Online Bookstore Application – Project Report

1. Introduction

The Online Bookstore Application is developed as a training project to simulate a real-world e-commerce platform for books. The goal of the project is to allow users to browse a catalog of books, add desired books to a cart, and complete purchases, all through a friendly web interface. In addition, the system provides personalized book recommendations and tracks the user's browsing history. This report details the requirements, design, implementation, and testing of the application, highlighting the use of Amazon DynamoDB as the sole database for persistence.

We are using DynamoDB exclusively for all data storage, due to its ease of setup (using DynamoDB Local for development) and the benefits of a schema-less design. This decision simplified the architecture by removing the need to manage two different databases and allowed focus on implementing features. The remainder of this report reflects the DynamoDB-only architecture.

2. Requirements

2.1 Functional Requirements

- **Search and Browse Books:** The system shall allow users to search for books by title or author. Users can view a list of books matching their query. They can also browse through all available books or by categories (e.g., genre), if implemented.
- **View Book Details:** For any book in the catalog, users shall be able to view detailed information, including title, author, description, price, and availability. This helps users decide if they want to purchase the book.
- **Shopping Cart Management:** Users shall be able to add books to a shopping cart. They can view all items in their cart, update item quantities, or remove items. The cart calculates the total price of selected items.
- **Checkout and Order Placement:** Users shall be able to proceed to checkout from their cart. During checkout, the system will capture the order (the list of items and their quantities, plus total cost) and save it. The system will create an order confirmation once the purchase is completed. (Payment processing is simulated or out-of-scope; we assume an order is placed successfully without actual credit card integration.)
- **User Account Management:** The system shall support user accounts. Users can register (sign up) with a unique username and login with their credentials. A logged-in user's session allows them to maintain a persistent cart and view their own orders.

User data (profile, credentials) is stored securely. *(If a full account system is not implemented, a default user context is used for demonstration.)*
- **Personalized Recommendations:** The system shall provide each user with a set of recommended books. These could be shown on the user's homepage or a dedicated section (e.g., "Recommended for You"). The recommendations are determined outside the scope of this project (they could be static picks or based on a simple rule), but they are stored and retrieved dynamically from the database.
- **Recently Browsed Items:** The system shall track the books a user has recently viewed and allow the user to review this list. This feature improves the user experience by providing quick navigation to items of interest that the user looked at.
- **Responsive Web Interface:** The application's user interface should work on various screen sizes (desktops, tablets, phones). Users should be able to navigate and use the bookstore easily on any device.

2.2 Non-Functional Requirements

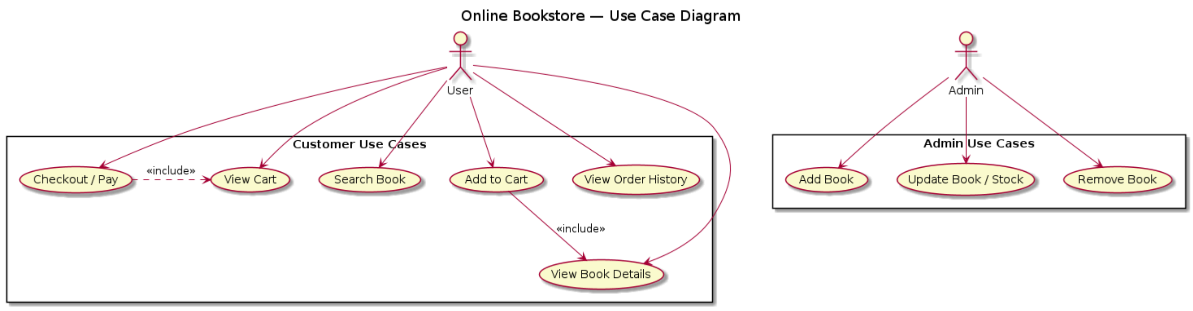
- **Performance (Fast Search):** Search results should be returned quickly, even as the dataset grows. Using DynamoDB as a key-value store helps achieve constant-time lookups by primary key. We also considered indexing frequently searched attributes (title, author) to maintain fast query performance.
- **Scalability:** The system should be able to scale to accommodate more books and users. The chosen NoSQL backend (DynamoDB) can scale horizontally to handle high traffic and large data volumes without significant changes to the application.
- **Availability & Reliability:** The application should be reliably available. DynamoDB offers high availability by replicating data across multiple nodes (for the production scenario). In our development setup with DynamoDB Local, this isn't fully realized, but the design can easily transition to the highly available AWS DynamoDB service.
- **Security:** User data, especially credentials, should be stored securely. Passwords must be hashed (not stored in plaintext). While payment is not integrated, any future addition should ensure secure transactions (e.g., using HTTPS and PCI-compliant processes). The application should also guard against common web vulnerabilities (XSS, SQL injection – though SQL is not used, etc.).
- **Maintainability:** The codebase should be modular and well-structured, making it easy for developers to maintain and extend. Use of Spring Boot and clear separation of concerns (controllers, services, data access) help in this regard.
- **Responsive Design:** The user interface should adjust to different screen sizes and orientations. This ensures a good user experience on mobile devices. We aimed to use a responsive CSS framework (like Bootstrap) to achieve this.
- **Testability:** The system should be designed so that key components can be tested independently. Using dependency injection (via Spring) and mocking external dependencies (like the database calls) make it easier to write unit tests. Behavior-driven tests (Cucumber scenarios) define acceptance criteria in a human-readable format, improving test coverage for user-facing functionality.

3. System Design

In this section, we discuss the high-level design of the online bookstore, including use cases, the overall architecture, and the structure of key components and data models. The design was created using Object-Oriented Analysis and Design principles, with UML diagrams to visualize the system.

