

**Bookeroo**

**Documentation**

**Folie á deux**

## Table of Contents

<b>Introduction .....</b>	<b>3</b>
<b>Design and architecture specifications .....</b>	<b>3</b>
System Architecture.....	3
<b>Acceptance Criteria.....</b>	<b>4</b>
Sprint 1 Issues:.....	4
<b>Test Report.....</b>	<b>7</b>
Front-End Testing.....	7
Back-End Testing.....	7
Books Microservice Testing .....	8
Login Microservice Testing .....	11
Book Page Testing.....	13
<b>Burndown Charts.....</b>	<b>16</b>
Sprint 1.....	16
Sprint 2.....	16

## Introduction

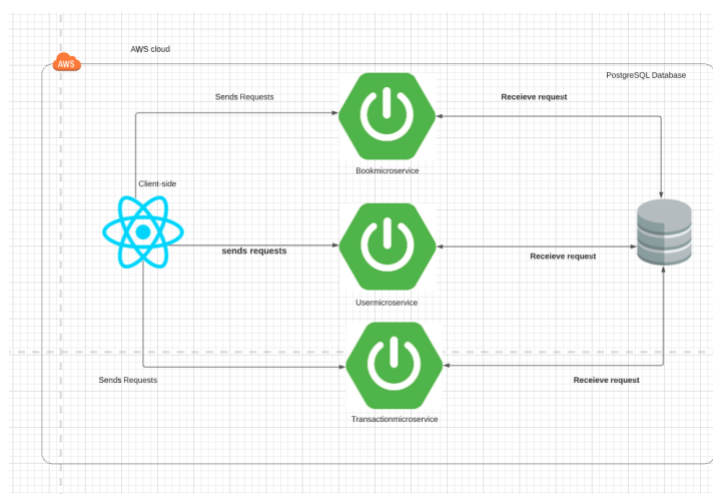
This document serves the purpose of introducing the system's design, architecture, and testing of all related components to show the application's capabilities and performance.

This application is build using Java and JavaScript as primary development stack and will soon be deployed unto AWS using CircleCi as the CI/CD tool. The application is communicating data with an Amazon RDS system using PostgreSQL as the database engine.

## Design and architecture specifications

### System Architecture

The application is build using microservices architecture with each services is a standalone application that communicates together as a single entity. The back-end is built using Java SpringBoot framework. The backend is build using 2 microservices, login and books. Each microservices contains a model or models, controller, service and a data access layer specific to the service it provides. Each microservice is connected to an RDS database on amazon that communicates data back and forth through the data access layer to invoke the APIs created in the controller through the service. Meanwhile, the frontend is a third microservice that serves an interactive interface built with React Js framework where the app uses library called Axios to invoke each microservice to communicate user's actions and data to perform several tasks such as login, register, create book listing, display books and so on. The app will soon will contain extra features such as integrating the payment portal API, PayPal, to the books microservice to support book specific transactions.



*System architecture of Bookeroo*

## Acceptance Criteria

This section will feature the acceptance criteria of each of the user stories completed in sprint 1 & 2.

### Sprint 1 Issues:

- SEPT 57 - As a registered user, I want to browse books on the website so that I can rent, buy, review, or rate books online.

Acceptance Criteria :

**Given that** the user has registered their account

**When** the user logs in the system

**Then** they are able to rent, buy, review, or rate books

- SEPT 56 -As a public user, I want to browse books on the website without logging so that I can view books without the obligation of registering.

Acceptance Criteria :

**Given that** a public user launches the website

**When** the page loads and the landing page appears

**Then** the public user is able to browse book freely

- SEPT 23 - As an Admin, I want to type up the ISBN of a book, so that I can see if the result matches the corresponding book.

Acceptance Criteria :

**Given that** an admin is logged in the application

**When** the admin types in the ISBN of a book

**Then** the book with that ISBN will appear

- SEPT 22 -As a Public User, I want to search the name of an author, so that I can see all the books they have authored.

