DEVOPS CA

CCT College Dublin

Assessment Cover Page

Module Title:	Devops
Assessment Title:	Capstone Assignment
Lecturer Name:	Esteban Garcia
Student Full Name:	JAMES SCOTT
Student Number:	SBA24070
Assessment Due Date:	08/08/25
Date of Submission:	08/08/25

Declaration

By submitting this assessment, I confirm that I have read the CCT policy on Academic Misconduct and understand the implications of submitting work that is not my own or does not appropriately reference material taken from a third party or other source. I declare it to be my own work and that all material from third parties has been appropriately referenced. I further confirm that this work has not previously been submitted for assessment by myself or someone else in CCT College Dublin or any other higher education institution.

DEVOPS CA

Table of Contents

1. https://github.com/Jamesscott34/DevopsCA.git

2. PROJECT OVERVIEW

The Book Catalogue App is a full-stack web platform designed to enable users to manage, organize, and keep track of their personal or shared book collections. This application (Figure 1 and Figure 2) was developed as the central focus of my DevOps Capstone assignment at CCT College Dublin, with the primary aim of demonstrating practical experience with modern DevOps technologies. These technologies include continuous integration and delivery (CI/CD), containerization through Docker, orchestration with Kubernetes, and templated deployment via Helm.

At the heart of the application is the Django web framework, chosen for its scalability, flexibility, and built-in support for database models and RESTful APIs. For persistent data storage, I opted for PostgreSQL, which integrates well with Django and is widely used in enterprise-grade deployments. During the development lifecycle, I used Docker Compose to simplify the local development experience by bundling both the Django app and the PostgreSQL database into isolated, reproducible containers.

Once the application was running smoothly in Docker Compose, I moved to a more production-like environment using Minikube, a local Kubernetes cluster. This was then paired with Helm, which allowed me to deploy, upgrade, and manage my Kubernetes resources with greater flexibility and minimal manual repetition. I designed Helm charts that allowed environment-specific overrides, which is crucial for real-world use cases where development, staging, and production each have their own configurations.

To implement automation and reliability across the build and deployment stages, I configured GitHub Actions as my CI/CD pipeline. This workflow is responsible for automatically installing project dependencies, running unit tests, applying database migrations, building Docker images, and deploying them directly to the Kubernetes cluster. The CI/CD pipeline triggers on every push to the main branch, which guarantees that changes are immediately validated, packaged, and shipped without requiring any manual steps.

Rather than being just another Django CRUD project, this system represents a full end-to-end DevOps pipeline. Through it, I was able to practice everything from secure coding, version control, and automated testing to scalable deployment, infrastructure management, and secret handling. I learned how each tool in the DevOps toolkit fits into the software lifecycle and how to make them work together in harmony.

From a user perspective, the application provides a streamlined experience. Registered users can log in and manage their profile, add new books, edit existing ones, and categorize them using tags. Each book

can also have an optional cover image and is tracked for how often it is viewed or marked as read. ISBN validation is built-in to help prevent duplicate entries or data entry errors.

For administrators, the platform includes a dedicated dashboard that gives full control over all books and users. Admins can assign books to individual users, write notifications, and see analytics about reading trends and the most viewed titles. There is also a robust notification system, which ensures users receive real-time updates about assigned books or system announcements.

One of the standout features is the integration with the Open Library API. This allows users to search external databases for books and import their metadata including title, author, description, and cover image into their personal catalogue with a single click. This significantly reduces manual effort and improves data consistency.

The Book Catalogue App supports multiple deployment environments, all of which are backed by automated shell scripts. This approach allows the same codebase to be deployed in different contexts depending on the user's need whether they are a local developer, using Docker Compose in a shared team setting, or running the app on Kubernetes in a cloud-like setup.

Table 1 is showing a summary of the supported environments and the tools used in each:

Environment	**Startup Command**	**Access URL**	**Database**	**Approx. Setup Time**
Local Development	./custom_scripts/run_django.sh	**http://127.0.0.1:8000**	SQLite	~2 minutes
Docker Compose	./custom_scripts/run_docker.sh	**http://localhost:8000**	PostgreSQL	~3 minutes
Kubernetes (Minikube)	./custom_scripts/setup.sh	Minikube IP with NodePort	PostgreSQL	~5 minutes
Helm	(via helm install)	Minikube or cloud ingress	PostgreSQL	~5

Environment	Startup Command	Access URL	Database	Approx. Setup Time
Deployment		URL		minutes

Table 1 Summary of the supported environments and the tools

To support all these environments seamlessly, I created a small Python utility script called host_helper.py. This script dynamically changes the database host configuration depending on the target environment. For example, Django connects to localhost in local development mode, db in Docker Compose, and postgres in Kubernetes. This approach avoids hardcoded values and removes the need for manual intervention, which is especially important in automated pipelines.

