

Add interactivity to REST API



Estimated time needed: 40 minutes

Overview

In this lab, you will clone and load the Library Spring Boot application code to the project workspace. You will then add interactivity to the Library web application with additional REST API endpoints. The application is a variation of what you created in the previous lab and uses Java HashMap to maintain a database of the books, members, and borrowing records.

Learning objectives

After completing this lab, you will be able to:

- Clone a Spring project using Git
- Load the project
- Build and run the Spring Boot application
- Add more endpoints to the Controller besides CRUD using path parameters and request parameters
- Test the endpoints with POSTman

Prerequisites (optional)

You need to have an understanding of basic Java programming before you get started with this lab. Please ensure that you complete the labs sequentially as they are constructed progressively.

Clone the Library Application

1. The Maven spring application starter code with all the functionalities has been provided on a Git repository. Clone the application from the remote repository by running the following command:

```
git clone https://github.com/ibm-developer-skills-network/bkwcw-spring_labs.git
```

2. Change to the project directory.

```
cd /home/project/bkwcw-spring_labs/library
```

3. Use `mvn clean` (maven clean) to clean any existing class files and `mvn install` (maven install) to compile the files in the project directory and generate the runnable jar file.

```
mvn clean install
```

4. Run the following command to start the application. It will start in port 8080.

```
mvn exec:java -Dexec.mainClass="com.app.library.LibraryApplication"
```

This application provides the following endpoints:

- @GetMapping("/books") - To get all the books
- @GetMapping("/books/{id}") - To get a specific book by its ID
- @PostMapping("/books") - To add a book
- @PutMapping("/books/{id}") - To modify a book
- @DeleteMapping("/books/{id}") - To delete a book
- @GetMapping("/members") - To get all the members
- @GetMapping("/members/{id}") - To get a specific member
- @PostMapping("/members") - To add a member
- @PutMapping("/members/{id}") - To modify a member
- @DeleteMapping("/members/{id}") - To delete a member
- @GetMapping("/borrowing-records") - To get all borrowing records
- @PostMapping("/borrow") - To add a borrowing record
- @PutMapping("/return/{recordId}") - To modify a borrowing record

3. Click the button below to open the app.

Library Application

4. Copy the URL and note it down in a notepad or an editor. You will be using this url in POSTman.



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Mar 02 02:51:07 EST 2025

There was an unexpected error (type=Not Found, status=404).

Use the REST APIs on POSTman to add data

1. On the POSTman [homepage](#), sign in with your credentials.
2. On the homepage, click New Request to start sending requests to the Library app's REST APIs.

Get started



Send an API request

Quickly send and test any type of API request: HTTP, GraphQL, gRPC, WebSocket

3. On the Request page, set the request option to POST. Type the URL that you had copied and suffix it with /books. Choose the Body option for input and configure the input to be raw,json. Paste a json, like the option provided below. Click Send to post the book.

```
{
  "id": 1,
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
  "publicationYear": 1925,
  "genre": "Fiction",
  "availableCopies": 5
}
```

If the request goes through, you will see a response as shown in the image.

The screenshot shows a REST client interface with a POST request to `https://lavanyas-8080.theianext-0-labs-prod-misc-tools-us-east-0.proxy.cognitivecla`. The request body is a JSON object representing a book record. The response is also a JSON object, identical to the request body.

Request:

```
POST https://lavanyas-8080.theianext-0-labs-prod-misc-tools-us-east-0.proxy.cognitivecla
```

Body:

```
{
  "id": 1,
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
  "isbn": "9780743273565",
  "publicationYear": 1925,
  "genre": "Fiction",
  "availableCopies": 5
}
```

Response:

```
{
  "id": 1,
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
  "isbn": "9780743273565",
  "publicationYear": 1925,
  "genre": "Fiction",
  "availableCopies": 5
}
```

3. Add a few more records to the books following the same procedure.
- [Click here for sample](#). You can add the one after the other through postman
4. Add members to the library, using the `/members` endpoint with data as in the sample provided below.

► [Click here for a sample of what the URL should look like](#)

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john.doe@example.com",
  "phoneNumber": "123-456-7890",
  "startDate": "2023-01-01",
  "endDate": "2025-01-01"
}
```

5. Next, add borrowing records to the library, using the `/borrow`, as in the sample provided below.

Ensure the id of books and members exist before you add them.

► [Click here for a sample of what the URL should like lik](#)

A borrowing record will have the record ID, member ID, and book ID. The date of borrowing will be the current date and the due date will be calculated. A sample borrowing record has been provided.