Acceptance Criteria :

**Given that** public user is browsing the application

**When** the public user types in the author name

**Then** all the books made by that author will appear

- SEPT 21 - As a Public User, I want to search for a book that contains a particular word in the title, so that I can see a list of books containing that word in the title.

Acceptance Criteria :

**Given that** a public user is on the website

**When** a public user types in any word in the search box

**Then** all the book where the title contains that keyword will appear

- SEPT 9 - As a non-registered user I want to register an account, so I can post reviews on other registered users I have interacted with.

Acceptance Criteria :

**Given that** a public user launches the web application

**When** a public user clicks on "Register"

**And** the registration form appear for the user and enter their details

**And** the public user clicks "Submit"

**And** there details are transferred to the database

**And** the user successfully logs in with their new details

**Then** they are able to post reviews and interact with other users

- SEPT 8 - As a non-registered user I want to register an account, so I can post reviews on books I have read or purchased from shop owners.

Acceptance Criteria :

**Given that** a public user launches the web application

**When** a public user clicks on "Register"

**And** the registration form appear for the user and enter their details

**And** the public user clicks "Submit"

**And** there details are transferred to the database

**And** the user successfully logs in with their new details

**Then** they are able to post reviews on books they have purchased

- SEPT 7 - As a non-registered user I want to register an account, so I can sell my preowned books to other registered public users/customers.

Acceptance Criteria :

**Given that** a public user launches the web application

**When** a public user clicks on "Register"

**And** the registration form appear for the user and enter their details

**And** the public user clicks "Submit"

**And** there details are transferred to the database

**And** the user successfully logs in with their new details

**Then** they are able to post books for sale to other users

- SEPT 6 - As a shop owner I want to register an account so I can sell my inventory of books to registered users.

Acceptance Criteria :

**Given that** a public user launches the web application

**When** a public user clicks on "Register"

**And** the registration form appear for the user and enter their details

**And** the public user clicks "Submit"

**And** there details are transferred to the database

**And** the user successfully logs in with their new details

**Then** they are able to post their books online for other users to browse and buy.

- SEPT 5 - As a non-registered customer, I want to register an account, so I can purchase books from other registered public users or bookstores.

Acceptance Criteria :

**Given that** a public user launches the web application

**When** a public user clicks on "Register"

**And** the registration form appear for the user and enter their details

**And** the public user clicks "Submit"

**And** there details are transferred to the database

**And** the user successfully logs in with their new details

**Then** they are able to purchase books online

- SEPT 3 - As a registered user, I want to log in, so I can purchase or share books.

Acceptance Criteria :

**Given that** a user is already registered

**When** a user logs in the system

**Then** they are able to purchase books

## Test Report

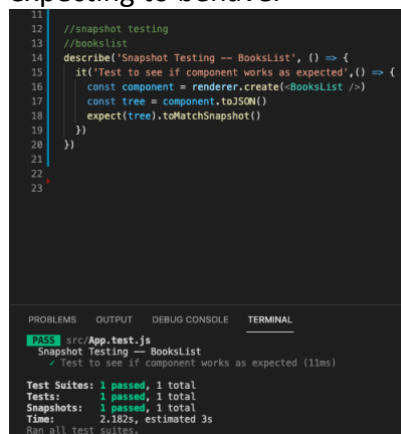
This section will include several tests performed over the SprintBoot and React application to make sure every aspect of the application is working well.

### Front-End Testing

For React, a JavaScript testing framework called JEST will be used. More information can be found in <https://jestjs.io/docs/tutorial-react>. This section will feature testing some components using different types of tests.

#### Test1 – Snapshot Testing for BooksList

Snapshot testing is used to determine if the component will behave the way we are expecting to behave.



```
11 //snapshot testing
12 //booklist
13
14 describe('Snapshot Testing -- BooksList', () => {
15   it('Test to see if component works as expected', () => {
16     const component = renderer.create(<BooksList />)
17     const tree = component.toJSON()
18     expect(tree).toMatchSnapshot()
19   })
20 })
21
22
23
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PASS src/App.test.js
Snapshot Testing -- BooksList
  Test to see if component works as expected (11ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:  1 passed, 1 total
Time:        2.182s, estimated 3s
Run all test suites.
```

### Back-End Testing

Each microservice will be tested using Junit tests executed to test several functionalities of the microservice.