Figure 1 *GitHub repository homepage*

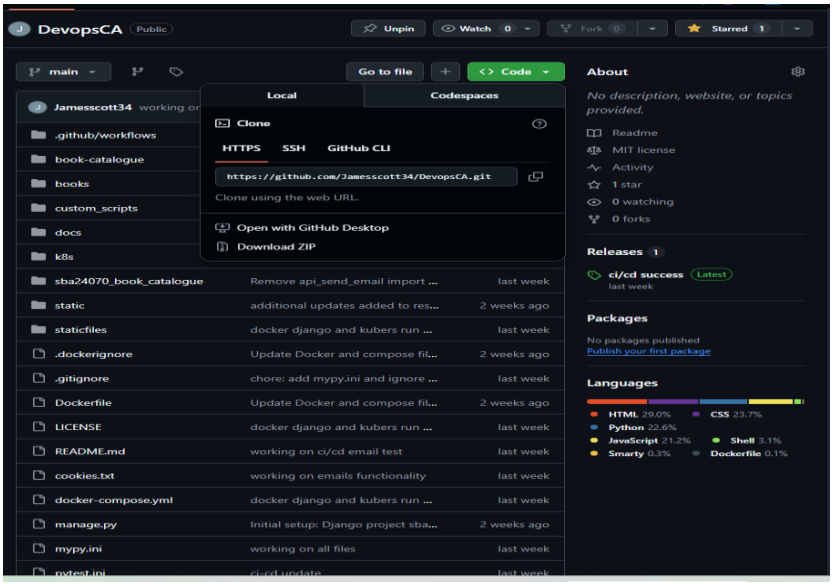
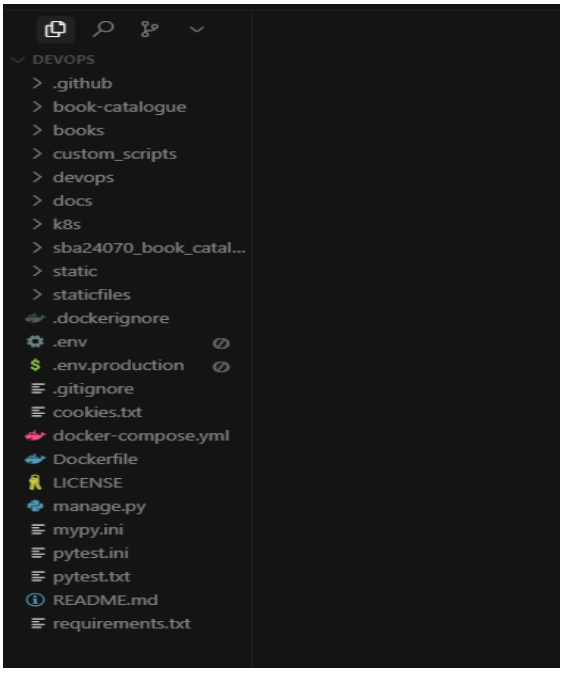


Figure 2 *Screenshot of the project structure.*


DEVOPS
.github
book-catalogue
books
custom_scripts
devops
docs
k8s
sba24070_book_catal...
static
staticfiles
.dockerignore
.env
.env.production
.gitignore
cookies.txt
docker-compose.yml
Dockerfile
LICENSE
manage.py
mypy.ini
pytest.ini
pytest.txt
README.md
requirements.txt

3. ARCHITECTURE

The system architecture for the Book Catalog App was carefully planned to support both local development and production-style deployment. From the beginning, I wanted to make sure that the way I wrote, tested, and deployed the application could scale as the project evolved. I started with a local development setup using Docker Compose and expanded it into a fully automated Kubernetes deployment pipeline that runs through GitHub Actions and Helm.

For day-to-day development, I run the entire application stack using Docker Compose. This environment includes two main containers: one for the Django web application and another for the PostgreSQL database. By containerizing both services, I avoided a lot of the usual set-up and compatibility issues that come with running software across different machines or operating systems. With a single command, I can bring up the database, web server, and static file support, and start building features immediately. This setup is defined in my docker-compose.yml file, and I added a custom script called run_docker.sh to simplify the launch process even further.

The real backbone of the project is how everything connects starting from writing code to running the app in Kubernetes. Once I complete a feature or fix and commit my changes, I push the update to GitHub, which triggers an automated CI/CD pipeline built with GitHub Actions. The pipeline handles everything automatically: it installs project dependencies, runs all the unit tests, checks for database migrations, and builds a Docker image from the current state of the application. That image is then pushed to DockerHub, my public image registry, where it becomes available for deployment.

What happens next is what makes this system more than just a coding exercise. Once the Docker image is published, the pipeline then uses Helm to deploy the latest version of the app into a local Minikube Kubernetes cluster. Helm is used to define the deployment strategy, service exposure, secrets, and environmental configuration, all from reusable templates. I created a Helm chart that controls everything from the number of replicas to the port configuration and even integrates secure credentials using Kubernetes secrets.

The overall architecture of the system follows a clear flow:

1)I write and commit code on my local machine.

2)I push that code to GitHub.

3)GitHub Actions automatically kicks off the CI/CD process.

4)Tests are executed, static checks are run, and a new Docker image is built.

5)The image is uploaded to DockerHub.

6)Helm then uses that image to perform a rolling update in the Kubernetes cluster.

All of this is done automatically with no manual deployment steps required. This makes it extremely easy to ship changes frequently and safely exactly what modern DevOps practices are designed to support.

In terms of deployment environments, the system supports running both in development (via Docker Compose) and in production-like environments (via Kubernetes). Locally, everything is containerized and predictable. In the Kubernetes cluster, the same app and database are deployed as separate pods, exposed via Kubernetes services. These services can be accessed using a Minikube service URL or optionally configured with Ingress for advanced routing.

One important part of this architecture is the way the system handles environment-specific configurations. I wrote a script called host_helper.py, which rewrites the database host settings in my .env file depending on whether I'm using Docker, Kubernetes, or local development. This ensures that Django knows where to look for the PostgreSQL database no matter where the app is running. Without this kind of flexibility, it would be difficult to maintain separate environments using the same codebase.

In summary, the Book Catalog App is built to support a full DevOps lifecycle. It combines modular development with automated deployment, scalable infrastructure, and flexible configuration. I found this architecture extremely valuable for testing real-world scenarios, improving my DevOps skills, and gaining experience in managing a production-ready application using open-source tools.

Figure 3 *Diagram showing the flow from developer to GitHub, DockerHub, and Kubernetes. Screenshots.*

