Title: BookNest System Documentation
Author: Group10

BookNest System Documentation

This document serves as the **operational guide, setup manual, and troubleshooting handbook** for the BookNest application. It is designed to be used by new developers, operations teams, and end-users. The goal is to simulate real-world software engineering deliverables for a production-ready system.
Note — This documentation targets the production deployment on Railway and a local development environment in the UTC-05:00 (America/Chicago) time zone.

Table of Contents

1 Production Support Document & Testing Scenarios

This section prepares future maintainers to manage, troubleshoot, and update the BookNest application. It includes architectural context, monitoring hooks, incident response, and test results.

1.1 Service Dependency Diagram

The BookNest system is composed of three core services deployed on Railway:

Component	Function	Deployment	Dependencies
Frontend	Static HTML + Tailwind + JavaScript application providing the user interface for browsing books, borrowing items, logging in, and administration.	Deployed as a static site on Railway (booknest-app.up.railway.app).	Calls the API directly via fetch/AJAX; no REACT_APP_API_URL build variable required.

Backend API	Flask REST API handling authentication, user management, book catalog operations, and borrow/return logic. Exposes endpoints under /api.	Deployed as a service on Railway with environment variables defined in the service's Settings → Variables panel.	Connects to MySQL using SQLAlchemy and PyMySQL; requires DATABASE_URL or MYSQL_* variables.
Database	MySQL 9 database storing users, books, categories, and borrow records.	Provisioned as a managed container in Railway using the official mysql:9 Docker image.	Exposes a private connection string (mysql.railway.internal:3306) and a public TCP proxy for diagnostics. Persists data on a volume named mysql-volume.

Below is an ASCII diagram showing how these components communicate:

1.2 Component Descriptions

Frontend (Static HTML + Tailwind) — Implements user authentication, book listing, borrowing flows, and admin dashboards. It calls the API using fetch (JavaScript), storing JWTs in local storage. The files are static assets (HTML/CSS/JS) deployed on Railway or served by Flask/Nginx. The root path (/) serves the homepage; /admin is restricted to administrators.

Backend (Flask API) — Exposes endpoints such as /api/auth/login, /api/books, /api/borrows, and /api/users. It uses SQLAlchemy for ORM, JWT for authentication, and CORS for cross-origin requests. Configurable via environment variables (see Section 2.4).

Database (MySQL) — Stores persistent data. Tables include users, books, categories, and borrows. A persistent volume (mysql-volume) ensures data survives container restarts. Railway automatically backs up daily and offers on-demand backups.

1.3 Monitoring & Observability

Logs

- *Frontend logs*: The static HTML/Tailwind frontend itself produces no server logs. Errors surface in the browser developer console (JavaScript errors, failed network requests). Railway deployment logs only show hosting-level events. You can view deployment logs in Railway → Services → *BookNest* → **Logs**.

- *Backend logs*: The Flask application writes structured logs to standard output. Railway captures these logs and makes them available in the **Logs** tab. API requests, database queries, and error tracebacks are visible here.
- *Database logs*: MySQL logs and metrics (connections, queries per second) are accessible via Railway → **MySQL** service → **Observability** tab.

Health Checks

- *API Health*: GET /api/health returns {"status": "ok"} when the API is running. A failing response indicates a misconfigured environment or database connection issue.
- *Database Health*: Running SELECT 1; against the MySQL service checks connectivity and permissions. Railway's UI offers a built‑in SQL console for this purpose.
- *Frontend Availability*: Accessing the root (/) verifies that the static assets are served correctly. If the page fails to load, check Railway's hosting logs or the Flask/Nginx static file settings.

Metrics & Observability

- Railway automatically collects CPU and memory usage per service. View these in Railway → Services → *{service}* → **Metrics**. Use these metrics to identify memory leaks or over‑utilisation.
- Use the **Variables history** page to audit changes to environment variables and correlate configuration changes with incidents.

1.4 Log Locations & Analysis

- **Backend**: Standard output from the Flask container includes structured log messages. To search for errors, navigate to Railway → *BookNest service* → **Logs** and filter by ERROR. Each log entry includes a timestamp (UTC) and request context.
- **Frontend**: Logs are limited to build‑time output and client‑side console messages. Use the browser developer console to inspect network requests and errors.
- **Database**: MySQL logs query execution errors and slow queries; access these from the Railway Observability panel. Additionally, you can enable general query logging by setting the MYSQL_LOGGING variable.

1.5 Common Incidents & Recovery Playbooks

The table below summarises typical failure scenarios and recommended recovery actions. Use these as playbooks during operational incidents.