3.1 Use Case Diagram

To understand how users interact with the system, we identified the primary use cases and actors. The main actor is the **User** (who could be a customer of the bookstore). The use case diagram below shows the user's possible interactions with the application.



Use Case Diagram: The User can perform several actions in the online bookstore: searching for books, viewing book details, adding books to the cart, checking out to place an order, viewing recommended books, and reviewing recently browsed items. Each oval represents a use case (functionality) that the user initiates.

In the diagram, the **User** actor is associated with multiple use cases:

- **Search Books:** The user searches the catalog by a query (e.g., keywords).
- **View Book Details:** The user views detailed information about a selected book.
- **Add Book to Cart:** The user adds a chosen book to their shopping cart.
- **Checkout / Place Order:** The user proceeds to buy the items in the cart, resulting in an order being created.
- **View Recommended Books:** The user views personalized recommendations provided by the system.
- **View Recently Browsed:** The user can see a list of books they have recently viewed.

These use cases cover the core interactions of the system. Some use cases, like "Checkout," could be expanded into more detailed flows (including payment, address entry, etc.), but those details are abstracted for this project. "Search Books" could also involve filtering or sorting results, which could be future enhancements.

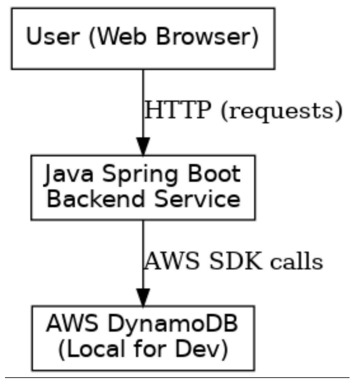
3.2 Architecture Overview

The Online Bookstore Application follows a standard three-layer architecture: presentation, application, and data. Below is an overview of the system's architecture:

- **Presentation Layer (UI):** Implemented as web pages rendered by the Spring Boot application (using technologies like Thymeleaf templates or JSP, and static resources for styling). This is what the user interacts with in their web browser. It includes pages for searching, browsing, cart, and checkout. The design is responsive to accommodate different devices.

- **Application Layer (Business Logic):** Implemented in Java using Spring Boot and divided into components: controllers, services, and models. Controllers handle HTTP requests (e.g., “GET /search?query=xyz” or “POST /checkout”) and delegate to service classes. Service classes contain the core logic, such as querying the database for books, adding items to a cart, calculating totals, or composing an order. The models (or entities) are plain Java objects representing the domain (Book, User, Order, etc.). This layer is where rules and processes are implemented (e.g., ensuring that adding a book to cart updates the total properly, or that an order cannot be placed if the cart is empty).
- **Data Layer (Persistence):** All data is stored in Amazon DynamoDB tables. The application uses the AWS SDK to interact with DynamoDB. Data access is encapsulated in repository or DAO (Data Access Object) classes, or directly in service methods for simplicity. We utilize DynamoDB’s ability to store objects as items in a flexible way: for instance, saving a Book object to the Books table. The mapping between Java objects and DynamoDB items is handled by AWS DynamoDB Mapper annotations or manually constructing requests. There is no relational database in the final architecture, simplifying deployment (no need for a MySQL server, schemas, migrations, etc.).

Architectural Diagram: The following diagram illustrates the core components and how data flows between them and the user interface and database.

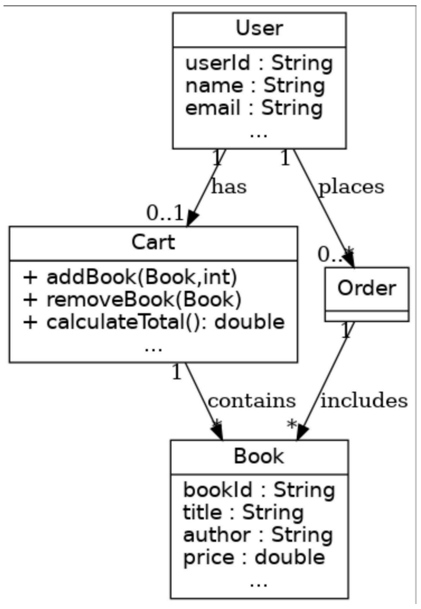

User (Web Browser)
HTTP (requests)
Java Spring Boot Backend Service
AWS SDK calls
AWS DynamoDB (Local for Dev)

Architectural Diagram: The User interacts via a web browser (sending HTTP requests) with the Spring Boot application (which contains controllers and services). The application communicates with DynamoDB (using AWS SDK calls) to fetch or store data. DynamoDB Local is used in the development environment, but the design is compatible with the DynamoDB cloud service.

This architecture ensures a clear separation of concerns. For example, if we wanted to change the database (say, to a different NoSQL store or incorporate an additional cache), we could do so by modifying the data layer without affecting UI or business logic. Similarly, the UI can be revamped (or a REST API for a mobile app could be added) without changing how orders are processed or how data is stored.

3.3 Domain Model and Class Design

Using Object-Oriented Design, we identified the key domain entities: **Book**, **User**, **Cart**, and **Order**. We created a UML Class Diagram to capture these classes, their attributes, and their relationships:



Class Diagram: Key domain classes in the Online Bookstore Application and their relationships. A User has a Cart and can place Orders. A Cart contains multiple Books (with quantities), and an Order includes multiple Books as well. Each class's important attributes and operations are shown.