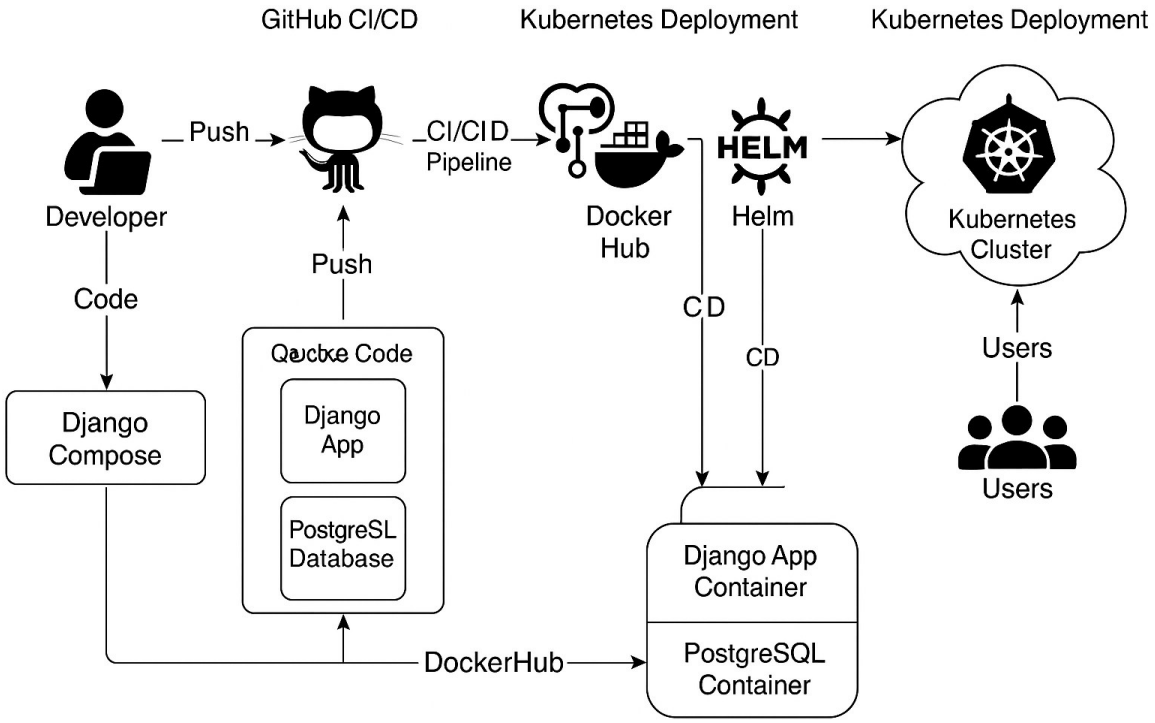
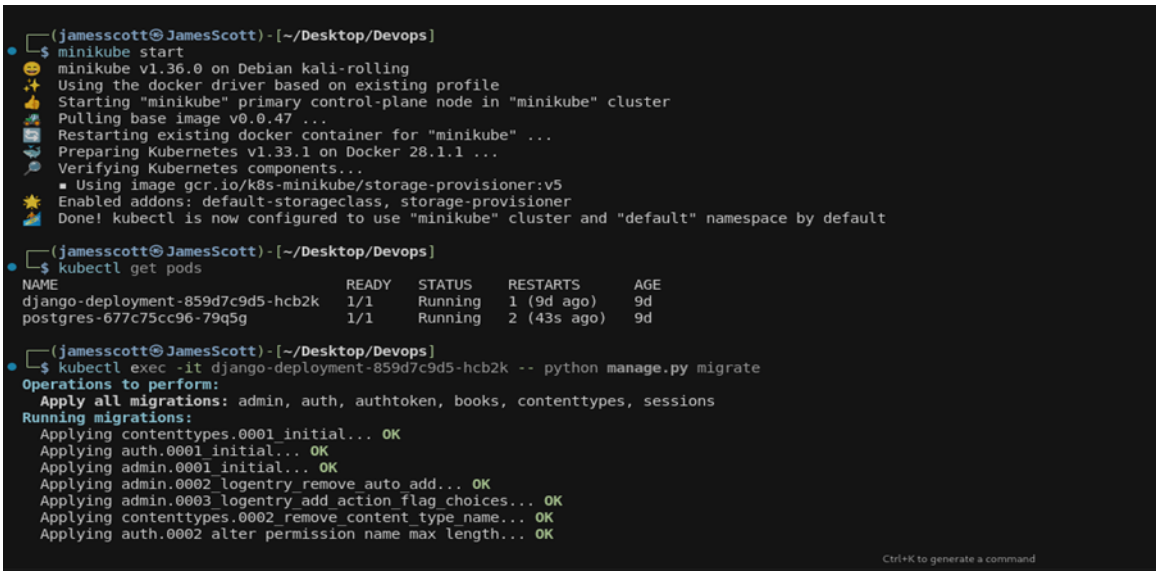
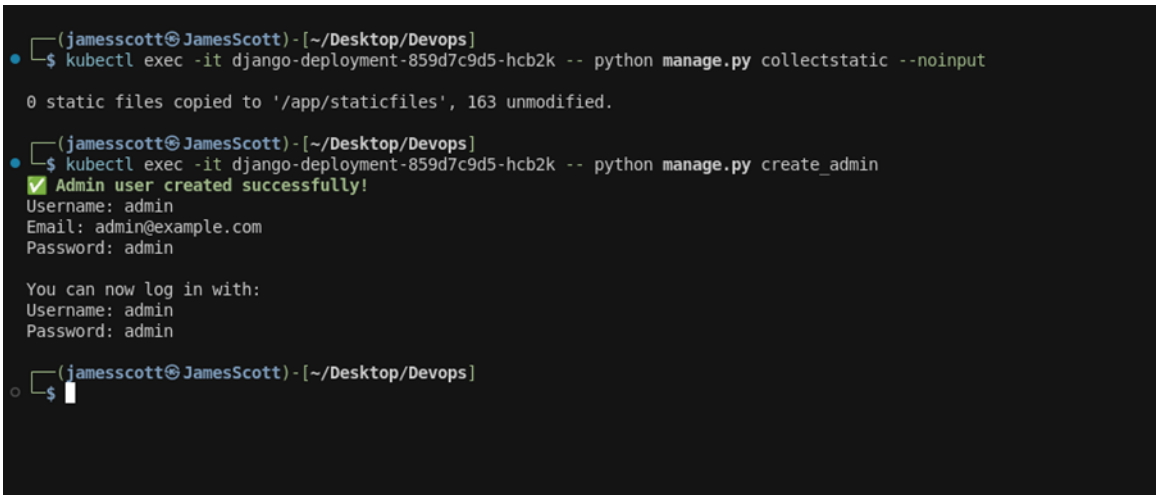
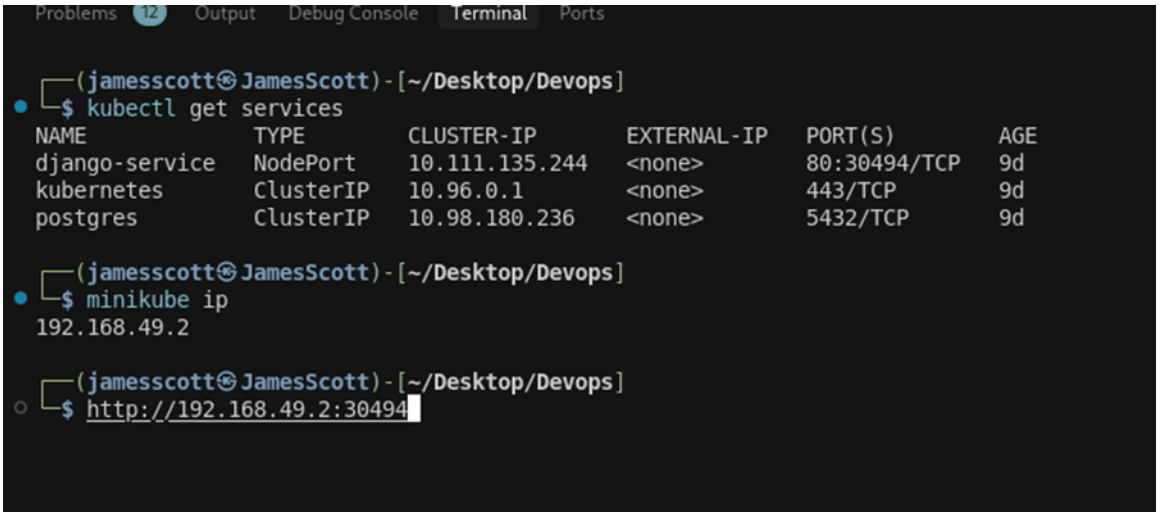