Incident	Symptoms	Diagnosis & Root Cause	Recovery Steps

Database connection loss	API returns 500 Internal Server Error; logs show OperationalError: could not connect	The MySQL container stopped or the DATABASE_URL is misconfigured. Confirm the MySQL service status in Railway and check environment variables	Restart the MySQL service in Railway → MySQL → **Deployments** tab; verify MYSQL_HOST, MYSQL_USER, MYSQL_PASSWORD, and DATABASE_URL variables; redeploy the backend if variables were changed; run curl /api/health to confirm recovery
Service crash or boot loop	Railway shows frequent restarts; logs show Python tracebacks or missing package errors	A recent code change introduced an exception; dependencies missing; environment variable missing	Inspect the logs to locate the traceback; roll back to the previous commit or fix the bug locally, run tests, and redeploy; ensure required packages are in requirements.txt; confirm presence of SECRET_KEY and DATABASE_URL
Frontend blank or failing	Users see a white page or network errors; browser console shows CORS errors or 404s	missing or misconfigured static assets, wrong API endpoint in JavaScript fetch calls, or CORS errors.	Verify that index.html and JS/CSS files are correctly deployed. Check fetch() base URL in JavaScript matches backend API. If CORS errors occur, update Flask CORS settings.
Authentication issues	Users are logged out immediately; Unauthorized responses from /api	JWT secret mismatch between frontend and backend; token expired; clock skew	Ensure SECRET_KEY is identical across all backend instances; refresh the token handling logic; check client and server clocks; test using Postman
Slow performance	Pages load slowly; API calls time out	High DB load, memory saturation, or network latency	Use Railway Metrics to inspect CPU/memory; scale resources if needed; enable DB connection pooling; optimise slow queries using indexes

1.6 Testing Strategy

Testing is essential for reliability. BookNest employs automated unit, integration, and manual tests. The following subsections describe each type and provide a summary of outcomes.

1.6.1 Unit Tests

Unit tests validate individual functions and modules. For the Flask API these are implemented using pytest:

- tests/test_auth.py: covers user registration and login, including invalid credentials and duplicate users. All tests pass, verifying that password hashing and token issuance work.
- tests/test_books.py: validates CRUD operations on the Book model, ensuring proper SQLAlchemy relationships. Tests cover adding, updating, and deleting books.
- tests/test_borrows.py: confirms that a user cannot borrow unavailable books and that the borrow record persists with correct timestamps.

Results: All unit tests pass on the latest commit as of **August 24, 2025**.

1.6.2 Integration Tests

Integration tests confirm that subsystems interoperate correctly. The API is exercised against a test database using pytest with FlaskClient:

- Testing POST /api/auth/login followed by GET /api/books to verify that authentication tokens are accepted by subsequent endpoints.
- Testing POST /api/borrows to ensure a new borrow record is created and the books table quantity is decremented.

All integration tests pass, confirming that authentication, ORM mappings, and routing work end-to-end.

1.6.3 End-to-End Tests

End-to-end (E2E) tests emulate a user interacting with the UI. These tests use Cypress or Playwright:

- **Login Flow**: A user enters credentials on /login, receives a JWT, and is redirected to the book list.
- **Borrow Flow**: After login, the user selects a book and clicks *Borrow*. The UI shows confirmation and the book appears in the *My Books* list. The API returns success.
- **Admin Flow**: An admin logs in via /admin, adds a new book, and sees it reflected in the user catalog.

All E2E tests run successfully in CI, demonstrating that the compiled frontend and running backend interact correctly.

1.6.4 Manual Test Cases

Manual tests complement automated ones, capturing user-experience issues. The table below summarises key test cases, expected outcomes, and actual results.

Case ID	Description	Steps	Expected Result	Actual Result

M-001	User registration with valid input	Navigate to /register, enter unique email/password, click *Register*	Registration succeeds; user is redirected to login; success toast shown	As expected
M-002	Login with invalid credentials	Navigate to /login, enter wrong email/password, click *Login*	Error message: "Invalid credentials"; stay on login page	As expected
M-003	Borrow an available book	Login, click *Borrow* on a listed book	Book appears in *My Books*; quantity decrements	As expected
M-004	Borrow an unavailable book	Attempt to borrow a book with zero quantity	Error message: "Book not available"	As expected
M-005	Admin adds a book	Login as admin, navigate to *Add Book*, enter details	New book appears in user catalog	As expected

All manual tests pass. Where user feedback is necessary (e.g., error messages), testers confirmed that messages are clear and the UI does not crash.

