Product Requirements Document (PRD): Bookmarked

Version: 1.5
Date: June 20, 2025
Project Name: Bookmarked

1. Overview & Vision

1.1. Vision:

To build "Bookmarked," a scalable, cloud-native application powered by a standalone, production-grade web API. The project will serve a dual purpose: providing a rich, personalized book discovery experience for readers via a web interface, and offering a robust, well-documented API for potential third-party developers.

1.2. Problem Statement:

Readers need better tools for discovering books that match their tastes. Developers need access to well-built recommendation APIs. This project addresses both needs by creating a powerful backend service and a polished frontend application that consumes it, all built on a modern, multi-database architecture.

1.3. Target Audience:

- **Readers (Primary):** Avid and casual readers looking for personalized recommendations and a community connection.
- **Developers (Secondary):** Developers who could potentially integrate the Bookmarked API into their own reading applications.
- **Recruiters (Implicit):** Hiring managers evaluating skills in API design, system architecture, cloud services, and DevOps.

2. User Personas & Stories

- **Alex, The Avid Reader:** "I want a system that understands my tastes and can show me *why* a book is recommended, such as 'People who loved *Dune* also loved this.'"
- **Ben, The Developer:** "I'm building a mobile reading tracker. I want a reliable API I can call to get book metadata and recommendation data for my users without building the backend myself."
- **Chloe, The Hiring Manager:** "I'm looking for engineers who build for production. I want to see a well-architected, deployed application with clear API documentation, robust testing, and automated CI/CD."

3. Features & Requirements (MVP)

Feature ID	Feature Name	User Story / Description	Requirements & Acceptance Criteria
F-101	**User Authentication**	As a user, I want to sign up and log in using my Google account for a seamless experience.	- Users can authenticate via OAuth2 using Google. - Integration is handled by AWS Cognito. - User profile data is stored in PostgreSQL.
F-102	**Book Catalog & Search**	As a user, I want to browse and search for books so I can find specific titles to rate.	- API provides paginated book lists. - API supports search by title and author. - Frontend displays results in a responsive grid.
F-103	**Trending Books**	As a user, I want to see a list of globally top-rated or trending books to discover popular titles.	- An API endpoint GET /api/books/trending is available. - This endpoint returns the top 20 books based on average rating and number of ratings. - The frontend displays this list on the homepage for all users.
F-104	**Book Rating System**	As a user, I want to rate books on a 1-5 star scale to get better recommendations.	- Rating is written to PostgreSQL (for record-keeping) AND to Neo4j as a [:RATED] relationship (for real-time queries).
F-105	**Personalized Recommendations**	As a user, I want to see a list of books recommended	- Recommendations are generated by the Python collaborative

		specifically for me based on my ratings history.	filtering model. - Requires the user to have rated at least 10 books.
F-106	**Graph-Powered Recommendations**	As a user viewing a book, I want to see what other books are liked by people who also liked this book.	- On the Book Details page, a section shows real-time recommendations. - Powered by a real-time Cypher query to the Neo4j database.
F-107	**API Documentation**	As a developer, I want to view clear, interactive documentation for the API so I can understand how to use it.	- The Spring Boot backend automatically generates an OpenAPI 3.0 specification. - Interactive documentation is publicly available via a /swagger-ui/index.html endpoint.

4. Technical & Operational Requirements

This section defines the non-functional, engineering-focused requirements that ensure the project is production-grade.

- **Testing Strategy:**
  - **Unit & Integration Tests:** The Java backend will be tested using JUnit and Mockito.
  - **Test Coverage:** The target for server-side test coverage is **>85%**.
- **Secrets Management:** All sensitive credentials (database passwords, API keys, OAuth secrets) will be managed using **AWS Secrets Manager**, not hardcoded or stored in Git.
- **Logging & Monitoring:**
  - Application logs will be streamed to **AWS CloudWatch Logs**.
  - Key application metrics (e.g., latency, error rates) and infrastructure health will be monitored via **AWS CloudWatch Metrics**.
  - CloudWatch Alarms will be configured to notify on critical events like a spike in 500 errors or sustained high latency.
- **Containerization & Deployment:**
  - All services (Java API, Python ML) will be containerized using **Docker**.
  - Deployment will be to **AWS ECS Fargate**, orchestrated with **AWS CodeDeploy** for safe deployments (e.g., Blue/Green).

5. Success Criteria

The project will be considered a success for the portfolio when the following are met:

1. The full application (frontend, Java API, Python service) is deployed and publicly accessible on AWS.
2. All MVP features are fully functional.
3. The backend API is clearly documented with a live Swagger/OpenAPI endpoint.
4. The backend has a unit and integration test suite with >85% code coverage.
5. The GitHub repository contains a comprehensive README with setup instructions and an architecture diagram.
6. The CI/CD pipeline automatically deploys changes from the main branch to production.

6. Tech Stack Summary

- **Frontend:** React with TypeScript
- **Backend API:** Java 21 & Spring Boot 3
- **ML Service:** Python 3 & FastAPI
- **Databases:** PostgreSQL (Relational) & Neo4j (Graph)
- **Authentication:** AWS Cognito
- **Cloud & DevOps:** AWS (ECS Fargate, RDS, S3), Docker, GitHub Actions

7. Recommended Development Roadmap for a Solo Engineer

This roadmap is designed to be iterative, ensuring tangible progress at every stage.

