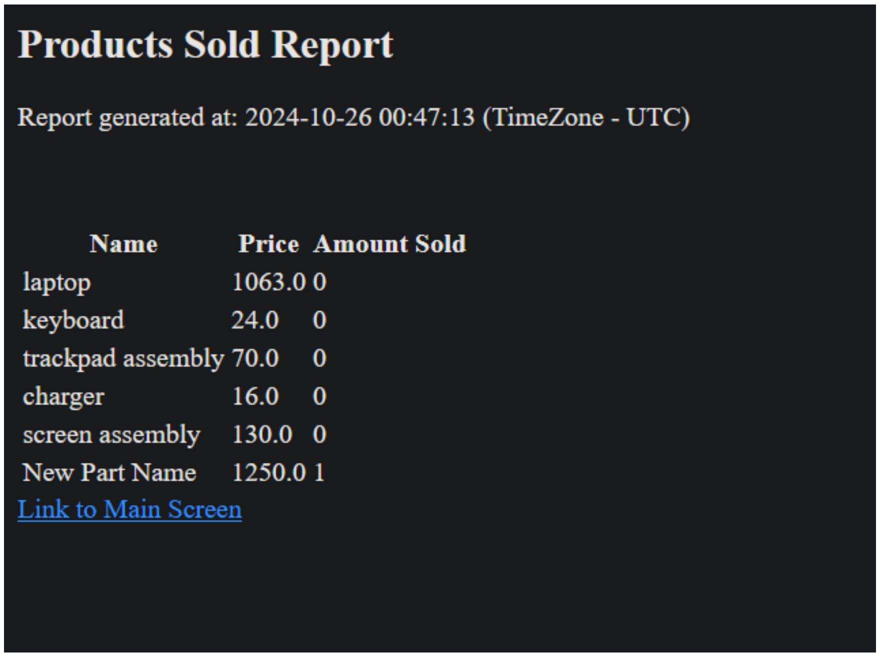


The product you selected has been purchased.

Main Screen

10. The main page has a report button available to see the products that have been sold on the site.





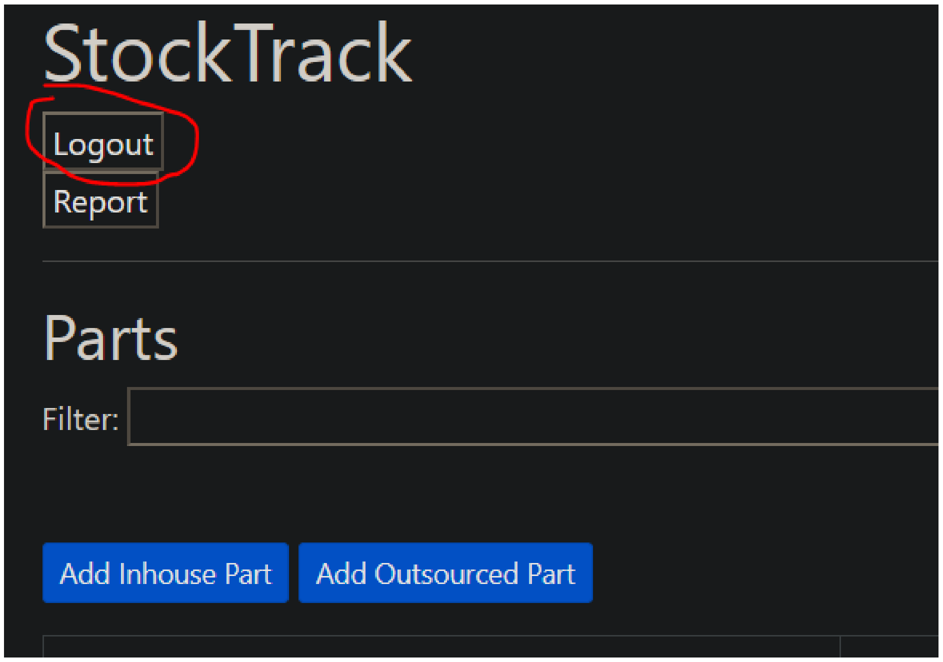
Products Sold Report

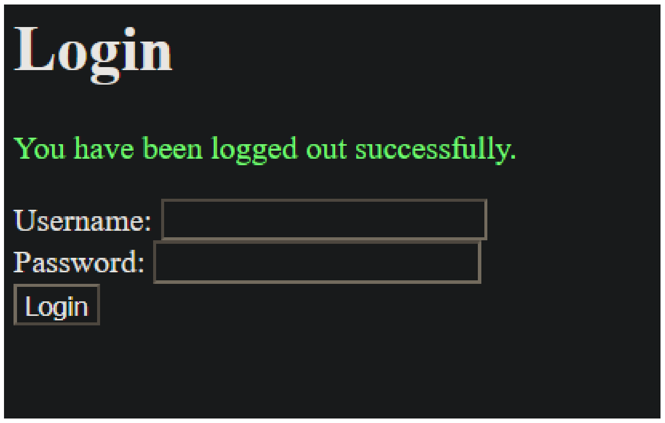
Report generated at: 2024-10-26 00:47:13 (TimeZone - UTC)

Name	Price	Amount Sold
laptop	1063.0	0
keyboard	24.0	0
trackpad assembly	70.0	0
charger	16.0	0
screen assembly	130.0	0
New Part Name	1250.0	1

Link to Main Screen

11. The logout button will log you out and bring you back to the login page with a successful logout message.





You have been logged out successfully.

****No sources were used in the creation of this document****