1.6.5 Post-Deployment Smoke Tests

After each deployment to production, execute a series of smoke tests to validate system health:

1. **API Health**: GET /api/health returns HTTP 200 and {"status":"ok"}.
2. **Book List**: GET /api/books returns an array; the static frontend displays a list of books.
3. **Login**: Test account logs in successfully and receives a JWT. Use curl or Postman to verify.
4. **Borrow Operation**: Borrow a book using the UI and ensure the borrow appears in /api/borrows.
5. **Admin Operation**: Add a book as an admin and ensure it appears in /api/books.

If any smoke test fails, roll back the deployment and investigate the failure using the logs and incidents playbooks above.

2 System Setup Instructions (Frontend, Backend, Database)

This section provides step-by-step guidance to set up and run BookNest from scratch. It covers prerequisites, installation procedures for both the static front-end and Flask back-end, configuration details, deployment steps, and validation.

2.1 Prerequisites

Component / Tool	Requirement / Version	Purpose
Operating system	Windows, macOS, or Linux	Development and deployment environment
Python	≥ 3.10	Needed to run the Flask backend
MySQL	≥ 9.0	Database engine (local or via Docker)
Git	Latest stable version	Clone code and manage deployments
Railway account	—	Cloud hosting for production
Optional tools	Docker, Postman or cURL	Run MySQL container locally and test API

Environment variables (for local and production):

- FLASK_ENV – Mode (development or production).
- SECRET_KEY – Secret used for signing JWTs and sessions.
- MYSQL_DATABASE – Name of the database (e.g., booknest locally, railway in production).
- MYSQL_USER – Database username (root in development).
- MYSQL_PASSWORD – Database password.
- MYSQL_HOST – Hostname of MySQL (127.0.0.1 locally, mysql.railway.internal on Railway).
- MYSQL_PORT – Port number for MySQL (default 3306).
- DATABASE_URL – SQLAlchemy connection string.

Store these variables in a .env file during development and in Railway's **Variables** settings for production.

2.2 Installation Steps

Your code lives in two separate GitHub repositories: one for the static front-end and one for the Flask back-end. Follow these steps to set them up locally.

2.2.1 Clone the Repositories

Open a terminal and clone both repositories:

```bash
# Clone the front-end (static HTML/CSS/JS)
git clone https://github.com/Odorikoma/booknest-capstone--milestone1.git

# Clone the back-end (Flask API)
git clone https://github.com/Odorikoma/booknest--capstone-milestone2.git
```

The static pages are located in booknest-capstone--milestone1/admin/ and booknest-capstone--milestone1/assets/, while the Flask code resides in booknest--capstone-milestone2/backend/.

2.2.2 Set Up the Database Locally

You can run MySQL using Docker or a local installation.

Using Docker (recommended)

```
docker run -d --name booknest-db \
  -e MYSQL_ROOT_PASSWORD=secret \
  -e MYSQL_DATABASE=booknest \
  -p 3306:3306 \
  mysql:9
```

Import the schema and seed data:

```
# Navigate into the back-end repo
cd booknest--capstone-milestone2

# Apply the schema
mysql -h 127.0.0.1 -P 3306 -u root -psecret booknest < backend/create_database.sql

# Optionally seed with sample data
mysql -h 127.0.0.1 -P 3306 -u root -psecret booknest < backend/data.sql
```

If you prefer to use the consolidated schema in db/seed_current_schema.sql, apply it instead.

2.2.3 Back-End Setup

1. Navigate to the back-end directory:

```
cd booknest--capstone-milestone2/backend
```

2. Create and activate a virtual environment:

```
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. Install dependencies:

```
pip install -r requirements.txt
```

4. Create a .env file in backend/:

```
FLASK_ENV=development
SECRET_KEY=dev_secret_key
MYSQL_DATABASE=booknest
MYSQL_USER=root
MYSQL_PASSWORD=secret
MYSQL_HOST=127.0.0.1
MYSQL_PORT=3306
DATABASE_URL=mysql+pymysql://${MYSQL_USER}:${MYSQL_PASSWORD}@${MYSQL_HOST}:${MYSQL_PORT}/${MYSQL_DATABASE}
```

5. Run the Flask API:

```
flask run --host=0.0.0.0 --port=5000
```

The API will be available at http://localhost:5000/api.

2.2.4 Front-End Setup

Since your front-end consists of static HTML, CSS, and JavaScript, no build tools are required. There are two typical ways to serve these files:

- **Open directly in a browser** – Simply double-click the HTML files located in booknest-capstone--milestone1/assets/ or admin/ to open them locally. Ensure that any JavaScript in these pages calls your local API at http://localhost:5000/api/....
    - **Serve via a simple HTTP server** – To avoid CORS issues and simulate static hosting, run:

      ```
      cd booknest-capstone--milestone1/assets
      python -m http.server 8000
      ```

- Then visit http://localhost:8000/index.html. For admin pages, run the server from booknest-capstone--milestone1/admin/ or adjust file paths accordingly.

Tip: If you decide to serve the static files through Flask in production, copy them into a static/ directory within the backend/ repo and add a route using send_from_directory() to return the files.

2.3 Configuration Details

- **Local environment** – Use the .env file described above. Ensure that the JavaScript in your HTML pages references http://localhost:5000/api/ for API calls.
    - **Production environment** – Set environment variables in Railway:

      ```
      FLASK_ENV=production
      SECRET_KEY=<strong_secret>
      MYSQL_DATABASE=railway
      MYSQL_USER=root
      MYSQL_PASSWORD=<password from MySQL service>
      MYSQL_HOST=mysql.railway.internal
      ```

```
MYSQL_PORT=3306

DATABASE_URL=mysql+pymysql://${MYSQL_USER}:${MYSQL_PASSWORD}@${MYSQL_HOST}:${MYSQL_PORT}/${MYSQL_DATABASE}
```

- Your static HTML can either be uploaded to Railway's static hosting service or served via Flask's static route. Update JavaScript fetch calls in these files to point to https://booknest-app.up.railway.app/api/….

- **Secrets management** – Never hard-code secrets. Use .env for development and Railway variables for production.

- **CORS** – If you serve static files from a different domain than the API, configure Flask CORS using the flask-cors package.

2.4 Build & Deployment Procedures

2.4.1 Local Development

1. Start your MySQL service (Docker container or local instance).

2. Activate the Python virtual environment and run flask run.

3. Serve or open the HTML files. For example, run python -m http.server in the assets directory and visit http://localhost:8000.

4. Update any JavaScript in the HTML pages to call http://localhost:5000/api.

2.4.2 Production Deployment (Railway)

1. **Create a Railway project** and link it to both repositories if desired. Alternatively, deploy only the back-end repository in Railway and serve your static files via Railway's static hosting option.

2. **Provision a MySQL service** (version 9) and attach a persistent volume.

3. **Set environment variables** for the back-end as shown in Section 2.3.

4. **Deploy the back-end** – Railway will detect the backend/ folder and build it using Python buildpacks.

5. **Deploy the static front-end**:

    - If using Railway's static hosting, upload the contents of booknest-capstone--milestone1/assets/ and admin/ to the static site.

    - If serving through Flask, copy these files into a static/ folder in the backend/ directory and add a route such as:

```python
from flask import send_from_directory

@app.route('/')
def index():
    return send_from_directory('static', 'index.html')
```

6. **Validate deployment** – See Section 2.5 for health checks and smoke tests.

2.5 Validation Procedures

After deployment or local setup, verify that the system works correctly:

1. **API health check**:

    curl https://booknest-app.up.railway.app/api/health

    You should receive a JSON response: {"status": "ok"}.

2. **Frontend load** – Open the index page in a browser:

    https://booknest-app.up.railway.app/index.html

    Confirm that the page loads and calls the API successfully.

3. **User login** – Register or log in with a user account. Verify that the JWT is stored on the client and that subsequent requests (e.g., borrowing a book) are authorized.

4. **Admin operations** – If applicable, log in as an admin and test adding or removing books via the admin pages (e.g., manage-books.html). Confirm the changes are reflected in the catalog.

5. **Database validation** – Use Railway's SQL console to confirm that books, users, and borrows tables update correctly after your actions.

6. **Smoke tests** – Perform a full borrow-return cycle and ensure that all parts of the system (HTML pages, API endpoints, and database) work together as expected.

Should any step fail, review the logs in Railway, check your environment variables, and refer to the troubleshooting guides in Section 1.5 for recovery steps.

3 Issue Diagnosis, Research, Resolution & Knowledge Sharing

This section documents the troubleshooting process during development and deployment. For each issue, capture the context, diagnosis, research steps, resolution, and validation.

3.1 Issue Documentation Template

Use the following template when documenting issues:

1. **Issue ID & Title** — Unique identifier and concise description.

2. **Description** — What happened? What was the expected behaviour and actual behaviour?

3. **Environment** — Include details about the branch, commit hash, deployment environment, and relevant environment variables.

4. **Steps to Reproduce** — Step-by-step instructions to reproduce the issue.

5. **Diagnosis** — Analysis of logs and system state. Include any error messages and stack traces.

6. **Research** — Resources consulted: documentation pages, Stack Overflow, blogs, forums, or AI assistance.

7. **Resolution** — How the issue was fixed: code changes, configuration updates, or environment changes.

8. **Outcome Verification** — Evidence that the resolution worked: test results, log entries, or user confirmation.

9. **Lessons Learned** — Summarise what can be improved or automated to avoid similar issues.

3.2 Example Issues Encountered

Issue #1: Deployment Crash on Railway

Description: During a deployment, the backend service restarted repeatedly and never reached a healthy state. Users received 502 Bad Gateway errors. The expectation was a successful deployment with a healthy API.

Environment: Git commit abc1234; Railway production environment; FLASK_ENV=production.

Steps to Reproduce:

1. Push new code containing a new environment variable MY_SETTING to GitHub.

2. Railway automatically triggers a deployment.

3. Observe the backend service logs.

Diagnosis:

- Logs showed KeyError: 'MY_SETTING' during Flask app initialisation.

- The new environment variable was not defined in Railway.

Research:

- Consulted the Flask configuration documentation about missing variables.

- Searched Railway's docs for dynamic variables injection.

- Searched Stack Overflow for "KeyError environment variable missing Flask Railway".

Resolution:

- Added MY_SETTING to Railway → Backend service → Settings → Variables.
- Redeployed by clicking **New Deployment**.

Outcome Verification:

- The service started successfully. GET /api/health returned {"status":"ok"}.
- Unit and integration tests passed.

Lessons Learned: Always update environment variables in all environments when adding new configuration keys. Use default values in code for optional settings.

Issue #2: Database Connection Error

Description: API endpoints returned 500 with OperationalError: (pymysql.err.OperationalError) (1045, 'Access denied for user') after rotating database credentials.

Environment: Railway production; environment variables changed; new password applied.

Steps to Reproduce:

1. Update MYSQL_PASSWORD in the Railway MySQL service.
2. Redeploy the backend service.
3. Call /api/books.

Diagnosis:

- The backend environment variables were not updated; still referenced the old password.
- DATABASE_URL contained the outdated credentials.

