Project Report

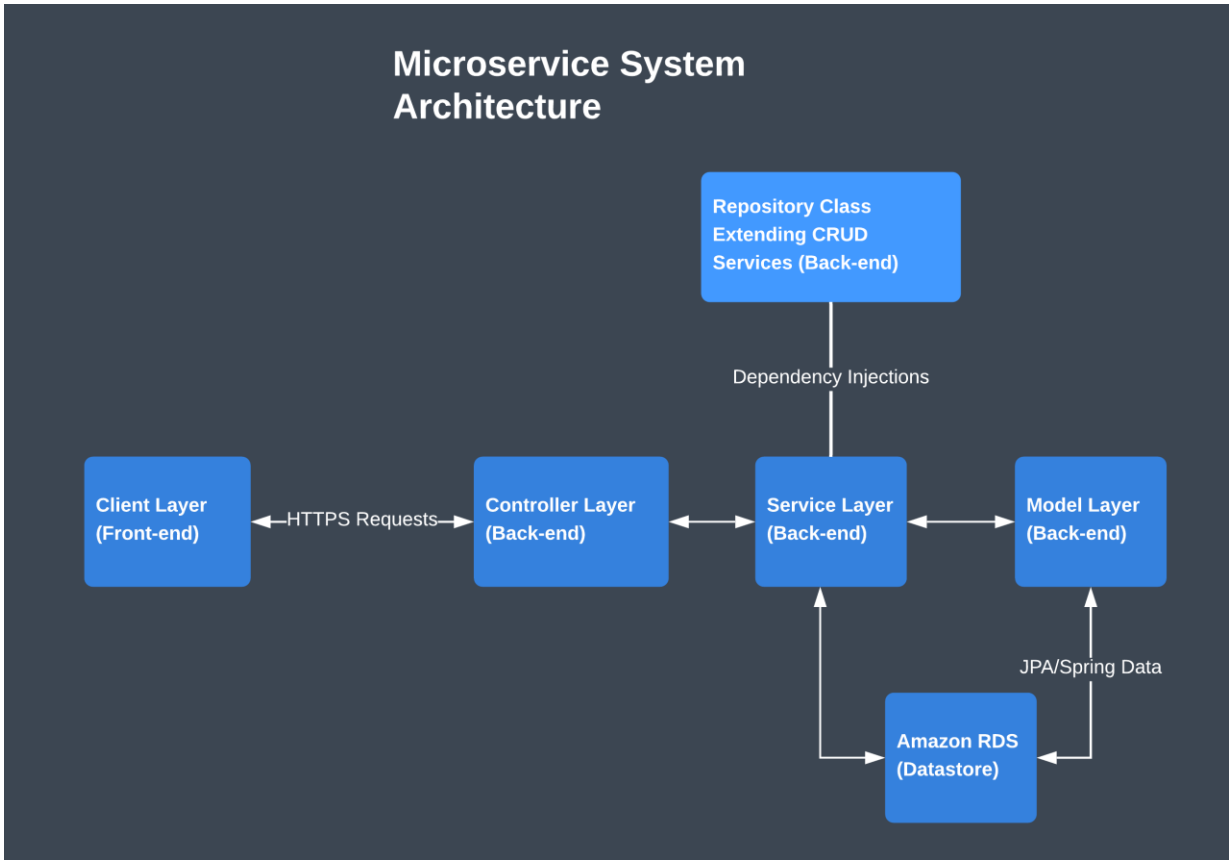
Vision Statement:

We believe that our product is valuable as it demonstrates the team's ability to create a product as requested by a client or product owner. Although we were initially very inexperienced with the language and framework that was required, our team overcame challenges together and was able to quickly learn new technologies such as AWS cloud deployment, CircleCI and Docker. Our work shows that every person on the team is capable, resilient, determined and will address the needs of a customer quickly and effectively. Having documented all our progress, meetings, and retrospectives for each sprint, we are able to illustrate how our initial setbacks and slow progress was only temporary, as our velocity for each sprint increased which shows our adaptability and rapid improvement in skills over time.

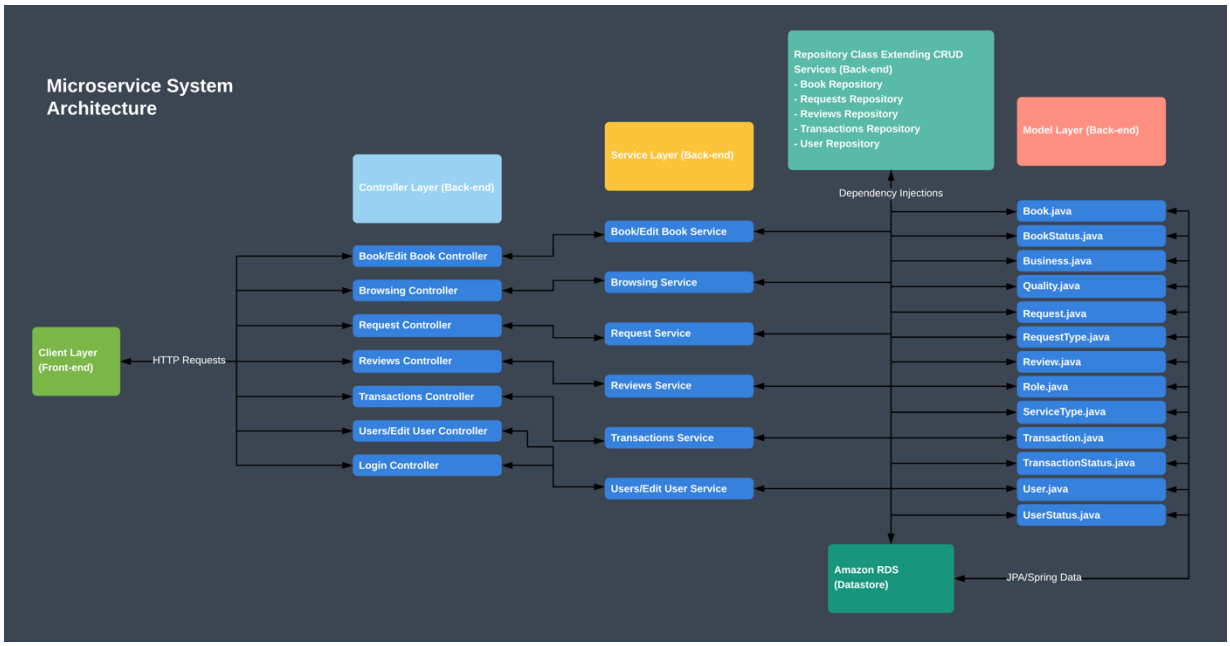
While our product is not suitable for full-scale commercial use, it has the groundwork and potential to be used as a fully functioning e-commerce website for any business that is interested in buying, sharing, and selling books for users and businesses, while admins can oversee and manage all business operations. Given more time, we believe we could continue to adjust and improve our application as required and continue to deliver on all promised functionality as we have done.

System Architecture/Design

Base Diagram:



Full Diagram: (for better quality, this image can be found in the Scrum folder in the root directory)



Refactoring Report

In the original base code provided to us, the starting microservice given was the login microservice, which included a user class and methods for saving new users to a user repository. Due to our unfamiliarity and lack of knowledge in implementing Spring Boot as a back-end framework, all our new classes and services were added to this login microservice, including the books, browsing, transactions, requests, and reviews microservices. This monolithic architecture did have some advantages, such as easy and fast development, testing and tracing, and deployment. These factors were especially helpful as it allowed us to quickly build our starting application with some basic features while we were still learning how to write code for Spring Boot.

However, as our experience and knowledge become more consolidated over the course of the project, we found that there were several disadvantages to our monolithic architecture, such as dependencies between services; for example, if there was an error or bug with compiling any classes related to the users microservice, this would create a ripple effect and inhibit us from running all our other services. This slowed down our development speed and make scaling our services more difficult as most services were dependent on others, making our code unreliable at times and inflexible. To solve these issues, we decided to refactor our design pattern to follow a microservice architecture.

To refactor our code, we initially planned to create a common library or service, such as the new independent users microservice, and then have all other microservices include it as a dependency, making use of asynchronous communication. However, we believed that this would not help us solve all our issues associated with the monolithic architecture and decided to create each microservice so that they could be developed and deployed independently, without being dependent on other services. This allowed us to make changes and updates to scale each service when required and prevented any issues with microservices from causing other services to crash, making our code more resilient.

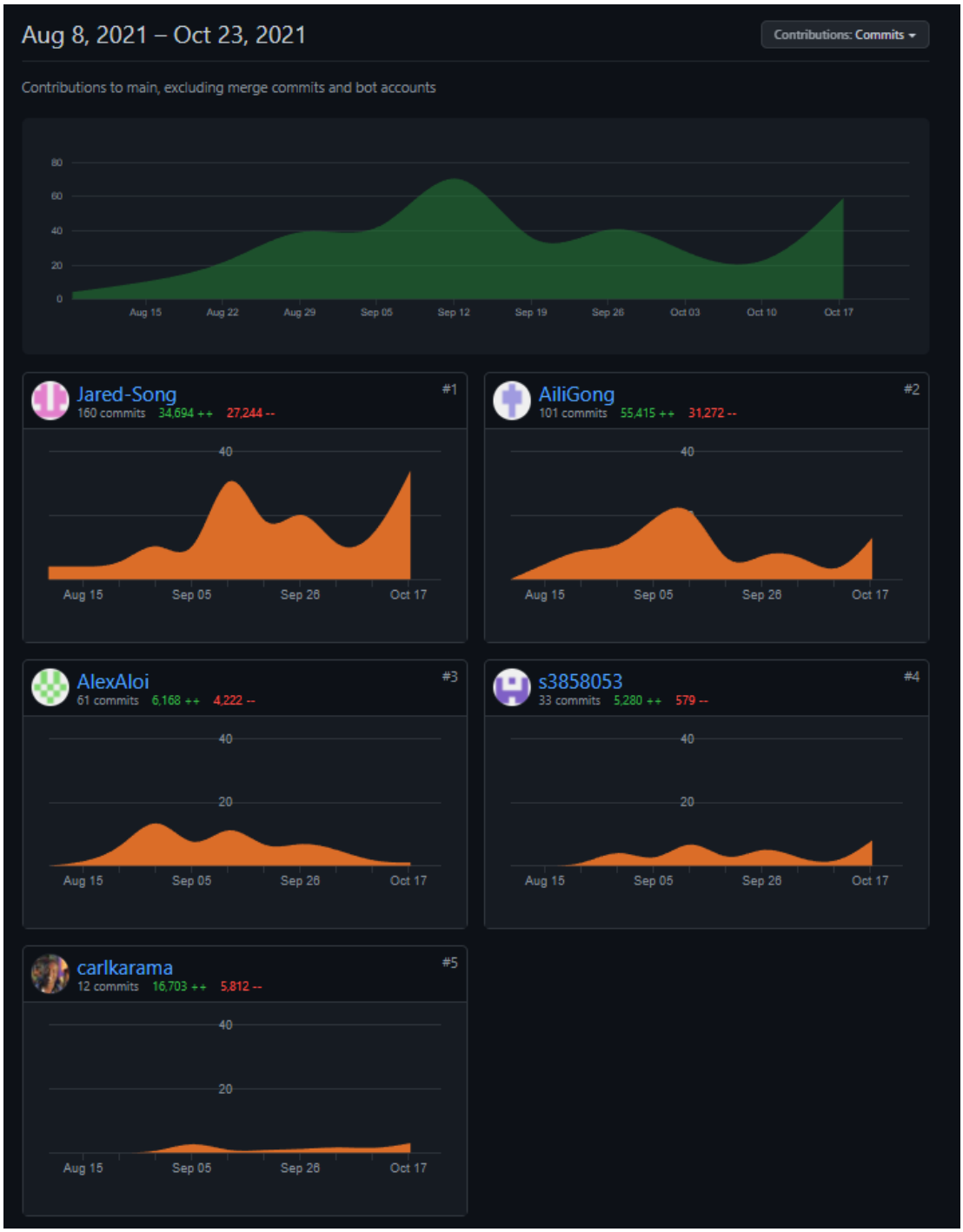
Currently, each microservice is responsible for handling queries only related to their microservice, for example the users microservice is required to register a new user, edit user information, or delete users, while the reviews microservice is required to leave reviews and ratings for users and books. If any microservice is not deployed and run, the other services will still function as required, but the terminated service will not be able to be called or used. Each service is also connected to the same datastore we set up on Amazon RDS, and thus any changes each service makes will be shared across all services.

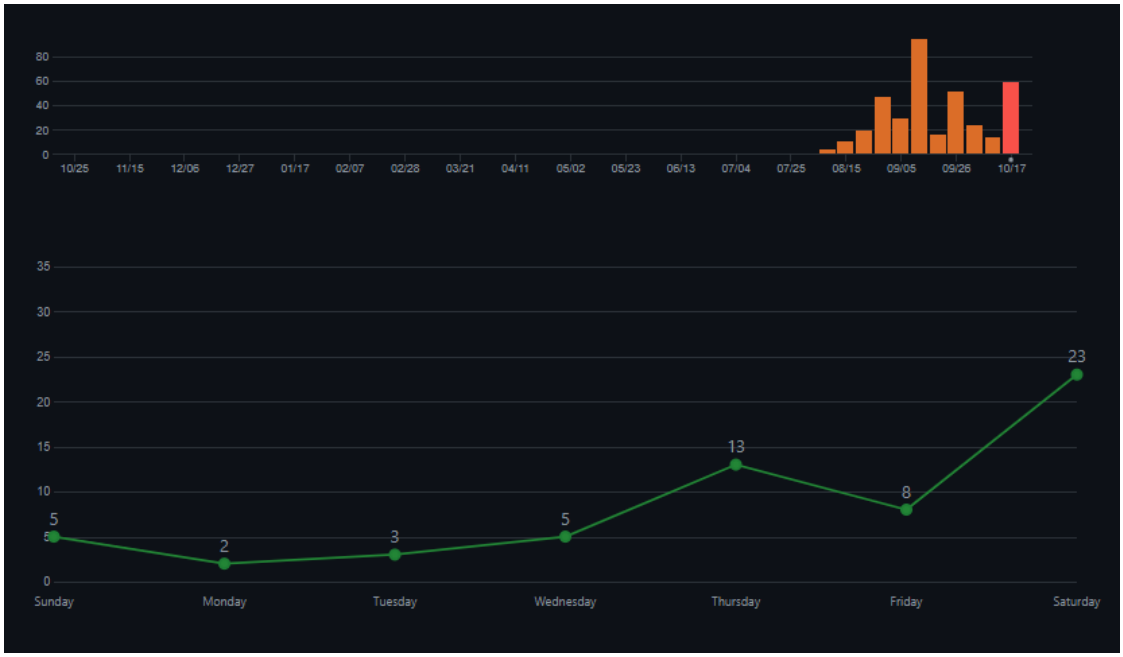
The microservices we have implemented are as follow:

- Books
- Browsing
- Login
- Reviews
- Requests
- Transactions
- Users

A microservice was created for incentives, however as this was not an original requirement for our product, the directory only contains our models for our possible future implementation of this new service.

Git Organisation




80
60
40
20
0
10/25
11/15
12/06
12/27
01/17
02/07
02/28
03/21
04/11
05/02
05/23
06/13
07/04
07/25
08/15
09/05
09/26
10/17
35
30
25
20
15
10
5
0
5
2
3
5
13
8
23
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

Scrum Process:

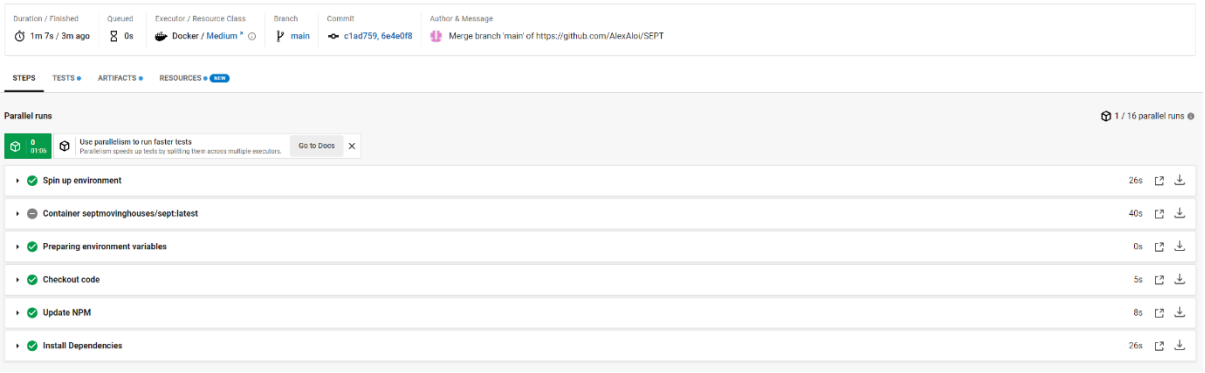
Scrum master: Jared

Our group met at least twice every week typically on Tuesdays and Fridays. Each meeting often went for two hours, while additional sporadic meetings were held each week depending on progress, bugs, errors, or issues with code. If a major issue was present, or there were tasks still incomplete close to deadlines, then meetings would be held to solve these issues immediately, and in some cases we would have more than four meetings a week to deal with these problems as quickly as possible. However, when the velocity of the team was higher than expected, and the group was outperforming predictions, then we followed a standard number of two meetings a week.

Our communication was primarily through Microsoft Teams, where messages were typically sent each day for updates on tasks, progress, or questions about each member's tasks. For tasks that would require two or three members to work together and solve, smaller meetings were held to address these concerns, as often meeting with the entire group was not necessary, and results would be shared at the next scheduled group meeting regardless. Every group member attended all team meetings, with some apologies due to health issues that were communicated ahead of time. Overall, the scrum master is extremely content and happy with the scrum process of the project and group.

Deployment Setup:

After pushing to GitHub, CircleCI will run the front-end tests.



This also ensures the latest Docker container is working.

The next part in the deployment is a batch file, that packages and tests all backend micro-services, and runs them.

```
@echo off
for /d %%i in (%~dp0\*) do (
    cd %%i
    echo %%i
    if exist %%~fi\target\*.jar start cmd /C "mvnw package && cd %%~fi\target\ && for /f "delims=" %%x in ('dir /b *.jar') do java -jar %%x"
)
```

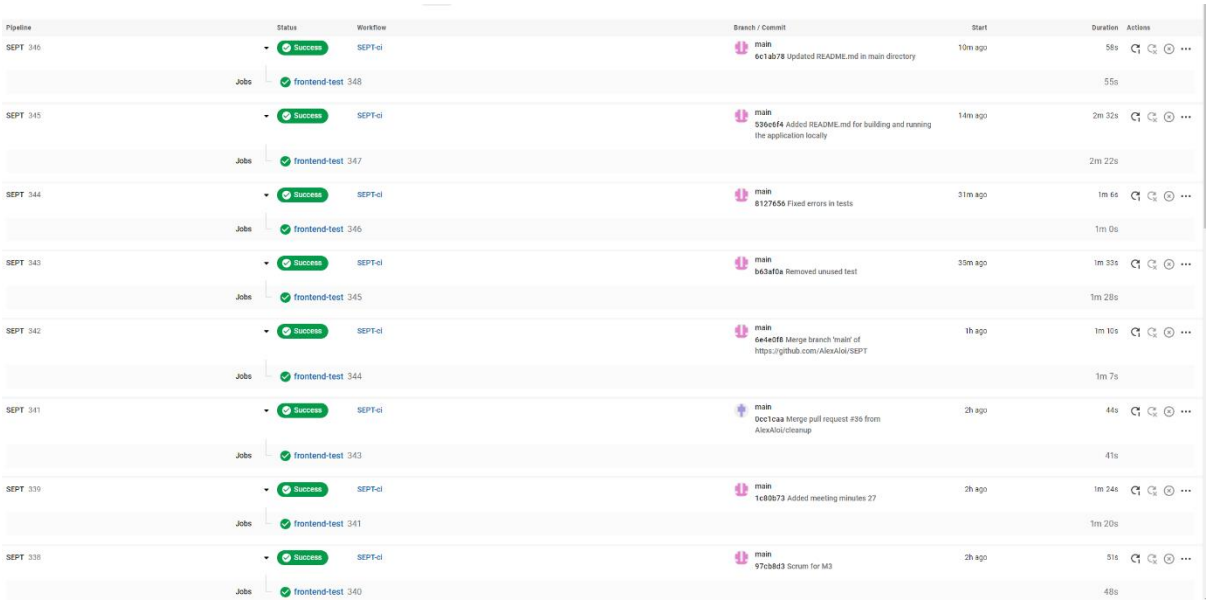
If everything above works, then Docker images are built for each micro-service that has been updated.

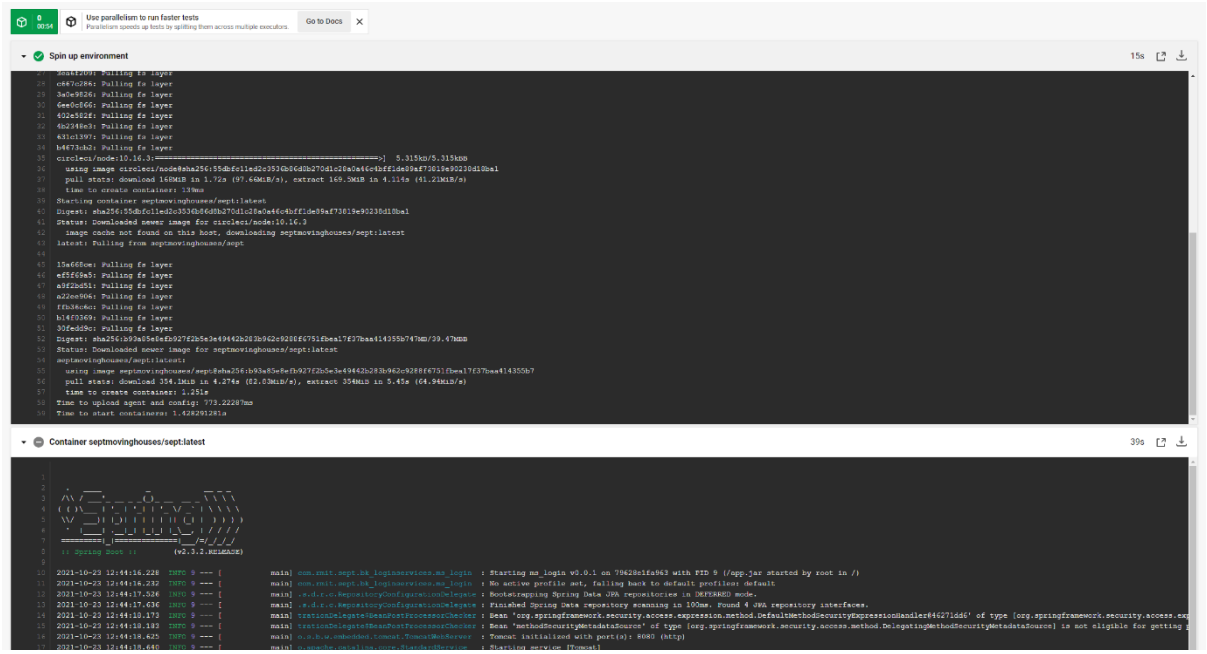
```
alex@Dell:~/SEPT/BackEnd/Books$ sudo docker build -t moving-houses/books .
[sudo] password for alex:
Sending build context to Docker daemon  84.94MB
Step 1/6 : FROM openjdk:11
 ---> dd8edf47a855
Step 2/6 : VOLUME /tmp
 ---> Using cache
 ---> 8e86371532b6
Step 3/6 : ARG JAR_FILE=target/*.jar
 ---> Using cache
 ---> 5e4b0094c40d
Step 4/6 : COPY ${JAR_FILE} app.jar
 ---> Using cache
 ---> 024eab0943f9
Step 5/6 : EXPOSE 8082
 ---> Using cache
 ---> b3e261982ea7
Step 6/6 : ENTRYPOINT ["java", "-jar", "/app.jar"]
 ---> Using cache
 ---> c24ba848a465
Successfully built c24ba848a465
Successfully tagged moving-houses/books:latest
```

These containers are then run on AWS, able to be accessed

```
alex@Dell:~/SEPT/BackEnd/Books$ sudo docker run -p 8082:8082 moving-houses/books

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.2.RELEASE)

2021-10-23T12:45:23.180+0000 INFO Starting BookApplication v1.0.0 on 898081224a2b with PID 1 (/app.jar started by root in /)
2021-10-23T12:45:23.204+0000 INFO No active profile set, falling back to default profiles: default
2021-10-23T12:45:25.756+0000 INFO Bootstrapping Spring Data JPA repositories in DEFERRED mode.
2021-10-23T12:45:25.995+0000 INFO Finished Spring Data repository scanning in 206ms. Found 4 JPA repository interfaces.
2021-10-23T12:45:26.949+0000 INFO Bean 'org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler@4ae33a11' of type
[org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler] is not eligible for getting processed by all BeanPostPro
cessors (for example: not eligible for auto-proxying)
2021-10-23T12:45:26.976+0000 INFO Bean 'methodSecurityMetadataSource' of type [org.springframework.security.access.method.DelegatingMethodSecurityMetad
ataSource] is not eligible for getting processed by all BeanPostProcessors (for example: not eligible for auto-proxying)
2021-10-23T12:45:27.737+0000 INFO Tomcat initialized with port(s): 8082 (http)
2021-10-23T12:45:27.764+0000 INFO Initializing ProtocolHandler ["http-nio-8082"]
2021-10-23T12:45:27.767+0000 INFO Starting service [Tomcat]
2021-10-23T12:45:27.769+0000 INFO Starting Servlet engine: [Apache Tomcat/9.0.37]
2021-10-23T12:45:27.927+0000 INFO Initializing Spring embedded WebApplicationContext
2021-10-23T12:45:27.927+0000 INFO Root WebApplicationContext: initialization completed in 4560 ms
2021-10-23T12:45:28.315+0000 WARN spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Ex
plicitly configure spring.jpa.open-in-view to disable this warning
2021-10-23T12:45:28.590+0000 INFO Initializing ExecutorService 'applicationTaskExecutor'
2021-10-23T12:45:28.725+0000 INFO HHH000204: Processing PersistenceUnitInfo [name: default]
2021-10-23T12:45:28.979+0000 INFO HHH000412: Hibernate ORM core version 5.4.18.Final
2021-10-23T12:45:29.659+0000 INFO HCANN000001: Hibernate Commons Annotations {5.1.0.Final}
2021-10-23T12:45:30.105+0000 INFO HikariPool-1 - Starting...
2021-10-23T12:45:30.588+0000 INFO Creating filter chain: any request, [org.springframework.security.web.context.request.async.WebAsyncManagerIntegratio
nFilter@37095ded, org.springframework.security.web.context.SecurityContextPersistenceFilter@6eafb10e, org.springframework.security.web.header.HeaderWri
terFilter@469d003c, org.springframework.web.filter.CorsFilter@2eb79cbe, org.springframework.security.web.authentication.logout.LogoutFilter@2e1792e7, c
om.rmit.sept.books.security.JwtAuthenticationFilter@1500e009, org.springframework.security.web.savedrequest.RequestCacheAwareFilter@464a4442, org.sprin
gframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@4303b7f0, org.springframework.security.web.authentication.AnonymousAuthentic
```

CircleCI:




0
00:54
Use parallelism to run faster tests
Parallelism speeds up tests by splitting them across multiple executors.
Go to Docs
Spin up environment
15s
c847c286: Pulling fs layer
3a7e9826: Pulling fs layer
6ee0c866: Pulling fs layer
402e302f: Pulling fs layer
4b2348e3: Pulling fs layer
631c1397: Pulling fs layer
b4473cb2: Pulling fs layer
circleci/node:10.16.3: Pulling from circleci/node
using image circleci/node@sha256:55dbfc11ed2c353cb6d8b270d1c28a0a46c4bff1de99af73019e90230d18ba1
pull stats: download 168MiB in 1.72s (97.66MiB/s), extract 169.5MiB in 4.11s (41.21MiB/s)
time to create container: 139ms
Starting container septmovinghouses/sept:latest
Digest: sha256:55dbfc11ed2c353cb6d8b270d1c28a0a46c4bff1de99af73019e90230d18ba1
Status: Downloaded newer image for circleci/node:10.16.3
image cache not found on this host, downloading septmovinghouses/sept:latest
latest: Pulling from septmovinghouses/sept
15a665ce: Pulling fs layer
ed3f69a5: Pulling fs layer
a9f2bd51: Pulling fs layer
a22ee906: Pulling fs layer
ffb386c6c: Pulling fs layer
b14f0369: Pulling fs layer
30fedd90: Pulling fs layer
Digest: sha256:b30a85efb927f2b5e3e49462b283b962c9288f6751fbea17f37baa414355b747mm/39.47MBMB
Status: Downloaded newer image for septmovinghouses/sept:latest
septmovinghouses/sept:latest
using image septmovinghouses/sept@sha256:b30a85efb927f2b5e3e49462b283b962c9288f6751fbea17f37baa414355b7
pull stats: download 354.1MiB in 4.27s (82.83MiB/s), extract 354MiB in 5.45s (64.94MiB/s)
time to create container: 1.251s
Time to upload agent and config: 773.22287ms
Time to start containers: 1.42829128ls
Container septmovinghouses/sept:latest
39s
.   ____   _   __ _ _
/\\ / ___'_ __ _ _(_)_ __ __ _ \ \ \ \
(( )___ | '_ | '_| | '_ \/ _` | \ \ \ \
\\/ ___)| |_)| | | | | || (_| |))))
' |____| .__|_| |_|_| |___, | / / / /
=========|_|==============|___/=/_/_/_/
:: Spring Boot ::   (v2.3.2.RELEASE)
2021-10-23 12:44:16.228 INFO 9 --- [main] com.xmit.sept.bk_loginservices.ms_login : Starting ms_login v0.0.1 on 79628e1fa963 with PID 9 (/app.jar started by root in /)
2021-10-23 12:44:16.232 INFO 9 --- [main] com.xmit.sept.bk_loginservices.ms_login : No active profile set, falling back to default profiles: default
2021-10-23 12:44:17.326 INFO 9 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFERRED mode.
2021-10-23 12:44:17.636 INFO 9 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 150ms. Found 4 JPA repository interfaces.
2021-10-23 12:44:18.173 INFO 9 --- [main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler@4627ldd6' of type [org.springframework.security.access.exp
2021-10-23 12:44:18.193 INFO 9 --- [main] trationDelegate$BeanPostProcessorChecker : Bean 'methodSecurityMetadataSource' of type [org.springframework.security.access.method.DelegatingMethodSecurityMetadataSource] is not eligible for getting
2021-10-23 12:44:18.625 INFO 9 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2021-10-23 12:44:18.640 INFO 9 --- [main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]

Testing Documentation

Unit Testing

Testing: Reviews

Tester: Carl Karama

Types of tests for Review Controller

Test: *addReviewIsBadRequest()*

Test Result: *Pass*

Implementation Specification: When a bad request is made i.e. bad syntax the server fails to understand triggers this which has a HTTP status code of 400.