```
{
  "id": 1,
  "bookId": 1,
  "memberId": 1
}
```

Add additional end points

To enhance the functionality of the library application, you need to add additional API end points:

- `@GetMapping("/books/genre")` - This takes the genre as a request parameter and gets all the books that belong to the specified genre.
- `@GetMapping("/books/author/{author}")` - This takes the author name as a path variable and genre optionally as a request variable and gets all the books for that author, optionally filtered by genre.
- `@GetMapping("/books/dueondate")` - This takes the date as a request parameter and gets all books due on the specified date.
- `@GetMapping("/bookavailabledate")` - This takes the bookId as the request parameter and returns the earliest date on which the book will be available.

1. Open the `LibraryServices.java` file under the `com/app/library/services` directory. You will need to provide the services in this file, for the end points to be functional.
2. Add the following code in `LibraryServices.java` for getting books by genre.

```
public Collection<Book> getBooksByGenre(String genre) {
    Collection<Book> allBooks = (Collection<Book>)(books.values());
    return allBooks.stream()
        .filter(book -> (book.getGenre()).toLowerCase().contains(genre.toLowerCase()))
        .collect(Collectors.toList());
}
```

In this code:

- The `allBooks` list is converted into a stream.
- For each book in the stream:
 - Convert the genre to lowercase.
 - Check if the book's genre converted to lowercase contains the genre passed (also converted to lowercase). For example, `Fantasy Fiction`, `Science Fiction`, and `Fiction`; all three contain `fiction`.
 - If true, include the book in the filtered stream.
- The filtered stream contains books in genres that match the condition.
- These books are added into a new list.

3. Add the following code in `LibraryServices.java` for getting books by author, optionally filtered by genre.

```
public Collection<Book> getBooksByAuthorAndGenre(String author, String genre) {
    Collection<Book> allBooks = (Collection<Book>)(books.values());
    return allBooks.stream()
        .filter(book -> book.getAuthor().equalsIgnoreCase(author)) // Filter by author
        .filter(book -> genre == null || book.getGenre().toLowerCase().contains(genre.toLowerCase())) // Optional filter by genre
        .collect(Collectors.toList());
}
```

4. Add the following code in `LibraryServices.java` for getting books by due date.

```
public Collection<Book> getBooksDueOnDate(LocalDate dueDate) {
    Collection<BorrowingRecord> allRecords = (Collection<BorrowingRecord>)(borrowingRecords.values());
    ArrayList<Book> dueBooks = new ArrayList<Book>();
    Collection<BorrowingRecord> tempRecords = allRecords.stream()
        .filter(record -> record.getDueDate().equals(dueDate)) // Filter by dueDate
        .collect(Collectors.toList());
    // For each filtered record, find the corresponding book and add it to the dueBooks list
    for (BorrowingRecord record : tempRecords) {
        Book book = books.get(record.getBookId());
        if (book != null) {
            dueBooks.add(book); // Add the book to the dueBooks list
        }
    }
    return dueBooks;
}
```

In this code:

- An empty `ArrayList<Book>` is created to store the books that are due on the specified dueDate.
- A stream is created from the records list, obtained from the hashmap of borrowing records.
- The filter method is used to select only those `BorrowingRecord` objects where the dueDate matches the specified dueDate.
- The filtered records are collected into a new list called `tempRecords`.
- Iterate through `tempRecords` and get the books with the id and add them to the list to be returned.

4. Add the following code in `LibraryServices.java` for getting the earliest date on which a book is available.

```
public LocalDate checkAvailability(Long bookId) {
    Collection<BorrowingRecord> allRecords = (Collection<BorrowingRecord>)(borrowingRecords.values());
    Book bookToCheck = books.get(bookId);
    if (bookToCheck == null) {
        return null;
    } else {
        if (bookToCheck.getAvailableCopies() >= 1) {
            return LocalDate.now();
        } else {
            List<BorrowingRecord> sortedRecords = allRecords.stream()
                .filter(record -> record.getBookId() == bookId) // Filter by bookId
                .sorted((b1, b2) -> b1.getDueDate().compareTo(b2.getDueDate())) // Sort by dueDate (soonest to latest)
                .collect(Collectors.toList());
            return sortedRecords.get(0).getDueDate();
        }
    }
}
```

In this code:

- If the book with the given id doesn't exist in the `HashMap`, it returns null.
- If the book exists, and it's available copies are greater than or equal to 1, the current date is returned.
- If the book has no available copies, the method filters the `allRecords` list to find borrowing records for the specified bookId.
- The filtered records are sorted by dueDate in ascending order, soonest to latest.
- The method retrieves the first record from the sorted list that has the earliest due date and returns this dueDate.

Now that the services are added for all the four interactions, you will add the endpoints in the Controller in the next step.

Add the endpoints in the controller

1. Open `LibraryController.java` in `com.app.library.controllers` package.

2. Add the following code to the Controller:

```
@GetMapping("/books/genre")
public ResponseEntity<Collection<Book>> getBooksByGenre(
    @RequestParam String genre) {
    Collection<Book> books = libraryService.getBooksByGenre(genre);
    logger.info("The books retrieved for genre "+genre);
    return new ResponseEntity<>(books, HttpStatus.OK);
}
```

```

@GetMapping("/books/author/{author}")
public ResponseEntity<Collection<Book>> checkGenreForAuthor(
    @PathVariable String author,
    @RequestParam(required = false) String genre) {
    Collection<Book> books = libraryService.getBooksByAuthorAndGenre(author, genre);
    logger.info("The books retrieved for the author and genre "+author+" - " + genre);
    return new ResponseEntity<>(books, HttpStatus.OK);
}
@GetMapping("/books/dueondate")
public ResponseEntity<Collection<Book>> getBooksDueOnDate(
    @RequestParam("dueDate") @DateTimeFormat(pattern = "dd/MM/yyyy") LocalDate dueDate) {
    Collection<Book> books = libraryService.getBooksDueOnDate(dueDate);
    logger.info("The books retrieved by due date "+dueDate);
    return new ResponseEntity<>(books, HttpStatus.OK);
}
@GetMapping("/bookavailableDate")
public ResponseEntity<LocalDate> checkAvailability(
    @RequestParam Long bookId) {
    LocalDate avlDate = libraryService.checkAvailability(bookId);
    if(avlDate == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    } else {
        return new ResponseEntity<>(avlDate, HttpStatus.OK);
    }
}
}

```

In this code:

@GetMapping("/books/genre")

- The endpoint is accessed via a GET request to /books/genre.
- It accepts a query parameter genre, for example, /books/genre?genre=Fiction.
- The libraryService.getBooksByGenre(genre) method is called to retrieve all books of the specified genre.
- The retrieved books are logged using a logger.
- The method returns a ResponseEntity containing the list of books and an HTTP status code of 200 OK.

@GetMapping("/books/author/{author}")

- The endpoint is accessed via a GET request to /books/author/{author}. For example: /books/author/Harper%20Lee).
- It accepts a path variable author, for example, Harper Lee.
- An optional query parameter genre (e.g., /books/author/Harper%20Lee?genre=Fiction).
- The libraryService.getBooksByAuthorAndGenre(author, genre) method is called to retrieve books by the specified author and genre.
- The retrieved books are logged using a logger.
- The method returns a ResponseEntity containing the list of books and an HTTP status code of 200 OK.

@GetMapping("/books/dueondate")

- The endpoint is accessed via a GET request to /books/dueondate.
- It accepts a query parameter dueDate in the format dd/MM/yyyy. For example, /books/duondate?dueDate=20/03/2025).
- The libraryService.getBooksDueOnDate(dueDate) method is called to retrieve books due on the specified date.
- The retrieved books are logged using a logger.
- The method returns a ResponseEntity containing the list of books and an HTTP status code of 200 OK.

@GetMapping("/bookavailableDate")

- The endpoint is accessed via a GET request to /bookavailableDate.
- It accepts a query parameter bookId (e.g., /bookavailableDate?bookId=1).
- The libraryService.checkAvailability(bookId) method is called to check the availability of the book:
 - If the book is available, it returns the current date (LocalDate.now()). If the book is not available, it returns the earliest due date for the book. If the book does not exist, it returns null.
 - If the book does not exist (avlDate == null), the method returns an HTTP status code of 404 NOT FOUND. Otherwise, it returns a ResponseEntity containing the availability date and an HTTP status code of 200 OK.

3. Use mvn clean (maven clean) to clean any existing class files and mvn install (maven install) to compile the files in the project directory and generate the runnable jar file.