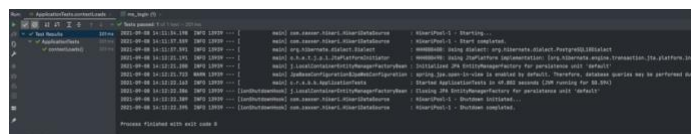
## Books Microservice Testing

### Test 1 - Acceptance Testing for books controller using Junit test

The purpose of this test is to check if the controller is working as expected while connected to the database.

```
/**
 * Test 1
 */
@DisplayName("Test to check if the controller is not null and is working")
@Test
void contextLoads() throws Exception {
    assertThat(booksController).isNotNull();
}
```

*Junit test to testing the controller*



*Result of Junit Test*

### Test 2

#### Test 2.1 Acceptance testing for books controller using ARC(Advanced REST Client) & Pg-admin(PostgreSQL app)

The purpose of this testing is to check whether the controllers build on SpringBoot is effectively and accurately communicating data to database. The test will involve invoking the API via ARC and depending on the response, we will be able to determine if the operation was successful or not. We will also validate our test by checking the data inside the database.

```
@RestController
@RequestMapping("/books")
// @CrossOrigin(origins="http://localhost:3886")
@CrossOrigin("*")
public class BooksController {

    @Autowired
    private BookService bookService;

    @GetMapping
    public ResponseEntity<Collection<Book>> findAll() {
        return new ResponseEntity<>(bookService.findAllBooks(), HttpStatus.OK);
    }

    @GetMapping(path =("/{ISBN}")
    public ResponseEntity<Book> findById(@PathVariable("ISBN") String ISBN) {
        return new ResponseEntity<>(bookService.findBookByISBN(ISBN), HttpStatus.OK);
    }

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Book> save(@RequestBody Book book) {
        return new ResponseEntity<>(bookService.addBook(book), HttpStatus.CREATED);
    }

    @PutMapping(consumes = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Book> update(@RequestBody Book book) {
        return new ResponseEntity<>(bookService.updateBook(book), HttpStatus.OK);
    }

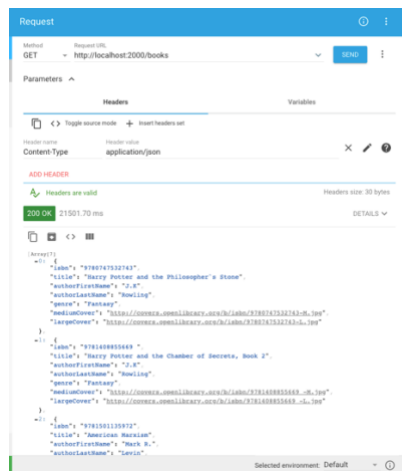
    @DeleteMapping(path =("/{ISBN}")
    @Transactional
    public ResponseEntity<?> deleteById(@PathVariable("ISBN") String ISBN) {
        bookService.deleteBook(ISBN);
        return new ResponseEntity<>(HttpStatus.OK);
    }
}
```

*Api controllers for books microservice*



## Test 2.2 GET method Test

This test will attempt to retrieve all books present in the database via GET method in the controller.