Implementation Notes: I assert an HTTP.BAD_REQUEST as expected and the actual test results affirm that when I pass bad syntax through the JSON string which hits simulated mock api endpoint.

```
@Test
@DisplayName("Test: addReviewIsBadRequest() [Fail]")
public void testAddReviewIsBadRequest() throws Exception {

    when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class))).thenReturn(null);
    when(reviewService.addReview(ArgumentMatchers.any(Review.class))).thenReturn(reviews.get(0));

    String input = "";

    RequestBuilder requestBuilder = MockMvcRequestBuilders.post( urlTemplate: "/api/review/addReview")

            .accept(MediaType.APPLICATION_JSON).content(input).contentType(MediaType.APPLICATION_JSON);

    MvcResult result = mockMvc.perform(requestBuilder).andReturn();
    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.BAD_REQUEST.value(), response.getStatus());

}
```

Test: *getAllReviews()*

Test Result: *Pass*

Implementation Specification: Returns all the reviews from the database

```
188     @Test
189     public void testGetAllReview() throws Exception {
190
191         when(reviewService.getAllReviews()).thenReturn(reviews);
192
193         mockMvc.perform(
194                 MockMvcRequestBuilders.get( urlTemplate: "/api/review/all").accept(MediaType.APPLICATION_JSON))
195                 .andExpect(MockMvcResultMatchers.status().isOk());
196     }
197
198
199 }
```

Test: *addReview()*

Test Result: *Pass*

Implementation Specification: This tests the methods ability to successfully add a review

Implementation Notes: I assert an HTTP.OK as expected and the actual test results affirm that response successfully when I pass a payload through the JSON string which hits the simulated mock API endpoint for addReview.

```
96      @Test
97      @DisplayName("Test: addReview() [Success]")
98      public void testAddReview() throws Exception {
99
100         when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class))).thenReturn(null);
101         when(reviewService.addReview(ArgumentMatchers.any(Review.class))).thenReturn(reviews.get(0));
102
103
104         String input = "{\n" +
105                 "    \"userRating\": 2,\n" +
106                 "    \"bookRating\": 2,\n" +
107                 "    \"transactionId\": 3,\n" +
108                 "    \"review\": \"It was not the best\"\n" +
109                 "}";
110
111         RequestBuilder requestBuilder = MockMvcRequestBuilders.post( urlTemplate: "/api/review/addReview")
112
113         .accept(MediaType.APPLICATION_JSON).content(input).contentType(MediaType.APPLICATION_JSON);
114
115         MvcResult result = mockMvc.perform(requestBuilder).andReturn();
116         MockHttpServletResponse response = result.getResponse();
117
118         assertEquals(HttpStatus.OK.value(), response.getStatus());
119     }
120
```

Test: *transactionIDIsNull()*

Test Result: *Pass*

Implementation Specification: This tests the methods ability to successfully add a review

Implementation Notes: I assert an HTTP.NOT as expected and the actual test results affirm that response successfully when I pass a payload through the JSON string which hits the simulated mock API endpoint for addReview.

```
@Test
@DisplayName("Test: transactionIDIsNull() [Fail]")
public void testTransactionIDIsNull() throws Exception {

    when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class))).thenReturn(null);
    when(reviewService.addReview(ArgumentMatchers.any(Review.class))).thenReturn(reviews.get(0));

    String input = "{\n" +
            "    \"userRating\": 2,\n" +
            "    \"bookRating\": 2,\n" +
            "    \"transactionId\": null,\n" +
            "    \"review\": \"It was not the best\"\n" +
            "}";

    RequestBuilder requestBuilder = MockMvcRequestBuilders.post( urlTemplate: "/api/review/addReview")

            .accept(MediaType.APPLICATION_JSON).content(input).contentType(MediaType.APPLICATION_JSON);

    MvcResult result = mockMvc.perform(requestBuilder).andReturn();
    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.NOT_ACCEPTABLE.value(), response.getStatus());
}
```

Types of Tests for Service

Test: *getAllReviews()*

Test Result: *Pass*

Implementation Specification: This tests the methods ability to successfully get all reviews

```java
@Test
@DisplayName("Test: getAllReviews() [Success]")
public void getAllReviews() throws Exception {
    given(reviewRepository.findAll()).willReturn(reviews);
    Iterable<Review> reviewIterable = reviewService.getAllReviews();
    Assert.assertNotNull(reviewIterable);
}
```

Test: addReview*()*

Test Result: *Pass*

Implementation Specification: This tests the methods ability to successfully add reviews

```java
@Test
@DisplayName("Test: addReview() [Success]")
public void testAddReview() {
    when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class))).thenReturn(null);
    when(reviewService.addReview(ArgumentMatchers.any(Review.class))).thenReturn(reviews.get(0));
    Review review = reviewService.addReview(reviews.get(0));
    Assert.assertNotNull(review);
}
```

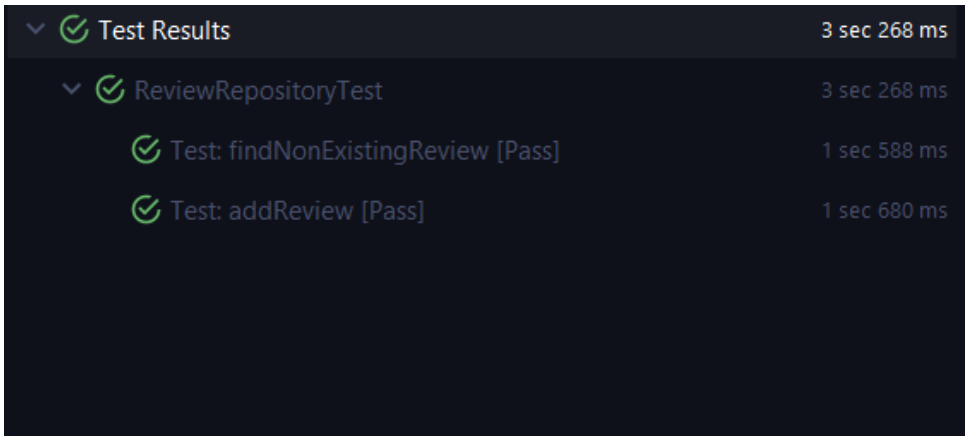
Test: incrementRating*()*

Test Result: *Pass*

Implementation Specification: This tests the methods ability to successfully get increment rating reviews

```java
@Test
@DisplayName("Test: incrementRating() [Success]")
public void incrementRating() {
    when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class))).thenReturn(null);
    when(reviewService.addReview(ArgumentMatchers.any(Review.class))).thenReturn(reviews.get(0));

    reviewService.incrementRating(reviews.get(0));
}
```

Types of Tests for Repo

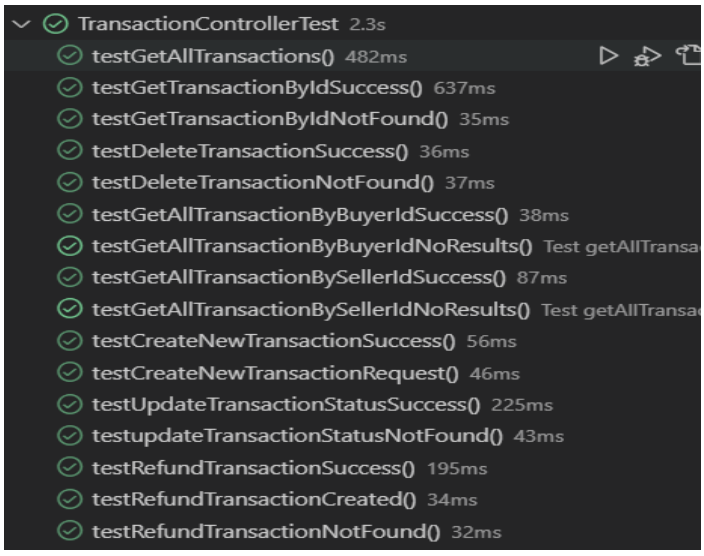


```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-jar-plugin:3.2.0:jar (default-jar) @ reviews ---
[INFO] Building jar: C:\Users\kg\Desktop\CK\Subjects\SEPT\Assignments\A10\SEPT\BackEnd\Reviews\target\reviews-1.0.0.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.3.2.RELEASE:repackage (repackage) @ reviews ---
[INFO] Replacing main artifact with repackaged archive
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:25 min
[INFO] Finished at: 2021-10-23T21:24:37+11:00
[INFO] ------------------------------------------------------------------------
PS C:\Users\kg\Desktop\CK\Subjects\SEPT\Assignments\A10\SEPT\BackEnd\Reviews>
PS C:\Users\kg\Desktop\CK\Subjects\SEPT\Assignments\A10\SEPT\BackEnd\Reviews>
Git  Run  TODO  Problems  Profiler  Terminal  Endpoints  Build  Dependencies  Services  Spring
```

Testing: Transactions

Tester: Shannon Dann

Tests for Transaction Controller



```java
@Test
@DisplayName("Test getAllTransactions") // test for getting all books
void testGetAllTransactions() throws Exception {
    // Mocking service
    when(transactionService.findAllTransactions()).thenReturn(transactions);
    mockMvc.perform(get("/api/transactions/all").contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$[0].id", is(1)))
            .andExpect(jsonPath("$[0].buyer.id", is(1)))
            .andExpect(jsonPath("$[0].book.id", is(1)))
            .andExpect(jsonPath("$[0].price", is(99.99)))
            .andExpect(jsonPath("$[0].status", is(TransactionStatus.PROCESSING.toString())))
            .andExpect(jsonPath("$[0].quantity", is(3)))
            .andExpect(jsonPath("$[0].isReviewed", is(false)))

            .andExpect(jsonPath("$[1].id", is(2)))
            .andExpect(jsonPath("$[1].buyer.id", is(1)))
            .andExpect(jsonPath("$[1].book.id", is(1)))
            .andExpect(jsonPath("$[1].price", is(99.99)))
            .andExpect(jsonPath("$[1].status", is(TransactionStatus.PROCESSING.toString())))
            .andExpect(jsonPath("$[1].quantity", is(3)))
            .andExpect(jsonPath("$[1].isReviewed", is(false)));
}
```

Test: *testGetAllTransactions()*

Test Result: *Pass*

Implementation Specification: Tests api get call “/api/transactions/all” for returning all transactions in the system.

Implementation Notes: Checks the response and makes sure it is HTTP status OK and that all the returned information is correct and in the correct format.

```java
@Test
@DisplayName("Test getTransactionById success") // test for getting a transaction sucessfully
void testGetTransactionByIdSuccess() throws Exception {
    // Mocking service
    when(transactionService.findById(1L)).thenReturn(transactions.get(0));
    mockMvc.perform(get("/api/transactions/1").contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("id", is(1))).andExpect(jsonPath("buyer.id", is(1)))
            .andExpect(jsonPath("book.id", is(1))).andExpect(jsonPath("price", is(99.99)))
            .andExpect(jsonPath("status", is(TransactionStatus.PROCESSING.toString())))
            .andExpect(jsonPath("quantity", is(3)))
            .andExpect(jsonPath("isReviewed", is(false)));
}
```

Test: *testGetTransactionByIdSuccess()*

Test Result: *Pass*

Implementation Specification: Tests api get call “/api/transactions/{id}” for returning a transaction by its ID.

Implementation Notes: Checks the response and makes sure it is HTTP status OK and that all the returned information is correct and in the correct format.

```java
@Test
@DisplayName("Test getTransactionById not found") // test for getting a transaction that doesn't exist
void testGetTransactionByIdNotFound() throws Exception {
    // Mocking service
    when(transactionService.findById(3L)).thenReturn(null);

    RequestBuilder requestBuilder = MockMvcRequestBuilders.get("/api/transactions/3")
            .contentType(MediaType.APPLICATION_JSON);
    MvcResult result = mockMvc.perform(requestBuilder).andReturn();

    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());

}
```

Test: *testGetTransactionByIdNotFound()*

Test Result: *Pass*

Implementation Specification: Tests api call get "/api/transactions/{id}" throwing not found when transaction does not exist.

Implementation Notes: Checks the response and makes sure it returns HTTP NOT FOUND as expected.

```java
@Test
@DisplayName("Test deleteTransaction success") // test for successfully deleting a transaction
void testDeleteTransactionSuccess() throws Exception {
    // Mocking service
    when(transactionService.findById(1L)).thenReturn(transactions.get(0));

    RequestBuilder requestBuilder = MockMvcRequestBuilders.delete("/api/transactions/1");

    MvcResult result = mockMvc.perform(requestBuilder).andReturn();
    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.OK.value(), response.getStatus());
    assertEquals("Transaction with ID 1 was deleted", response.getContentAsString());
}
```

Test: *testDeleteTransactionSuccess()*

Test Result: *Pass*

Implementation Specification: Tests api delete call "/api/transactions/{id}" successfully deleting a transaction.

Implementation Notes: Checks the response and makes sure it returns HTTP OK as expected and text saying the transaction was deleted.

```java
@Test
@DisplayName("Test deleteTransaction not found") // test for deleting a transaction that doesn't exist
void testDeleteTransactionNotFound() throws Exception {
    // Mocking service
    when(transactionService.findById(3L)).thenReturn(null);

    RequestBuilder requestBuilder = MockMvcRequestBuilders.delete("/api/transactions/3")
            .contentType(MediaType.APPLICATION_JSON);
    MvcResult result = mockMvc.perform(requestBuilder).andReturn();

    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
}
```

Test: *testDeleteTransactionNotFound()*

Test Result: *Pass*

Implementation Specification: Tests api delete call “/api/transactions/{id}” throwing not found when transaction does not exist

Implementation Notes: Checks the response and makes sure it returns HTTP NOT FOUND as expected.

```
@Test
@DisplayName("Test getAllTransactionByBuyerId success") // test for getting all transactions by buyer ID sucessfully
void testGetAllTransactionByBuyerIdSuccess() throws Exception {
    // Mocking service
    when(transactionService.getAllByBuyerID(1L)).thenReturn(transactions);
    mockMvc.perform(get("/api/transactions/buyer/1").contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$[0].id", is(1))).andExpect(jsonPath("$[0].buyer.id", is(1)))
        .andExpect(jsonPath("$[0].book.id", is(1))).andExpect(jsonPath("$[0].price", is(99.99)))
        .andExpect(jsonPath("$[0].status", is(TransactionStatus.PROCESSING.toString())))
        .andExpect(jsonPath("$[0].quantity", is(3)))
        .andExpect(jsonPath("$[0].isReviewed", is(false)))

        .andExpect(jsonPath("$[1].id", is(2))).andExpect(jsonPath("$[1].buyer.id", is(1)))
        .andExpect(jsonPath("$[1].book.id", is(1))).andExpect(jsonPath("$[1].price", is(99.99)))
        .andExpect(jsonPath("$[1].status", is(TransactionStatus.PROCESSING.toString())))
        .andExpect(jsonPath("$[1].quantity", is(3)))
        .andExpect(jsonPath("$[1].isReviewed", is(false)));
}
```

Test: *testGetAllTransactionByBuyerIdSuccess()*

Test Result: *Pass*

Implementation Specification: Tests api get call “/api/transactions/buyer/{id}” for returning transactions with given buyer ID.

Implementation Notes: Checks the response and makes sure it is HTTP status OK and that all the returned information is correct and in the correct format.

```
@Test
@DisplayName("Test getAllTransactionByBuyerId no results") // test for getting a transaction when there are no results
void testGetAllTransactionByBuyerIdNoResults() throws Exception {
    // Mocking service
    Iterable<Transaction> emptyTransactions = new ArrayList<Transaction>();
    when(transactionService.getAllByBuyerID(3L)).thenReturn(emptyTransactions);

    RequestBuilder requestBuilder = MockMvcRequestBuilders.get("/api/transactions/buyer/3")
            .contentType(MediaType.APPLICATION_JSON);
    MvcResult result = mockMvc.perform(requestBuilder).andReturn();

    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.OK.value(), response.getStatus());
    assertEquals("[]", response.getContentAsString());
}
```

Test: *testGetAllTransactionByBuyerIdNoResults()*

Test Result: *Pass*

Implementation Specification: Tests api get call “/api/transactions/buyer/{id}” for returning empty list when no transactions are found.

Implementation Notes: Checks the response and makes sure it is HTTP status OK and that an empty string is returned.

```java
@Test
@DisplayName("Test getAllTransactionBySellerId success") // test for getting all transactions by seller ID sucessfully
void testGetAllTransactionBySellerIdSuccess() throws Exception {
    // Mocking service
    when(transactionService.getAllBySellerID(1L)).thenReturn(transactions);
    mockMvc.perform(get("/api/transactions/seller/1").contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$[0].id", is(1))).andExpect(jsonPath("$[0].buyer.id", is(1)))
        .andExpect(jsonPath("$[0].book.id", is(1))).andExpect(jsonPath("$[0].price", is(99.99)))
        .andExpect(jsonPath("$[0].status", is(TransactionStatus.PROCESSING.toString())))
        .andExpect(jsonPath("$[0].quantity", is(3)))
        .andExpect(jsonPath("$[0].isReviewed", is(false)))

        .andExpect(jsonPath("$[1].id", is(2))).andExpect(jsonPath("$[1].buyer.id", is(1)))
        .andExpect(jsonPath("$[1].book.id", is(1))).andExpect(jsonPath("$[1].price", is(99.99)))
        .andExpect(jsonPath("$[1].status", is(TransactionStatus.PROCESSING.toString())))
        .andExpect(jsonPath("$[1].quantity", is(3)))
        .andExpect(jsonPath("$[1].isReviewed", is(false)));
}
```

Test: *testGetAllTransactionBySellerIdSuccess()*

Test Result: *Pass*

Implementation Specification: Tests api get call “/api/transactions/seller/{id}” for returning transactions with given seller ID.

Implementation Notes: Checks the response and makes sure it is HTTP status OK and that all the returned information is correct and in the correct format.

```java
@Test
@DisplayName("Test getAllTransactionBySellerId no results") // test for getting a transaction when there are no results
void testGetAllTransactionBySellerIdNoResults() throws Exception {
    // Mocking service
    Iterable<Transaction> emptyTransactions = new ArrayList<Transaction>();
    when(transactionService.getAllBySellerID(3L)).thenReturn(emptyTransactions);

    RequestBuilder requestBuilder = MockMvcRequestBuilders.get("/api/transactions/seller/3")
            .contentType(MediaType.APPLICATION_JSON);
    MvcResult result = mockMvc.perform(requestBuilder).andReturn();

    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.OK.value(), response.getStatus());
    assertEquals("[]", response.getContentAsString());
}
```

Test: *testGetAllTransactionBySellerIdNoResults()*

Test Result: *Pass*

Implementation Specification: Tests api get call “/api/transactions/seller/{id}” for returning empty list when no transactions are found.

Implementation Notes: Checks the response and makes sure it is HTTP status OK and that an empty string is returned.

```java
@Test
@DisplayName("Test createNewTransaction success") // test for creating a new transaction successfully
void testCreateNewTransactionSuccess() throws Exception {
    // Mocking service
    when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
            .thenReturn(null);
    when(transactionService.saveTransaction(ArgumentMatchers.any(Transaction.class)))
            .thenReturn(transactions.get(0));
    String inputJson = "{\n" + " \"buyerID\":\"1\",\n" + "\"bookID\":\"1\",\n"
            + " \"price\":\"99.99\",\n" + " \"quantity\":\"3\"\n" + "}";
    RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/transactions/new")
            .accept(MediaType.APPLICATION_JSON)
            .content(inputJson).contentType(MediaType.APPLICATION_JSON);

    MvcResult result = mockMvc.perform(requestBuilder).andReturn();
    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.CREATED.value(), response.getStatus());
}
```

Test: *testCreateNewTransactionSuccess()*

Test Result: *Pass*

Implementation Specification: Tests api post call “/api/transactions/new” for creating a transaction with given information.

Implementation Notes: Checks the response and makes sure it is HTTP status CREATED for the transaction being created successfully.

```
@Test
@DisplayName("Test createNewTransaction badRequest") // test for creating a new transaction with invalid information
void testCreateNewTransactionBadRequest() throws Exception {
    // Mocking service
    when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
            .thenReturn(null);

    String inputJson = "null";
    RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/transactions/new")
            .accept(MediaType.APPLICATION_JSON)
            .content(inputJson).contentType(MediaType.APPLICATION_JSON);

    MvcResult result = mockMvc.perform(requestBuilder).andReturn();
    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.BAD_REQUEST.value(), response.getStatus());
}
```

Test: *testCreateNewTransactionBadRequest()*

Test Result: *Pass*

Implementation Specification: Tests api post call “/api/transactions/new” throwing bad request when given information is invalid.

Implementation Notes: Checks the response and makes sure it is HTTP status BAD REQUEST.

```
@Test
@DisplayName("Test updateTransactionStatus success") // test for updating a transaction status successfully
void testUpdateTransactionStatusSuccess() throws Exception {
    // Mocking service
    when(transactionService.findById(1L)).thenReturn(transactions.get(0));
    when(transactionService.updateTransactionStatus(ArgumentMatchers.any(TransactionStatus.class),
                    ArgumentMatchers.any(Transaction.class)))
                    .thenReturn(transactions.get(0));

    String inputJson = "{\n" + "\"status\":\"PRE_ORDER\"\n" +"}";
    RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/transactions/updateStatus/1")
                    .accept(MediaType.APPLICATION_JSON).content(inputJson)
                    .contentType(MediaType.APPLICATION_JSON);

    MvcResult result = mockMvc.perform(requestBuilder).andReturn();
    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.OK.value(), response.getStatus());
}
```

Test: *testUpdateTransactionStatusSuccess()*

Test Result: *Pass*

Implementation Specification: Tests api post call “/api/transactions/updateStatus/{id}” updating the status of a given transaction with the given status.

Implementation Notes: Checks the response and makes sure it is HTTP status OK to show the transaction was updated successfully.

```java
@Test
@DisplayName("Test updateTransactionStatus not found") // test for updating a transaction status with invalid information
void testupdateTransactionStatusNotFound() throws Exception {
    // Mocking service
    when(transactionService.findById(3L)).thenReturn(null);
    when(transactionService.updateTransactionStatus(ArgumentMatchers.any(TransactionStatus.class),
                ArgumentMatchers.any(Transaction.class)))
                .thenReturn(transactions.get(0));

    String inputJson = "{\n" + "\"status\":\"PRE_ORDER\"\n" +"}";
    RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/transactions/updateStatus/1")
                    .accept(MediaType.APPLICATION_JSON).content(inputJson)
                    .contentType(MediaType.APPLICATION_JSON);

    MvcResult result = mockMvc.perform(requestBuilder).andReturn();
    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
}
```

Test: *testupdateTransactionStatusNotFound()*

Test Result: *Pass*

Implementation Specification: Tests api post call “/api/transactions/updateStatus/{id}” throwing not found when transaction doesn’t exist.

Implementation Notes: Checks the response and makes sure it is HTTP status NOT FOUND as expected.

```java
@Test
@DisplayName("Test refundTransaction success") // test for refunding transcation successfully
void testRefundTransactionSuccess() throws Exception {
    // Mocking service
    when(transactionService.findById(1L)).thenReturn(transactions.get(0));
    when(transactionService.refundTransaction(transactions.get(0))).thenReturn(true);
    MvcResult result = mockMvc.perform(get("/api/transactions/refund/1")
                .contentType(MediaType.APPLICATION_JSON)).andReturn();
    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.OK.value(), response.getStatus());
    assertEquals("Refund successful.", response.getContentAsString());
}
```

Test: *testRefundTransactionSuccess()*

Test Result: *Pass*

Implementation Specification: Tests api get call “/api/transactions/refund/{id}” successfully refunding.

Implementation Notes: Checks the response and makes sure it is HTTP status OK and text saying refund successful.

```java
@Test
@DisplayName("Test refundTransaction created") // test for refunding transcation successfully
void testRefundTransactionCreated() throws Exception {
    // Mocking service
    when(transactionService.findById(1L)).thenReturn(transactions.get(0));
    when(transactionService.refundTransaction(transactions.get(0))).thenReturn(false);
    MvcResult result = mockMvc.perform(get("/api/transactions/refund/1")
                .contentType(MediaType.APPLICATION_JSON)).andReturn();
    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.OK.value(), response.getStatus());
    assertEquals("Refund request was created.", response.getContentAsString());
}
```

Test: *testRefundTransactionCreated()*

Test Result: *Pass*

Implementation Specification: Tests api get call "/api/transactions/refund/{id}" creating a refund request.

Implementation Notes: Checks the response and makes sure it is HTTP status OK and text saying refund request was created.

```
@Test
@DisplayName("Test refundTransaction not found") // test for refunding transcation and transaction not found
void testRefundTransactionNotFound() throws Exception {
    // Mocking service
    when(transactionService.findById(3L)).thenReturn(null);
    MvcResult result = mockMvc.perform(get("/api/transactions/refund/1")
                .contentType(MediaType.APPLICATION_JSON)).andReturn();
    MockHttpServletResponse response = result.getResponse();

    assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
}
```

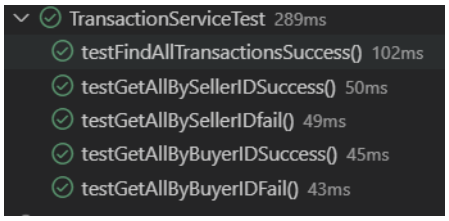
Test: *testRefundTransactionNotFound()*

Test Result: *Pass*

Implementation Specification: Tests api get call "/api/transactions/refund/{id}" throwing not found if transaction is not found.