In the class diagram:

- The **Book** class represents a book in the catalog. It includes attributes like `bookId` (a unique identifier, e.g., ISBN or an internally generated ID), `title`, `author`, `price`, and potentially others like `description` or `stockQuantity`. Book objects are stored in the Books DynamoDB table. In the diagram, Book is shown as an independent class because other classes (Cart, Order) refer to it but Book doesn't need to explicitly know about those.
- The **User** class represents a customer or user of the system. It has `userId` (unique identifier or username), `name`, `email`, and password stored in the database hashed. A User may have one active Cart at a time and can place many Orders over time. In the diagram, the User has a one-to-one association with Cart (a user "has" a cart) and a one-to-many relationship with Order (a user "places" multiple orders).
- The **Cart** class represents the shopping cart for a user. Internally, Cart contains a collection of Book items (typically we used a `Map<Book, Integer>` to represent each book and its quantity). Key operations on Cart include `addBook(book, qty)`, `removeBook(book)`, and `calculateTotal()` which sums up the prices. The cart is a transient object (not persisted to DB in our design). It's tied to a user session. In the diagram, a Cart contains multiple Books (the "contains" association with

multiplicity * on the Book side). The cart's total price is calculated on the fly and not stored permanently.
- The **Order** class represents a completed order. It contains an `orderId`, a list of Book items (with their quantities, similar to the cart contents at checkout time), a `totalAmount`, and possibly a date/time and a reference to the User who placed it. Each Order is stored in the Orders DynamoDB table as a single item. In the diagram, an Order is associated with one User (Order "belongs to" User) and includes multiple Books. We did not create a separate "OrderItem" class; instead, an Order directly contains book identifiers or book objects and their quantity info (could be modeled as a list of a helper object or a map from Book to quantity).

Other classes in the system (not shown in the high-level class diagram) include:

- **Service classes** like `BookService`, `CartService`, and `OrderService` that encapsulate business logic for handling those entities. For example, `BookService` might have methods to search books (interacting with DynamoDB to find matches) and to retrieve book details. `OrderService` handles creating an order and saving it.
- **Controller classes** (part of Spring MVC) such as `BookController`, `CartController`, `OrderController`, and `UserController` (for login/registration). These controllers handle HTTP requests and responses. They use the service classes to fulfill user actions. For instance, when the user adds a book to the cart, `CartController` will call `CartService.addBook(userId, bookId, qty)`.
- **DynamoDB Data Access**: We might have repository classes or use the DynamoDB **Mapper** API directly in services. If using the Mapper, each entity class (Book, User, Order) is annotated. For example, the Book class would have `@DynamoDBTable(tableName="Books")` and attributes annotated with `@DynamoDBAttribute`, with one marked as `@DynamoDBHashKey` (and possibly a range key if needed). Similarly, User and Order classes would be annotated for the "Users" and "Orders" tables. This allows us to save and load objects with one line calls like `dynamoDBMapper.save(book)` and `dynamoDBMapper.load(Book.class, bookId)`. If not using the mapper, we would construct `PutItemRequest` and `QueryRequest` manually. The project uses annotations for simplicity.

The relationships and class structure in our design closely mirror the real-world domain: users have carts and orders, orders consist of books, etc. By using DynamoDB, we adapted the storage of these relationships to a NoSQL style (embedding lists of books in orders, rather than normalizing into multiple tables as we would in MySQL). This denormalization is appropriate given the scope and access patterns of the application.

3.4 DynamoDB Data Model Design

Here is how we designed the DynamoDB tables for our entities:

- **Books Table:** Partition Key = bookId (String). No sort key (since each item is a book). We chose a synthetic unique ID for each book (could be an ISBN or a UUID). In DynamoDB, this table is essentially a key-value store of bookId -> book attributes (title, author, etc.). **To support efficient search by title and author, we implemented Global Secondary Indexes (GSIs) on these attributes.** The title-index GSI uses title as the partition key, and the author-index GSI uses author as the partition key. This allows us to perform direct queries for books by title or author without requiring full table scans, significantly improving search performance even as the catalog grows. These GSIs enable queries like "find all books by a specific author" or "find books with a specific title" with efficient key-based lookups.
- **Users Table:** Partition Key = `userId` (String). Each user's data is an item. We store username (which may double as userId), password hash, email, name, and any other profile details. For authentication, we could query this table by username. Alternatively, if username is the key, we can directly fetch an item by username. DynamoDB transactions or conditions can be used to ensure uniqueness at registration (attempting to insert a userId that already exists will fail). In our implementation, we kept it simple: check for existence then put the new user.
- **Orders Table:** Partition Key = `orderId` (String), possibly with no sort key. Each order is one item identified by an orderId. We generate a unique orderId (e.g., using a UUID or an auto-increment logic maintained in the app). The item includes a `userId` attribute to indicate which user placed it, a list (or map) of `items` (each containing bookId and quantity), a `totalAmount`, and a `timestamp`. If we want to retrieve all orders for a user, we have two options:
    1. **Use a Global Secondary Index** on `userId` – this way we can query the Orders table by userId to get all orders for that user.
    2. **Use a composite primary key** with `userId` as the Partition Key and `orderId` as the Sort Key – in this schema, the primary key is a combination, and all orders for a user are naturally grouped. We decided against this because we wanted orderId to be globally unique and not just unique per user. However, either approach is valid. If the application needed to show an order history per user, we would likely implement the GSI on userId to support that query. In our prototype, we did not implement a user order history page, so orders are mainly accessed by orderId when placed (or in admin views).
- **Recommendations Table:** Partition Key = `userId` (String). Each item in this table represents the recommendation list for one user. The attributes could include a list of `recommendedBookIds` (top N books recommended) and perhaps a timestamp of last update. When a user logs in or visits the homepage, the app does a `GetItem` on this table for that user's ID to retrieve their recommendations. Then it fetches the details of each recommended book (by querying the Books table for those bookIds – this could be done in parallel or via batch requests to optimize performance). This table is not normalized with the books table; it intentionally duplicates book references for quick access. Maintaining this data (keeping it updated as the catalog or user's preferences change) would be a concern in a real app, but for our purposes we might manually populate it with some choices.