*Test shows successful request*

The screenshot shows a PostgreSQL query editor with the query `select * from books;` and its results. The results are as follows:

isbn	author_first_name	author_last_name	book_genre	title
9780747532743	J.K.	Rowling	Fantasy	Harry Potter and the Philosopher's Stone
9781406855669	J.K.	Rowling	Fantasy	Harry Potter and the Chamber of Secrets, Book 2
9781501520972	Mark R.	Levin	Fantasy	American Marxism
9780441172719	Frank	Hobert	Drama	Dune
9780451524895	George	Orwell	Thriller	1984 (Signet Classics)
9781451673819	Rap	Bradbury	Drama	Fahrenheit 451
9780316769174	J.D.	Salinger	Imagination	The Catcher in the Rye

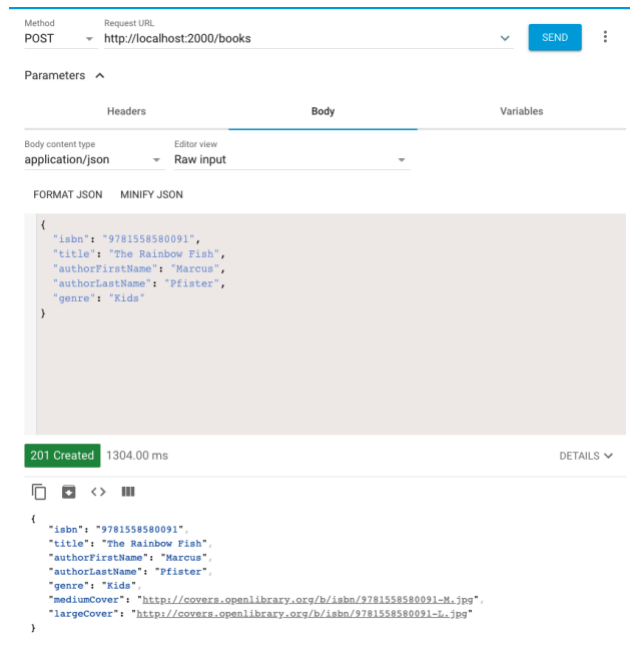
*Validate data present in database*

## Test 2.3 POST method Test

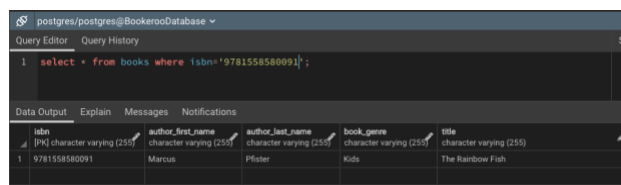
This test will attempt to post a new book in the database via GET method in the controller.



*Data that will be posted using the POST method*



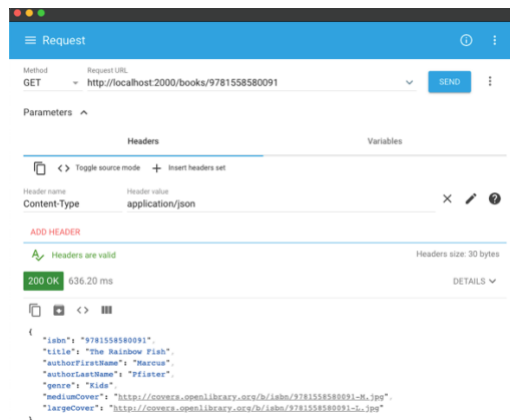
*Data successfully posted to database*



*Validate data posted in database*

## Test 2.4 GET method by ISBN Test

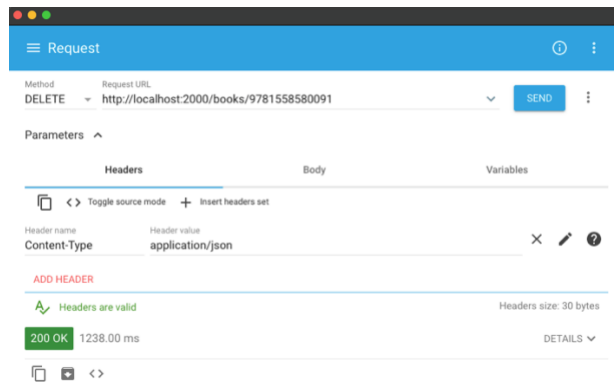
This test will attempt to get the new book by ISBN that was added to the database via POST method in the controller.



*Successful retrieval of the book by ISBN that was just posted in the previous test.*

## Test 2.5 DELETE method by ISBN Test

This test will attempt to delete the new book by ISBN that was added to the database via POST method in the controller.