Implementation Notes: Checks the response and makes sure it is HTTP status NOT FOUND.

Tests for Transaction Service




```
@Test
@DisplayName("Test findAllTranscations success") // test for finding all transcation
public void testFindAllTransactionsSuccess() throws Exception {
    given(transactionRepository.findAll()).willReturn(transactions);
    Iterable<Transaction> allTransactions = transactionService.findAllTransactions();
    Assert.assertNotNull(allTransactions);
}
```

Test: *testFindAllTransactionsSuccess()*

Test Result: *Pass*

Implementation Specification: Test for method findAllTransactions successfully returning transactions.

Implementation Notes: Checks to make sure that the transactions returned are not null.

```java
@Test
@DisplayName("Test getAllBySellerID success") // test for finding all transaction by seller ID successfully
public void testGetAllBySellerIDSuccess() throws Exception {
    User user = new User();
    given(userRepository.getById(1L)).willReturn(user);
    given(bookRepository.findBySeller(user)).willReturn(books);
    given(transactionRepository.findByBookIn(books)).willReturn(transactions);
    Iterable<Transaction> allTransactions = transactionService.getAllBySellerID(1L);
    Assert.assertNotNull(allTransactions);
}
```

Test: *testGetAllBySellerIDSuccess()*

Test Result: *Pass*

Implementation Specification: Test for method getAllBySellerID successfully returning transactions.

Implementation Notes: Checks to make sure that the transactions returned is not null.

```java
@Test
@DisplayName("Test getAllBySellerID fail") // test for finding all transaction by seller ID that doesn't exist
public void testGetAllBySellerIDfail() throws Exception {
    given(userRepository.getById(null)).willReturn(null);
    given(bookRepository.findBySeller(null)).willReturn(null);
    given(transactionRepository.findByBookIn(null)).willReturn(null);
    Iterable<Transaction> allTransactions = transactionService.getAllBySellerID(null);
    Assert.assertNull(allTransactions);
}
```

Test: *testGetAllBySellerIDfail()*

Test Result: *Pass*

Implementation Specification: Test for method getAllBySellerID returning null when there are no transactions.

Implementation Notes: Checks to make sure that no transactions were returned.

```java
@Test
@DisplayName("Test getAllByBuyerID success") // test for finding all transaction by buyer ID successfully
public void testGetAllByBuyerIDSuccess() throws Exception {
    User user = new User();
    given(userRepository.getById(1L)).willReturn(user);
    given(transactionRepository.findByBuyer(user)).willReturn(transactions);
    Iterable<Transaction> allTransactions = transactionService.getAllByBuyerID(1L);
    Assert.assertNotNull(allTransactions);
}
```

Test: *testGetAllByBuyerIDSuccess()*

Test Result: *Pass*

Implementation Specification: Test for method getAllByBuyerID successfully returning transactions.

Implementation Notes: Checks to make sure that the transactions returned is not null.

```java
@Test
@DisplayName("Test getAllByBuyerID fail") // test for finding all transaction by buyer ID that doesn't exist
public void testGetAllByBuyerIDFail() throws Exception {
    given(userRepository.getById(null)).willReturn(null);
    given(transactionRepository.findByBuyer(null)).willReturn(null);
    Iterable<Transaction> allTransactions = transactionService.getAllByBuyerID(null);
    Assert.assertNull(allTransactions);
}
```

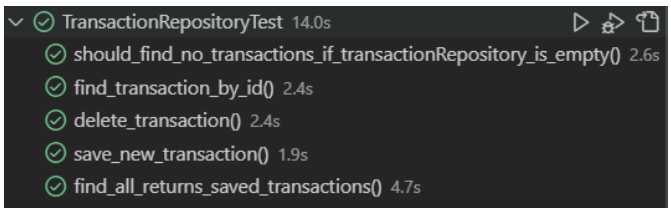
Test: *testGetAllByBuyerIDFail()*

Test Result: *Pass*

Implementation Specification: Test for method getAllByBuyerID returning null when there are no transactions.

Implementation Notes: Checks to make sure that no transactions were returned.

Tests for Transaction Repository




```java
// testing for empty transaction repository
@Test
@Rollback(true)
public void should_find_no_transactions_if_transactionRepository_is_empty() {
    Iterable<Transaction> transactions = transactionRepository.findAll();
    if (transactions.iterator().hasNext()) {
        assertThat(transactions).isNotEmpty();
    } else {
        assertThat(transactions).isEmpty();
    }
}
```

Test: *should_find_no_transactions_if_transactionRepository_is_empty()*

Test Result: *Pass*

Implementation Specification: Tests repository to check if it is empty.

Implementation Notes: goes through all transactions to make sure they are in the returned list.

```java
//test to find a transaction by id
@Test
@Rollback(true)
public void find_transaction_by_id() {
    Book book = new Book();
    book.setId(1L);
    User buyer = new User();
    buyer.setId(1L);

    Transaction transaction = new Transaction();
    transaction.setId(1L);
    transaction.setBuyer(buyer);
    transaction.setBook(book);
    transaction.setPrice(99.99);
    transaction.setStatus(TransactionStatus.PROCESSING);
    transaction.setQuantity(3);
    transaction.setIsReviewed(false);
    transaction.setCreatedAt(new Date());
    transactionRepository.save(transaction);

    Transaction findTransaction = transactionRepository.getById(1L);
    assertNotNull(findTransaction);

    assertThat(findTransaction).hasFieldOrPropertyWithValue("id", 1L);
    assertThat(findTransaction).hasFieldOrPropertyWithValue("price", 99.99);
    assertThat(findTransaction).hasFieldOrPropertyWithValue("status", TransactionStatus.PROCESSING);
    assertThat(findTransaction).hasFieldOrPropertyWithValue("quantity", 3);
    assertThat(findTransaction).hasFieldOrPropertyWithValue("isReviewed", false);
    assertThat(findTransaction.getBook()).hasFieldOrPropertyWithValue("id", 1L);
    assertThat(findTransaction.getBuyer()).hasFieldOrPropertyWithValue("id", 1L);
}
```

Test: *find_transaction_by_id()*

Test Result: *Pass*

Implementation Specification: Tests repository for finding a transaction by given ID.

Implementation Notes: checks to make sure a transaction was found and then checks the information of that transaction are correct.

```
// test to delete a transaction
@Test
@Rollback(true)
public void delete_transaction() {
   Book book = new Book();
   book.setId(1L);
   User buyer = new User();
   buyer.setId(1L);

   Transaction transaction = new Transaction();
   transaction.setId(1L);
   transaction.setBuyer(buyer);
   transaction.setBook(book);
   transaction.setPrice(99);
   transaction.setStatus(TransactionStatus.PROCESSING);
   transaction.setQuantity(3);
   transaction.setIsReviewed(false);
   transaction.setCreatedAt(new Date());
   Transaction savedTransaction = transactionRepository.save(transaction);

   transactionRepository.delete(savedTransaction);

   Transaction findTransaction = transactionRepository.getById(1L);
   assertNull(findTransaction);
}
```

Test: *delete_transaction()*

Test Result: *Pass*

Implementation Specification: Tests repository for deleting a transaction.

Implementation Notes: checks to make sure that no transaction is returned once deleted.

```
//test to find a transaction by id
@Test
@Rollback(true)
public void save_new_transaction() {
    Book book = new Book();
    book.setId(1L);
    User buyer = new User();
    buyer.setId(1L);

    Transaction transaction = new Transaction();
    transaction.setId(1L);
    transaction.setBuyer(buyer);
    transaction.setBook(book);
    transaction.setPrice(99.99);
    transaction.setStatus(TransactionStatus.PROCESSING);
    transaction.setQuantity(3);
    transaction.setIsReviewed(false);
    transaction.setCreatedAt(new Date());
    Transaction savedTransaction = transactionRepository.save(transaction);

    assertThat(savedTransaction).hasFieldOrPropertyWithValue("id", 1L);
    assertThat(savedTransaction).hasFieldOrPropertyWithValue("price", 99.99);
    assertThat(savedTransaction).hasFieldOrPropertyWithValue("status", TransactionStatus.PROCESSING);
    assertThat(savedTransaction).hasFieldOrPropertyWithValue("quantity", 3);
    assertThat(savedTransaction).hasFieldOrPropertyWithValue("isReviewed", false);
    assertThat(savedTransaction.getBook()).hasFieldOrPropertyWithValue("id", 1L);
    assertThat(savedTransaction.getBuyer()).hasFieldOrPropertyWithValue("id", 1L);
}
```

Test: *save_new_transaction()*

Test Result: *Pass*

Implementation Specification: Tests repository for saving a new transaction.

Implementation Notes: checks to make sure that the transaction is saved and that all the transaction information is correct.

```java
// test to find all users after saving a new user
@Test
@Rollback(true)
public void find_all_returns_saved_transactions() {
    Book book = new Book();
    book.setId(1L);
    User buyer = new User();
    buyer.setId(1L);

    Transaction transaction = new Transaction();
    transaction.setId(1L);
    transaction.setBuyer(buyer);
    transaction.setBook(book);
    transaction.setPrice(99.99);
    transaction.setStatus(TransactionStatus.PROCESSING);
    transaction.setQuantity(3);
    transaction.setIsReviewed(false);
    transaction.setCreatedAt(new Date());
    transactionRepository.save(transaction);

    Iterable<Transaction> transactions = transactionRepository.findAll();
    Iterator<Transaction> iter = transactions.iterator();
    Transaction savedTransaction = iter.next();
    while(iter.hasNext() && savedTransaction.getId() != 1L) {
        savedTransaction = iter.next();
    }
    assertThat(savedTransaction).hasFieldOrPropertyWithValue("id", 1L);
    assertThat(savedTransaction).hasFieldOrPropertyWithValue("price", 99.99);
    assertThat(savedTransaction).hasFieldOrPropertyWithValue("status", TransactionStatus.PROCESSING);
    assertThat(savedTransaction).hasFieldOrPropertyWithValue("quantity", 3);
    assertThat(savedTransaction).hasFieldOrPropertyWithValue("isReviewed", false);
    assertThat(savedTransaction.getBook()).hasFieldOrPropertyWithValue("id", 1L);
    assertThat(savedTransaction.getBuyer()).hasFieldOrPropertyWithValue("id", 1L);
}
```

Test: *find_all_returns_saved_transactions()*

Test Result: *Pass*

Implementation Specification: Tests repository returning all transactions after saving.

Implementation Notes: checks to find the transaction that was saved and make sure it is returned with all transactions.

Build success for transactions

```
Results:

Tests run: 22, Failures: 0, Errors: 0, Skipped: 0


--- maven-jar-plugin:3.2.0:jar (default-jar) @ transactions ---
Building jar: C:\Users\shann.DESKTOP-9P41NT8\Documents\sept\test\SEPT\BackEnd\Transactions\target\transactions-1.0.0.jar

--- spring-boot-maven-plugin:2.3.2.RELEASE:repackage (repackage) @ transactions ---
Replacing main artifact with repackaged archive
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  53.596 s
```

Books microservice Tests:

Tester: Jared

BookControllerTest.java

```java
    @Test
    @DisplayName("Test findAllBooks") // test for getting all books
    void testFindAllBooks() throws Exception {
        // Mocking service
        when(bookService.findAllBooks()).thenReturn(books);
        mockMvc.perform(get("/api/books/all").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
                .andExpect(jsonPath("$[0].id", is(1))).andExpect(jsonPath("$[0].title", is("Book Title")))
                .andExpect(jsonPath("$[0].authorName", is("Author Name"))).andExpect(jsonPath("$[0].sellerId", is(1)))
                .andExpect(jsonPath("$[0].isbn", is(123456))).andExpect(jsonPath("$[0].quantity", is(10)))
                .andExpect(jsonPath("$[0].category", is("Book Category")))
                .andExpect(jsonPath("$[0].quality", is(Quality.NEW.toString())))
                .andExpect(jsonPath("$[0].price", is(99.99))).andExpect(jsonPath("$[0].ratingNo", is(0)))
                .andExpect(jsonPath("$[0].ratingTotal", is(0)))
                .andExpect(jsonPath("$[0].serviceType", is(ServiceType.E_BOOK.toString())))

                .andExpect(jsonPath("$[1].id", is(2))).andExpect(jsonPath("$[1].title", is("Book Title 2")))
                .andExpect(jsonPath("$[1].authorName", is("Author Name 2"))).andExpect(jsonPath("$[1].sellerId", is(1)))
                .andExpect(jsonPath("$[1].isbn", is(1234567))).andExpect(jsonPath("$[1].quantity", is(5)))
                .andExpect(jsonPath("$[1].category", is("Book Category 2")))
                .andExpect(jsonPath("$[1].quality", is(Quality.USED.toString())))
                .andExpect(jsonPath("$[1].price", is(89.99))).andExpect(jsonPath("$[1].ratingNo", is(1)))
                .andExpect(jsonPath("$[1].ratingTotal", is(1)))
                .andExpect(jsonPath("$[1].serviceType", is(ServiceType.PRINT_ON_DEMAND.toString())));

    }
```

```java
    @Test
    @DisplayName("Test getBook success") // test for getting a book successfully
    void testGetBookSuccess() throws Exception {
        // Mocking service
        when(bookService.findById(1L)).thenReturn(books.get(0));

        mockMvc.perform(get("/api/books/1").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
                .andExpect(jsonPath("id", is(1))).andExpect(jsonPath("title", is("Book Title")))
                .andExpect(jsonPath("authorName", is("Author Name"))).andExpect(jsonPath("sellerId", is(1)))
                .andExpect(jsonPath("isbn", is(123456))).andExpect(jsonPath("quantity", is(10)))
                .andExpect(jsonPath("category", is("Book Category")))
                .andExpect(jsonPath("quality", is(Quality.NEW.toString()))).andExpect(jsonPath("price", is(99.99)))
                .andExpect(jsonPath("ratingNo", is(0))).andExpect(jsonPath("ratingTotal", is(0)))
                .andExpect(jsonPath("serviceType", is(ServiceType.E_BOOK.toString())));
    }

    @Test
    @DisplayName("Test getBook not found") // test for getting a book that doesn't exist
    void testGetBookNotFound() throws Exception {
        // Mocking service
        when(bookService.findById(3L)).thenReturn(null);

        RequestBuilder requestBuilder = MockMvcRequestBuilders.get("/api/books/3")
                .contentType(MediaType.APPLICATION_JSON);
        MvcResult result = mockMvc.perform(requestBuilder).andReturn();

        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
        assertEquals("Book with ID 3 was not found", response.getContentAsString());

    }
```

```java
    @Test
    @DisplayName("Test addNewBook success") // test for adding a new book successfully
    void testAddNewBookSuccess() throws Exception {
        // Mocking service
        when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
                .thenReturn(null);
        when(bookService.saveBook(ArgumentMatchers.any(Book.class))).thenReturn(books.get(0));
        String inputJson = "{\n" + " \"title\":\"Title\",\n" + "\"authorName\":\"Author Name\",\n"
                + " \"sellerId\":\"1\",\n" + " \"category\":\"Category\",\n" + " \"isbn\":\"12345678\",\n"
                + " \"quantity\":\"99\",\n" + " \"quality\":\"0\",\n" + " \"bookStatus\":\"0\",\n"
                + " \"serviceType\":\"PRINT_ON_DEMAND\",\n" + " \"price\":\"99.99\"\n" + "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/books/new").accept(MediaType.APPLICATION_JSON)
                .content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.OK.value(), response.getStatus());
    }

    @Test
    @DisplayName("Test addNewBook null seller error") // test for adding a new book without a seller
    void testAddNewBookNullSellerError() throws Exception {
        // Mocking service
        when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
                .thenReturn(null);
        when(bookService.saveBook(ArgumentMatchers.any(Book.class))).thenReturn(null);
        String inputJson = "{\n" + " \"title\":\"Book Title\",\n" + "\"authorName\":\"Author Name\",\n"
                + " \"category\":\"Book Category\",\n" + " \"isbn\":\"123456\",\n" + " \"quantity\":\"10\",\n"
                + " \"quality\":\"0\",\n" + " \"bookStatus\":\"0\",\n" + " \"serviceType\":\"E_BOOK\",\n"
                + " \"price\":\"99.99\"\n" + "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/books/new").accept(MediaType.APPLICATION_JSON)
                .content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_ACCEPTABLE.value(), response.getStatus());
        assertEquals("Unable to add the new book, User id not given!.", response.getContentAsString());
    }
```

```java
    @Test
    @DisplayName("Test addNewBook duplicate error") // test for adding a duplicate new book
    void testAddNewBookDuplicateError() throws Exception {
        // Mocking service
        when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
                .thenReturn(null);
        when(bookService.saveBook(ArgumentMatchers.any(Book.class))).thenReturn(null);
        String inputJson = "{\n" + " \"title\":\"Book Title\",\n" + "\"authorName\":\"Author Name\",\n"
                + " \"sellerId\":\"1\",\n" + " \"category\":\"Book Category\",\n" + " \"isbn\":\"123456\",\n"
                + " \"quantity\":\"10\",\n" + " \"quality\":\"0\",\n" + " \"bookStatus\":\"0\",\n"
                + " \"serviceType\":\"E_BOOK\",\n" + " \"price\":\"99.99\"\n" + "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/books/new").accept(MediaType.APPLICATION_JSON)
                .content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.CONFLICT.value(), response.getStatus());
        assertEquals("Unable to add the new book, a copy of the book already exists.", response.getContentAsString());
    }

    @Test
    @DisplayName("Test addNewBook badRequest") // test for adding a new book with invalid information
    void testAddNewBookBadRequest() throws Exception {
        // Mocking service
        when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
                .thenReturn(null);

        String inputJson = "null";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/books/new").accept(MediaType.APPLICATION_JSON)
                .content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.BAD_REQUEST.value(), response.getStatus());
    }
```

```java
    @Test
    @DisplayName("Test deleteBook success") // test for successfully deleting a book
    void testDeleteBookSuccess() throws Exception {
        // Mocking service
        when(bookService.findById(1L)).thenReturn(books.get(0));

        RequestBuilder requestBuilder = MockMvcRequestBuilders.delete("/api/books/1");

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.OK.value(), response.getStatus());
        assertEquals("Book with ID 1 was deleted", response.getContentAsString());
    }

    @Test
    @DisplayName("Test deleteBook not found") // test for deleting a book that doesn't exist
    void testDeleteBookNotFound() throws Exception {
        // Mocking service
        when(bookService.findById(3L)).thenReturn(null);

        RequestBuilder requestBuilder = MockMvcRequestBuilders.delete("/api/books/3");

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
        assertEquals("Book with ID 3 was not found", response.getContentAsString());
    }
}
```

EditBookControllerTest.java

```java
@Test
@DisplayName("Test editBook success") // test for editing a book successfully
void testEditBookSuccess() throws Exception {
        // Mocking service
        when(bookService.findById(1L)).thenReturn(books.get(0));
        when(editBookService.updateBook(ArgumentMatchers.any(BookForm.class), ArgumentMatchers.any(Book.class)))
                .thenReturn(books.get(0));

        String inputJson = "{\n" + "\"sellerId\":\"1\",\n" + "\"quality\":\"NEW\"\n" + "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/editBook/1")
                .accept(MediaType.APPLICATION_JSON).content(inputJson)
                .contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.OK.value(), response.getStatus());
        assertEquals("Successfully updated book details", response.getContentAsString());
}

@Test
@DisplayName("Test editBook null seller error") // test for editing a book so it is sold by an invalid seller
void testEditBookNullSellerError() throws Exception {
        // Mocking service
        when(bookService.findById(1L)).thenReturn(books.get(0));
        String inputJson = "{\n" + "\"quality\":\"NEW\"\n" + "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/editBook/1")
                .accept(MediaType.APPLICATION_JSON).content(inputJson)
                .contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
        assertEquals("Unable to add the new book details, User to tie to not found!.",
                response.getContentAsString());
}
```

```java
@Test
@DisplayName("Test editBook duplicate error") // test for editing a book so it becomes a duplicate copy of
                                              // another book
void testEditBookDuplicateError() throws Exception {
        // Mocking service

        when(bookService.findById(1L)).thenReturn(books.get(0));
        when(editBookService.updateBook(ArgumentMatchers.any(BookForm.class), ArgumentMatchers.any(Book.class)))
                .thenReturn(null);
        String inputJson = "{\n" + " \"title\":\"Book Title\",\n" + "\"authorName\":\"Author Name\",\n"
                + " \"sellerId\":\"1\",\n" + " \"category\":\"Book Category\",\n"
                + " \"isbn\":\"123456\",\n" + " \"quantity\":\"10\",\n" + " \"quality\":\"0\",\n"
                + " \"bookStatus\":\"0\",\n" + " \"serviceType\":\"E_BOOK\",\n"
                + " \"price\":\"99.99\"\n" + "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/editBook/1")
                .accept(MediaType.APPLICATION_JSON).content(inputJson)
                .contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.CONFLICT.value(), response.getStatus());
        assertEquals("Unable to save details for book, a copy of the book already exists.",
                response.getContentAsString());
}

@Test
@DisplayName("Test editBook not found") // test for editing a book that doesn't exist
void testEditBookNotFound() throws Exception {
        // Mocking service
        when(userService.findById(3L)).thenReturn(null);

        String inputJson = "{\n" + "\"sellerId\":\"1\",\n" + "\"quality\":\"NEW\"\n" + "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/editBook/3")
                .accept(MediaType.APPLICATION_JSON).content(inputJson)
                .contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
        assertEquals("Book with ID 3 was not found", response.getContentAsString());
}
```

BookRepositoryTest.java

```java
    // testing for empty book repository
    @Test
    @Rollback(true)
    public void should_find_no_books_if_bookRepository_is_empty() {
        Iterable<Book> books = bookRepository.findAll();
        if (books.iterator().hasNext()) {
            assertThat(books).isNotEmpty();
        } else {
            assertThat(books).isEmpty();
        }
    }

    // test to find a book by id
    @Test
    @Rollback(true)
    public void find_book_by_id() {
        Book book1 = new Book();
        book1.setId(1L);
        book1.setTitle("Book Title");
        book1.setAuthorName("Author Name");
        book1.setISBN(123456);
        book1.setQuantity(10);
        book1.setCategory("Book Category");
        book1.setQuality(Quality.NEW);
        book1.setPrice(99.99);
        book1.setRatingNo(0);
        book1.setRatingTotal(0);
        book1.setServiceType(ServiceType.E_BOOK);
        bookRepository.save(book1);

        Book findBook = bookRepository.findById(1L).orElse(null);
        assertNotNull(findBook);
        assertThat(findBook).hasFieldOrPropertyWithValue("id", 1L);
        assertThat(findBook).hasFieldOrPropertyWithValue("title", "Book Title");
        assertThat(findBook).hasFieldOrPropertyWithValue("authorName", "Author Name");
        assertThat(findBook).hasFieldOrPropertyWithValue("isbn", 123456);
        assertThat(findBook).hasFieldOrPropertyWithValue("quantity", 10);
        assertThat(findBook).hasFieldOrPropertyWithValue("category", "Book Category");
        assertThat(findBook).hasFieldOrPropertyWithValue("quality", Quality.NEW);
        assertThat(findBook).hasFieldOrPropertyWithValue("price", 99.99);
        assertThat(findBook).hasFieldOrPropertyWithValue("ratingNo", 0);
        assertThat(findBook).hasFieldOrPropertyWithValue("ratingTotal", 0);
        assertThat(findBook).hasFieldOrPropertyWithValue("serviceType", ServiceType.E_BOOK);
    }
```

```java
    // test to delete a book
    @Test
    @Rollback(true)
    public void delete_book() {
        Book book1 = new Book();
        book1.setId(0L);
        book1.setTitle("Book Title");
        book1.setAuthorName("Author Name");
        book1.setISBN(123456);
        book1.setQuantity(10);
        book1.setCategory("Book Category");
        book1.setQuality(Quality.NEW);
        book1.setPrice(99.99);
        book1.setRatingNo(0);
        book1.setRatingTotal(0);
        book1.setServiceType(ServiceType.E_BOOK);
        bookRepository.save(book1);
        bookRepository.delete(book1);
        Book findBook = bookRepository.findById(0L).orElse(null);
        assertNull(findBook);
    }

    // test to save a new book
    @Test
    @Rollback(true)
    public void save_new_book() {
        Book book1 = new Book();
        book1.setId(1L);
        book1.setTitle("Book Title");
        book1.setAuthorName("Author Name");
        book1.setISBN(123456);
        book1.setQuantity(10);
        book1.setCategory("Book Category");
        book1.setQuality(Quality.NEW);
        book1.setPrice(99.99);
        book1.setRatingNo(0);
        book1.setRatingTotal(0);
        book1.setServiceType(ServiceType.E_BOOK);
        bookRepository.save(book1);

        Book findBook = bookRepository.findById(1L).orElse(null);
        assertNotNull(findBook);
    }
```

```java
    // test to see if a book exists with valid params
    @Test
    @Rollback(true)
    public void book_exists_returns_true_if_book_exists() {
        Book book1 = new Book();
        book1.setId(1L);
        book1.setTitle("Book Title");
        book1.setAuthorName("Author Name");
        book1.setISBN(123456);
        book1.setQuantity(10);
        book1.setCategory("Book Category");
        book1.setQuality(Quality.NEW);
        book1.setPrice(99.99);
        book1.setRatingNo(0);
        book1.setRatingTotal(0);
        book1.setServiceType(ServiceType.E_BOOK);
        book1.setBookStatus(BookStatus.AVAILABLE);
        User seller = new User();
        seller.setId(1L);
        book1.setSeller(seller);
        Request request = new Request();
        request.setId(1L);
        book1.setRequest(request);
        bookRepository.save(book1);

        boolean newbookExists = bookRepository.bookExists(seller, "Book Title".toLowerCase(),
                "Author Name".toLowerCase(), "Book Category".toLowerCase(), 123456, Quality.NEW);
        assertEquals(true, newbookExists);
    }
```

```java
    // test to see if a book exists with invalid params
    @Test
    @Rollback(true)
    public void book_exists_returns_false_if_book_does_not_exist() {
        Book book1 = new Book();
        book1.setId(1L);
        book1.setTitle("Book Title");
        book1.setAuthorName("Author Name");
        book1.setISBN(123456);
        book1.setQuantity(10);
        book1.setCategory("Book Category");
        book1.setQuality(Quality.NEW);
        book1.setPrice(99.99);
        book1.setRatingNo(0);
        book1.setRatingTotal(0);
        book1.setServiceType(ServiceType.E_BOOK);
        book1.setBookStatus(BookStatus.AVAILABLE);
        User seller = new User();
        seller.setId(1L);
        book1.setSeller(seller);
        Request request = new Request();
        request.setId(1L);
        book1.setRequest(request);
        bookRepository.save(book1);

        boolean newbookExists = bookRepository.bookExists(seller, "Wrong Book Title".toLowerCase(),
                "Author Name".toLowerCase(), "Book Category".toLowerCase(), 123456, Quality.NEW);
        assertEquals(false, newbookExists);
    }
```

```java
    // test to find a book with valid params
    @Test
    @Rollback(true)
    public void find_with_params_returns_correct_book_if_book_exists() {
        Book book1 = new Book();
        book1.setId(1L);
        book1.setTitle("Book Title");
        book1.setAuthorName("Author Name");
        book1.setISBN(123456);
        book1.setQuantity(10);
        book1.setCategory("Book Category");
        book1.setQuality(Quality.NEW);
        book1.setPrice(99.99);
        book1.setRatingNo(0);
        book1.setRatingTotal(0);
        book1.setServiceType(ServiceType.E_BOOK);
        book1.setBookStatus(BookStatus.AVAILABLE);
        User seller = new User();
        seller.setId(1L);
        book1.setSeller(seller);
        Request request = new Request();
        request.setId(1L);
        book1.setRequest(request);
        bookRepository.save(book1);

        Book findBook = bookRepository.findWithParams(seller, "Book Title".toLowerCase(), "Author Name".toLowerCase(),
                "Book Category".toLowerCase(), 123456, Quality.NEW);
        assertEquals(book1.getTitle(), findBook.getTitle());
        assertEquals(book1.getAuthorName(), findBook.getAuthorName());
        assertEquals(book1.getISBN(), findBook.getISBN());
        assertEquals(book1.getQuantity(), findBook.getQuantity());
        assertEquals(book1.getCategory(), findBook.getCategory());
        assertEquals(book1.getQuality(), findBook.getQuality());
        assertEquals(book1.getPrice(), findBook.getPrice());
        assertEquals(book1.getRatingNo(), findBook.getRatingNo());
        assertEquals(book1.getRatingTotal(), findBook.getRatingTotal());
        assertEquals(book1.getServiceType(), findBook.getServiceType());
        assertEquals(book1.getBookStatus(), findBook.getBookStatus());
        assertEquals(book1.getSeller().getId(), findBook.getSeller().getId());
    }
```

```java
    // test to find a book with valid params
    @Test
    @Rollback(true)
    public void find_with_params_returns_null_book_if_book_does_not_exist() {
        User seller = new User();
        seller.setId(1L);

        Book findBook = bookRepository.findWithParams(seller, "Book Title".toLowerCase(), "Author Name".toLowerCase(),
                "Book Category".toLowerCase(), 123456, Quality.NEW);
        assertNull(findBook);

    }
}
```