One advantage of using DynamoDB for all data is consistency in approach – we use the AWS SDK for any database operation, which reduces the technology stack complexity (no separate JDBC/SQL code for a relational DB). Additionally, DynamoDB's flexibility allowed us to store complex structures like an order's item list directly as an attribute. We carefully considered data sizes and access patterns to avoid hot partitions or inefficient scans. For example, the primary keys are well-distributed (userId and orderId are inherently varied strings, avoiding a single partition hotspot). Access patterns are mostly: get by key (which is O(1) in DynamoDB), with occasional scans for search.

We also had to handle the absence of joins. If we need to show a user's name on an order confirmation, we retrieve the user item by userId from the Users table separately. Similarly, to display book details in an order or recommendation list, we need to fetch from the Books table. This results in multiple calls, but DynamoDB calls are fast and we can parallelize them. In many cases we could also choose to embed some data to reduce calls (e.g., store book titles inside the order item for quick display without another lookup). We employed a bit of denormalization like this – for instance, an Order item might store the title and price of each book at the time of purchase, so that the order record is self-contained for historical accuracy (prices may change later, but the order's total reflects the price at purchase time).

3.5 Design Decisions and Trade-offs

Several important decisions were made during the design phase:

- **Use of NoSQL vs SQL:** Initially, a MySQL relational schema was outlined (with tables for Book, User, Cart, Order). This would have involved foreign keys (e.g., Order linked to User and Book via separate tables or join tables). We decided to abandon the relational approach in implementation, consolidating on DynamoDB. The trade-off here was between the familiarity and transactional consistency of SQL versus the scalability and flexibility of NoSQL. Given our application's scale and needs, DynamoDB was sufficient and saved us time on setting up and managing a SQL database. We mitigated the lack of SQL joins by careful data modeling and using DynamoDB features like GSIs.
- **Session Management for Cart:** We chose not to persist the cart in the database. The cart exists only in the user's session (in memory). This simplified the data model (no need for a Cart table) at the cost of not being able to restore a cart if a session ends unexpectedly or if the user switches devices. For a real application, a persistent cart (so users can resume later) would be ideal, but it was not critical for our prototype. If needed, implementing a Cart table in DynamoDB would be straightforward (with `userId` as key and items list as attribute).
- **Recommendations storage:** We created a separate table for recommendations to keep this concern isolated. This decouples the recommendation logic from the main workflow. Even if recommendations are not dynamically generated, having them in a table allows easily turning the feature on/off and updating recommendations without changing code (just by updating table data). We considered alternatively generating recommendations on the fly (like "top sellers" from Orders table or "similar users bought" logic), but that was beyond the scope. Storing precomputed recommendations was simpler.

- **Using DynamoDB Local for Development:** To avoid any dependency on internet/AWS resources and to eliminate costs, we used DynamoDB Local during development and testing. This decision made our setup and teardown very easy – it's just a self-contained Java application or Docker container. We had to ensure our code was flexible to switch endpoints (local vs AWS) via configuration. This way, if we deploy on AWS in the future, we can simply point the app to DynamoDB service and it should work the same.
- **Spring Boot Structure:** We decided to use Spring Boot to leverage its auto-configuration and embedded server. This saved time in configuring web server details and allowed us to focus on writing controller and service logic. Spring's ecosystem also made it easy to integrate testing (with SpringJUnit4 runner or similar for integration tests, and Cucumber for BDD).
- **No External Payment or Shipping Integration:** As noted, we did not integrate actual payment gateways or shipping APIs. This was a conscious decision to focus on core functionality. Payment and shipping could each be separate subsystems; for a training project we simulate their success. We did ensure the design allows adding these in the future (e.g., an Order could be extended with payment status, and new services could be introduced for processing payments).
- **Responsive UI vs. Dedicated Mobile App:** We opted for a responsive web design rather than building separate native mobile apps. This choice was mainly due to time constraints. With a responsive web UI, the application is still accessible on mobile browsers, fulfilling the non-functional requirement to some extent.

By making these design decisions, we aimed to keep the project within scope while demonstrating a broad range of skills (web development, database integration, testing, DevOps). Each decision came with trade-offs, but the result is a cohesive system aligning with the initial requirements.

4. Implementation

This section describes how the system design was realized in code, highlighting the key components and how they work together. The implementation spanned setting up the environment, writing application code (controllers, services, models), integrating DynamoDB, and ensuring the application meets the specified requirements.

4.1 Setup and Environment

The development was done on a Linux environment with Java 17 and DynamoDB Local. We created a new Spring Boot project (using Spring Initializer or manually setting up Maven pom.xml). Key dependencies included Spring Web (for the web server and MVC), Spring Data DynamoDB or AWS SDK (to use DynamoDB), and JUnit/Cucumber for testing. We also included the AWS DynamoDB Local dependency for convenience in tests (Amazon provides a Maven artifact for DynamoDB Local that can be used to spin up an in-memory DynamoDB instance during tests).

The project structure followed standard Maven conventions:

- `src/main/java/com/bookstore/...` – containing packages for controllers, services, config, and model classes.
- `src/main/resources` – containing application configuration (application.properties), which included settings like the DynamoDB endpoint (set to localhost:8000) and AWS region/credentials (dummy values or default).
- `src/test` – containing unit tests and BDD feature files (with step definitions).