Successful deletion of the book by ISBN that was just posted in previous tests.



Validation that book with that particular ISBN no longer exists in the database.

## Login Microservice Testing

Testing this microservice will be tested using the UI and ARC f. We will still test the microservice using the controller's Api but through the interface built to make sure that our application meets the user stories' acceptance criteria. ARC testing was not used for this API testing as the structure doesn't support JSON data consuming or returning.

```

import static com.mit.sept.bk_loginServices.security.SecurityConstant.TOKEN_PREFIX;

//controller api
@RestController
@CrossOrigin("*")
// path of api
@RequestMapping(path = "api/users")
public class UserController {

    @Autowired
    private MapValidationErrorService mapValidationErrorService;

    //instantiate a user object to invoke methods
    @Autowired
    private UserService userService;

    @Autowired
    private UserValidator userValidator;

    //register api
    @PostMapping("/register")
    public ResponseEntity<?> registerUser(@Valid @RequestBody User user, BindingResult result){
        // Validate passwords match
        userValidator.validate(user, result);

        ResponseEntity<?> errorMap = mapValidationErrorService.MapValidationService(result);
        if(errorMap != null)return errorMap;

        User newUser = userService.saveUser(user);

        return new ResponseEntity<User>(newUser, HttpStatus.CREATED);
    }

    @GetMapping("/register")
    public ResponseEntity<User> register() {
        return new ResponseEntity(HttpStatus.OK);
    }
}

```

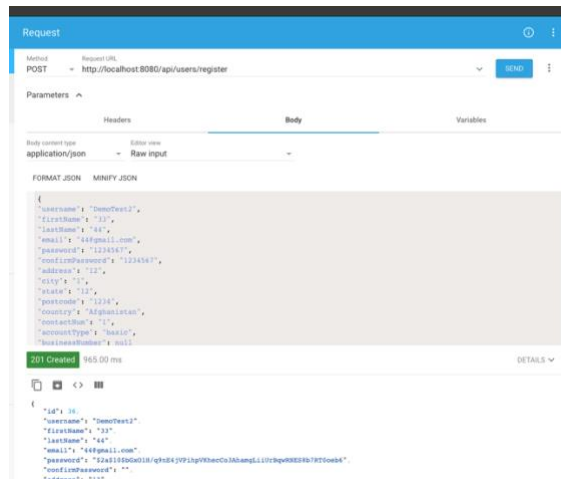
### Test 1 – Acceptance test for registering an Account

### Test credentials :

**Username : DemoTest2**

**Password : 1234567**

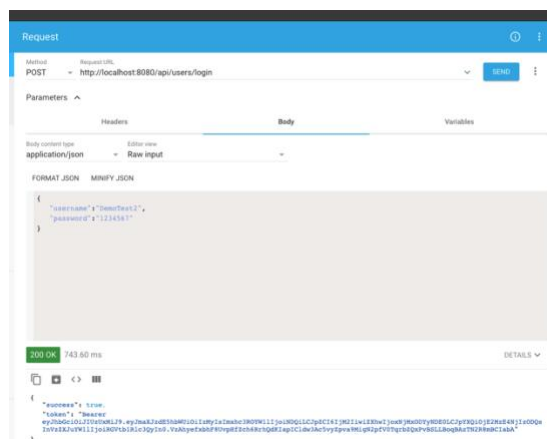
Using the Sign Up page on the UI, registering an account with the provided credentials, we can clearly see that details are added into the database.