BookServiceTest.java

```java
    @Test
    @DisplayName("Test findAllBooks success") // test for finding all books
    public void testFindAllBooksSuccess() throws Exception {
        given(bookRepository.findAll()).willReturn(books);
        Iterable<Book> allBooks = bookService.findAllBooks();
        Assert.assertNotNull(allBooks);
    }

    @Test
    @DisplayName("Test findBySeller success") // test for finding all books by seller successfully
    public void testFindBySellerSuccess() throws Exception {
        User user = new User();
        given(bookRepository.findBySeller(user)).willReturn(books);
        Iterable<Book> allBooks = bookService.getAllBySeller(user);
        Assert.assertNotNull(allBooks);
    }

    @Test
    @DisplayName("Test findBySeller fail") // test for finding all books by seller which doesn't exist
    public void testFindBySellerFail() throws Exception {
        given(bookRepository.findBySeller(null)).willReturn(null);
        Iterable<Book> allBooks = bookService.getAllBySeller(null);
        Assert.assertNull(allBooks);
    }
```

Browsing microservice Tests:

Tester: Jared

BrowsingControllerTest.java

```java
    @Test
    @DisplayName("Test getByTitle") // test for getting by book title
    void testGetByTitle() throws Exception {
            // Mocking service
            when(browsingService.findAllByTitle("title")).thenReturn(books);
            mockMvc.perform(get("/api/browse/title/title").contentType(MediaType.APPLICATION_JSON))
                        .andExpect(status().isOk()).andExpect(jsonPath("$[0].title", is("Book Title")))
                        .andExpect(jsonPath("$[1].title", is("Book Title 2")));
    }

    @Test
    @DisplayName("Test getByAuthorName") // test for getting by author name
    void testGetByAuthorName() throws Exception {
            // Mocking service
            when(browsingService.findAllByAuthorName("author")).thenReturn(books);
            mockMvc.perform(get("/api/browse/authorName/author").contentType(MediaType.APPLICATION_JSON))
                        .andExpect(status().isOk()).andExpect(jsonPath("$[0].authorName", is("Author Name")))
                        .andExpect(jsonPath("$[1].authorName", is("Author Name 2")));
    }

    @Test
    @DisplayName("Test getBySeller") // test for getting by seller
    void testGetBySeller() throws Exception {
            // Mocking service
            when(browsingService.findAllBySeller(ArgumentMatchers.any(User.class))).thenReturn(newBooks);
            mockMvc.perform(get("/api/browse/sellerId/1").contentType(MediaType.APPLICATION_JSON))
                        .andExpect(status().isOk())
                        .andExpect(jsonPath("$[0].quality", is(Quality.NEW.toString())));
    }

    @Test
    @DisplayName("Test getByCategory") // test for getting by category
    void testGetByCategory() throws Exception {
            // Mocking service
            when(browsingService.findAllByCategory("category")).thenReturn(books);
            mockMvc.perform(get("/api/browse/category/category").contentType(MediaType.APPLICATION_JSON))
                        .andExpect(status().isOk()).andExpect(jsonPath("$[0].category", is("Book Category")))
                        .andExpect(jsonPath("$[1].category", is("Book Category 2")));
    }
```

```java
    @Test
    @DisplayName("Test getByISBN") // test for getting by isbn
    void testGetByISBN() throws Exception {
            // Mocking service
            when(browsingService.findAllByISBN(123)).thenReturn(books);
            mockMvc.perform(get("/api/browse/isbn/123").contentType(MediaType.APPLICATION_JSON))
                    .andExpect(status().isOk()).andExpect(jsonPath("$[0].isbn", is(123456)))
                    .andExpect(jsonPath("$[1].isbn", is(1234567)));
    }

    @Test
    @DisplayName("Test getNewBooks") // test for getting all new books
    void testGetNewBooks() throws Exception {
            // Mocking service
            when(browsingService.findAllNewBooks()).thenReturn(books);
            mockMvc.perform(get("/api/browse/new").contentType(MediaType.APPLICATION_JSON))
                    .andExpect(status().isOk())
                    .andExpect(jsonPath("$[0].quality", is(Quality.NEW.toString())));
    }

    @Test
    @DisplayName("Test getUsedBooks") // test for getting all used books
    void testGetUsedBooks() throws Exception {
            // Mocking service
            when(browsingService.findAllUsedBooks()).thenReturn(usedBooks);
            mockMvc.perform(get("/api/browse/used").contentType(MediaType.APPLICATION_JSON))
                    .andExpect(status().isOk())
                    .andExpect(jsonPath("$[0].quality", is(Quality.USED.toString())));
    }

    @Test
    @DisplayName("Test sortByPriceHighToLow") // test for getting books sorted from highest to lowest price
    void testSortByPriceHighToLow() throws Exception {
            // Mocking service
            when(browsingService.sortByHighestPrice()).thenReturn(books);
            mockMvc.perform(get("/api/browse/price/high").contentType(MediaType.APPLICATION_JSON))
                    .andExpect(status().isOk()).andExpect(jsonPath("$[0].price", is(99.99)))
                    .andExpect(jsonPath("$[1].price", is(89.99)));
    }
```

```java
    @Test
    @DisplayName("Test sortByPriceLowToHigh") // test for getting all sorted from lowest to highest price
    void testSortByPriceLowToHigh() throws Exception {
            // Mocking service
            when(browsingService.sortByLowestPrice()).thenReturn(books2);
            mockMvc.perform(get("/api/browse/price/low").contentType(MediaType.APPLICATION_JSON))
                    .andExpect(status().isOk()).andExpect(jsonPath("$[0].price", is(89.99)))
                    .andExpect(jsonPath("$[1].price", is(99.99)));
    }

    @Test
    @DisplayName("Test sortByAlphabet") // test for getting all sorted in alphabetical title order
    void testSortByAlphabet() throws Exception {
            // Mocking service
            when(browsingService.sortByAlphabet()).thenReturn(books);
            mockMvc.perform(get("/api/browse/alphabet").contentType(MediaType.APPLICATION_JSON))
                    .andExpect(jsonPath("$[0].title", is("Book Title")))
                    .andExpect(jsonPath("$[1].title", is("Book Title 2")));
    }

    @Test
    @DisplayName("Test bestSellers") // test for getting all sorted by highest rating
    void testBestSellers() throws Exception {
            // Mocking service
            when(browsingService.sortByHighestRating(2)).thenReturn(books2);
            mockMvc.perform(get("/api/browse/bestSellers/2").contentType(MediaType.APPLICATION_JSON))
                    .andExpect(jsonPath("$[0].ratingTotal", is(1)))
                    .andExpect(jsonPath("$[1].ratingTotal", is(0)));
    }

    @Test
    @DisplayName("Test random") // test for getting two randpo
    void testRandom() throws Exception {
            // Mocking service
            when(browsingService.random(2)).thenReturn(books2);
            mockMvc.perform(get("/api/browse/random/2").contentType(MediaType.APPLICATION_JSON))
                    .andExpect(jsonPath("$[0].id", is(2))).andExpect(jsonPath("$[1].id", is(1)));
    }
}
```

BookRepositoryTest.java

```java
    // testing for browsing when books have a given title
    @Test
    @Rollback(true)
    public void test_if_books_exist_with_title() {
        Iterable<Book> books = bookRepository.findByTitle("peril");
        assertThat(books).isNotEmpty();
    }

    // testing for browsing when no books have a given title
    @Test
    @Rollback(true)
    public void test_empty_result_if_no_books_with_title() {
        Iterable<Book> books = bookRepository.findByTitle("TestTitle");
        assertThat(books).isEmpty();
    }

    // testing for browsing when books have a given author's name
    @Test
    @Rollback(true)
    public void test_if_books_exist_with_author_name() {
        Iterable<Book> books = bookRepository.findByAuthorName("dave");
        assertThat(books).isNotEmpty();
    }

    // testing for browsing when no books have a given author's name
    @Test
    @Rollback(true)
    public void test_empty_result_if_no_books_with_author_name() {
        Iterable<Book> books = bookRepository.findByAuthorName("authorName");
        assertThat(books).isEmpty();
    }

    // testing for browsing when books have a given category
    @Test
    @Rollback(true)
    public void test_if_books_exist_with_category() {
        Iterable<Book> books = bookRepository.findByCategory("business");
        assertThat(books).isNotEmpty();
    }

```

```java
    // testing for browsing when no books have a given category
    @Test
    @Rollback(true)
    public void test_empty_result_if_no_books_with_category() {
        Iterable<Book> books = bookRepository.findByCategory("TestCategory");
        assertThat(books).isEmpty();
    }

    // testing for browsing when books have a given isbn
    @Test
    @Rollback(true)
    public void test_if_books_exist_with_isbn() {
        Iterable<Book> books = bookRepository.findByisbn("198218291");
        assertThat(books).isNotEmpty();
    }

    // testing for browsing when no books have a given isbn
    @Test
    @Rollback(true)
    public void test_empty_result_if_no_books_with_isbn() {
        Iterable<Book> books = bookRepository.findByisbn("falseisbn");
        assertThat(books).isEmpty();
    }

    // testing for browsing when books have a given quality - new
    @Test
    @Rollback(true)
    public void test_if_books_exist_with_quality_new() {
        Iterable<Book> books = bookRepository.findByQuality(Quality.NEW);
        Iterator<Book> iter = books.iterator();
        while (iter.hasNext()) {
            Book book = iter.next();
            assertThat(book.getQuality()).isEqualTo(Quality.NEW);
        }
    }

    // testing for browsing when books have a given quality - used
    @Test
    @Rollback(true)
    public void test_if_books_exist_with_quality_used() {
        Iterable<Book> books = bookRepository.findByQuality(Quality.USED);
        Iterator<Book> iter = books.iterator();
        while (iter.hasNext()) {
            Book book = iter.next();
            assertThat(book.getQuality()).isEqualTo(Quality.USED);
        }
    }
```

```java
    // testing for browsing by books sorted by highest price
    @Test
    @Rollback(true)
    public void test_sort_by_highest_price() {
        Iterable<Book> books = bookRepository.sortByHighestPrice();
        Iterator<Book> iter = books.iterator();
        double maxPrice = Integer.MAX_VALUE;
        while (iter.hasNext()) {
            Book book = iter.next();
            double price = book.getPrice();
            assertThat(price <= maxPrice).isTrue();
            maxPrice = price;
        }
    }

    // testing for browsing by books sorted by lowest price
    @Test
    @Rollback(true)
    public void test_sort_by_lowest_price() {
        Iterable<Book> books = bookRepository.sortByLowestPrice();
        Iterator<Book> iter = books.iterator();
        double minPrice = Integer.MIN_VALUE;
        while (iter.hasNext()) {
            Book book = iter.next();
            double price = book.getPrice();
            assertThat(price >= minPrice).isTrue();
            minPrice = price;
        }
    }

    // testing for browsing by books sorted by alphabetical title order
    @Test
    @Rollback(true)
    public void test_sort_by_alphabet() {
        Iterable<Book> books = bookRepository.sortByAlphabet();
        Iterator<Book> iter = books.iterator();
        String prevTitle = "a";
        while (iter.hasNext()) {
            Book book = iter.next();
            String title = book.getTitle();
            assertThat(prevTitle.compareTo(title)).isGreaterThan(-1);
            title = prevTitle;
        }
    }
```

```java
    // testing for browsing by books sorted by newest release
    @Test
    @Rollback(true)
    public void test_sort_by_newest_release() {
        Iterable<Book> books = bookRepository.sortByNewestRelease(10);
        Iterator<Book> iter = books.iterator();
        Date newestDate = new Date();
        while (iter.hasNext()) {
            Book book = iter.next();
            Date date = book.getcreated_At();
            assertThat(newestDate.compareTo(date)).isGreaterThan(-1);
            newestDate = date;
        }
    }

    // testing for browsing by books sorted by highest rating
    @Test
    @Rollback(true)
    public void test_sort_by_highest_rating() {
        Iterable<Book> books = bookRepository.sortByHighestRating(10);
        Iterator<Book> iter = books.iterator();
        double maxRating = 5.0;
        while (iter.hasNext()) {
            Book book = iter.next();
            double rating = book.getRatingTotal()/book.getRatingNo();
            assertThat(maxRating >= rating).isTrue();
            maxRating = rating;
        }
    }
}
```

BrowsingServiceTest.java

```java
    @Test
    @DisplayName("Test findByTitle") // test for finding by title
    public void testFindByTitle() throws Exception {
        given(bookRepository.findByTitle("title")).willReturn(books);
        Iterable<Book> books = browsingService.findAllByTitle("title");
        Assert.assertNotNull(books);
    }

    @Test
    @DisplayName("Test findByAuthorName") // test for finding by author name
    public void testFindByAuthorName() throws Exception {
        given(bookRepository.findByAuthorName("author")).willReturn(books);
        Iterable<Book> books = browsingService.findAllByTitle("author");
        Assert.assertNotNull(books);
    }

    @Test
    @DisplayName("Test findByisbn") // test for finding by isbn
    public void testFindByisbn() throws Exception {
        given(bookRepository.findByisbn("123456")).willReturn(books);
        Iterable<Book> books = browsingService.findAllByISBN(123456);
        Assert.assertNotNull(books);
    }

    @Test
    @DisplayName("Test findByCategory") // test for finding by category
    public void testFindByCategory() throws Exception {
        given(bookRepository.findByCategory("book")).willReturn(books);
        Iterable<Book> books = browsingService.findAllByTitle("book");
        Assert.assertNotNull(books);
    }

    @Test
    @DisplayName("Test findAllNewBooks") // test for finding all new books
    public void testFindAllNewBooks() throws Exception {
        given(bookRepository.findByQuality(Quality.NEW)).willReturn(newBooks);
        Iterable<Book> books = browsingService.findAllNewBooks();
        Iterator<Book> iter = books.iterator();
        while (iter.hasNext()) {
            Book book = iter.next();
            assertThat(book.getQuality()).isEqualTo(Quality.NEW);
        }
    }
```

```java
    @Test
    @DisplayName("Test findAllUsedBooks") // test for finding all used books
    public void testFindAllUsedBooks() throws Exception {
        given(bookRepository.findByQuality(Quality.USED)).willReturn(usedBooks);
        Iterable<Book> books = browsingService.findAllUsedBooks();
        Iterator<Book> iter = books.iterator();
        while (iter.hasNext()) {
            Book book = iter.next();
            assertThat(book.getQuality()).isEqualTo(Quality.USED);
        }
    }

    @Test
    @DisplayName("Test sortByHighestPrice") // test for sorting by highest price
    public void testSortByHighestPrice() throws Exception {
        given(bookRepository.sortByHighestPrice()).willReturn(books);
        Iterable<Book> books = browsingService.sortByHighestPrice();
        Iterator<Book> iter = books.iterator();
        double maxPrice = Integer.MAX_VALUE;
        while (iter.hasNext()) {
            Book book = iter.next();
            double price = book.getPrice();
            assertThat(price <= maxPrice).isTrue();
            maxPrice = price;
        }
    }

    @Test
    @DisplayName("Test sortByLowestPrice") // test for sorting by lowest price
    public void testSortByLowestPrice() throws Exception {
        given(bookRepository.sortByHighestPrice()).willReturn(books2);
        Iterable<Book> books = browsingService.sortByLowestPrice();
        Iterator<Book> iter = books.iterator();
        double minPrice = Integer.MIN_VALUE;
        while (iter.hasNext()) {
            Book book = iter.next();
            double price = book.getPrice();
            assertThat(price >= minPrice).isTrue();
            minPrice = price;
        }
    }

```

```java
    @Test
    @DisplayName("Test sortByAlphabet") // test for sorting by alphabetical title order
    public void testSortByAlphabet() throws Exception {
        given(bookRepository.sortByAlphabet()).willReturn(books);
        Iterable<Book> books = bookRepository.sortByAlphabet();
        Iterator<Book> iter = books.iterator();
        String prevTitle = "a";
        while (iter.hasNext()) {
            Book book = iter.next();
            String title = book.getTitle();
            assertThat(prevTitle.compareTo(title)).isGreaterThan(-1);
            title = prevTitle;
        }
    }

    @Test
    @DisplayName("Test sortByNewestRelease") // testing for browsing by books sorted by newest release
    public void testSortByNewestRelease() {
        Iterable<Book> books = bookRepository.sortByNewestRelease(10);
        Iterator<Book> iter = books.iterator();
        Date newestDate = new Date();
        while (iter.hasNext()) {
            Book book = iter.next();
            Date date = book.getcreated_At();
            assertThat(newestDate.compareTo(date)).isGreaterThan(-1);
            newestDate = date;
        }
    }

    @Test
    @DisplayName("Test sortByHighestRating") // testing for browsing by books sorted by highest rating
    public void testSortByHighestRating() {
        Iterable<Book> books = bookRepository.sortByHighestRating(10);
        Iterator<Book> iter = books.iterator();
        double maxRating = 5.0;
        while (iter.hasNext()) {
            Book book = iter.next();
            double rating = book.getRatingTotal()/book.getRatingNo();
            assertThat(maxRating >= rating).isTrue();
            maxRating = rating;
        }
    }
}
```

Login microservice Tests:

Tester: Jared

LoginControllerTest.java

```java
    @Test
    @DisplayName("Test login success") // test for logging in successfully
    void testLoginSuccess() throws Exception {
        // Mocking service
        when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
                .thenReturn(null);

        String inputJson = "{\n" + "    \"username\":\"JohnDoe\",\n" + "    \"password\":\"password\"\n" + "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/users/login")
                .accept(MediaType.APPLICATION_JSON).content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andExpect(jsonPath("success", is(true))).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.OK.value(), response.getStatus());
    }

    @Test
    @DisplayName("Test login failure") // test for failing to log in with incorrect user information
    void testLoginFailure() throws Exception {
        // Mocking service
        String inputJson = "{\n" + "    \"username\":\"WrongUsername\",\n" + "    \"password\":\"password\"\n" + "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/users/login")
                .accept(MediaType.APPLICATION_JSON).content(inputJson).contentType(MediaType.APPLICATION_JSON);
        MvcResult result = mockMvc.perform(requestBuilder).andExpect(jsonPath("token", is("Bearer null"))).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.OK.value(), response.getStatus());
    }

}

```

Requests microservice Tests:

Tester: Jared

RequestsControllerTest.java

```java
    @Test
    @DisplayName("Test findAllRequests") // test for getting all requests
    void testFindAllRequests() throws Exception {
        // Mocking service
        when(requestService.findAllRequests()).thenReturn(requests);
        mockMvc.perform(get("/api/requests/all").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
                .andExpect(jsonPath("$[0].id", is(1))).andExpect(jsonPath("$[0].objectId", is(1)))
                .andExpect(jsonPath("$[0].request", is("New Book"))).andExpect(jsonPath("$[0].requestType", is(RequestType.NEW_BOOK_LISTING.toString())))
                .andExpect(jsonPath("$[0].user.id", is(1)))

                .andExpect(jsonPath("$[1].id", is(2))).andExpect(jsonPath("$[1].objectId", is(2)))
                .andExpect(jsonPath("$[1].request", is("New Book"))).andExpect(jsonPath("$[1].requestType", is(RequestType.NEW_BOOK_LISTING.toString())))
                .andExpect(jsonPath("$[1].user.id", is(1)));

    }

    @Test
    @DisplayName("Test getRequest success") // test for getting a request successfully
    void testGetRequestSuccess() throws Exception {
        // Mocking service
        when(requestService.findById(1L)).thenReturn(requests.get(0));
        mockMvc.perform(get("/api/requests/1").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
                .andExpect(jsonPath("$.id", is(1))).andExpect(jsonPath("$.objectId", is(1)))
                .andExpect(jsonPath("$.request", is("New Book")))
                .andExpect(jsonPath("$.requestType", is(RequestType.NEW_BOOK_LISTING.toString())))
                .andExpect(jsonPath("$.user.id", is(1)));
    }

    @Test
    @DisplayName("Test getRequest not found") // test for getting a request that doesn't exist
    void testGetRequestNotFound() throws Exception {
        // Mocking service
        when(requestService.findById(0L)).thenReturn(null);

        RequestBuilder requestBuilder = MockMvcRequestBuilders.get("/api/requests/0")
                .contentType(MediaType.APPLICATION_JSON);
        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
        assertEquals("Request with ID 0 was not found", response.getContentAsString());
    }

```

```java
    @Test
    @DisplayName("Test addNewRequest success") // test for adding a new request successfully
    void testAddNewRequestSuccess() throws Exception {
        // Mocking service
        when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
                .thenReturn(null);
        when(requestService.saveRequest(ArgumentMatchers.any(Request.class))).thenReturn(requests.get(0));
        String inputJson = "{\n" + " \"userId\":\"1\",\n" + "\"objectId\":\"1\",\n"
                + " \"requestType\":\"1\"\n" + "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/requests/new").accept(MediaType.APPLICATION_JSON)
                .content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.CREATED.value(), response.getStatus());
    }

    @Test
    @DisplayName("Test addNewRequest null user error") // test for adding a new request without a user
    void testAddNewRequestNullUserError() throws Exception {
        // Mocking service
        when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
                .thenReturn(null);
        when(requestService.saveRequest(ArgumentMatchers.any(Request.class))).thenReturn(requests.get(0));
        String inputJson = "{\n" + "\"objectId\":\"1\",\n"
                + " \"requestType\":\"1\"\n" + "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/requests/new").accept(MediaType.APPLICATION_JSON)
                .content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_ACCEPTABLE.value(), response.getStatus());
        assertEquals("Unable to add the new request, User id not given!.", response.getContentAsString());
    }
```

```java
    @Test
    @DisplayName("Test addNewRequest duplicate error") // test for adding a duplicate new request
    void testAddNewRequestDuplicateError() throws Exception {
        // Mocking service
        when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
                .thenReturn(null);
                when(requestService.saveRequest(ArgumentMatchers.any(Request.class))).thenReturn(null);
                String inputJson = "{\n" + " \"userId\":\"1\",\n" + "\"objectId\":\"1\",\n"
                        + " \"requestType\":\"1\"\n" + "}";
                RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/requests/new").accept(MediaType.APPLICATION_JSON)
                        .content(inputJson).contentType(MediaType.APPLICATION_JSON);

                MvcResult result = mockMvc.perform(requestBuilder).andReturn();
                MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.CONFLICT.value(), response.getStatus());
        assertEquals("Unable to add the new request, a copy of the request already exists.", response.getContentAsString());
    }

    @Test
    @DisplayName("Test addNewRequest badRequest") // test for adding a new request with invalid information
    void testAddNewRequestBadRequest() throws Exception {
        // Mocking service
        when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
                .thenReturn(null);

        String inputJson = "null";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/requests/new").accept(MediaType.APPLICATION_JSON)
                .content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.BAD_REQUEST.value(), response.getStatus());
    }
```

```java
    @Test
    @DisplayName("Test deleteRequest success") // test for successfully deleting a request
    void testDeleteRequestSuccess() throws Exception {
        // Mocking service
        when(requestService.findById(1L)).thenReturn(requests.get(0));

        RequestBuilder requestBuilder = MockMvcRequestBuilders.delete("/api/requests/1");

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.OK.value(), response.getStatus());
        assertEquals("Request with ID 1 was deleted", response.getContentAsString());
    }

    @Test
    @DisplayName("Test deleteRequest not found") // test for deleting a request that doesn't exist
    void testDeleteRequestNotFound() throws Exception {
        // Mocking service
        when(requestService.findById(0L)).thenReturn(null);

        RequestBuilder requestBuilder = MockMvcRequestBuilders.delete("/api/requests/0");

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
        assertEquals("Request with ID 0 was not found", response.getContentAsString());
    }
}
```

RequestsRepositoryTest.java

```java
    // testing for empty request repository
    @Test
    @Rollback(true)
    public void should_find_no_requests_if_requestRepository_is_empty() {
        Iterable<Request> requests = requestRepository.findAll();
        if (requests.iterator().hasNext()) {
            assertThat(requests).isNotEmpty();
        } else {
            assertThat(requests).isEmpty();
        }
    }

    // test to find a request by id
    @Test
    @Rollback(true)
    public void find_request_by_id() {
        Request request1 = new Request();
        request1.setId(1L);
        requestRepository.save(request1);

        Request findRequest = requestRepository.findById(1L).orElse(null);
        assertNotNull(findRequest);
    }

    // test to find a request by non-existing id
    @Test
    @Rollback(true)
    public void find_request_by_nonexisting_id() {
        Request findRequest = requestRepository.findById(0L).orElse(null);
        assertNull(findRequest);
    }

```

```java
    // test to delete a request
    @Test
    @Rollback(true)
    public void delete_request() {
        Request request1 = new Request();
        request1.setId(1L);
        requestRepository.save(request1);
        requestRepository.delete(request1);
        Request findRequest = requestRepository.findById(1L).orElse(null);
        assertNull(findRequest);
    }

    // test to save a new request
    @Test
    @Rollback(true)
    public void save_new_request() {
        Request request1 = new Request();
        request1.setId(1L);
        requestRepository.save(request1);

        Request findRequest = requestRepository.findById(1L).orElse(null);
        assertNotNull(findRequest);

        Request request2 = new Request();
        request1.setId(2L);
        requestRepository.save(request2);

        Request findRequest2 = requestRepository.findById(2L).orElse(null);
        assertNotNull(findRequest2);
    }
}

```

RequestsServiceTest.java

```java
    @Test
    @DisplayName("Test findAllRequests success") // test for finding all requests
    public void testFindAllRequestsSuccess() throws Exception {
        given(requestRepository.findAll()).willReturn(requests);
        Iterable<Request> allRequests = requestService.findAllRequests();
        Assert.assertNotNull(allRequests);
    }

    @Test
    @DisplayName("Test saveRequest success") // test for saving a request
    public void testSaveRequestSuccess() throws Exception {
        given(requestRepository.save(request1)).willReturn(request1);
        Request approvedRequest = requestService.saveRequest(request1);
        Assert.assertNotNull(approvedRequest);
    }

    @Test
    @DisplayName("Test saveRequest failure") // test for failing to save a request
    public void testSaveRequestFailure() throws Exception {
        given(requestRepository.save(request2)).willReturn(null);
        Request approvedRequest = requestService.saveRequest(request1);
        Assert.assertNull(approvedRequest);
    }
}
```

Users microservice Tests:

Tester: Jared

EditUserControllerTest.java

```java
    @Test
    @DisplayName("Test editUser success") // test for editing a user successfully
    void testEditUserSuccess() throws Exception {
        // Mocking service
        when(userService.findById(1L)).thenReturn(users.get(0));
        when(userService.updateUser(ArgumentMatchers.any(UserForm.class), ArgumentMatchers.any(User.class)))
                .thenReturn(users.get(0));

        String inputJson = "{\n" +
        "    \"username\":\"newUsername\",\n" +
        "    \"password\":\"newPassword\"\n" +
        "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/editUser/1")
                .accept(MediaType.APPLICATION_JSON).content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.OK.value(), response.getStatus());
        assertEquals("Successfully updated user details", response.getContentAsString());
    }

    @Test
    @DisplayName("Test editUser username taken") // test for trying to change a username to one that is already taken
    void testEditUserUsernameTaken() throws Exception {
        // Mocking service
        when(userService.findById(1L)).thenReturn(users.get(0));
        when(userService.updateUser(ArgumentMatchers.any(UserForm.class), ArgumentMatchers.any(User.class)))
                .thenReturn(null);

        String inputJson = "{\n" +
        "    \"username\":\"JohnDoe2\"\n" +
        "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/editUser/1")
                .accept(MediaType.APPLICATION_JSON).content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.CONFLICT.value(), response.getStatus());
        assertEquals("Unable to save details, Username 'JohnDoe2' already taken", response.getContentAsString());
    }
```

```java
    @Test
    @DisplayName("Test editUser not found") // test for editing a user that doesn't exist
    void testEditUserNotFound() throws Exception {
        // Mocking service
        when(userService.findById(3L)).thenReturn(null);

        String inputJson = "{\n" +
        "    \"username\":\"newUsername\",\n" +
        "    \"password\":\"newPassword\"\n" +
        "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/editUser/3")
                .accept(MediaType.APPLICATION_JSON).content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
        assertEquals("User with ID 3 was not found", response.getContentAsString());
    }

    @Test
    @DisplayName("Test editPassword success") // test for changing a user's password successfully
    void testEditPasswordSuccess() throws Exception {
        // Mocking service
        when(userService.findById(1L)).thenReturn(users.get(0));
        when(userService.updateUserPassword(ArgumentMatchers.any(UserForm.class), ArgumentMatchers.any(User.class)))
                .thenReturn(users.get(0));

        String inputJson = "{\n" +
        "    \"password\":\"newPassword\"\n" +
        "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/editUser/password/1")
                .accept(MediaType.APPLICATION_JSON).content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.OK.value(), response.getStatus());
        assertEquals("Successfully updated password", response.getContentAsString());
    }
```

```java
    @Test
    @DisplayName("Test editPassword invalid password") // test for changing a user's password to an invalid password
    void testEditPasswordInvalid() throws Exception {
        // Mocking service
        when(userService.findById(1L)).thenReturn(users.get(0));
        when(userService.updateUserPassword(ArgumentMatchers.any(UserForm.class), ArgumentMatchers.any(User.class)))
                .thenReturn(null);

        String inputJson = "{\n" +
        "    \"password\":\"newPassword\"\n" +
        "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/editUser/password/1")
                .accept(MediaType.APPLICATION_JSON).content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_ACCEPTABLE.value(), response.getStatus());
        assertEquals("Unable to update invalid password", response.getContentAsString());
    }

    @Test
    @DisplayName("Test editPassword user not found") // test for changing a user's password when the user doesn't exist
    void testEditPasswordUserNotFound() throws Exception {
        // Mocking service
        when(userService.findById(3L)).thenReturn(null);

        String inputJson = "{\n" +
        "    \"password\":\"newPassword\"\n" +
        "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/editUser/3")
                .accept(MediaType.APPLICATION_JSON).content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
        assertEquals("User with ID 3 was not found", response.getContentAsString());
    }
```

UserControllerTest.java

```java
    @Test
    @DisplayName("Test findAllUsers") // test for getting all users
    void testFindAllUsers() throws Exception {
        // Mocking service
        when(userService.findAllUsers()).thenReturn(users);
        mockMvc.perform(get("/api/users/all").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
                .andExpect(jsonPath("$[0].id", is(1))).andExpect(jsonPath("$[0].email", is("johndoe@gmail.com")))
                .andExpect(jsonPath("$[0].username", is("JohnDoe")))
                .andExpect(jsonPath("$[0].fullName", is("John Doe")))
                .andExpect(jsonPath("$[0].password", is("password")))
                .andExpect(jsonPath("$[0].address", is("1 John Street, Doeland")))

                .andExpect(jsonPath("$[1].id", is(2))).andExpect(jsonPath("$[1].email", is("johndoe2@gmail.com")))
                .andExpect(jsonPath("$[1].username", is("JohnDoe2")))
                .andExpect(jsonPath("$[1].fullName", is("John Doe")))
                .andExpect(jsonPath("$[1].password", is("password")))
                .andExpect(jsonPath("$[1].address", is("2 John Street, Doeland")));

    }

    @Test
    @DisplayName("Test getUser success") // test for getting a user successfully
    void testGetUserSuccess() throws Exception {
        // Mocking service
        when(userService.findById(1L)).thenReturn(users.get(0));

        mockMvc.perform(get("/api/users/1").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
                .andExpect(jsonPath("id", is(1))).andExpect(jsonPath("email", is("johndoe@gmail.com")))
                .andExpect(jsonPath("username", is("JohnDoe"))).andExpect(jsonPath("fullName", is("John Doe")))
                .andExpect(jsonPath("password", is("password")))
                .andExpect(jsonPath("address", is("1 John Street, Doeland")));
    }
```

```java
    @Test
    @DisplayName("Test getUser not found") // test for getting a user that doesn't exist
    void testGetUserNotFound() throws Exception {
        // Mocking service
        when(userService.findById(3L)).thenReturn(null);

        RequestBuilder requestBuilder = MockMvcRequestBuilders.get("/api/users/3")
                .contentType(MediaType.APPLICATION_JSON);
        MvcResult result = mockMvc.perform(requestBuilder).andReturn();

        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
        assertEquals("User with ID 3 was not found", response.getContentAsString());

    }

    @Test
    @DisplayName("Test registerUser success") // test for registering a user successfully
    void testRegisterUserSuccess() throws Exception {
        // Mocking service
        when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
                .thenReturn(null);
        when(userService.saveUser(ArgumentMatchers.any(User.class))).thenReturn(users.get(0));

        String inputJson = "{\n" + "    \"username\":\"TestUsername\",\n" + "    \"password\":\"password\"\n" + "}";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/users/register")
                .accept(MediaType.APPLICATION_JSON).content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.CREATED.value(), response.getStatus());
    }
```

```java
    @Test
    @DisplayName("Test registerUser badRequest") // test for registering a user with invalid credentials
    void testRegisterUserBadRequest() throws Exception {
        // Mocking service
        when(mapValidationErrorService.MapValidationService(ArgumentMatchers.any(BindingResult.class)))
                .thenReturn(null);
        when(userService.saveUser(ArgumentMatchers.any(User.class))).thenReturn(users.get(0));

        String inputJson = "null";
        RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/api/users/register")
                .accept(MediaType.APPLICATION_JSON).content(inputJson).contentType(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.BAD_REQUEST.value(), response.getStatus());
    }

    @Test
    @DisplayName("Test deleteUser success") // test for successfully deleting a user
    void testDeleteUserSuccess() throws Exception {
        // Mocking service
        when(userService.findById(1L)).thenReturn(users.get(0));

        RequestBuilder requestBuilder = MockMvcRequestBuilders.delete("/api/users/1");

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.OK.value(), response.getStatus());
        assertEquals("User with ID 1 was deleted", response.getContentAsString());
    }
```

```java
    @Test
    @DisplayName("Test deleteUser not found") // test for deleting a user that doesn't exist
    void testDeleteUserNotFound() throws Exception {
        // Mocking service
        when(userService.findById(3L)).thenReturn(null);

        RequestBuilder requestBuilder = MockMvcRequestBuilders.delete("/api/users/3");

        MvcResult result = mockMvc.perform(requestBuilder).andReturn();
        MockHttpServletResponse response = result.getResponse();

        assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());
        assertEquals("User with ID 3 was not found", response.getContentAsString());
    }
}
```

UserRepositoryTest.java

```java
    // testing for empty user repository
    @Test
    @Rollback(true)
    public void should_find_no_users_if_userRepository_is_empty() {
        Iterable<User> users = userRepository.findAll();
        if (users.iterator().hasNext()) {
            assertThat(users).isNotEmpty();
        } else {
            assertThat(users).isEmpty();
        }
    }

    // test to find a user by username
    @Test
    @Rollback(true)
    public void find_user_by_username() {
        User user = new User();
        user.setId(1L);
        user.setEmail("johndoe@gmail.com");
        user.setUsername("JohnDoe");
        user.setFullName("John Doe");
        user.setPassword("password");
        user.setAddress("1 John Street, Doeland");
        user.setRole(Role.USER_NORMAL);
        user.setUserStatus(UserStatus.ENABLED);
        userRepository.save(user);

        User findUser = userRepository.findByUsername("JohnDoe");
        assertNotNull(findUser);
        assertThat(findUser).hasFieldOrPropertyWithValue("email", "johndoe@gmail.com");
        assertThat(findUser).hasFieldOrPropertyWithValue("username", "JohnDoe");
        assertThat(findUser).hasFieldOrPropertyWithValue("fullName", "John Doe");
        assertThat(findUser).hasFieldOrPropertyWithValue("password", "password");
        assertThat(findUser).hasFieldOrPropertyWithValue("address", "1 John Street, Doeland");
        assertThat(findUser).hasFieldOrPropertyWithValue("role", Role.USER_NORMAL);
        assertThat(findUser).hasFieldOrPropertyWithValue("userStatus", UserStatus.ENABLED);
    }
```

```java
    // test to delete a user
    @Test
    @Rollback(true)
    public void delete_user() {
        User user = new User();
        user.setId(0L);
        user.setEmail("johndoe@gmail.com");
        user.setUsername("JohnDoe");
        user.setFullName("John Doe");
        user.setPassword("password");
        user.setAddress("1 John Street, Doeland");
        user.setRole(Role.USER_NORMAL);
        user.setUserStatus(UserStatus.ENABLED);
        User savedUser = userRepository.save(user);
        userRepository.delete(savedUser);

        Iterable<User> users = userRepository.findAll();
        Iterator<User> iter = users.iterator();
        while(iter.hasNext()) {
            User iterUser = iter.next();
            assertThat(iterUser.getUsername()).isNotEqualTo(savedUser.getUsername());
        }
    }
```

UserServiceTest.java

```java
    @Test
    @DisplayName("Test findByUsername success") // test for finding a user by username successfully
    public void testFindByUsernameSuccess() throws Exception {
        String username = user1.getUsername();
        given(userRepository.findByUsername(username)).willReturn(user1);
        User user = userService.findByUsername(username);
        Assert.assertEquals(user.getEmail(), "johndoe@gmail.com");
        Assert.assertEquals(user.getUsername(), "JohnDoe");
        Assert.assertEquals(user.getFullName(), "John Doe");
        Assert.assertEquals(user.getPassword(), "password");
        Assert.assertEquals(user.getAddress(), "1 John Street, Doeland");
    }

    @Test
    @DisplayName("Test findByUsername fail") // test for finding a user by username which doesn't exist
    public void testFindByUsernameFail() throws Exception {
        String username = "null";
        given(userRepository.findByUsername(username)).willReturn(null);
        User user = userService.findByUsername(username);
        Assert.assertNull(user);
    }
}
```

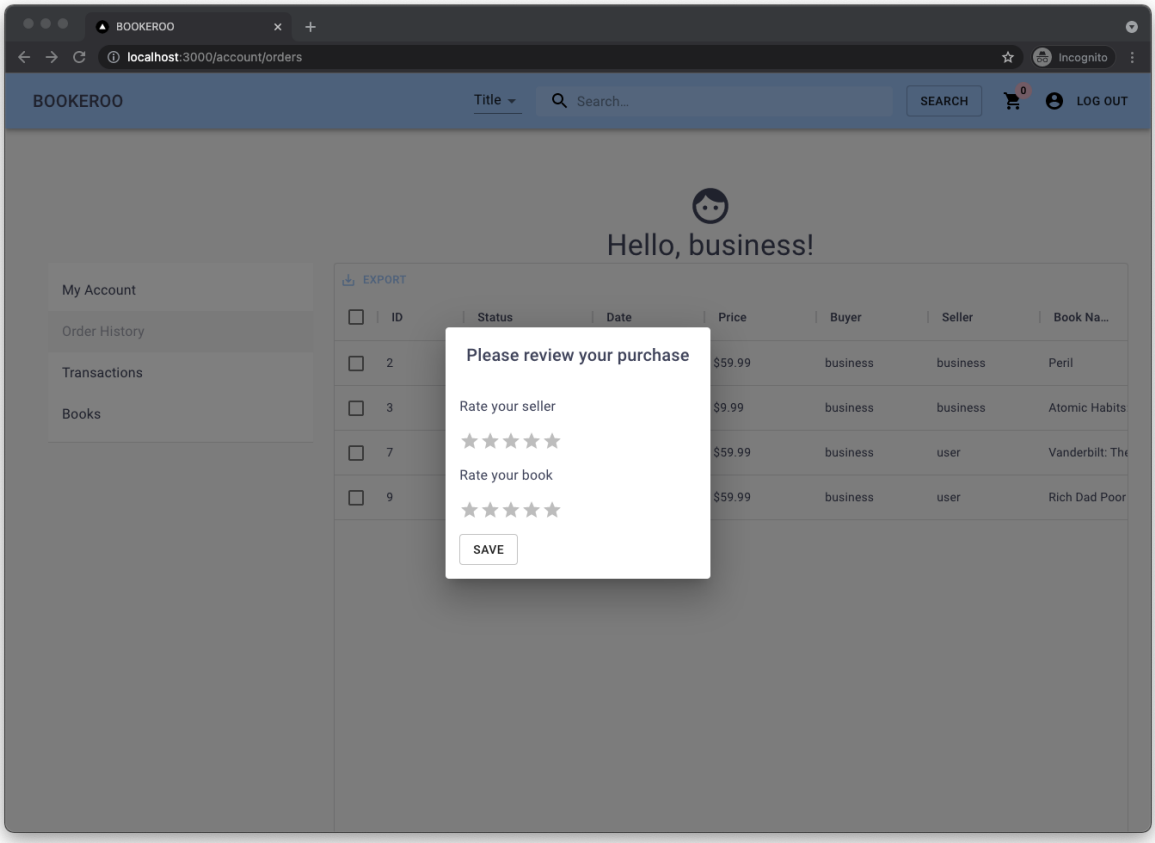
```java
    // test to save a new user
    @Test
    @Rollback(true)
    public void save_new_user() {
        User user = new User();
        user.setId(1L);
        user.setEmail("johndoe@gmail.com");
        user.setUsername("JohnDoe");
        user.setFullName("John Doe");
        user.setPassword("password");
        user.setAddress("1 John Street, Doeland");
        user.setRole(Role.USER_NORMAL);
        user.setUserStatus(UserStatus.ENABLED);
        userRepository.save(user);

        User savedUser = userRepository.save(user);

        assertThat(savedUser).hasFieldOrPropertyWithValue("email", "johndoe@gmail.com");
        assertThat(savedUser).hasFieldOrPropertyWithValue("username", "JohnDoe");
        assertThat(savedUser).hasFieldOrPropertyWithValue("fullName", "John Doe");
        assertThat(savedUser).hasFieldOrPropertyWithValue("password", "password");
        assertThat(savedUser).hasFieldOrPropertyWithValue("address", "1 John Street, Doeland");
        assertThat(savedUser).hasFieldOrPropertyWithValue("role", Role.USER_NORMAL);
        assertThat(savedUser).hasFieldOrPropertyWithValue("userStatus", UserStatus.ENABLED);
    }

```

```java
    // test to find all users after saving a new user
    @Test
    @Rollback(true)
    public void find_all_returns_saved_user() {
        User user = new User();
        user.setId(1L);
        user.setEmail("johndoe@gmail.com");
        user.setUsername("JohnDoe");
        user.setFullName("John Doe");
        user.setPassword("password");
        user.setAddress("1 John Street, Doeland");
        user.setRole(Role.USER_NORMAL);
        user.setUserStatus(UserStatus.ENABLED);
        userRepository.save(user);

        Iterable<User> users = userRepository.findAll();

        User savedUser = users.iterator().next();
        assertThat(savedUser.getEmail().equals("johndoe@gmail.com"));
        assertThat(savedUser.getUsername().equals("JohnDoe"));
        assertThat(savedUser.getFullName().equals("John Doe"));
        assertThat(savedUser.getAddress().equals("1 John Street, Doeland"));
        assertThat(savedUser.getRole().equals(Role.USER_NORMAL));
        assertThat(savedUser.getUserStatus().equals(UserStatus.ENABLED));
    }
}

```

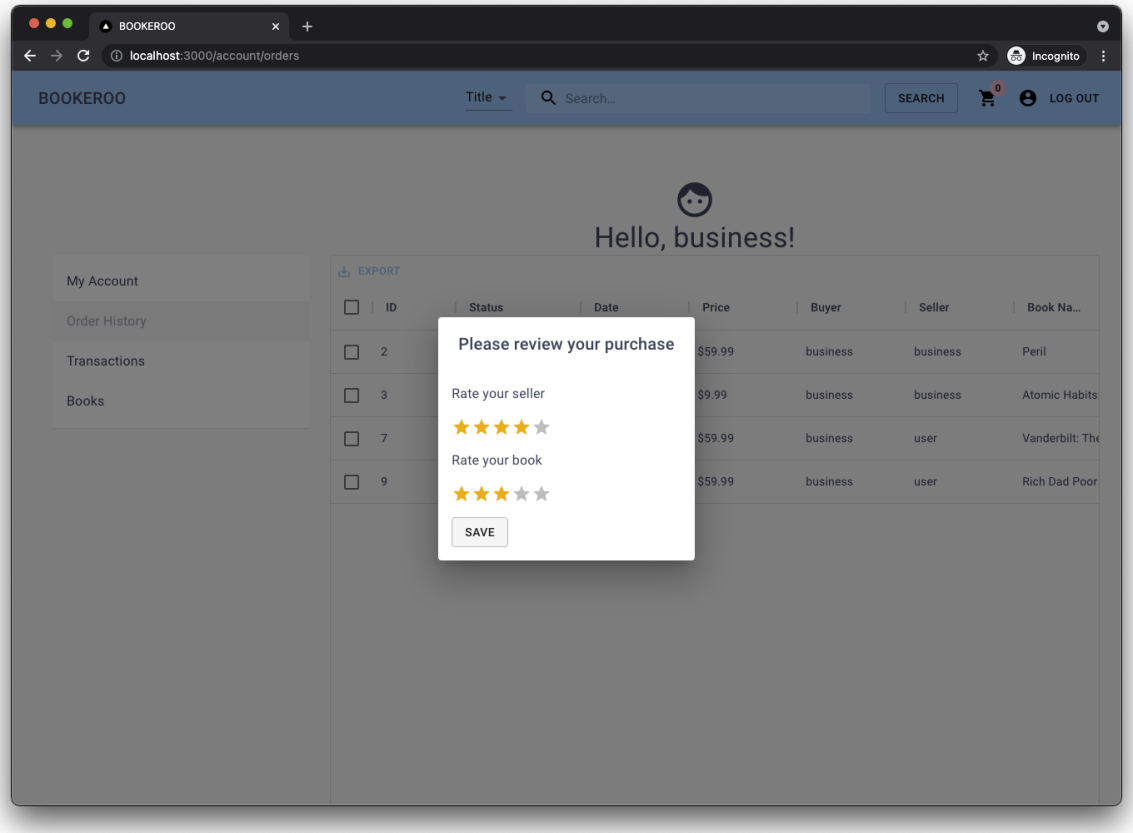
Acceptance Testing

*For new user stories in sprint 4

As a customer, I want to add a review to a book I purchased, so I can share my opinion.

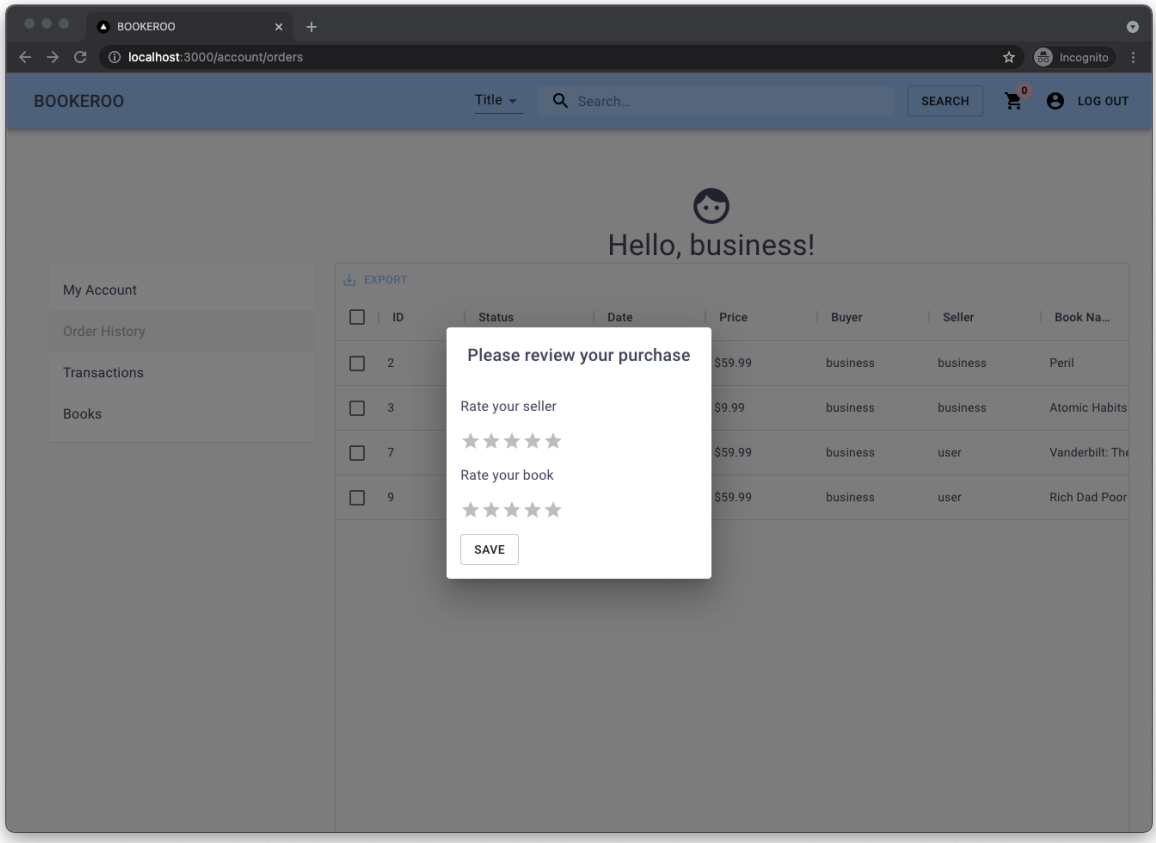
Given:
A user is login, and there is s transaction is ready to be review
When:
The user clicks the review button
Expect:
A review dialog will pop up
Screenshot:


Given:
A user is login, and the review dialog is open

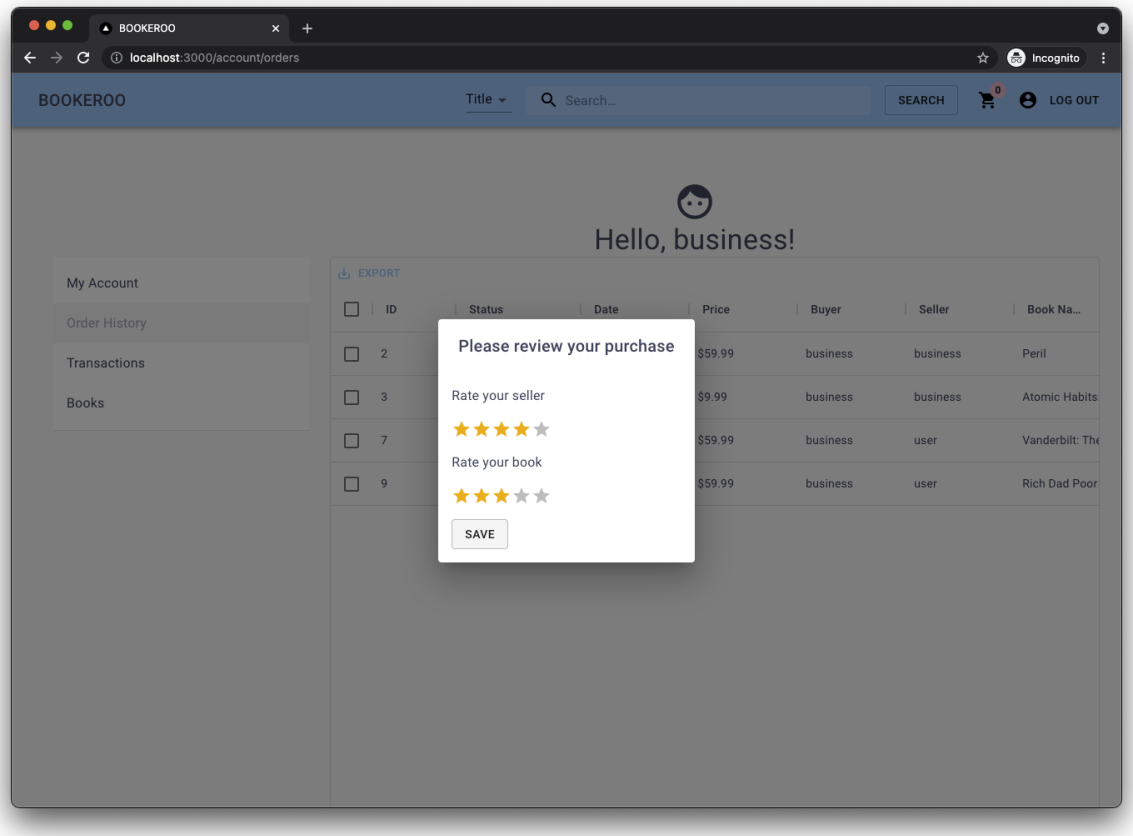
When:
The user adds the rating to the book and click save
Expect:
The review will be updated, and back end will be updated
Screenshot:


As a customer, I want to add a review to a user who sold an item to me, so I can rate the seller.