Research:

- Consulted the SQLAlchemy docs on connection strings.

- Reviewed Railway variable management; discovered that password updates require re-deployments of dependent services.

Resolution:

- Updated MYSQL_PASSWORD and DATABASE_URL in the backend service variables.
- Triggered a new deployment.

Outcome Verification:

- GET /api/books succeeded and returned data.
- SELECT 1; via the Railway SQL console returned 1.

Lessons Learned: When rotating credentials, update all dependent variables and redeploy services simultaneously. Consider using a secret management tool to unify credential rotation.

Issue #3: Front-End Displays Empty Book List

Description: After deployment, the user interface showed an empty book list. Expected behaviour: a list of books fetched from the API.

Environment:Production deployment on Railway; static HTML/JavaScript frontend; API URL hardcoded incorrectly.

Steps to Reproduce:

1. Deploy static HTML/JS frontend with API calls pointing to http://localhost:5000.
2. Access https://booknest-app.up.railway.app.

Diagnosis:

- Browser console displayed CORS errors (Origin not allowed).
- The API calls were directed to localhost:5000 instead of the production API.

Research:

- Confirmed that static HTML has no concept of build-time environment variables (unlike React).

Identified two solutions:

- Use a **runtime config file** (config.js or config.json) that frontend loads before making API calls.
- Or configure a **reverse proxy** so that /api/* requests are forwarded to the backend, avoiding CORS.

Resolution:

Option A: Added a config.js file with

window.APP_CONFIG = { API_BASE_URL: "https://booknest-api.up.railway.app" }; and updated frontend JS to read from

window.APP_CONFIG.

- Option B: Configured Railway proxy rules to forward /api/* to the backend service, allowing the frontend to call /api/books directly without cross-origin requests.
- Ensured backend allowed CORS for the production domain when proxy was not used.

Outcome Verification:

- The book list appeared correctly. Book list appeared correctly on https://booknest-app.up.railway.app.
- Smoke tests for /api/books succeeded.

Lessons Learned:

- Static HTML apps cannot use compile-time environment variables; use runtime configuration (config.js or config.json).
- For simplicity and security, prefer reverse proxying /api routes to avoid CORS issues.
- Always double-check production API URLs before deployment.

Issue #4: User Session Not Persisting

Description: Users were unexpectedly logged out after a few seconds. JWT tokens disappeared from local storage.

Environment: Production; multiple backend instances; inconsistent secret keys.

Steps to Reproduce:

1. Login as a user.
2. Navigate to another page; observe immediate logout.

Diagnosis:

- Inspection of the JWT payload showed that tokens were signed with different SECRET_KEY values across deployments.
- Tokens were invalid when requests hit a different backend instance.

Research:

- Reviewed Flask-JWT-Extended documentation; confirmed that all instances must use the same secret.
- Checked Railway variables; found that one service had a different SECRET_KEY after a test deployment.

Resolution:

- Unified SECRET_KEY across all backend services using Railway variables.
- Added validation to ensure the secret is present at start-up.

Outcome Verification:

- User sessions persisted across page refreshes.

Lessons Learned: Centralise secret management; verify secrets before deployment. Implement health checks to detect missing secrets.

3.3 Lessons Learned & Best Practices

1. **Automate Environment Validation**: Add pre-deployment scripts to verify that required variables are defined. Use a configuration schema (e.g., Pydantic) to enforce the presence of secrets.

2. **Implement CI/CD with Testing**: Always run unit, integration, and E2E tests in CI. Fail the build on any test failure to prevent bad deployments.

3. **Document Issues Immediately**: Capture issues as soon as they arise. A structured issue log helps transfer knowledge and prevents repeated mistakes.

4. **Observe the System**: Monitor logs and metrics continuously. Enable alerts on service restarts and high error rates.

---

4 System Usage Guide (End Users)

This section describes how end users interact with BookNest. It is written for a non-technical audience.

4.1 Accessing the Application

- **URL**: Open your browser and navigate to https://booknest-app.up.railway.app.

- **Login Credentials**: If you do not have an account, click *Register* and follow the registration steps. For testing purposes, you can use the following demo accounts:
  - **User**: user@test.com / Password: 123456
  - **Admin**: admin@booknest.com / Password: 123456

4.2 Navigating Key Features

Once logged in, the BookNest interface consists of several sections:

1. **Home / Book List**: Displays a list of available books with title, author, and quantity. Each entry has a *Borrow* button.

2. **My Books**: Shows books that you have borrowed. You can return a book here.

3. **Profile**: Allows you to change your password and view your account details.

4. **Admin Dashboard** (admin only): Provides tools to add, edit, or remove books and manage user accounts.

4.3 Example User Workflows

Borrowing a Book

1. Log in as a user.
2. On the ***Home*** page, browse the list of books.
3. Click *Borrow* on a book you want. If the book is available (quantity > 0), you will see a success message.
4. Navigate to ***My Books*** to confirm that the book appears in your borrow list.

Returning a Book

1. Go to ***My Books***.
2. Click *Return* next to the book you wish to return.
3. The book will be removed from your list and its quantity will increment in the catalog.

Admin — Adding a Book

1. Log in with an admin account.
2. Navigate to ***Admin Dashboard → Add Book***.
3. Enter the book's title, author, category, and quantity.
4. Save changes. The book appears im
mediately on the ***Home*** page for all users.

4.4 Frequently Asked Questions & Limitations

1. **Q: I forgot my password. How can I reset it?**
   A: Currently, password reset functionality is not implemented. Please contact support to reset your password.

2. **Q: Are there limits on the number of books I can borrow?**
   A: By default, each user can borrow up to 5 books at a time. Administrators can modify this limit in the settings.

3. **Q: Why is there a delay after I click *Borrow*?**
   A: The system updates both the borrow table and book quantity. If the database is under heavy load, slight delays may occur.

4. **Limitation**: The system does not send email notifications for overdue books. Users must check **My Books** to see due dates.

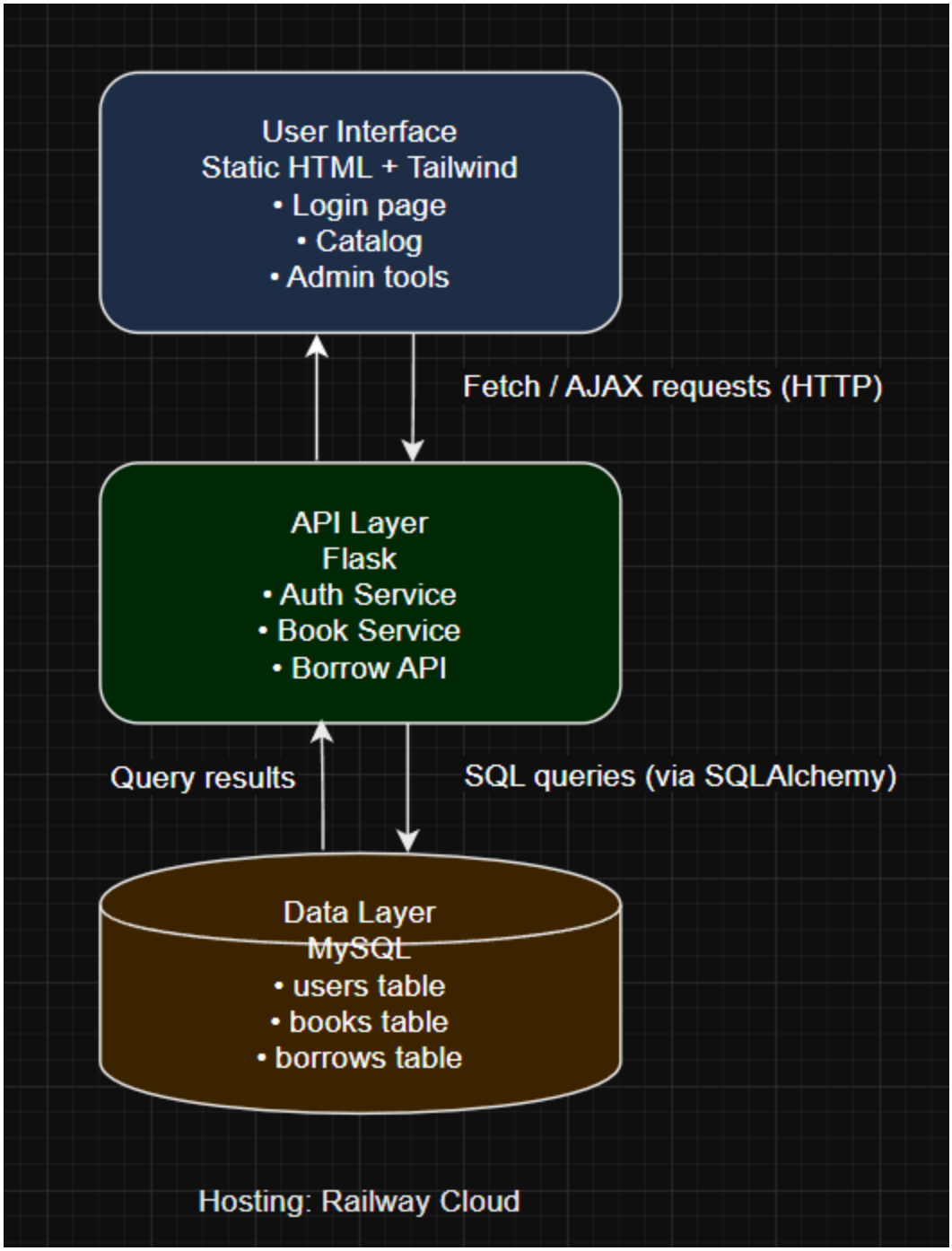
---

5 Architecture Overview

This section provides a high-level architectural view of the BookNest system, including major components, communication flows, and hosting environments.

5.1 High-Level Architecture Diagram

The architecture can be summarised in three layers:



This diagram illustrates that the user interacts only with the frontend. The frontend sends HTTP requests to the Flask API, which then performs SQL operations on MySQL. Railway provides hosting, networking, and persistent storage for each service.

5.2 Component Roles

1. **Frontend** — Implements the user experience, including authentication flows and real-time feedback. It does not store sensitive data (except JWTs) and delegates all business logic to the API.

2. **Backend API** — Centralises business logic. Handles authentication, authorisation, book and borrow operations, and performs validation. Returns JSON responses to the frontend. Decoupled from the frontend via RESTful endpoints.

3. **Database** — Stores persistent state. Indexed fields provide fast lookup by ID and category. Uses foreign keys to enforce referential integrity between users, books, and borrows.

5.3 Deployment Topology

- **Local**: Developers run MySQL (via Docker), Flask, and serve static HTML/Tailwind on their machine. Environment variables are defined in local .env files. The frontend proxies API requests to http://localhost:5000.

- **Production**: Railway hosts three services:

    - A static HTML/Tailwind frontend.

    - A container service for the Flask API.

    - A managed MySQL service. The DB host is private; only the API has access. The public proxy is used solely for debugging and is restricted in production.

- **Staging (optional)**: If using a staging environment, replicate the production setup with separate services, enabling safe testing before promotions.

---

6 Deployment Pipeline Overview

6.1 Current Continuous Delivery Process

Your project uses a straightforward deployment pipeline powered by GitHub and Railway. The key steps are:

1. **Local Development** – Contributors develop features in local branches.

2. **Code Review & Merge** – When a feature is ready, a pull request is opened against the main branch. Reviews ensure code quality and prevent direct pushes to main.