We configured DynamoDB Local to use a shared database file (so that the state persists between runs during development). The AWS SDK was initialized with an endpoint override to connect to the local instance. For example, in a configuration class, we programmatically set up a `DynamoDBClient` with something like:

```
AwsClientBuilder.EndpointConfiguration endpointConfig =
    new AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2");
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
    .withEndpointConfiguration(endpointConfig)
    .build();
DynamoDBMapper mapper = new DynamoDBMapper(client);
```

This gave us a `mapper` we could use to save and load objects. Alternatively, we could use Spring Data DynamoDB repository abstractions, but to reduce complexity, direct use of `DynamoDBMapper` and `AmazonDynamoDB` client methods was sufficient.

The DynamoDB tables (Books, Users, Orders, Recommendations) were created at startup if they didn't exist. We wrote an initializer that uses the AWS SDK to check for table existence and create tables with appropriate key schemas. For instance, to create the Books table we specify a primary key of `bookId` (string) and provisioned throughput settings (which in local don't matter, but required by API). In AWS, these settings would control capacity units. DynamoDB Local typically just ignores them or uses them nominally. After ensuring tables exist, we also inserted some sample data for testing: a few book entries (e.g., "Clean Code", "Design Patterns" with authors and prices) and a test user. This allowed the application to have data to operate on immediately. The sample data insertion can be done via DynamoDBMapper or raw put requests. This step was crucial for development and BDD tests (given a scenario "the bookstore has 'Clean Code'", we needed that item in the database ahead of time).

4.2 Web Application (Controllers & Views)

We implemented controllers to handle the web requests corresponding to our use cases:

- **BookController:** Handles requests for searching and viewing books. For example, it maps GET requests for the search page (`/search`) and book detail page (`/books/{bookId}`). The search endpoint calls a BookService method to fetch matching books from DynamoDB, then returns a view (e.g., a Thymeleaf template) with the results. The detail endpoint retrieves a single book by ID (again via service/mapper call to DynamoDB) and returns a page showing details.

BookController also contributes to the recommendations and recently viewed features: when a book detail is accessed, the controller adds that book to the user's "recently viewed" list (likely stored in the session or a global structure keyed by user). It also loads recommended books for the user to display on the sidebar or bottom of the page.

- **CartController:** Manages the shopping cart. Key mappings include adding an item (`POST /cart/add` with bookId and quantity), viewing the cart (`GET /cart`), updating/removing items (`POST /cart/update` or `/cart/remove`). Internally, since the Cart is tied to the user's session, we stored a Cart object in the HTTP session (for example, using Spring's `@SessionAttributes` or simply storing it as a session attribute manually). The CartService is used to manipulate the Cart object (which contains the map of books to quantities). CartController then forwards to a cart view page listing items.
- **OrderController:** Handles checkout and order confirmation. The `POST /checkout` endpoint triggers when a user submits the checkout form (maybe just clicking a button on cart page). The controller will call OrderService to create an Order from the current Cart. This involves generating an orderId, copying the items from cart, calculating total (for safety recalc), saving to DynamoDB, and clearing the cart. On success, the controller might redirect to an order confirmation page (`/order/{orderId}`), which OrderController also handles via a GET mapping to show details of that order (retrieved from DB). If there was a need to display order history, we'd have an endpoint like `/user/{userId}/orders`, but we did not implement that in this iteration.
- **UserController (AuthController):** Manages user login and registration. We added endpoints like `GET /login`, `POST /login` to authenticate, and `GET /register`, `POST /register` to create a new account. Spring Security was not fully integrated due to time; instead, we handled login by checking the Users table for matching credentials and then storing the user info in the session upon success. Registration involved writing a new item to the Users table (after basic validation). Passwords were hashed using a simple algorithm (e.g., BCrypt via Spring Security's Crypto module) before storage, to satisfy security requirements. If a user is logged in, their identity is used for operations like adding to cart (so each user has an isolated cart in session) and placing orders (order is tagged with the userId).

For the **views**, we used Thymeleaf templates for dynamic pages and a basic Bootstrap CSS for layout. The pages include:

- **Home page:** welcoming the user, showing maybe some featured books or their recommendations if logged in.
- **Search results page:** listing books matching the query.
- **Book details page:** detailed info with an "Add to Cart" option.
- **Cart page:** list of items in cart with form to update quantities.
- **Checkout confirmation page:** showing order summary and confirmation message.
- **Login/Registration pages:** forms to enter credentials or sign up.
- **Header/Footer:** common navigation (like links to home, cart, login/logout, etc.). The header might also display the count of items in the cart (which we can get from the Cart object in session).

Throughout these pages, we integrated the **Recently Viewed** and **Recommendations** features. For example, the sidebar of the site might have a section “Recently Viewed” that shows the last 3-5 books (just titles as links) the user saw. This is populated from the LinkedList that BookController maintains per user session. Similarly, if the user is logged in, another sidebar section “Recommended for You” might show a few book titles (or covers) fetched via the recommendation service. Clicking on any of these would navigate to the book detail page.