Given:
A user is login, and there is s transaction is ready to be review
When:
The user clicks the review button

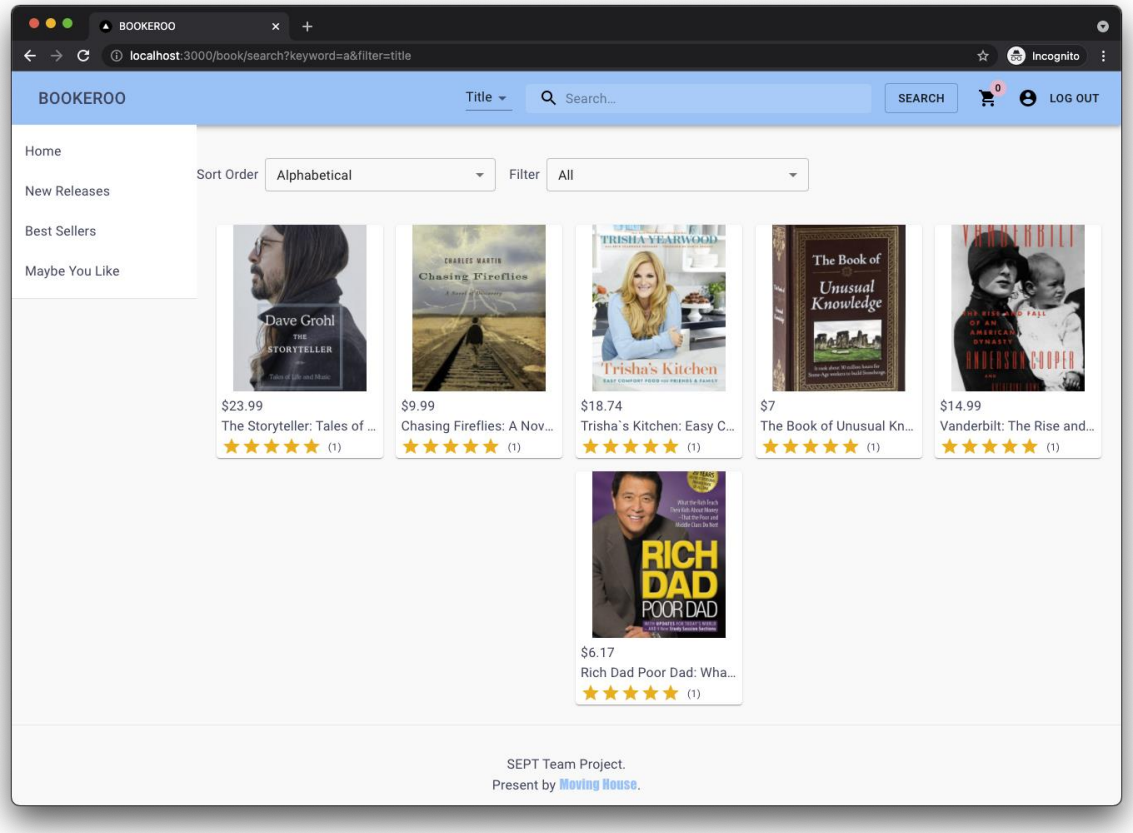
Expect:
A review dialog will pop up
Screenshot:


Given:
A user is login, and the review dialog is open
When:
The user adds the rating to the seller and click save
Expect:
The review will be updated, and back end will be updated
Screenshot:



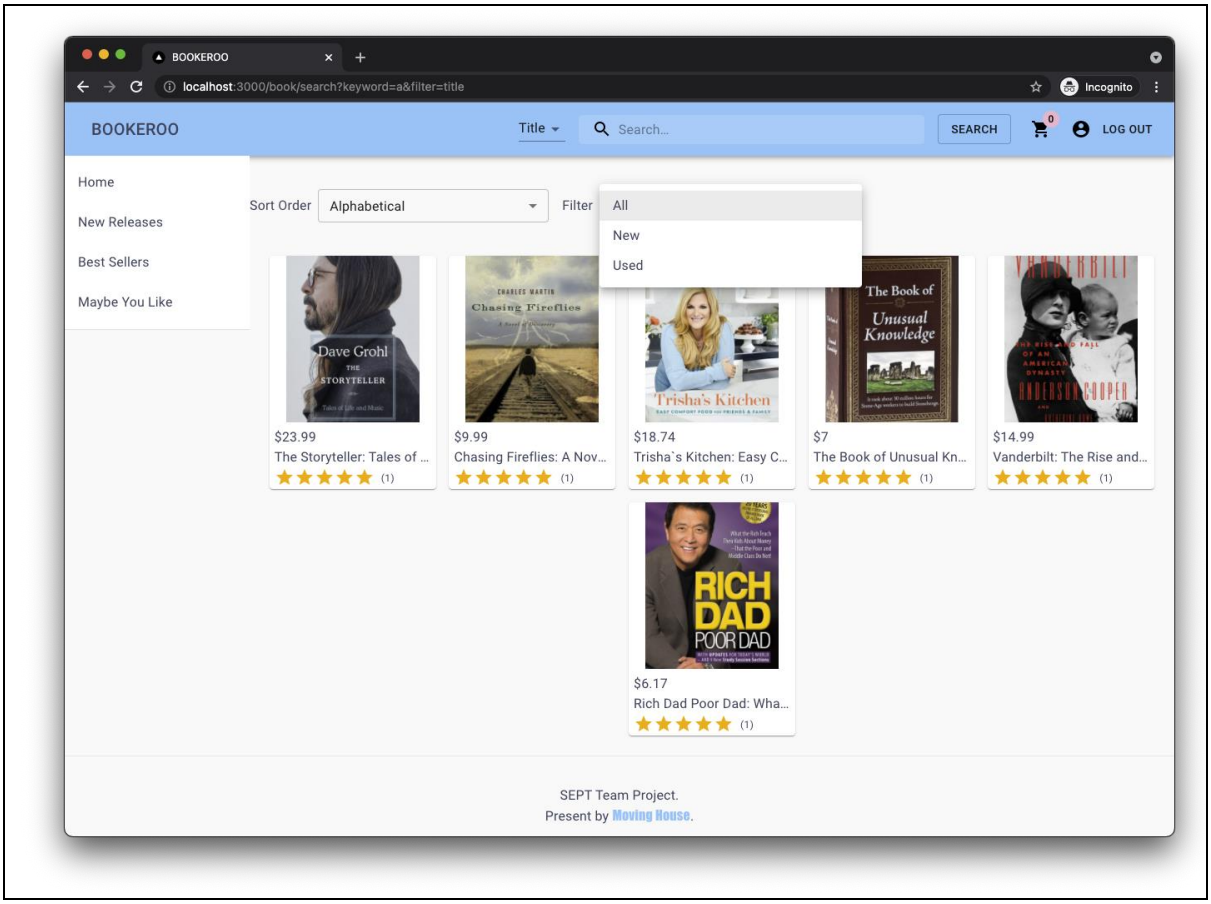
As a customer, I want to be able to filter my search, so I can find the items I am interested in more easily.

Given:
A user wants to use the search function
When:
The user is on the search result page
Expect:
There is a small search bar on the top
Screenshot:



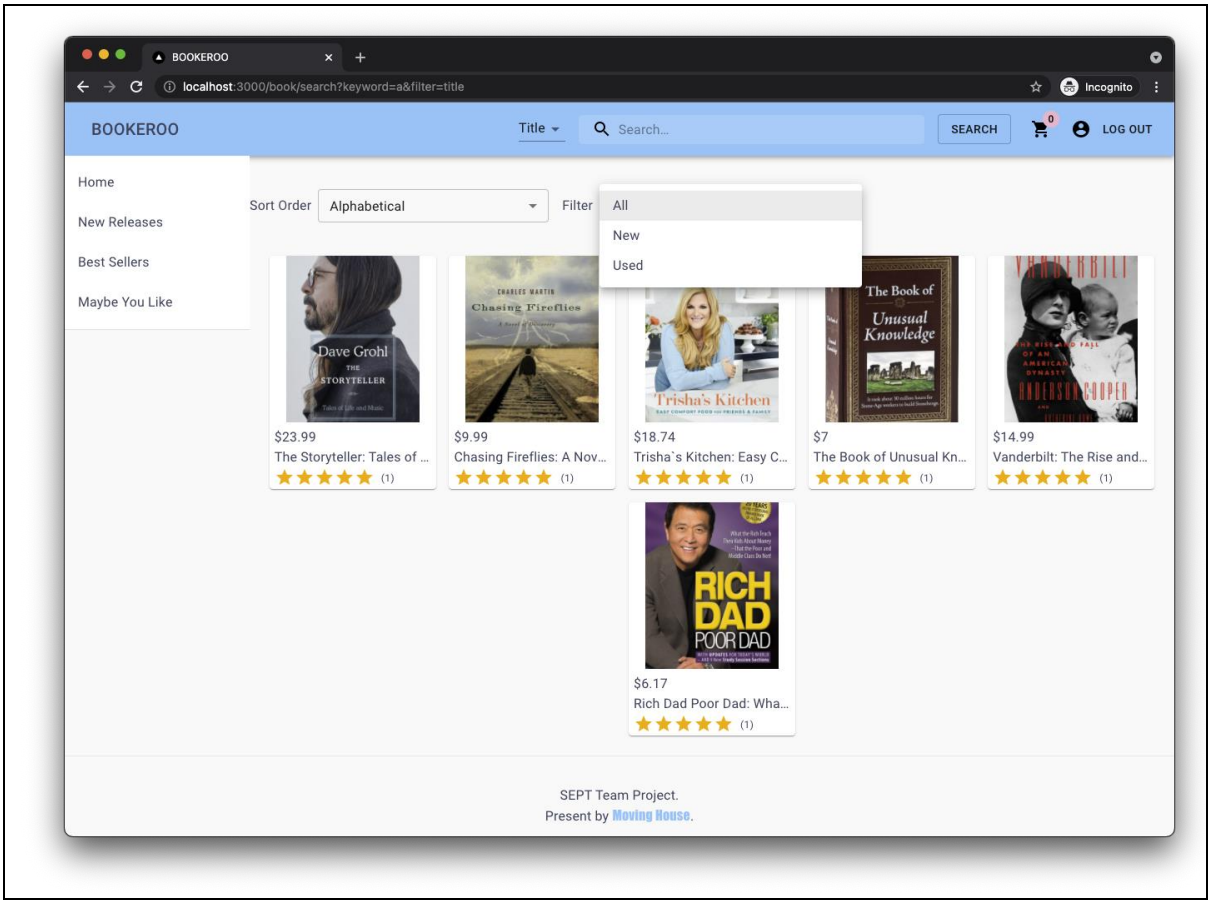
As a customer, I want to be able to filter my search results so that I can view only used books for sale.

Given:
The user is on the search result page
When:
The user clicks the filter button
Expect:
The result could be flited by USED
Screenshot:



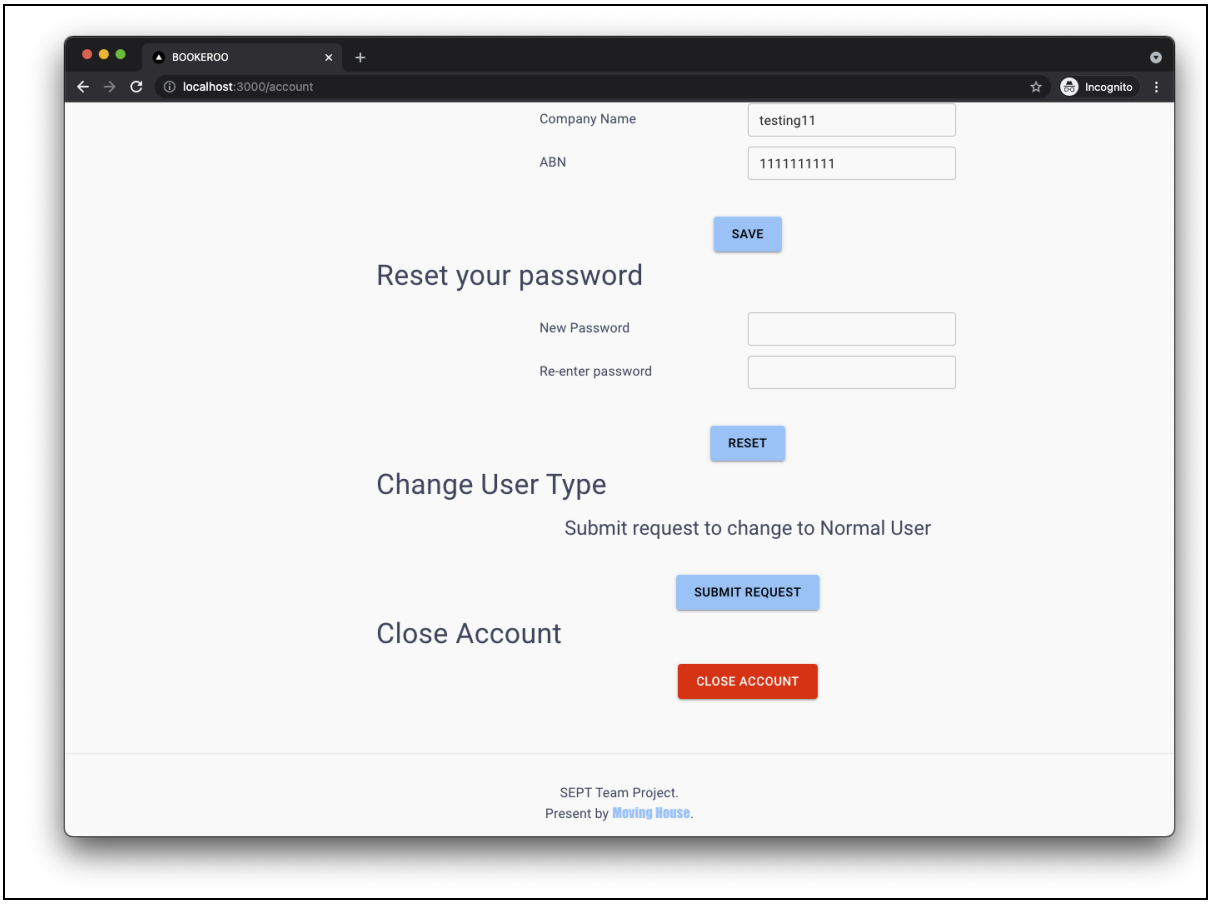
As a customer, I want to be able to filter my search results so that I can view only new books for sale.

Given:
The user is on the search result page
When:
The user clicks the filter button
Expect:
The result could be flited by NEW
Screenshot:



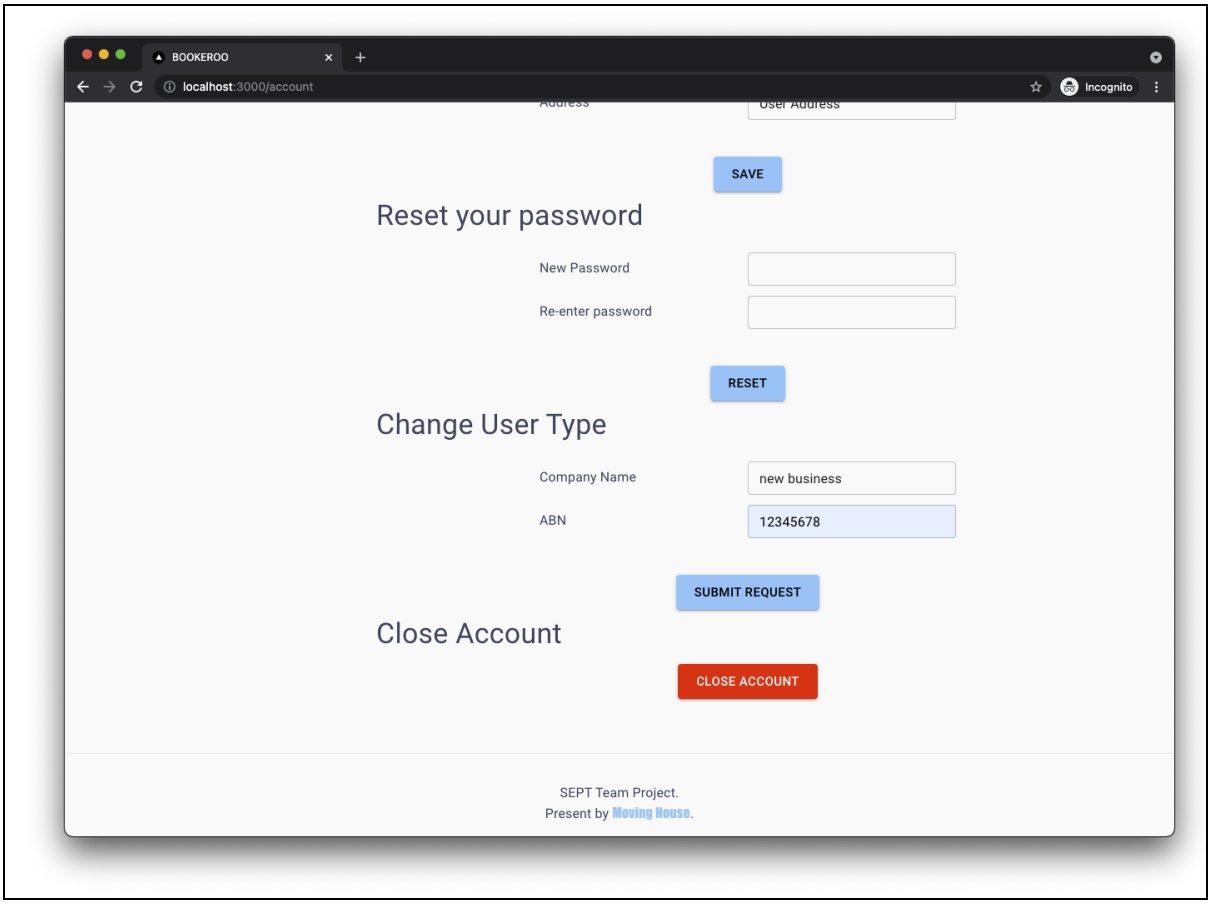
As a business user, I want to change my account type to a regular user account, so that I can remove my business' details from my account.

Given:
A business user is logged in, and on the account page
When:
The user clicks the button for change account type
Expect:
A new request will be created for admin' approval
Screenshot:



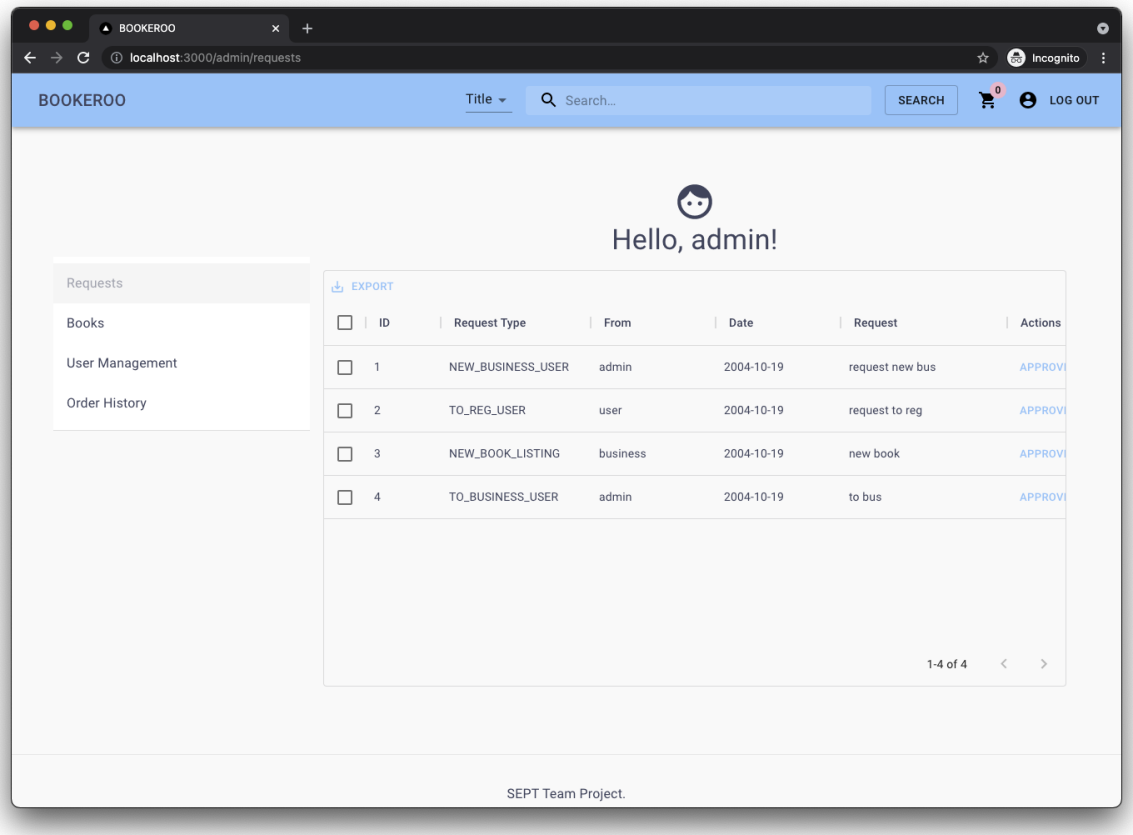
As a customer, I want to change my account type to a business account, so that I can add my business' name and ABN to my account details.

Given:
A normal user is logged in, and on the account page
When:
The user adds the new company name and abn, and clicks the button for change account type
Expect:
A new request will be created for admin' approval
Screenshot:



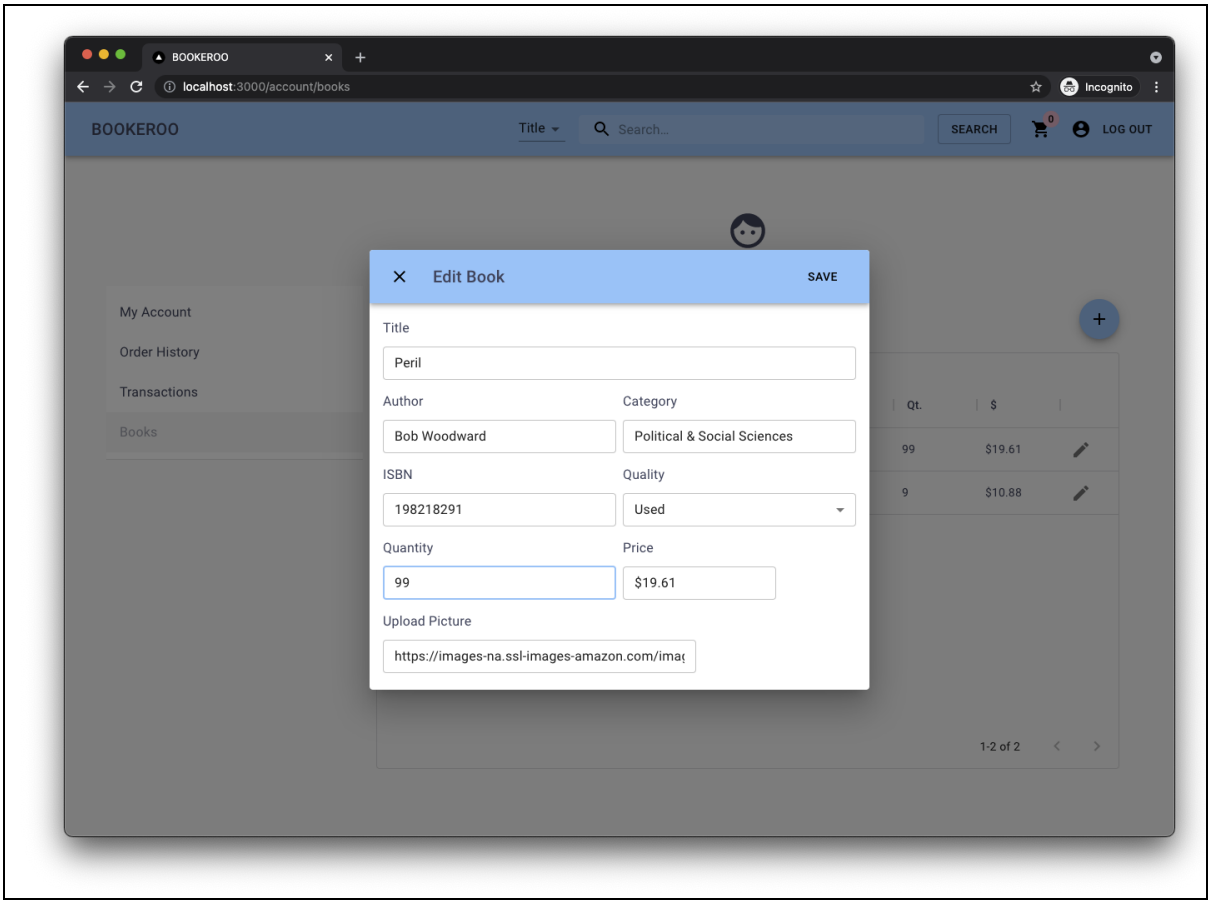
As an admin, I want to sort all pending registrations from business users so I can view the oldest one first.

Given:
The admin is logged in, and on the request page
When:
Admin clicks the arrow near the date column
Expect:
The table of request will be sorted by date
Screenshot:



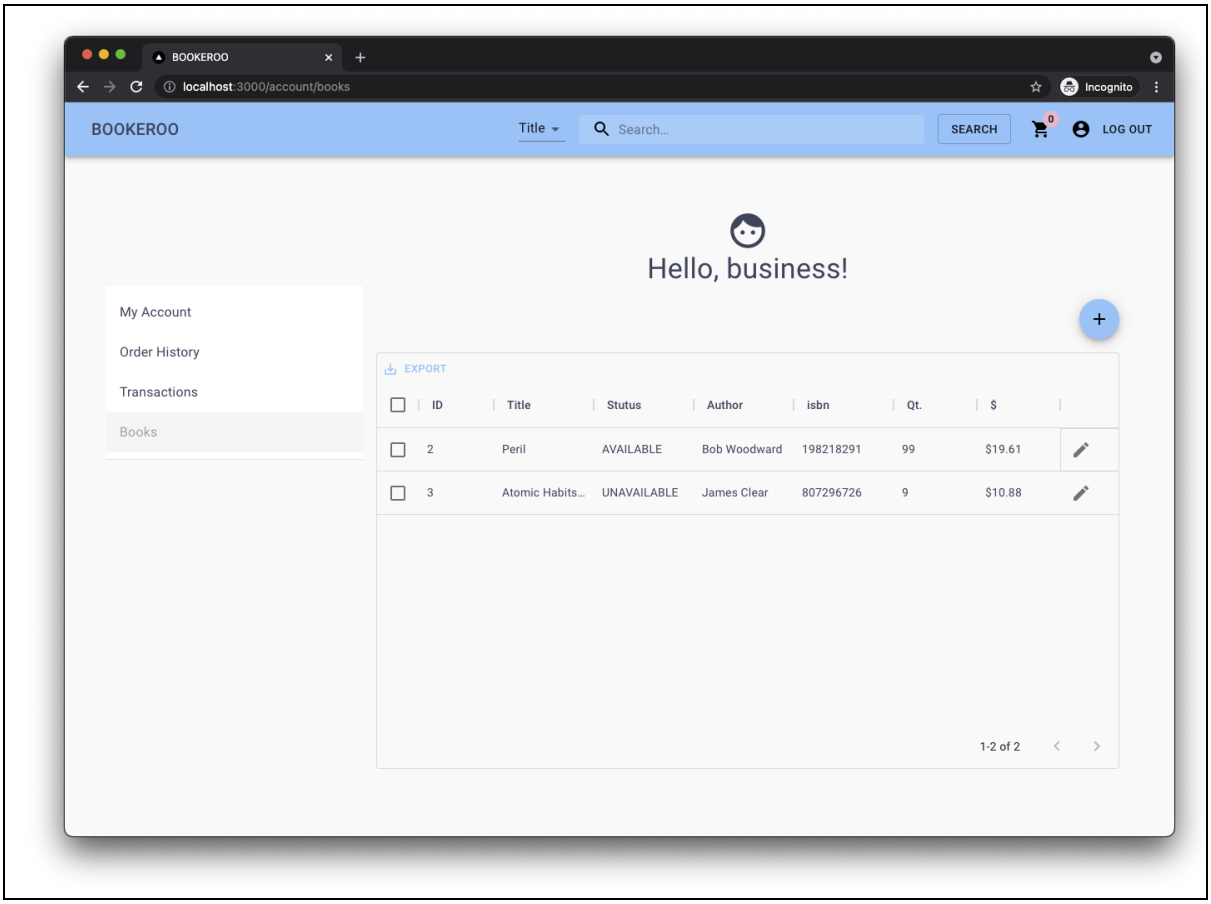
As a business user, I want to adjust the amount of stock for an item, so that item stock can be increased or decreased.