*Screenshot of data records on the database as controller response*

## Test 2 – Acceptance test for user signing in

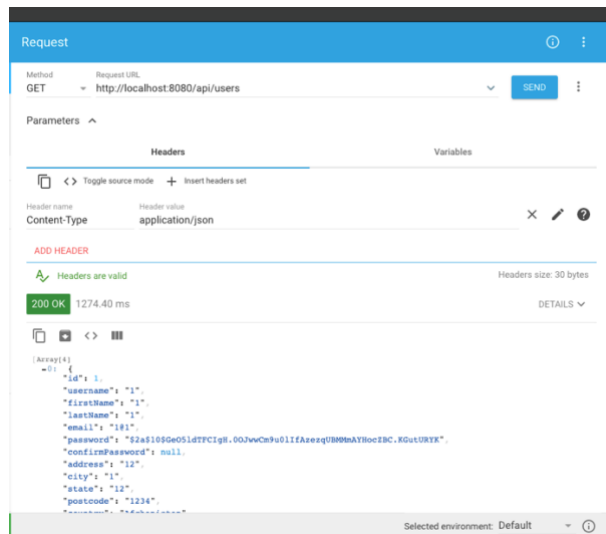
We can also observe here, upon logging in, the UI will create a log in token and can be viewed in the response of the controller response.



*token for the logged in user*

### Test 3- Testing the controller's connectivity to database

In this test, we test if our API controller could get all data from our “users” table in the database.



## Book Page Testing

Front end testing was also performed on the book page particularly the function which uses axios to get the particular book the page is for. This page uses the isbn of the book as the main id of the book and uses it together with the address of the books API to retrieve that particular book from the backend. In preparations for this test, we have planned several items to check. One of them is to check if the function is only being called once, to ensure that we are not sending multiple requests and place strain on the book microservice API.

```
// Calls axios and checks if its only been called a single time
it('calls axios and checks if its only been called a single time', async () => {

  // Setup
  mockAxios.get.mockImplementationOnce(() => Promise.resolve({
    data: {
      isbn: "9780747532743",
      title: "Harry Potter and the Philosopher's Stone ",
      authorFirstName: "J.K",
      authorLastName: "Rowling",
      largeCover: "http://covers.openlibrary.org/b/isbn/9780747532743-L.jpg",
      genre: "Adventure",
      mediumCover: "http://covers.openlibrary.org/b/isbn/9780747532743-M.jpg"
    }
  })))

  // Work
  const response = await BookPage.prototype.getBookFromApi(9780062390622);

  // Assertion / expects
  expect(mockAxios.get).toHaveBeenCalledTimes(1);
})
```

This test, like the following tests, uses a mock of axios to mock the response that may be received. The mock axios code uses Jest and simply resolves empty data. However using the function `mockImplementationOnce`, we can give it values which will mock whatever data we insert into it.

```
// calls axios and returns a book
it('calls axios and returns a book', async () => {

  // Setup
  mockAxios.get.mockImplementationOnce(() => Promise.resolve({
    data: {
      isbn: "9780747532743",
      title: "Harry Potter and the Philosopher's Stone ",
      authorFirstName: "J.K",
      authorLastName: "Rowling",
      largeCover: "http://covers.openlibrary.org/b/isbn/9780747532743-L.jpg",
      genre: "Adventure",
      mediumCover: "http://covers.openlibrary.org/b/isbn/9780747532743-M.jpg"
    }
  })))

  // Work
  const response = await BookPage.prototype.getBookFromApi(9780062390622);
  console.log(response);

  // Assertions / expects
  expect(response).toEqual({
    data: {
      isbn: "9780747532743",
      title: "Harry Potter and the Philosopher's Stone ",
      authorFirstName: "J.K",
      authorLastName: "Rowling",
      largeCover: "http://covers.openlibrary.org/b/isbn/9780747532743-L.jpg",
      genre: "Adventure",
      mediumCover: "http://covers.openlibrary.org/b/isbn/9780747532743-M.jpg"
    }
  });
});
```

Here we check if the function returns a response that is equal to a response value that is expected based on the ISBN used to find that particular book.

```
// Calls axios and checks if its been called using a parameter
it('calls axios and checks if its been called using a parameter', async () => {

  // Setup
  mockAxios.get.mockImplementationOnce(() => Promise.resolve({
    data: {
      isbn: "9780747532743",
      title: "Harry Potter and the Philosopher's Stone ",
      authorFirstName: "J.K",
      authorLastName: "Rowling",
      largeCover: "http://covers.openlibrary.org/b/isbn/9780747532743-L.jpg",
      genre: "Adventure",
      mediumCover: "http://covers.openlibrary.org/b/isbn/9780747532743-M.jpg"
    }
  })))

  // Work
  const response = await BookPage.prototype.getBookFromApi(9780062390622);

  // Assertions / expects
  expect(mockAxios.get).toHaveBeenCalledWith("http://localhost:2000/books/9780062390622");
});
```