Figure 4 *starting Minikube*




(jamesscott㉿JamesScott)-[~/Desktop/Devops]
$ kubectl exec -it django-deployment-859d7c9d5-hcb2k -- python manage.py collectstatic --noinput
0 static files copied to '/app/staticfiles', 163 unmodified.
(jamesscott㉿JamesScott)-[~/Desktop/Devops]
$ kubectl exec -it django-deployment-859d7c9d5-hcb2k -- python manage.py create_admin
✅ Admin user created successfully!
Username: admin
Email: admin@example.com
Password: admin
You can now log in with:
Username: admin
Password: admin
(jamesscott㉿JamesScott)-[~/Desktop/Devops]
$


Problems 12 Output Debug Console Terminal Ports
(jamesscott㉿JamesScott)-[~/Desktop/Devops]
$ kubectl get services
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
django-service NodePort 10.111.135.244 <none> 80:30494/TCP 9d
kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 9d
postgres ClusterIP 10.98.180.236 <none> 5432/TCP 9d
(jamesscott㉿JamesScott)-[~/Desktop/Devops]
$ minikube ip
192.168.49.2
(jamesscott㉿JamesScott)-[~/Desktop/Devops]
$ http://192.168.49.2:30494

4. GIT & GITHUB STRATEGY

Throughout the project, I used Git for version control and hosted everything on GitHub in a public repository. One of the first things I set up was a clear branching strategy. I wanted to make sure I could work on new features, experiment with ideas, and fix bugs without breaking anything in the main application and without losing track of what I was doing. So, I kept a main branch as the base for all stable, production-ready code, and then created feature-specific branches for development work.

As you can see from the repository (screenshot attached), I created several named branches to organize my work. The api-js-bridge branch was where I focused on improving how the backend API communicated with the frontend, and where I first experimented with the Open Library search logic. For everything related to sending emails from writing the logic and setting up templates to writing tests I used the emails branch. When I started working on deploying the app using Kubernetes and Helm, I used the kubernetes-setup branch to isolate all my manifests and testing. The open-library-integration branch was created just to handle the logic of connecting to the Open Library API and managing the import of book data into the system. I also created a more general-purpose branch called additional work, where I made polish and small improvements before merging them in. At one point, I spun up a temporary branch called my-feature-branch as a sort of scratchpad to test new features without the risk of interfering with my main setup.

This approach allowed me to work safely without fear of accidentally overwriting something important. I could easily switch contexts depending on what I was working on. When a branch was ready and tested, I would open a pull request to merge it into main. That gave me a chance to review everything, double-check for merge conflicts, and make sure tests passed in GitHub Actions before deploying.

I also followed the Conventional Commits style to keep my commit history clean and meaningful. For example, I used the feat: prefix when adding new functionality like the Open Library integration, and fix: for bugs like layout issues or broken email rendering. When I updated documentation or my README, I used docs: to clearly show that the commit was non-code related.

At key moments in the project, I tagged my repository with release markers like v1.0 for the first stable version and docker-stable once everything was fully working inside containers. These tags were extremely useful when I needed to roll back, test different setups, or just show progress during review.

This kind of workflow helped me stay organized and in control of my development process. It made me more confident to experiment and iterate, knowing I could always branch off or revert back. Overall, it gave the project a structure that mirrored how real-world teams manage software development.