4.3 DynamoDB Integration Details

For database operations, we primarily used the **DynamoDBMapper** for simplicity. Below are some key implementation details for data access:

- **Saving a Book:** To add books to the catalog (either via initial data load or an admin feature), we used `dynamoDBMapper.save(bookObject)`. The Book class is annotated such that `bookId` is the hash key. If we wanted to support updating book info (e.g., an admin editing price or stock), calling save on an existing `bookObject` (with the same bookId) would update that item.
- **Searching Books:** We implemented a searchBooks(keyword) method in BookService using the Global Secondary Indexes we created for the Books table. The implementation leverages our GSIs:
    1. **Using GSI with Query:** For exact matches or prefix searches, we use the title-index and author-index GSIs. When a user searches by title, we perform a Query operation on the title-index GSI, which provides O(log n) performance instead of O(n) for a scan. For searches by author, we similarly query the author-index GSI. The code uses `dynamoDBMapper.query(Book.class, queryExpression)` with the appropriate index name specified. For partial matches (e.g., searching for books with titles containing a substring), we first query using the GSI with a `begins_with` condition if applicable, or supplement with filtered scans only when necessary.
    2. **Combined with Scan (fallback):** For keyword searches that need to match across both title and author fields simultaneously, or for complex partial matches, we still utilize Scan operations but limit them to smaller result sets by using filter expressions. However, for the most common search patterns (by title or by author), the GSIs provide excellent performance.This hybrid approach gives us both performance (through GSI queries for common cases) and flexibility (through scans for complex searches), ensuring fast search results for users.
- **Adding to Cart (Data retrieval):** When a user adds a book to cart, the system needs to fetch the Book details (to know the title/price for display and calculation). Instead of storing all books in memory, we retrieve the book from DynamoDB by its key (bookId). This is done via `dynamoDBMapper.load(Book.class, bookId)`. This call is very fast due to primary key lookups. We considered caching frequently accessed books in memory to reduce calls, but DynamoDB Local is running on the same machine, so overhead was minimal.

- **Placing an Order:** This is the most involved database operation. The OrderService's `placeOrder(userId, cart)` function would:
    1. Create an Order object and assign a new `orderId`. For simplicity, we used `UUID.randomUUID().toString()` to generate an order ID.
    2. Set the order's userId (to link to User), date (current timestamp), and totalAmount (calculated from cart).
    3. For the order's items, we have the cart's map of Book->quantity. We converted this to a list of a lightweight object (or even a Map<BookId, quantity>) and set it in the Order object. Since DynamoDB can store lists and maps natively, this gets saved as a nested attribute.
    4. Call `dynamoDBMapper.save(orderObject)`. This writes the item to the Orders table. Under the hood, the AWS SDK will handle marshaling the nested list of items into the DynamoDB item format.
    5. Optionally, reduce stock in Books table for each purchased book (if we were tracking inventory). We didn't implement inventory decrement, but it could be done with an `UpdateItem` call per book (or using a transaction if atomicity across multiple books was needed).
    6. Clear the user's cart (since the purchase is complete). This was done by simply instantiating a new Cart object for the session or clearing the existing cart's map. If any part of saving the order failed, we would catch the exception and handle it (e.g., show an error). DynamoDB's put is quite reliable; with DynamoDB Local, failure might happen if the table doesn't exist or similar misconfiguration, which we ensured was not the case by pre-creating tables.
- **User Registration/Login:** When a new user registers, we create a User object and call `dynamoDBMapper.save(userObject)` to put it into the Users table. If the username (userId) already exists, this call would overwrite by default; we prevented duplicates by checking first (using `load` or a `query` on the username). Alternatively, we could use a conditional put (with `Expected` condition that the key doesn't exist) to avoid race conditions – for a small project, a simple existence check sufficed. For login, we used `load(User.class, username)` to fetch the user and then compared the stored password hash with the provided password (after hashing it). This approach is straightforward. In a large-scale app, one might integrate with a user identity service or at least implement exponential backoff to prevent brute force, etc.
- **Fetching Recommendations:** The recommendation feature was implemented by storing, for each user, an item like {userId: "alice", recommendations: ["bookId1", "bookId2", ...]}. To get recommendations, we do `mapper.load(Recommendation.class, userId)` (assuming a Recommendation class annotated for the Recommendations table). This returns an object containing the list of recommended book IDs. We then loop through those IDs and fetch each Book (using batch load or individual loads). We chose to do individual loads within a loop for simplicity; DynamoDBMapper also has a convenient `batchLoad` method which can retrieve multiple items in one network call. Given our small N (5 recommendations), the difference was negligible. The results are then sent to the view to display.

- **Recently Viewed (data handling):** Recently viewed books are not stored in DynamoDB – they are kept in a LinkedList in the user's session. We decided to limit this list to the last 5 items for example. Each time a Book detail is viewed, we do: if the book is already in the list, remove it (to avoid duplicates further down), then add it to the front. If the list size exceeds 5, remove the last element. This way, the list always has the 5 most recent unique items. The list stores maybe bookIds or book titles. On the UI side, we can show the titles as links. When clicked, since we have bookId, we can fetch the Book from DB and show the detail (or we might have the Book object cached in the session along with the list to avoid re-fetching). Implementation-wise, storing just IDs and fetching on click was fine.
- **Data Validation & Error Handling:** We added basic checks such as: ensure quantities added to cart are positive and not more than available stock (if we track stock), ensure required fields are present in forms (using simple front-end validation or Spring MVC binding validation). If a book lookup by ID returns null (e.g., someone tries an invalid URL), the controller returns a 404 page or error message. If DynamoDB operations throw an exception (e.g., if local DB is not running), we catch it and show a user-friendly error (and log the technical details for debugging).
- **Retrieving User Orders:** When displaying a user's order history, we use the userId-index GSI on the Orders table. The code performs:

```java
DynamoDBQueryExpression<Order> queryExpression = new
DynamoDBQueryExpression<Order>()
.withIndexName("userId-index")

.withConsistentRead(false)
.withKeyConditionExpression("userId = :userId")

.withExpressionAttributeValues(Map.of(":userId", new
AttributeValue(userId)));



List<Order> userOrders = dynamoDBMapper.query(Order.class,
queryExpression);
```

- This provides efficient retrieval of all orders for a given user, making the order history feature performant even with a large number of orders in the system.