```
mvn clean install
```

4. Run the following command to start the application. It will start in port 8080.

```
mvn exec:java -Dexec.mainClass="com.app.library.LibraryApplication"
```

If the previous instance of the application is running already, please stop it before you attempt to run.

5. Click the button below to open the app.

Library Application

6. Copy the URL and note it down in a notepad or an editor. You will use this URL in POSTman.

< > ↺ <https://lavanyas-8080.theianext-0-labs-prod-misc-tools-us-east-0.proxy.cognitiv>

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Mar 02 02:51:07 EST 2025

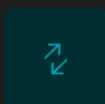
There was an unexpected error (type=Not Found, status=404).

Use POSTman to explore the REST APIs you just added

Please note that all the data, you had previously entered will not be stored as there is no storage we have used. You will need to add the details through JSON again.

1. On the postman homepage, click **New Request** to start sending requests to the Library app's REST APIs.

Get started



Send an API request

Quickly send and test any type of API request: HTTP, GraphQL, gRPC, WebSocket

2. On the request page, set the request option to **GET**. Type the URL that you had copied and suffix it with `/books/genre`. Choose the **Parameters** option for input and configure the key to be `genre` and the parameter to be the search parameter, such as `fiction`. Select the box preceding the parameter you want to pass to the request. Click **Send** to initiate the request.

GET ▼ <https://user-8080.theianext-0-labs-prod-misc-tools-us-east-0.proxy.cog>

Params ● Authorization Headers (9) Body ● Scripts Tests Settings

Query Params

<input type="checkbox"/>	Key	Value
<input type="checkbox"/>	genre	fiction
	Key	Value

If the request goes through, you will see a response showing the books matching the genre.


GET ▼ <https://user-8080.theianext-0-labs-prod-misc-tools-us-east-0.proxy.co>

Params ● Authorization Headers (7) Body Cookies Headers (5) Test Results

Query Params

<input checked="" type="checkbox"/>	Key	Value
<input checked="" type="checkbox"/>	genre	fiction
	Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON ▼ 