Figure 5 *branch history graph from GitHub,*

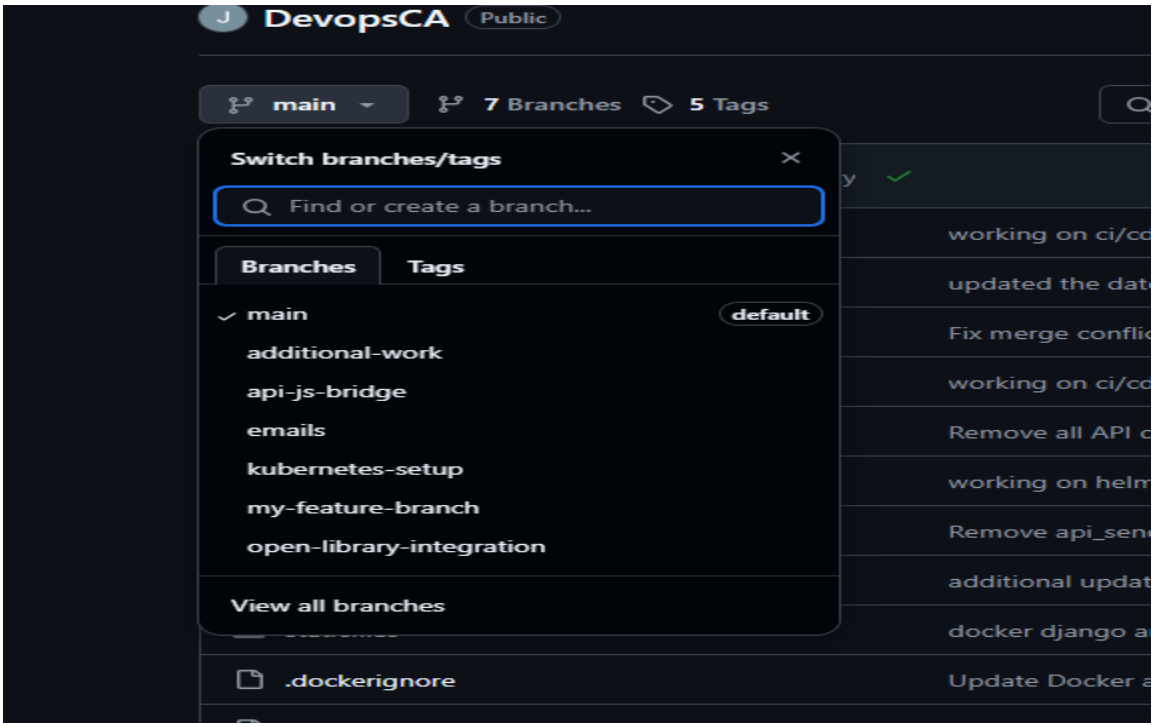


Figure 6 *a screenshot of the tags/releases page,*

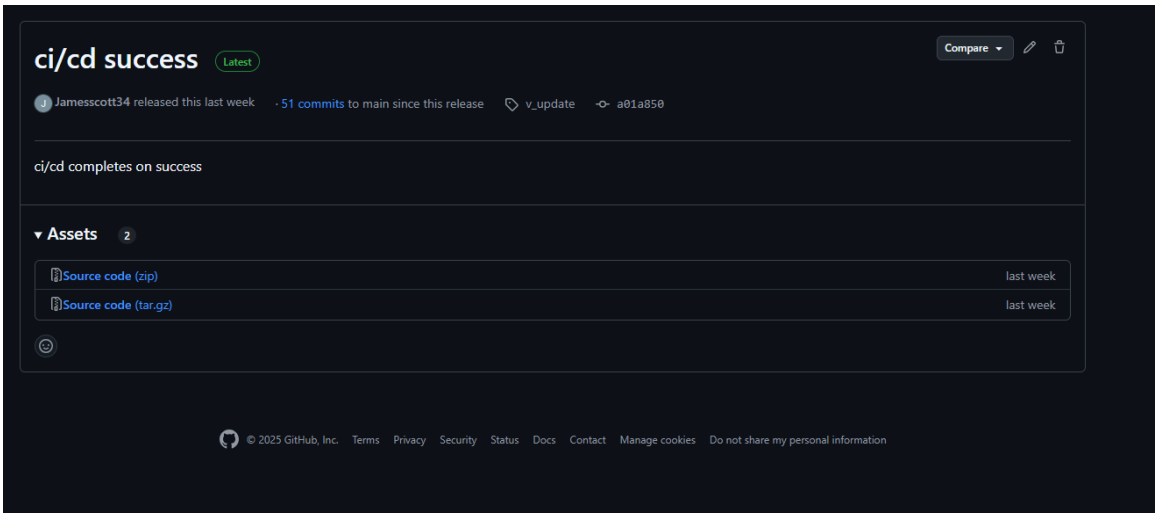
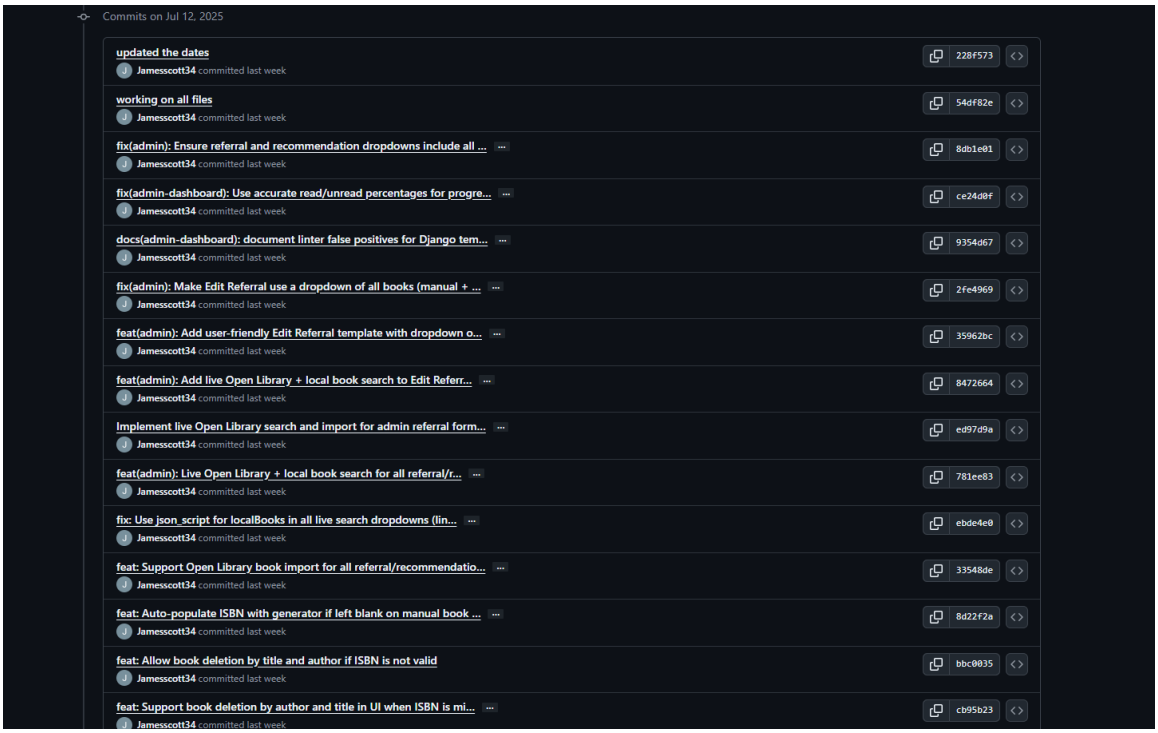