For the test above, we check if the axios get method is being called with the correct URL API to retrieve that book.

```
// Checks if returns nothing if no isbn is inputted in the parameter
it('calls axios with no parameters (no isbn) and returns nothing', async () => {

  // Setup
  mockAxios.get.mockImplementationOnce( () => Promise.resolve({}))

  // Work
  const response = await BookPage.prototype.getBookFromApi();

  // Assertions / expects
  expect(response).toEqual({});
})
```

For our last test, we simply are checking if the response returns nothing if the parameter has no ISBN, as it should not return a book without providing an ISBN.

```
TERMINAL  DEBUG CONSOLE  PROBLEMS  2  OUTPUT

PASS src/components/BookDisplay/__tests__/BookPage.js
✓ calls axios and checks if its only been called a single time
✓ calls axios and returns a book (2ms)
✓ calls axios and checks if its been called using a parameter
✓ calls axios with no parameters (no isbn) and returns nothing

console.log src/components/BookDisplay/__tests__/BookPage.js:45
{
  data: {
    isbn: '9780747532743',
    title: "Harry Potter and the Philosopher's Stone ",
    authorFirstName: 'J.K.',
    authorLastName: 'Rowling',
    largeCover: 'http://covers.openlibrary.org/b/isbn/9780747532743-L.jpg',
    genre: 'Adventure',
    mediumCover: 'http://covers.openlibrary.org/b/isbn/9780747532743-M.jpg'
  }
}

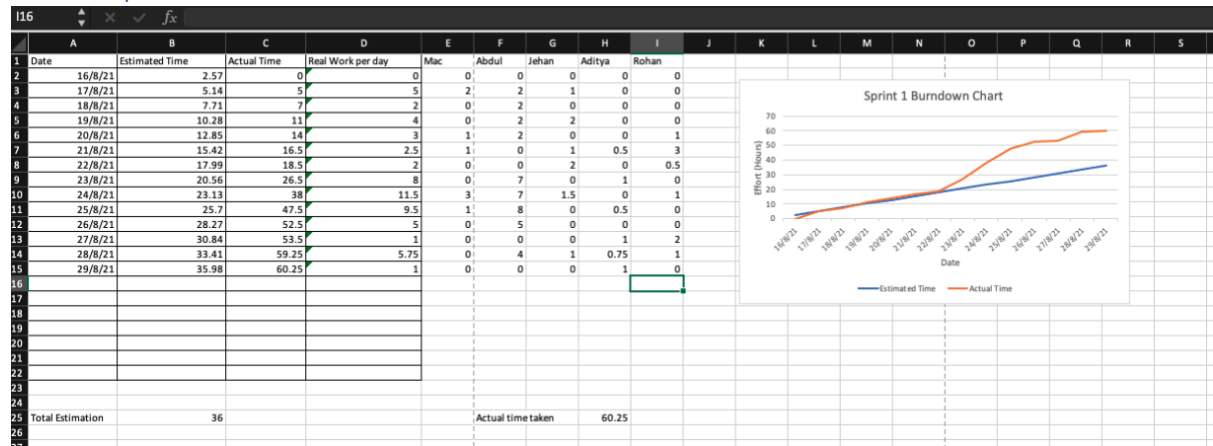
Test Suites: 1 passed, 1 total
Tests: 4 passed, 4 total
Snapshots: 0 total
Time: 0.357s, estimated 1s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.[]
```

Each one of these tests passed and satisfies our requirements for retrieving books from the backend API.

# Burndown Charts

## Sprint 1



## Sprint 2