Given:
A business user is logged in and on the book management page
When:
The user clicks the edit button for a book
Expect:
A editing dialog with a changeable quantity filed will be shown
Screenshot:



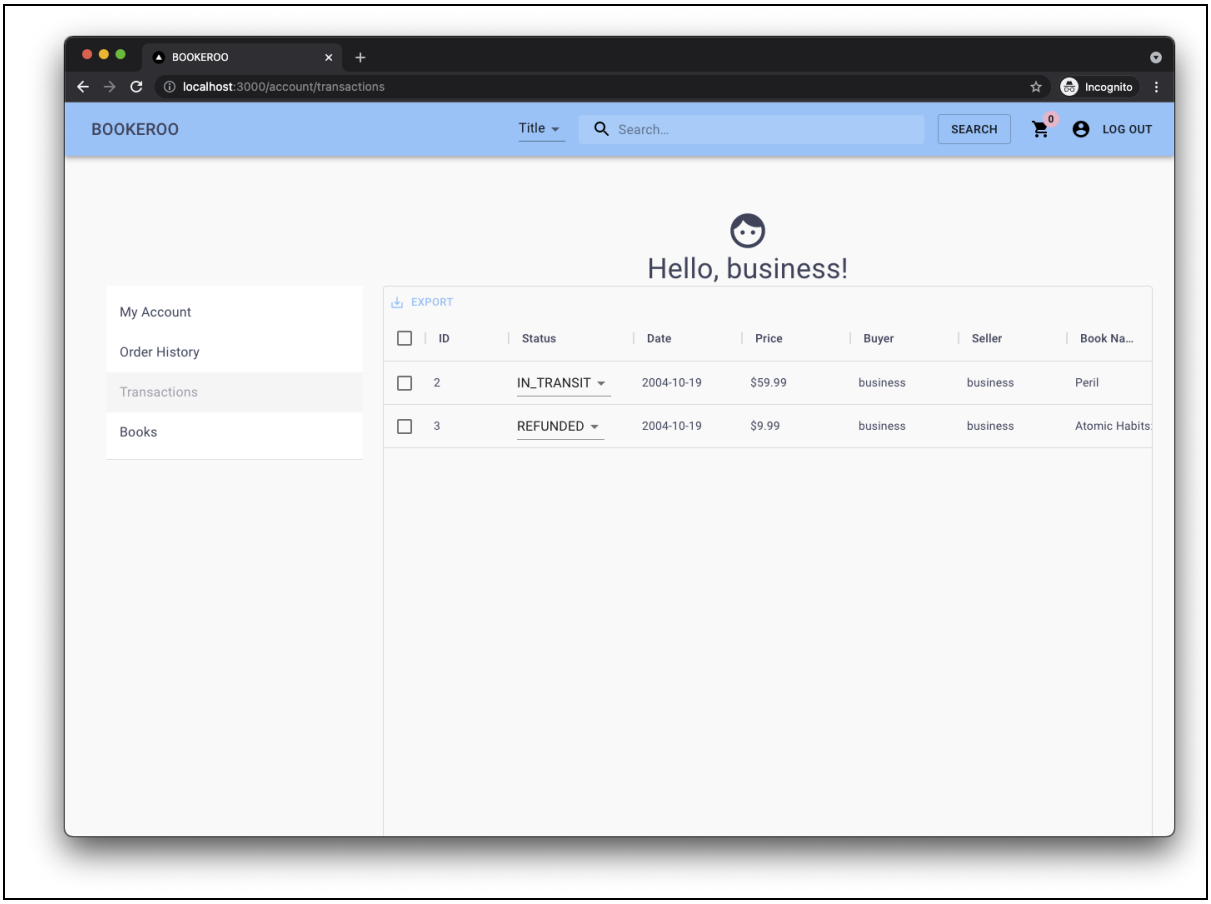
As a business user, I want to know how much inventory I have left on an item, so I can decide if I need to print more books.

Given:
A business user is logged in and on the book management page
When:
The user checks the book table
Expect:
There is a quantity column for each book
Screenshot:



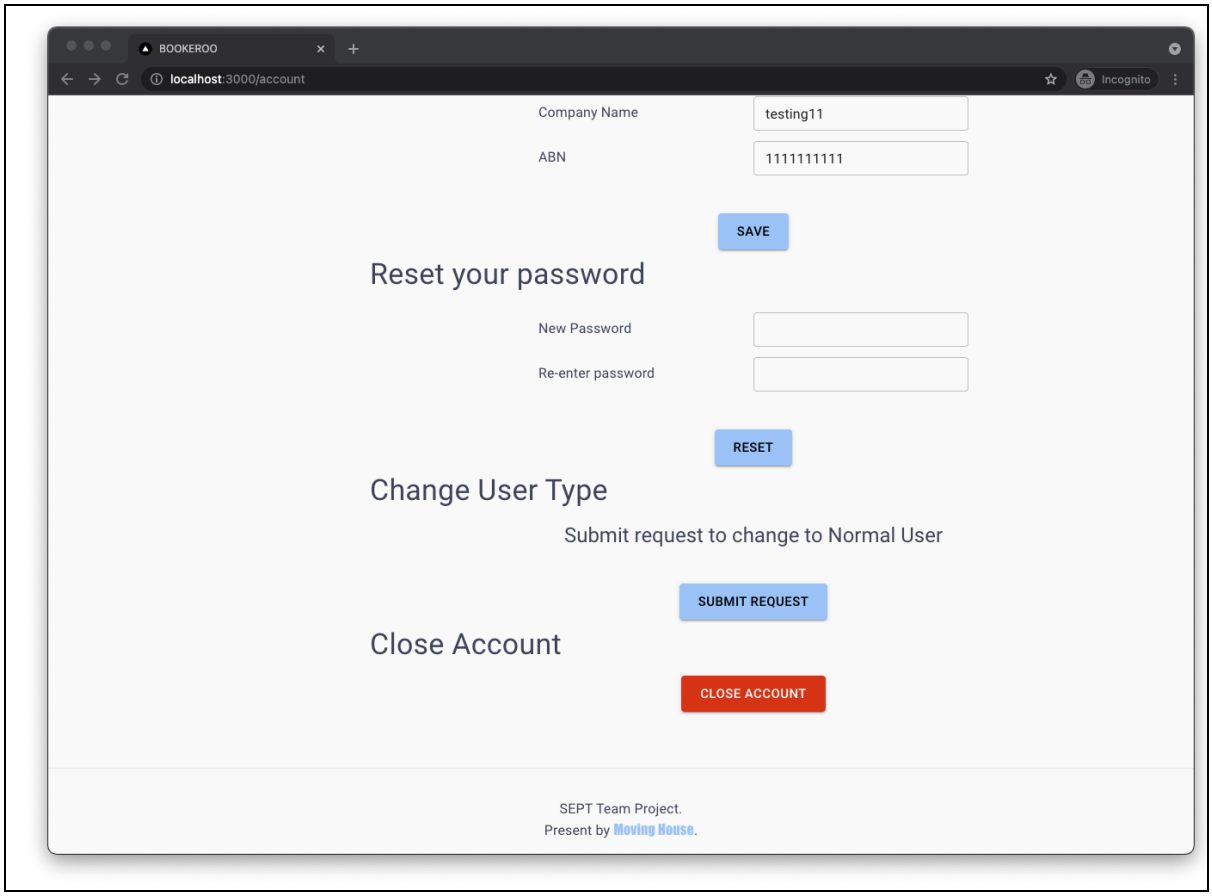
As a business user, I want to see the total amount of transactions, so I know if I am selling a large number of books.

Given:
A business user is logged in and on the transaction management page
When:
The user checks the transaction table
Expect:
There is a price column for each transaction occured
Screenshot:



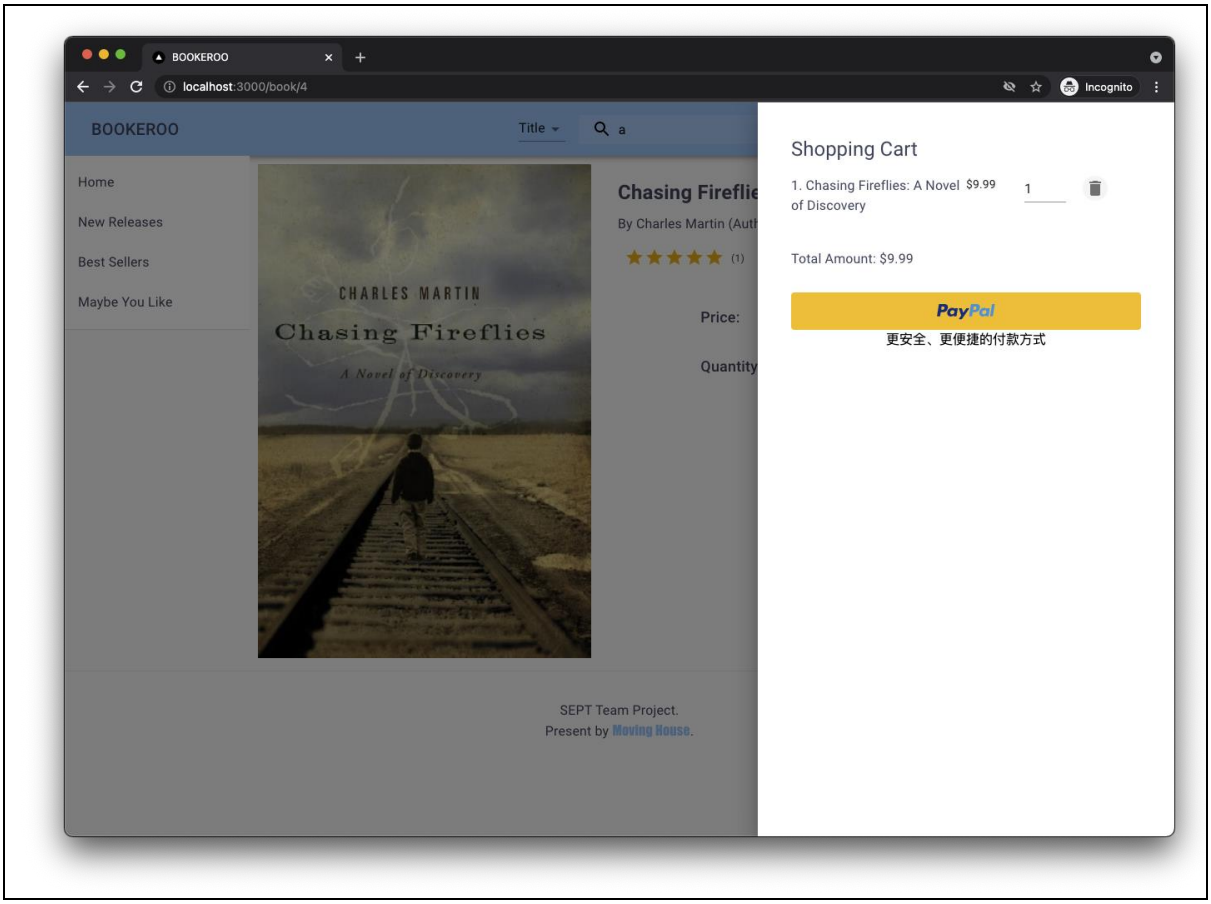
As a customer, I want to be able to shut down my account, so that I cannot worry about my data being stolen

Given:
A user is logged in and on the account page
When:
The user clicks the close account button
Expect:
The account will be terminated.
Screenshot:



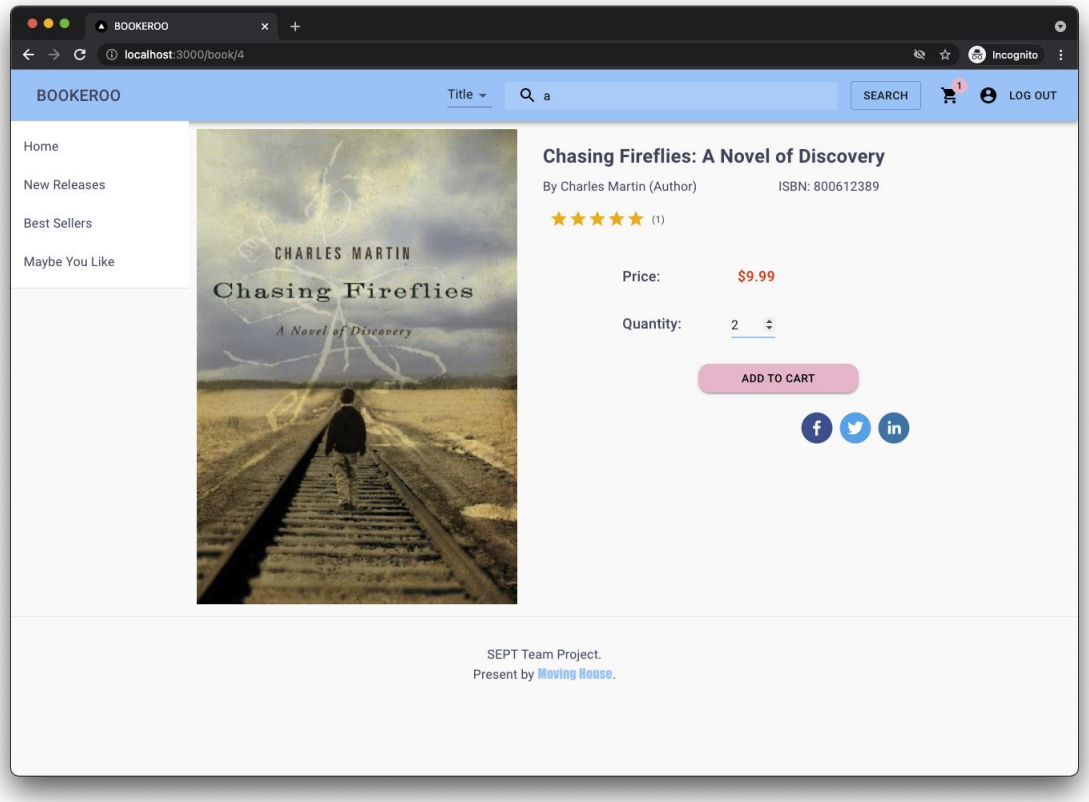
As a customer, I want to remove items from my shopping cart, so I can avoid buying items I don't want.

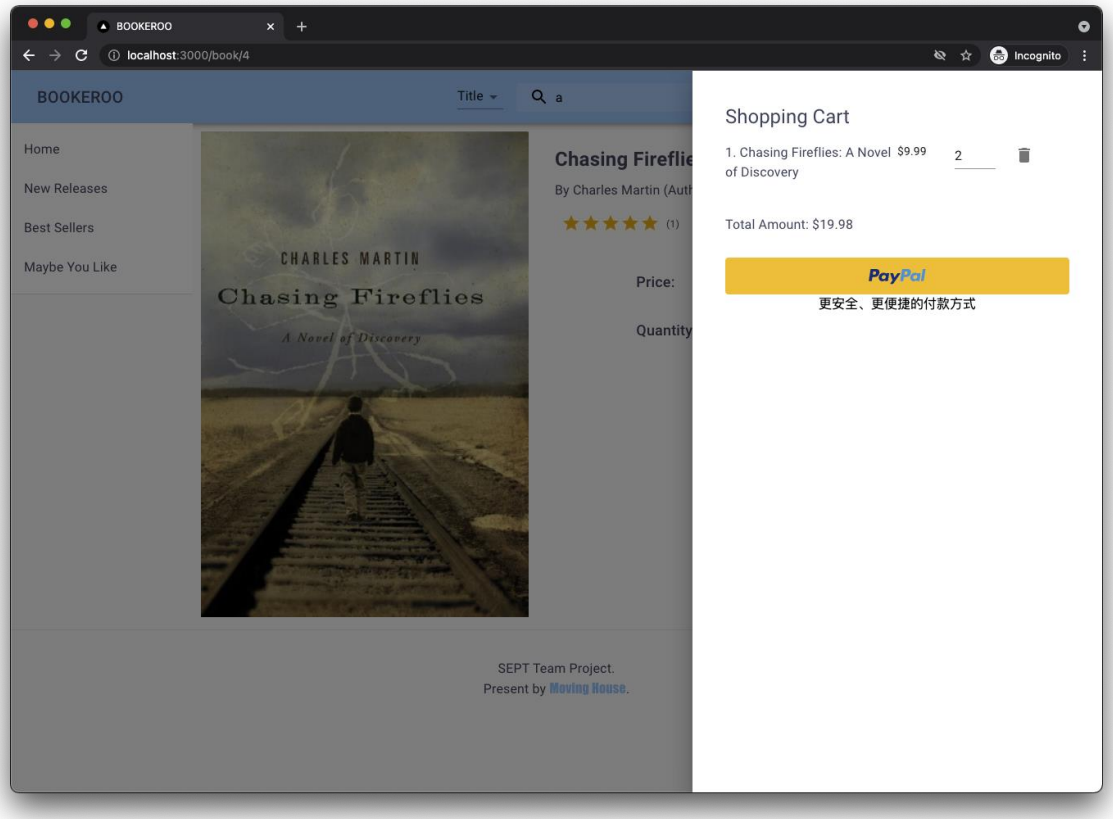
Given:
There is a book in the shopping cart
When:
The user clicks the delete button near the book item
Expect:
The item will be removed from the shopping cart
Screenshot:



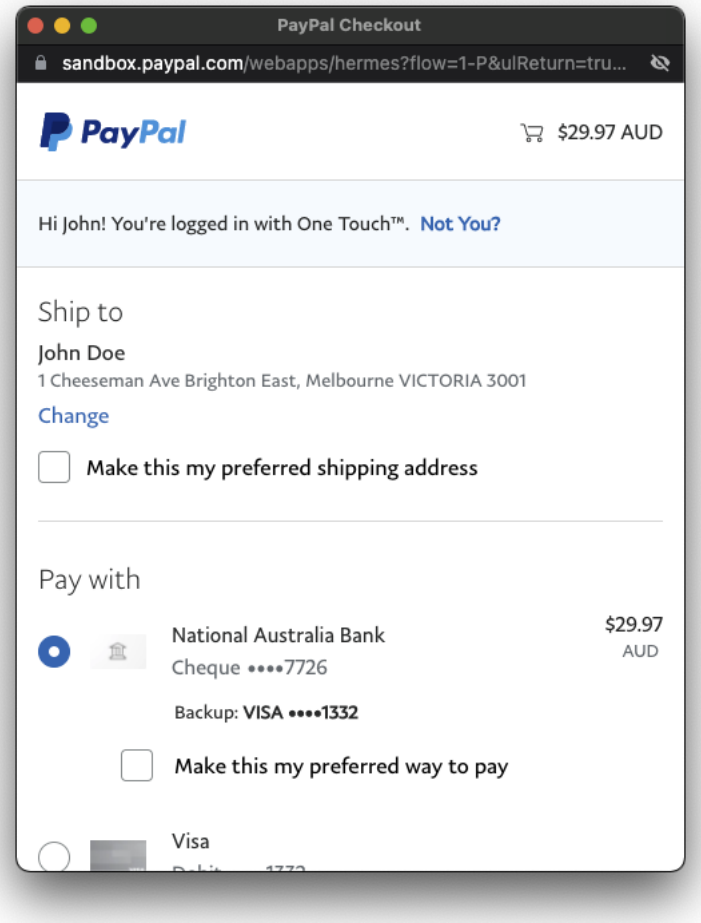
As a customer, I want to put items to my shopping cart, so I can buy them later

Given:
A user is on a book page
When:
The user clicks the add to cart button with some number of quantity
Expect:
The item will be added to the shopping cart
Screenshot:

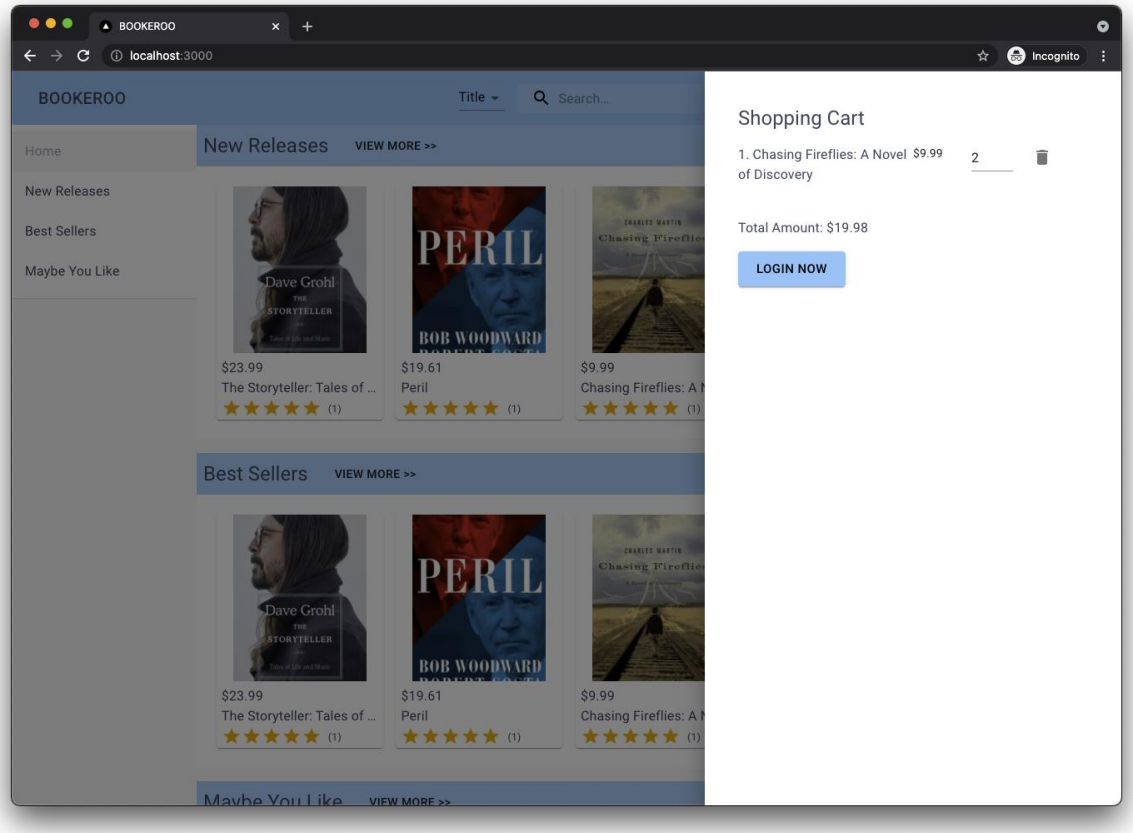

BOOKEROO
localhost:3000/book/4
Incognito
BOOKEROO
Title
a
SEARCH
LOG OUT
Home
New Releases
Best Sellers
Maybe You Like
CHARLES MARTIN
Chasing Fireflies
A Novel of Discovery
Chasing Fireflies: A Novel of Discovery
By Charles Martin (Author)
ISBN: 800612389
(1)
Price:
$9.99
Quantity:
2
ADD TO CART
SEPT Team Project.
Present by Moving House.


BOOKEROO
localhost:3000/book/4
Incognito
BOOKEROO
Title
a
Home
New Releases
Best Sellers
Maybe You Like
CHARLES MARTIN
Chasing Fireflies
A Novel of Discovery
Price:
Quantity
SEPT Team Project.
Present by Moving House.
Shopping Cart
1. Chasing Fireflies: A Novel of Discovery
$9.99
2
Total Amount: $19.98
PayPal
更安全、更便捷的付款方式

As a customer, I want to check out my shopping cart with PayPal, so I can finish my purchase

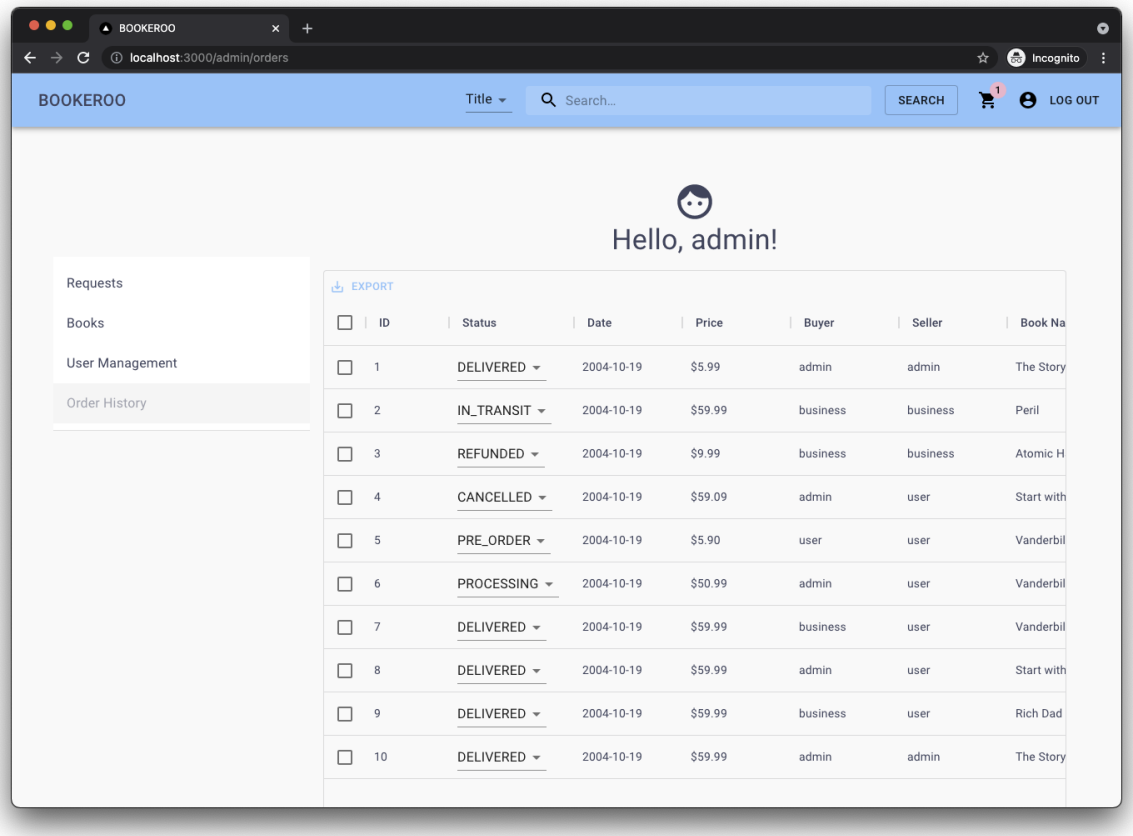
Given:
A user is logged in and on the shopping cart page
When:
The user clicks the PayPal button
Expect:
The user will take to PayPal to finish the purchase
Screenshot:


Given:
A user is not logged in
When:

The user opens the shopping cart page
Expect:
The user will see the login button
Screenshot:


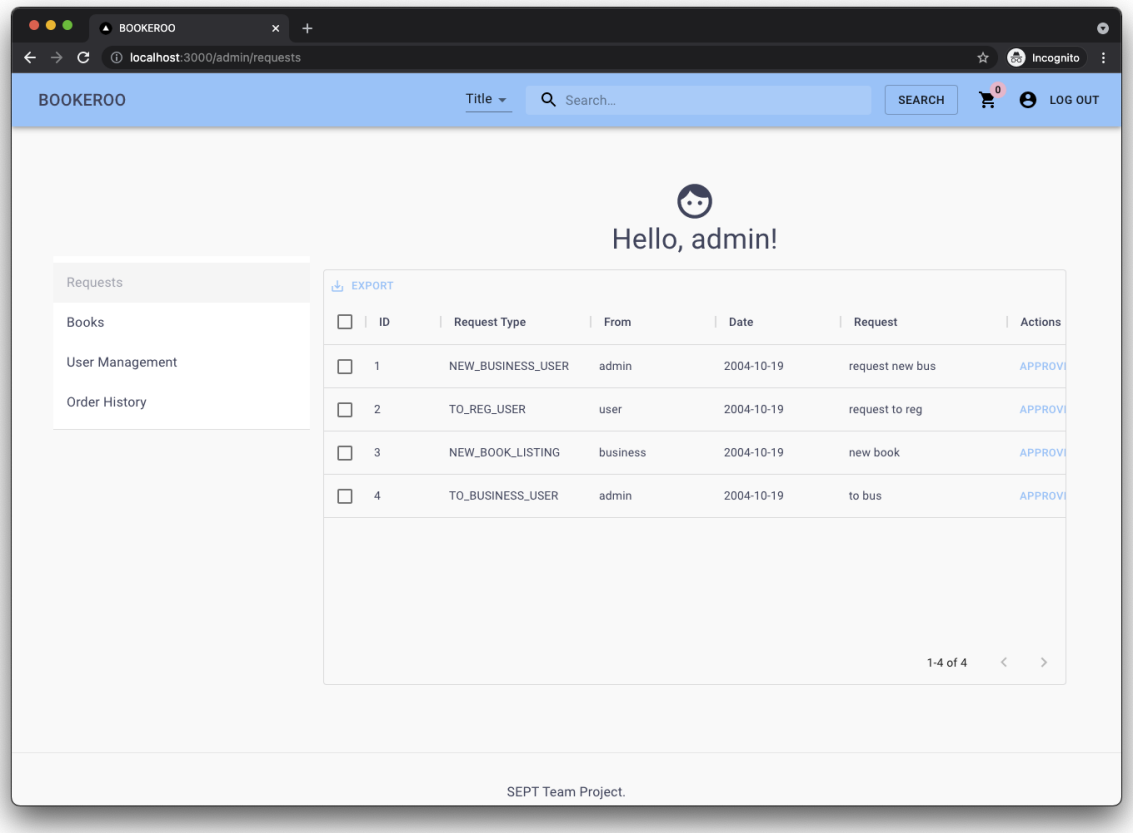
As an Admin, I want to see the transaction history of an item, so I know if the right amount of money is being transferred

Given:
The admin is logged in
When:
The admin checks the order history page
Expect:
Admin will be able to see all the transactions in a table
Screenshot:



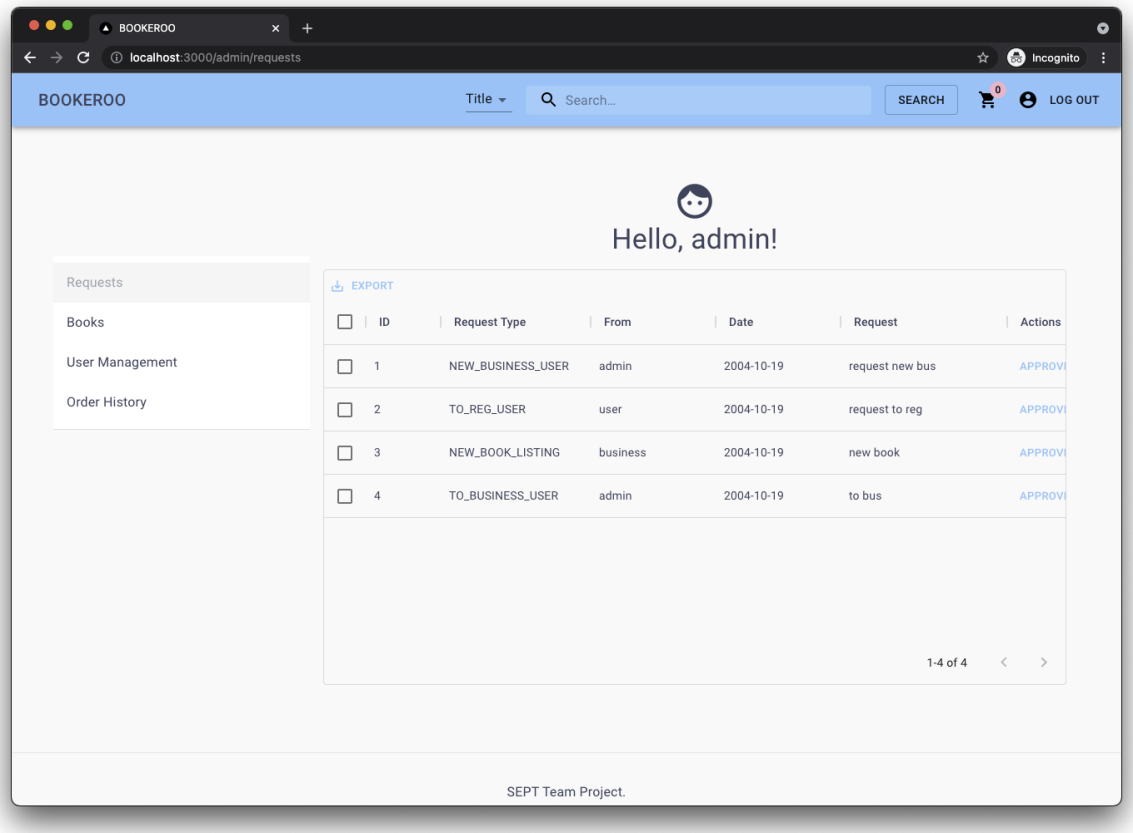
As an admin, I want to view the refund requests from users, so that I can manage all the refund requests.