Figure 7 *a sample pull request or merge commit.*



5. DOCKERIZATION

One of the key goals of this project was to make it easily portable and consistent across different environments. To achieve this, I used Docker to containerize the entire application. My Dockerfile is intentionally simple but structured in a way that clearly supports both development and production needs. It begins with a lightweight Python base image, which helps keep the final image size low while ensuring compatibility with Django and all dependencies.

In the Dockerfile, I first copy in the requirements.txt file and install all dependencies. This step is done before copying the rest of the project files because Docker caches layers so unless requirements.txt changes, this layer won't be rebuilt, which saves time. After that, I copy over the rest of the source code, set the working directory to /app, expose port 8000, and use Gunicorn as the entrypoint to serve the Django application. Gunicorn is more production-ready than Django's built-in server, so it's ideal for deployments.

For handling configuration, I used a .env file to store all sensitive values like database credentials and the Django SECRET_KEY. This file is **never** committed to GitHub thanks to the .gitignore, and in Docker it's loaded using the env_file directive in docker-compose.yml or injected at runtime in production. This allows the same image to work across different environments by just changing the environment file.

To avoid bloating the Docker image, I created a .dockerignore file that excludes files and folders that shouldn't be part of the container context. Things like __pycache__, .git, test logs, and compiled Python files are skipped when building the image, which keeps things clean and fast.

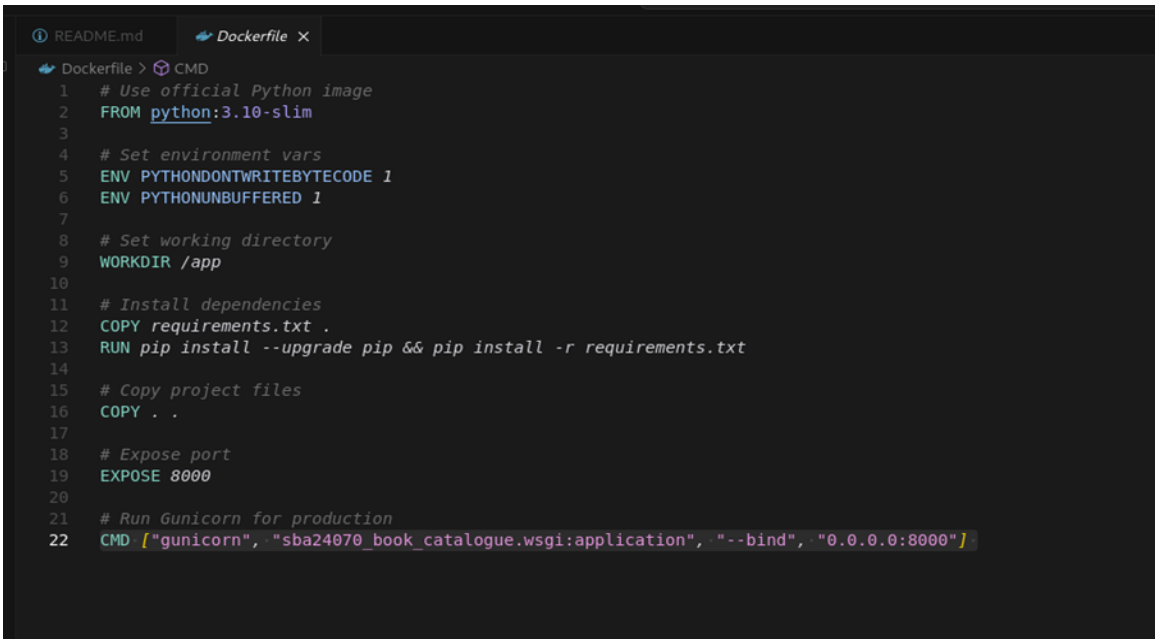
Although I didn't use a multi-stage build in this project, I'm aware of how it could improve performance by splitting build and runtime stages.

For local development, I used Docker Compose to manage both the Django application and the PostgreSQL database. This was really helpful because it allowed me to spin up the entire stack with a single command: docker-compose up. I configured service dependencies so that the web container only starts once the database is available, and I mapped all ports and volumes appropriately. This setup made it very easy to develop, test, and debug without needing to install PostgreSQL or Python directly on my host machine.

In practice, this approach made switching between environments painless. Whether I was testing locally, building images for production, or running the app inside Kubernetes, I knew that the underlying

container was identical in each case. This repeatability is one of the strongest arguments for using Docker, and it made a real difference in how I approached this project.