```
1  [  
2    {  
3      "id": 1,  
4      "title": "The Great Gatsby",  
5      "author": "F. Scott Fitzgerald",  
6      "publicationYear": 1925,  
7      "genre": "Fiction",  
8      "availableCopies": 6  
9    }  
10 ]
```


3. In the request page, set the request option to GET. Type the URL that you had copied and suffix it with /books/author/{author}, replacing {author} with an author name.
- This will return all the books by the author.

GET

https://user-8080.theianext-0-labs-prod-misc-tools-us-east-0.proxy.co

Params

Authorization

Headers (9)

Body ●

Scripts

Tests

Settings

Query Params

	Key	Value
	Key	Value

Body

Cookies

Headers (6)

Test Results

Pretty

Raw

Preview

Visualize

JSON

1

[

2

{

3

"id": 1,

4

"title": "The Great Gatsby",

5

"author": "F. Scott Fitzgerald",

6

"publicationYear": 1925,

7

"genre": "Fiction",

8

"availableCopies": 6

9

}

10

]

4. In the same request, choose the Parameters option for input and configure the key to be genreand the parameter to be the search parameter such as fiction. Click Send to initiate the request.
- This will return all the books of the author in the specified genre.

Note: {author} is a path parameter and genre is a query parameter here.

GET

▼

https://user-8080.theianext-0-labs-prod-misc-tools-us-east-0.proxy

Params

Authorization

Headers (9)

Body

Scripts

Tests

Settings

Query Params

<input checked="" type="checkbox"/>	Key	Value
<input checked="" type="checkbox"/>	genre	Mystery
	Key	Value

Body

Cookies

Headers (6)

Test Results

Pretty

Raw

Preview

Visualize

JSON ▼

1

[]

Practice exercise

1. Test /books/dueondate endpoint with a query parameter.

GET

▼

https://user-8080.theianext-0-labs-prod-misc-tools-us-east-0.proxy.co

Params

Authorization

Headers (7)

Body

Scripts

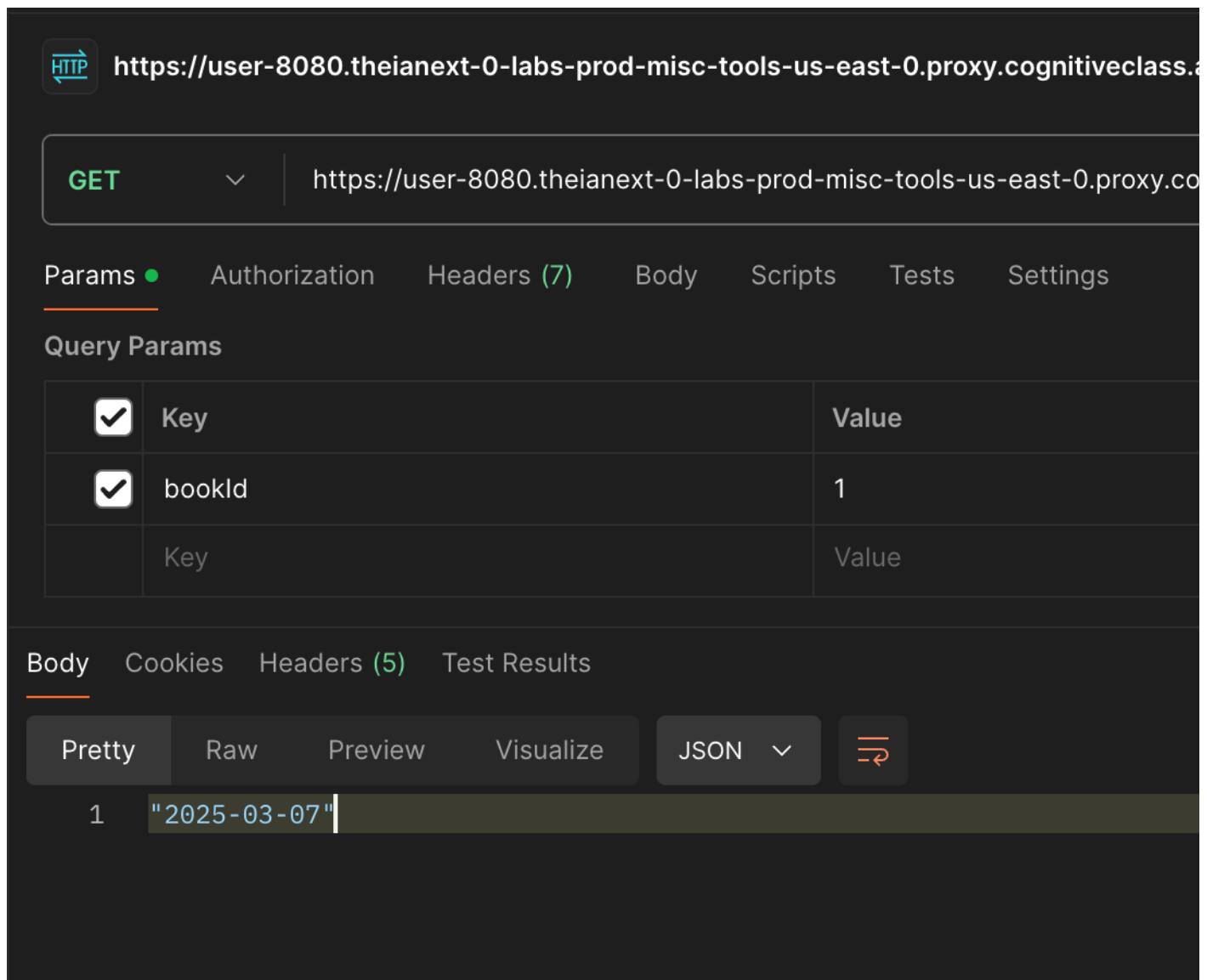
Tests

Settings

Query Params

<input checked="" type="checkbox"/>	Key	Value
<input checked="" type="checkbox"/>	dueDate	22/03/2025
	Key	Value

2. Test /bookavailableDate endpoint with a query parameter.



3. Modify the endpoint `@GetMapping("/books")`, to take optional request parameters for author and genre, and combine the following three queries into 1.

- `@GetMapping("/books")`
- `@GetMapping("/books/genre")`
- `@GetMapping("/books/author/{author}")`

► [Click here for hints](#)

Conclusion

In this lab, you have:

- Cloned a Spring project using Git, loaded the project in your IDE, and build and run the Spring Boot application
- Added more endpoints to the Controller in addition to CRUD using path parameters and request parameters.
- Tested the endpoints with POSTman

Author(s)

[Lavanya](#)