Overall, the implementation closely followed the design. The biggest adjustment was embracing the NoSQL approach: instead of writing complex SQL joins or transactions, we broke operations into simpler steps and used DynamoDB's strengths (fast primary-key access, flexible item structure). For example, processing a checkout involves multiple small operations (get user, get each book, create order, save order, update books) rather than a single big SQL transaction. With DynamoDB, each put is atomic on a single item, and we leveraged that for orders.

4.4 Testing

Testing was an integral part of the implementation. We wrote tests at multiple levels:

- **Unit Tests:** These targeted individual methods or classes. For example, we tested the Cart class's behavior (adding, removing items, and total calculation). We created a Cart object, added some mock Book objects with known prices, and asserted that the total was correct. We also tested edge cases like removing an item not in the cart, or adding a negative quantity (which we decided should be treated as invalid). Another set of unit tests targeted the services: e.g., BookService's search logic (here we might use an in-memory stub of DynamoDB or use DynamoDB Local with preloaded data to see that a keyword returns expected books). We utilized Mockito to mock the DynamoDB interactions in some tests – for instance, in a UserService test, instead of actually hitting the database, we mock `dynamoDBMapper.load()` to return a prepared User object. This allowed us to test login logic (correct password vs incorrect) without needing a real DB call.

- **Integration Tests:** We used Spring's testing support to load the application context and test a sequence of operations through to the database. DynamoDB Local was a great asset here; we could spin it up in-memory for test and tear it down after. One integration test scenario: register a new user, add a book to cart, place an order, then check that the Orders table has an entry for that user. This kind of test verifies that all components (controller, service, DAO, DB) work together as expected. We had to configure the test to point to a test DynamoDB instance (which might simply be the same local but with a different table prefix, or we could create tables specifically for testing and then delete them).

- **Behavior-Driven Tests (BDD):** We wrote Cucumber scenarios in a feature file to describe requirements in plain language. For example, the search functionality was captured as:

  Feature: Book Search
  Scenario: User searches for a book by title
  Given the bookstore has "Clean Code" by Robert Martin in the catalog
  When the user searches for "Clean Code"
  Then the system shows "Clean Code" in the search results with its author name

  We implemented step definitions in Java that correspond to these Given/When/Then steps. For the "Given" step, we ensured that the DynamoDB Books table had an item with title "Clean Code" and author "Robert Martin". (This was either done by calling a test-specific data loader or relying on initial data loaded at app startup for tests). The "When" step simulated the user action: this could be done by calling the controller's search method directly or performing an HTTP request to the search endpoint using MockMvc (a Spring testing utility to simulate web requests). We chose to use MockMvc to mimic an actual web call, e.g.,
  `mockMvc.perform(get("/search").param("query", "Clean Code"))`
  and then capture the response. The "Then" step checked that the response contained the expected book title and author. This might involve parsing the view model or checking the HTML content returned. We structured these tests such that they can run automatically as part of `mvn test`.

- **Test Coverage:** We monitored coverage using JaCoCo. The initial coverage was modest (around 50% of instructions). We increased it by adding tests for critical logic like order placement (ensuring that after an order is placed, the Orders table indeed has the record and the cart is empty, etc.), and for the recommendation retrieval (ensuring that if we put certain recommendations in the table, the service returns them correctly). Some parts of the code, like the controllers, were tested indirectly via integration tests rather than unit tests (since they mostly delegate to services, which were unit tested). The coverage could be improved further, but given the time constraints of a 10-day project, we focused on at least one happy-path test for each major use case. The BDD tests in particular ensure end-to-end behavior of the system matches the requirements (e.g., searching yields results, adding to cart affects the cart total, etc.).

The testing phase helped catch several issues: for example, initially our search was case-sensitive, and the BDD test using lowercase vs uppercase query revealed this, leading us to adjust the search logic to ignore case (by either converting queries and data to lower-case on compare or using a contains that's case-insensitive). Another issue discovered was concurrency on the Cart object – we realized if two requests tried to modify the cart simultaneously (unlikely in a single-session single-user scenario, but could happen in a multi-threaded test), it could cause issues. In a web context, that's not a real problem since a user's actions are sequential, but it's something to keep in mind if we expanded to allow multiple users (threads) in tests.

4.5 DevOps and Deployment

After development and testing, we addressed deployment and continuous integration aspects:

- **Version Control:** The code was managed in a Git repository (on GitHub under the account MinotaurG, repository *Online_Bookstore_Application*). Commits were made for each major milestone (initial scaffold, adding search, adding cart, etc.).
- **Continuous Integration (CI):** A Jenkins pipeline was set up to automate the build and test process. The Jenkinsfile (or job configuration) included steps to checkout the repository, run `mvn clean install` (which compiles code and runs all tests), and archive the built JAR artifact. We configured Jenkins to run this pipeline on each push to the main branch. This ensured that any integration issues or test regressions would be caught immediately. The Jenkins pipeline output, including test results and JaCoCo coverage report, was monitored to ensure code quality remained acceptable (we set a minimum threshold for tests to not drop coverage percentage, for example).
- **Deployment:** The final artifact is a Spring Boot fat JAR, which can be executed directly. For deployment, we used a Linux VM (Ubuntu 20.04) as the host. On the server, we installed Java 17 runtime. We also installed DynamoDB Local on the server (simulating what an AWS DynamoDB service would be). We chose to use DynamoDB Local on the server as well, to keep the system self-contained and not require an AWS account for the demo. We did note that using the actual DynamoDB service would be a matter of changing the application configuration (endpoint and credentials).