Figure 8 *screenshot of the Dockerfile, docker.yml*



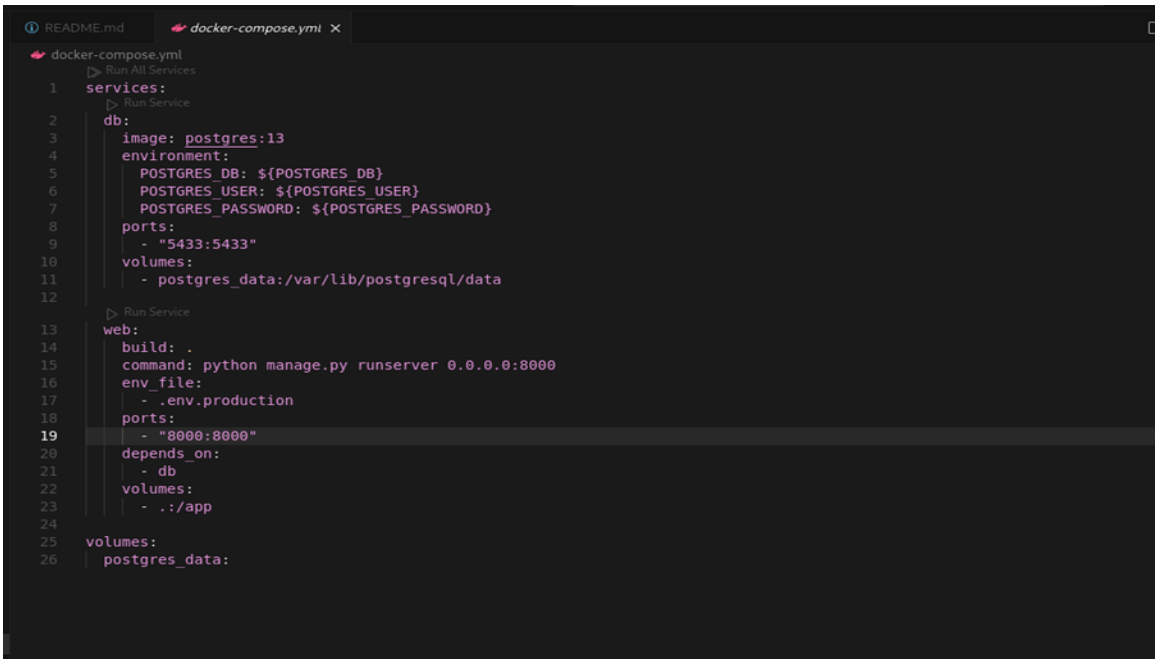
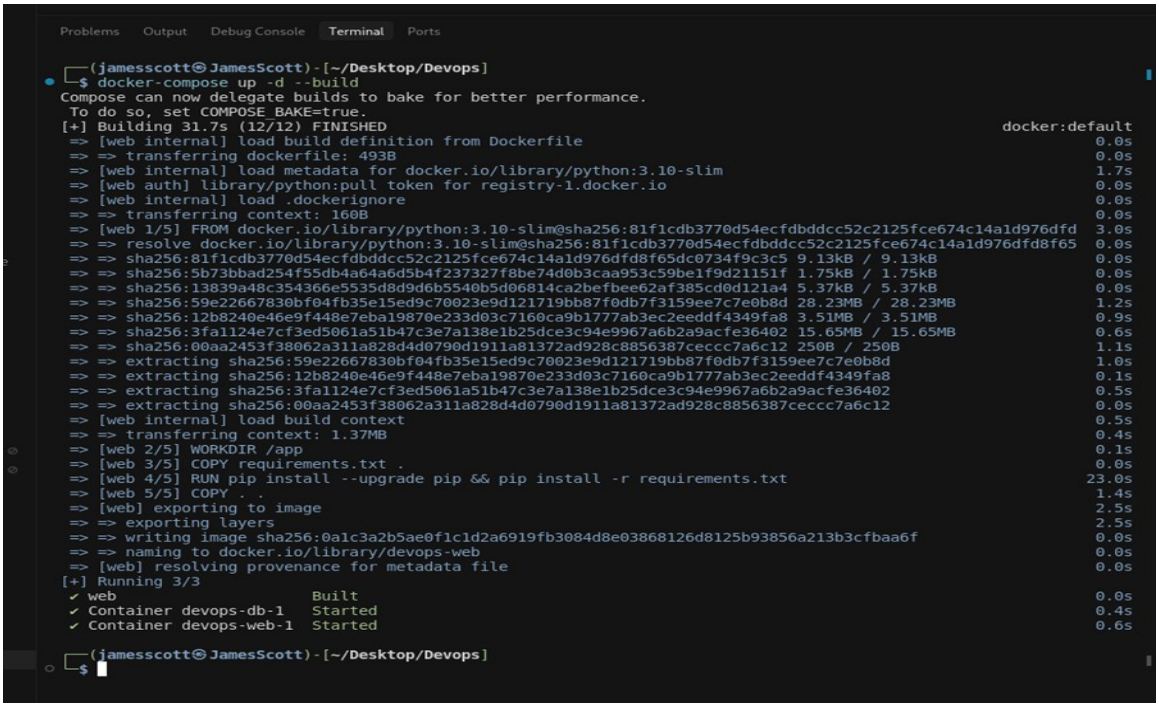