Phase 0: Foundation & Connectivity (The "Hello, World!" Phase)

- **Goal:** Establish and validate the connections between all core components of the stack.
- **Tasks:**
  1. Initialize a Git repository for the project.
  2. Set up local development environments for PostgreSQL and Neo4j using Docker Compose.
  3. Initialize the Spring Boot project, including dependencies for Web, JPA (PostgreSQL), and the Neo4j driver.
  4. Initialize the React (TypeScript) project.
  5. Create a single Book entity and a REST endpoint (GET /api/books) in Spring Boot to fetch placeholder data from PostgreSQL.
  6. Create a status endpoint (GET /api/status/graph) that confirms a successful connection to the Neo4j database.
  7. In the React app, create a basic component that successfully fetches and displays data from both new endpoints.
- **Outcome:** A validated, end-to-end connection from the frontend, through the backend, to both databases, proving the core setup works.

Phase 1: Backend-Driven MVP (The "API First" Phase)

- **Goal:** Build out the entire backend logic and test it independently of the frontend.
- **Tasks:**
  1. **Data Modeling & Population:** Define all JPA entities (User, Book, Rating, Tag) for PostgreSQL and the graph model (:User, :Book nodes; [:RATED] relationships) for Neo4j. Write one-time scripts to parse the Goodbooks-10k CSVs and populate both databases.
  2. **API Documentation:** Integrate Swagger/OpenAPI from the start. Annotate all endpoints as they are created.
  3. **Authentication:** Integrate AWS Cognito with Spring Security to handle OAuth2 user authentication.
  4. **API Endpoint Development:** Build all endpoints required for the MVP features (Search, Rating, Trending, Graph-Powered Recommendations). The rating endpoint must perform a dual-write to both PostgreSQL and Neo4j.
  5. **Testing:** As services and endpoints are built, write corresponding JUnit and Mockito tests to meet the >85% coverage target. Use a tool like Postman to perform integration tests on the live local API.
- **Outcome:** A complete, standalone, tested, and well-documented REST API that is ready for frontend consumption.

Phase 2: Frontend Integration (The "Make it Real" Phase)

- **Goal:** Build the complete user interface and connect it to the now-stable backend API.
- **Tasks:**
  1. **Component Scaffolding:** Create the full suite of React components required for the UI (e.g., Navbar, BookGrid, BookDetailsPage, LoginPage, RecommendationsPage).
  2. **Authentication Flow:** Implement the client-side authentication flow. On successful login via the backend, securely store the session/token and include it in subsequent authenticated API requests.
  3. **State Management:** Choose and implement a state management solution (e.g., React Context, Zustand) to handle global state like the logged-in user.
  4. **Connecting the UI:** Wire up all UI components to their corresponding API endpoints. Ensure data is fetched, displayed correctly, and user actions (like rating a book) successfully call the API and update the UI.
- **Outcome:** A fully functional web application running locally that fulfills all user-facing MVP requirements.

Phase 3: The Collaborative Filtering Engine (The "Magic" Phase)

- **Goal:** Build, containerize, and integrate the separate Python-based machine learning service.
- **Tasks:**
  1. **Model Training:** In a Jupyter Notebook, explore the dataset and train a collaborative filtering model (e.g., using matrix factorization with Scikit-learn). Save the trained model artifact (.pkl file).
  2. **ML API Service:** Create a new FastAPI project. Build an endpoint (e.g., GET /recommendations/{user_id}) that loads the saved model and returns a list of recommended book IDs.
  3. **Containerization:** Write a Dockerfile for the FastAPI service.
  4. **Integration:** Modify the main Spring Boot backend. The /api/recommendations/for-you endpoint will now make a server-to-server HTTP call to the Python ML service to get the list of book IDs, then query its own PostgreSQL database to enrich those IDs with full book details before returning them to the frontend.
- **Outcome:** The "For You" recommendations page is now powered by the ML model, and the application has a complete microservices architecture.

Phase 4: Cloud Deployment & CI/CD (The "Go Live" Phase)

- **Goal:** Deploy the entire multi-service application to the cloud and automate the deployment process.
- **Tasks:**
  1. **Dockerize the Backend:** Write a Dockerfile for the main Spring Boot application.
  2. **Infrastructure as Code (IaC):** Use the AWS CDK (or Terraform/CloudFormation) to define all cloud resources in code. This includes AWS ECS Fargate services and task definitions for both containers, an AWS RDS instance for PostgreSQL, an AWS S3 bucket, configuration for AWS Secrets Manager, and setting up AWS CloudWatch for logging and monitoring.
  3. **Cloud Database Setup:** Provision the RDS instance. For Neo4j, provision a cloud-hosted instance (e.g., from AuraDB) and configure the application to connect to it securely using credentials from AWS Secrets Manager.
  4. **CI/CD Pipeline:** Create a GitHub Actions workflow. Configure it to trigger on a push to the main branch. The workflow will: build the Docker images for both services, push them to Amazon ECR, and then trigger an AWS CodeDeploy process to update the Fargate services with the new images.
- **Outcome:** "Bookmarked" is live on the internet. The entire infrastructure is managed as code, and a fully automated CI/CD pipeline is in place for future updates.