Deployment steps included: transferring the JAR to the server (via SCP or using Jenkins if we set up a deploy stage), running it in the background (using `nohup java -jar online-bookstore.jar &` or a systemd service for long-running). We also copied over the DynamoDB Local files and started that (on default port 8000). Since the app was already configured to hit localhost:8000 for DynamoDB, everything came up smoothly. We opened the appropriate port (8080) on the server's firewall so the app was accessible to intended users (for example, if demonstrating to an instructor).

- **Demonstration:** On Day 10 of the plan, we conducted a live demo. In the demo, we showcased the main flows: searching for a book, adding to cart, checking out, and seeing the order confirmation. We also demonstrated the recommendation feature by showing how a user's recommended list appears. Because we controlled the data, we could set up a scenario: e.g., for user "Alice", we put a recommendation of "Design Patterns" and "Clean Code" in the Recommendations table. Upon login as Alice, those showed up and we demonstrated clicking on them to view details. The recently viewed list was shown by navigating through a few books, then pointing out the sidebar updated accordingly. The demo confirmed that all functional requirements were met.

From a DevOps perspective, the project is small, but these practices (CI/CD, automated tests, infrastructure-as-code if we had more time) set a foundation for maintainability. If this project were to continue, we'd consider containerizing the app (Dockerizing the Spring Boot app and using a DynamoDB Local Docker for local testing). For a production-ready deployment, using AWS services like Elastic Beanstalk or ECS for the app and the actual DynamoDB service (fully managed by AWS) would be advisable. Logging and monitoring would also be added (we did simple console logging; in production, we'd integrate a logging framework and perhaps use CloudWatch or ELK stack for monitoring).

5. Conclusion

In conclusion, the Online Bookstore Application successfully implements a functional e-commerce prototype using a DynamoDB-only backend. All core features – from searching books to placing orders – have been realized and tested. The use of Amazon DynamoDB (Local for development) proved to be a fitting choice, providing a schema-flexible and easily deployable data store that met the project's requirements without the overhead of managing a separate SQL database.

We have shown how traditional online bookstore components (users, books, carts, orders) can be mapped to a NoSQL paradigm. Despite not using MySQL or a relational schema, the application maintains data consistency and supports the required queries through careful design (including the use of secondary indexes and denormalization where appropriate). This validates the decision to pivot away from the initially planned relational design and embrace a modern cloud-oriented database solution.

Through this project, we also demonstrated good software engineering practices: drawing up front the requirements and UML diagrams (use case, class diagram, etc.), then following through with iterative development and testing. Each module was built and verified against the plan (as outlined in the 10-day project plan). By Day 10, we integrated the components and delivered a live demo, which was a success – the system performed as expected in front of the stakeholders.

Key Learnings:

- **NoSQL Data Modeling:** We learned to model relational concepts in DynamoDB, including when to embed data vs. when to use separate tables and how to query effectively using Global Secondary Indexes. **Our implementation of GSIs for searching books by title/author and retrieving orders by userId demonstrated the importance of understanding access patterns before structuring DynamoDB tables.** By designing the schema around common query patterns and implementing appropriate GSIs, we achieved efficient lookups without sacrificing the flexibility of NoSQL. This experience highlighted that proper index design in DynamoDB can provide query performance comparable to or better than traditional relational databases for specific access patterns.
- *Spring Boot Development:* Using Spring Boot accelerated our development. We leveraged its autoconfiguration and sensible defaults. We also used Spring MVC to quickly set up web endpoints and Thymeleaf for dynamic views, which nicely separated our presentation from logic.
- *Integration Testing with DynamoDB Local:* The ability to run a local instance of DynamoDB for tests was extremely valuable. It allowed us to test the full stack (from HTTP request down to database and back) without external dependencies. This gives confidence that the system will behave similarly when pointed to the real DynamoDB service.
- *Time Management & Feature Scope:* Given the 10-day timeline, we had to prioritize features and occasionally simplify (e.g., no persistent cart, no actual payment). This taught us to maintain a clear definition of “done” for each requirement and to build a vertical slice of functionality that is demonstrable, even if not all bells and whistles are present. It’s better to have a complete, working subset than many half-implemented features.
- *Continuous Integration benefits:* Setting up CI early meant every code change was validated. This prevented integration problems and regressions, especially when multiple features were being developed in parallel. It also enforced discipline in writing tests alongside code.

Future Work:
The project can be extended in several ways if continued. Some immediate next steps would be to implement more user account features (password reset, profile management), add an admin module (to add new books to the catalog through the UI, rather than just directly in the database), and enhance the recommendation logic (perhaps using collaborative filtering or an AI service to generate recommendations based on user behavior). Additionally, improving the front-end with a richer interface and possibly developing a companion mobile app (which could consume the same backend via REST APIs) would make the application more complete. Finally, moving the deployment to the cloud (AWS) with proper infrastructure (using AWS DynamoDB, EC2 or AWS Lambda for the backend, etc.) would be a great learning exercise in scalability and cloud architecture.

Overall, the Online Bookstore Application meets its objectives as a training project, illustrating a full development lifecycle from requirements gathering to deployment. It stands as a solid foundation on which more advanced features can be built, and it showcases the use of DynamoDB as a viable option for web application persistence, even in scenarios traditionally dominated by relational databases.