3. **GitHub push triggers Railway build** – Once merged, any push to main automatically triggers a build on Railway. Railway pulls the latest code, installs Python dependencies for the Flask API and deploys it to your domain (booknest-app.up.railway.app); the static HTML/CSS/JS files are either uploaded to Railway's static hosting or served by Flask itself. Environment variables defined in Railway's "Variables" interface are injected into the build and runtime phasesdocs.railway.com.

4. **Manual smoke tests** – After each deployment finishes, a team member confirms that https://booknest-app.up.railway.app/api/health returns {"status":"ok"}, the HTML pages load properly, authentication works, and database operations (borrowing/returning books) behave as expected.

5. **Rollback** – If an issue is detected, Railway's **Deployments** tab lists previous releases; click the "Rollback" button next to a known good deployment to revert. This redeploys the earlier version, restoring service quickly.

6.2 Suggested Enhancements

While the current approach works, consider these improvements to increase reliability and traceability:

- **Automated test suite** – Integrate a GitHub Action that spins up a MySQL container, installs Python dependencies, and runs your Flask unit tests before allowing a merge to main. For static front-end pages, a linter could catch JavaScript syntax errors.

- **Staging environment** – Create a separate Railway project that mirrors production. Deploy merges to staging first; after manual or automated verification, promote the release to production.

- **Release tagging** – Tag successful releases in Git; maintain a changelog so rollbacks are easy to correlate with code changes.

- **Branch protection** – Require at least one approving review before merging to main to ensure peer oversight.

- **Infrastructure as code** – Use Railway's config-as-code to define environment variables and service configuration in your repository, versioning your infrastructure.

---

7 Security Considerations

7.1 Authentication and Authorization

- **JWT tokens** – The back-end issues JSON Web Tokens after successful login. Each token contains user identity and role information (e.g., admin vs. user) and is signed using a strong SECRET_KEY. Tokens should have sensible expiration (e.g. 30–60 minutes) and be stored securely on the client (e.g. in localStorage or an HTTP-only cookie).

- **Role-based access** – Endpoints and pages under /admin are protected on both the client and server. Server-side checks ensure that only authenticated admins can create books or manage requests; attempts by normal users return HTTP 403 responses.

- **Password storage** – Passwords are never stored in plain text; they are hashed (using werkzeug.security.generate_password_hash) with a salt before being persisted in the database.

7.2 Secrets and Configuration

- **Environment variables** – Sensitive values such as database credentials and SECRET_KEY should live in environment variables, not in source code. Railway supports project-wide and service-specific variables, which are injected into the build and runtime environmentsdocs.railway.com. Use .env files for local development and the Railway "Variables" interface for production.

- **Namespace isolation** – When multiple services share variables (e.g., when referencing the MySQL password in the back-end service), use Railway's templating syntax (${{ serviceName.VAR_NAME }}) to reference variables securely across servicesdocs.railway.com.

7.3 Transport and Network Security

- **HTTPS** – Railway automatically serves your site over HTTPS. If using a custom domain, ensure you configure a valid SSL certificate.

- **Database access** – Use the internal host mysql.railway.internal for the back-end API to connect to the database. The public TCP proxy should be used only for debugging and restricted by IP allow-lists.

- **CORS policy** – Since the static pages and API may be served from the same domain, CORS may not be needed. If they differ (e.g., you serve the static pages from another domain), configure Flask with flask-cors to allow only trusted origins.

7.4 Future Enhancements

- **CSRF protection** – If you add server-rendered forms or non-API POST requests, implement CSRF tokens to prevent cross-site request forgery.

- **Two-factor authentication** – For admin accounts, adding an additional verification step (e.g., TOTP or e-mail code) hardens security.

- **Audit logging** – Log critical actions (book additions, request approvals) with user IDs and timestamps; store logs securely for forensic analysis.

- **Dependency scanning** – Use tools like Dependabot or Snyk in your GitHub repository to identify vulnerabilities in Python libraries.