Given:
The admin is logged in
When:
The admin checks the request history page
Expect:
The refund request will be shown if any
Screenshot:



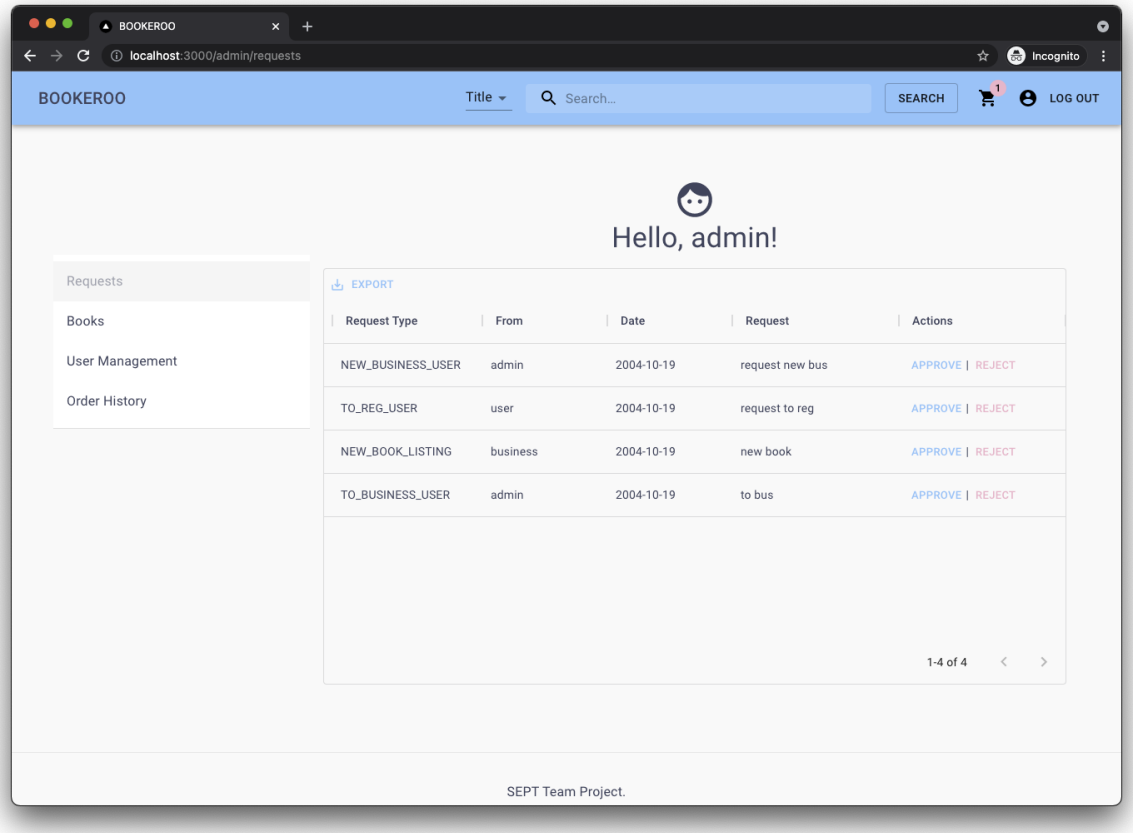
As an admin, I want to approve refund requests from users, so that I can allow the refund to the processed

Given:
The admin is logged in and on the request page
When:
The admin checks approve button for a refund request
Expect:
The refund request will be approved
Screenshot:



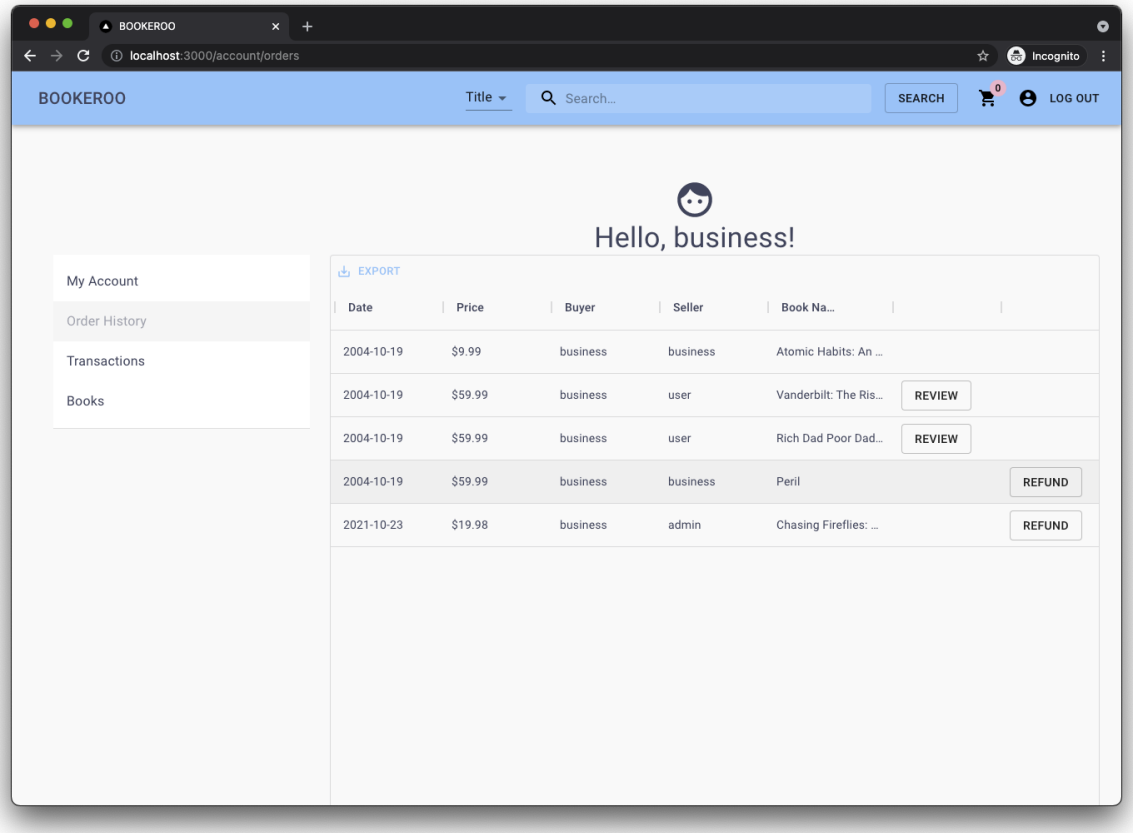
As an admin, I want to reject refund requests from users, so that I can deny the refund requests

Given:
The admin is logged in and on the request page
When:
The admin checks reject button for a refund request
Expect:
The refund request will be rejected
Screenshot:



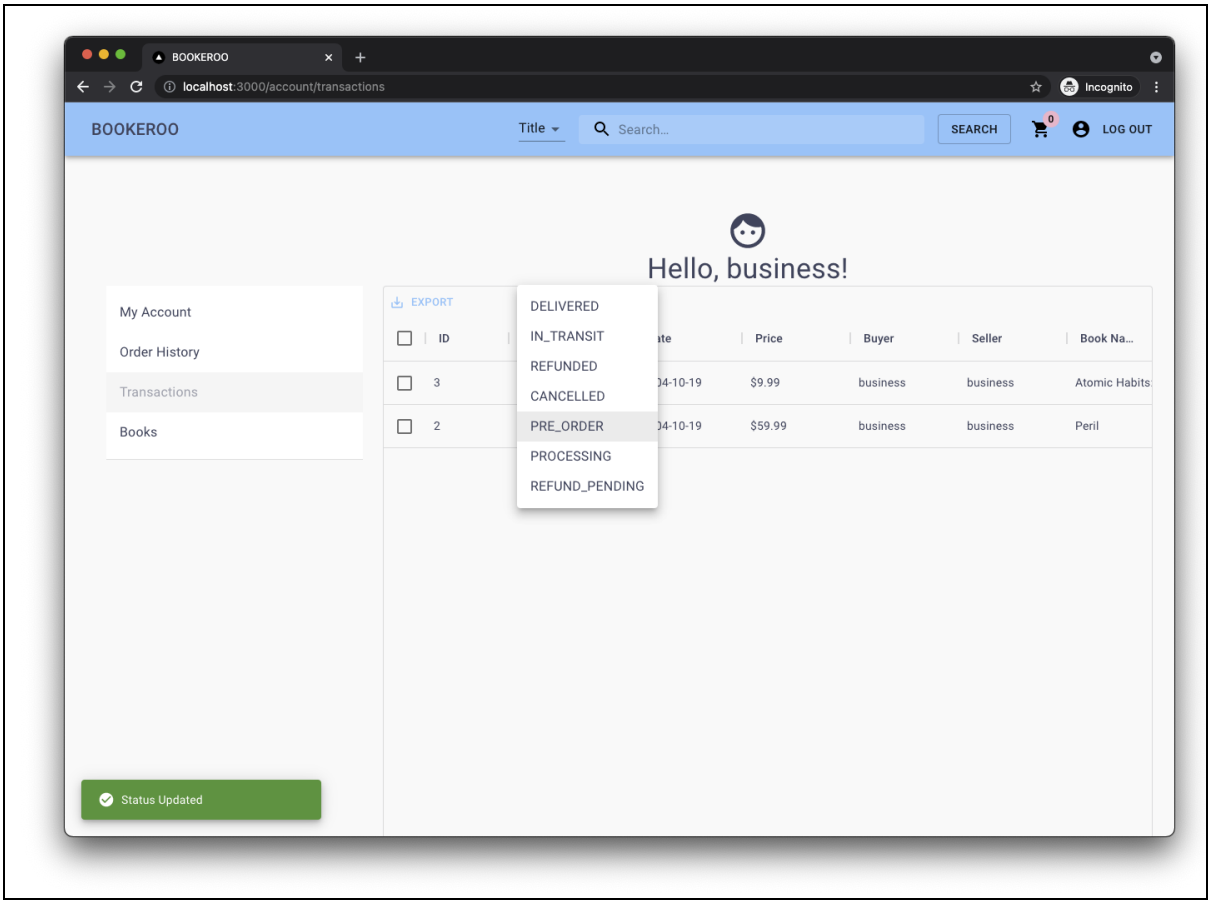
As a customer, I want to cancel my order within 2 hours of purchasing, so I can get a full refund.

Given:
The user is logged in and on the order page
When:
User clicks the refund button for order within 2 hours after purchase
Expect:
The transaction will be refunded
Screenshot:



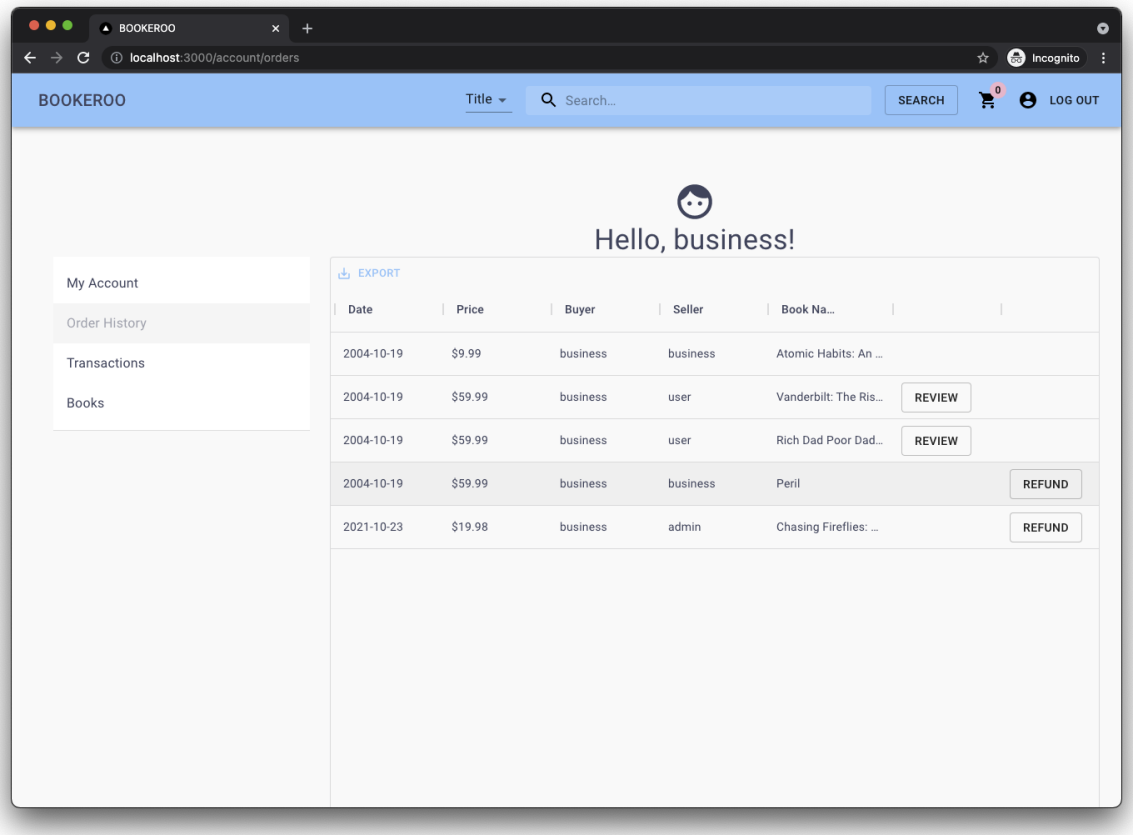
As a customer, I want to book items in advance, so I can get the book later when it is available.

Given:
Customer purchased a book which is not available
When:
The seller clicks the status on that order and change the status to pre-order
Expect:
The order will be a pre- order
Screenshot:



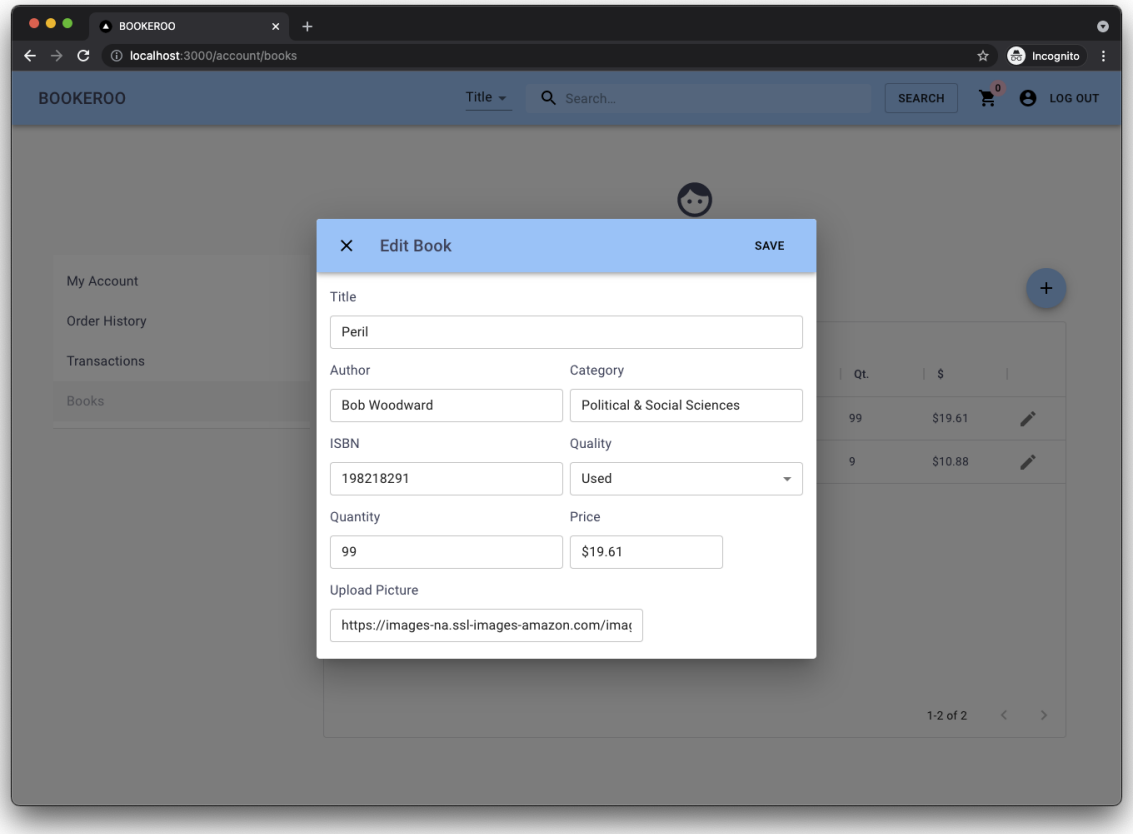
As a customer, I want to cancel my order after 2 hours of purchasing, so I can get a full refund.

Given:
The user is logged in and on the order page
When:
User clicks the refund button for order after 2 hours after purchase
Expect:
A refund request will be created for approval.
Screenshot:

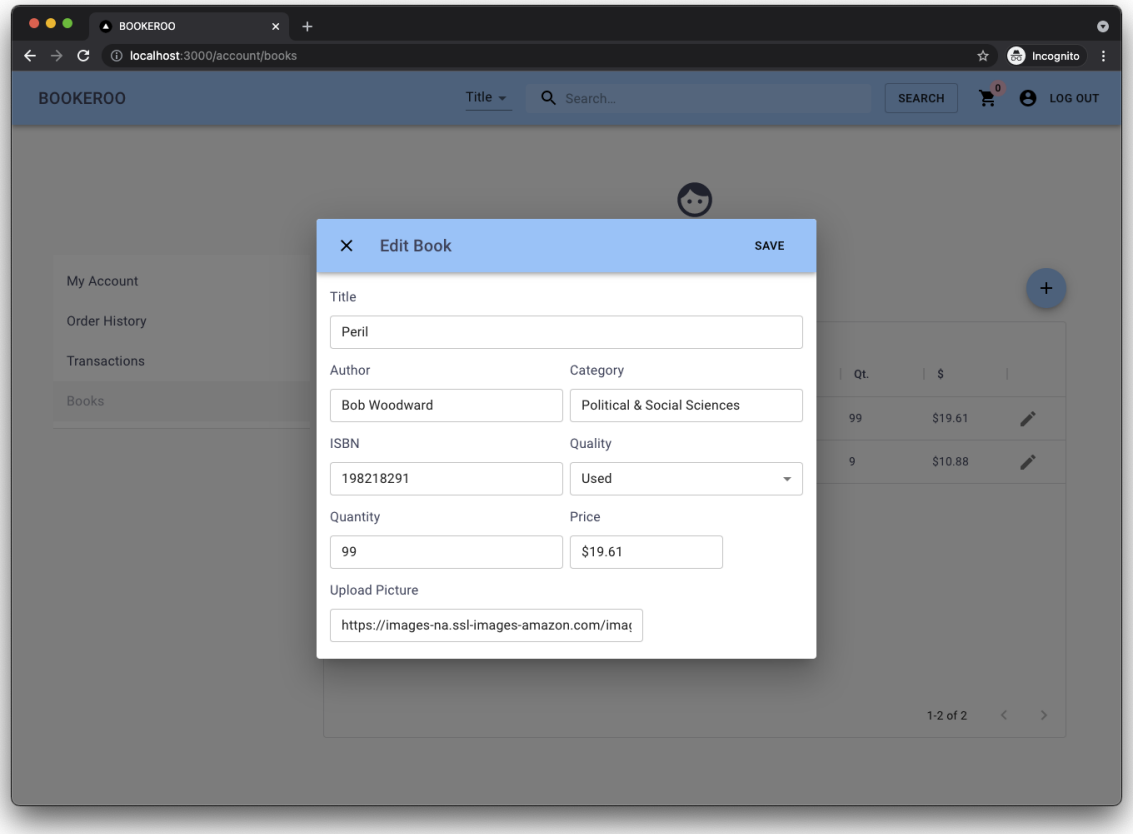


As a business user, I want to be able to upload images to my books, so that I can display previews for each book item.

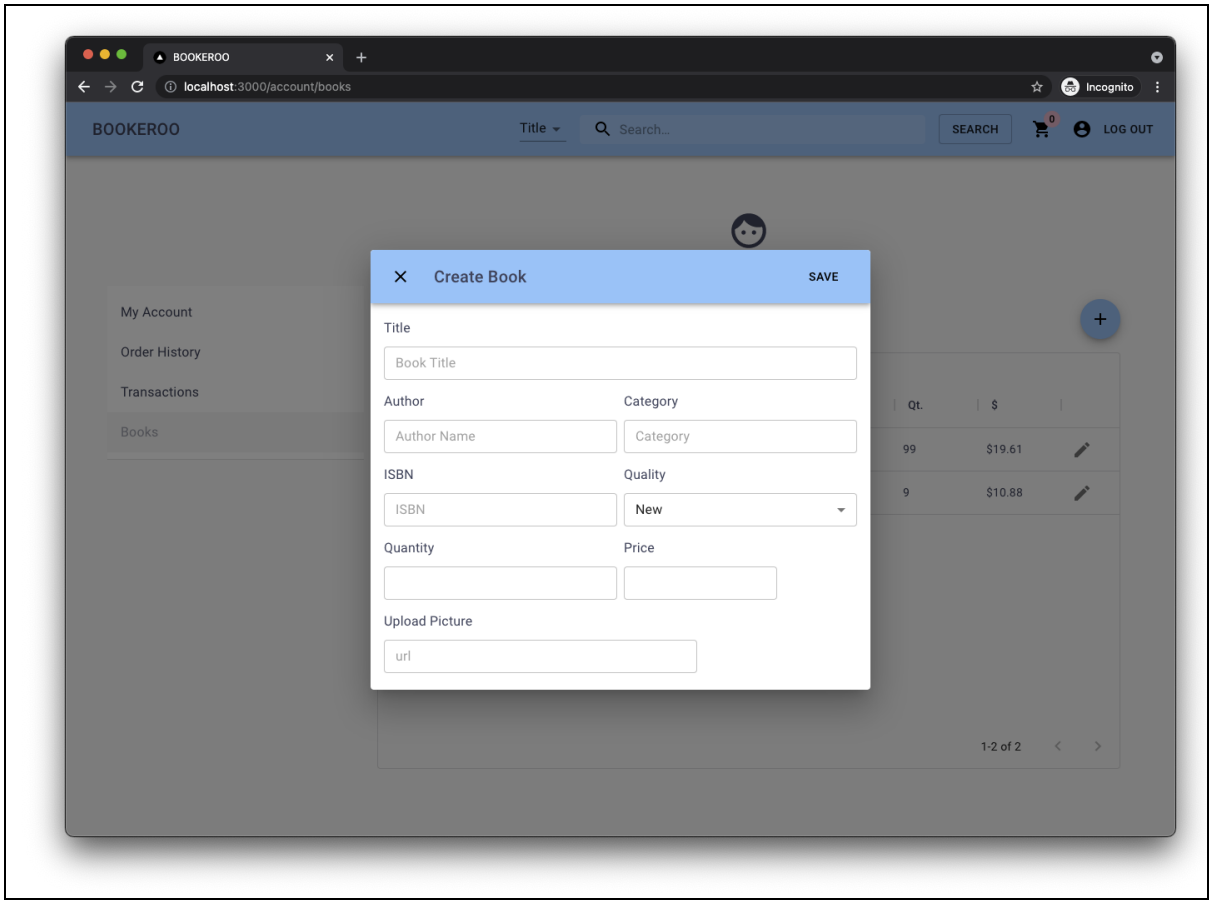
Given:
The business user is logged in and on the book management page
When:
User clicks the edit button for a book
Expect:
An edit dialog with an image URL field will pop up
Screenshot:



Given:
The business user is logged in and on the book management page
When:
User clicks the edit button for a book
Expect:
An edit dialog with an image URL field will pop up
Screenshot:



Given:
The business user is logged in and on the book management page
When:
User clicks the plus button to add a new book to sell
Expect:
An create dialog with an image URL field will pop up
Screenshot:


BOOKEROO
localhost:3000/account/books
Incognito
BOOKEROO
Title
Search...
SEARCH
LOG OUT
My Account
Order History
Transactions
Books
Create Book
SAVE
Title
Book Title
Author
Author Name
Category
Category
ISBN
ISBN
Quality
New
Quantity
Price
Upload Picture
url
Qt.
$
99
$19.61
9
$10.88
1-2 of 2