Figure 9 *docker-compose docker logs*



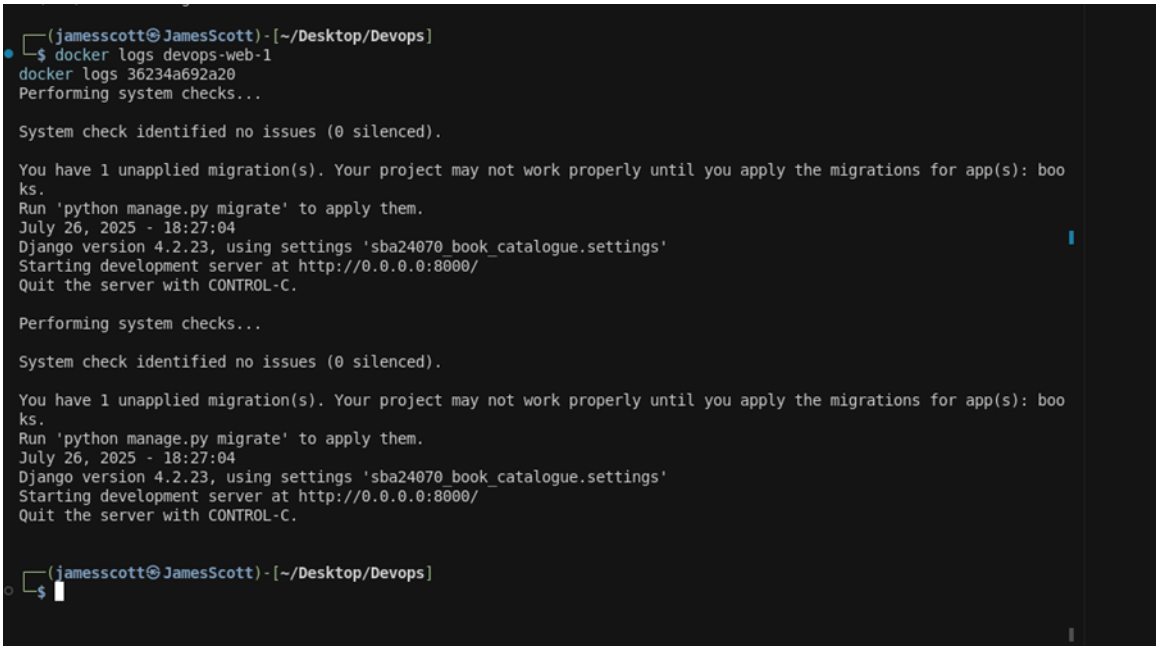
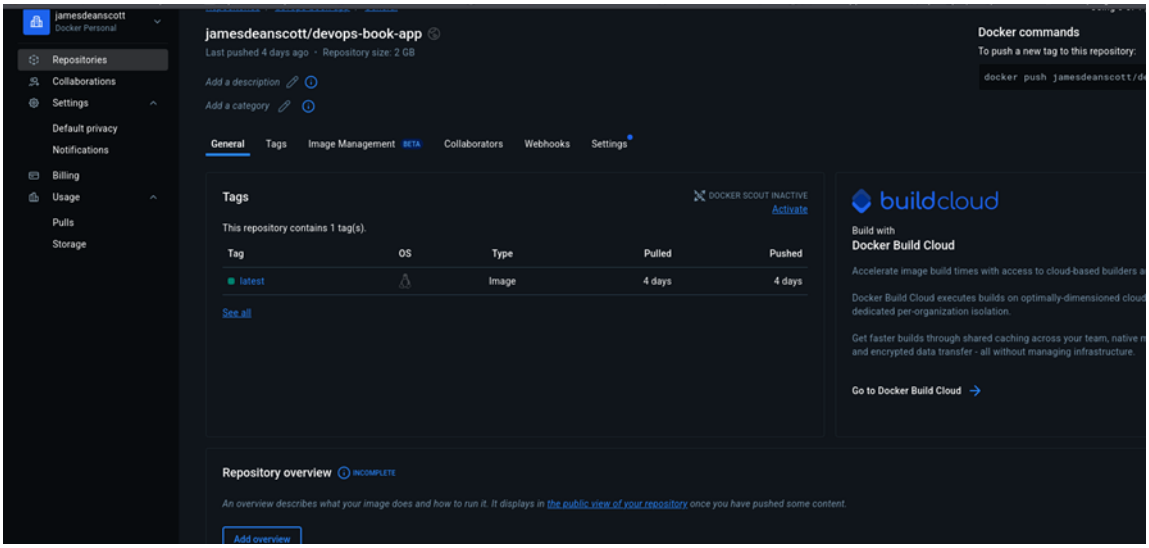


Figure 10 *the Docker Desktop UI showing running containers,*

Figure 11 *DockerHub repository page.*



6. KUBERNETES & HELM

Kubernetes was selected as the orchestration platform for this project because of its ability to manage containerized applications in a resilient, scalable, and automated fashion. While Docker is great for packaging and running applications locally, Kubernetes takes things further by enabling load balancing, automated rollouts and rollbacks, fault tolerance, and health monitoring. For a DevOps Capstone project like this where demonstrating a production-like deployment is essential Kubernetes allowed me to showcase a real-world deployment scenario that could scale under real usage.

In my case, I deployed the Book Catalogue App to a local Minikube cluster. This allowed me to simulate a cloud-based environment while maintaining full control for development and testing. Once the app was built and pushed to Docker Hub by my CI/CD pipeline, the new image was pulled into the Kubernetes cluster. Here, the app was managed and deployed through Helm, a package manager for Kubernetes that simplifies the configuration and deployment process.

The Helm chart I created for this project is structured into key configuration and template files. The values.yaml file contains all the environment-specific configuration options such as image name, replica count, secret values, environment variables, and service port definitions. This file acts as a central place to tweak settings without needing to touch the underlying templates.

The deployment.yaml file defines how the Django app should run inside Kubernetes. It specifies the container image to use, how many replicas to deploy, resource limits, probes for health checks, and volume mounts if needed. The service.yaml file exposes the application internally to the cluster and externally via a NodePort or Ingress when required. I also included an optional ingress.yaml file, which can be used to route traffic to the application via a domain name and TLS in more advanced setups.

To manage sensitive configuration data like database passwords and Django's SECRET_KEY, I used Kubernetes Secrets. I encoded the values using Helm's templating engine and passed them into the deployment securely. Similarly, I used ConfigMaps to inject non-sensitive environment variables such as DEBUG, POSTGRES_HOST, and ALLOWED_HOSTS. This allowed me to maintain the 12-factor app principle of separating config from code.

One of the major benefits of using Helm was how easily I could switch between development, test, and production setups. By simply maintaining different values.yaml files or using --set flags at install time, I could deploy the app with different configurations in seconds. If I needed to scale up the number of replicas, change the database host, or toggle debug mode, it was just a matter of modifying a single

variable. Helm also made upgrades and rollbacks incredibly simple using helm upgrade or helm rollback I could manage versioned deployments with confidence.

Additionally, this whole system integrates well with my CI/CD pipeline. After GitHub Actions builds the Docker image, it triggers a Helm upgrade as part of the automated deployment process. This means that pushing code to the main branch results in a live deployment in the Kubernetes cluster, with no manual intervention required.

Overall, adopting Kubernetes and Helm was a valuable learning experience. It gave me a much deeper understanding of cloud-native deployments, and the kind of automation used in real DevOps pipelines. It also showed me how much more manageable deployments become when templated properly using Helm.

Figure 12 *output from kubectl get pods*

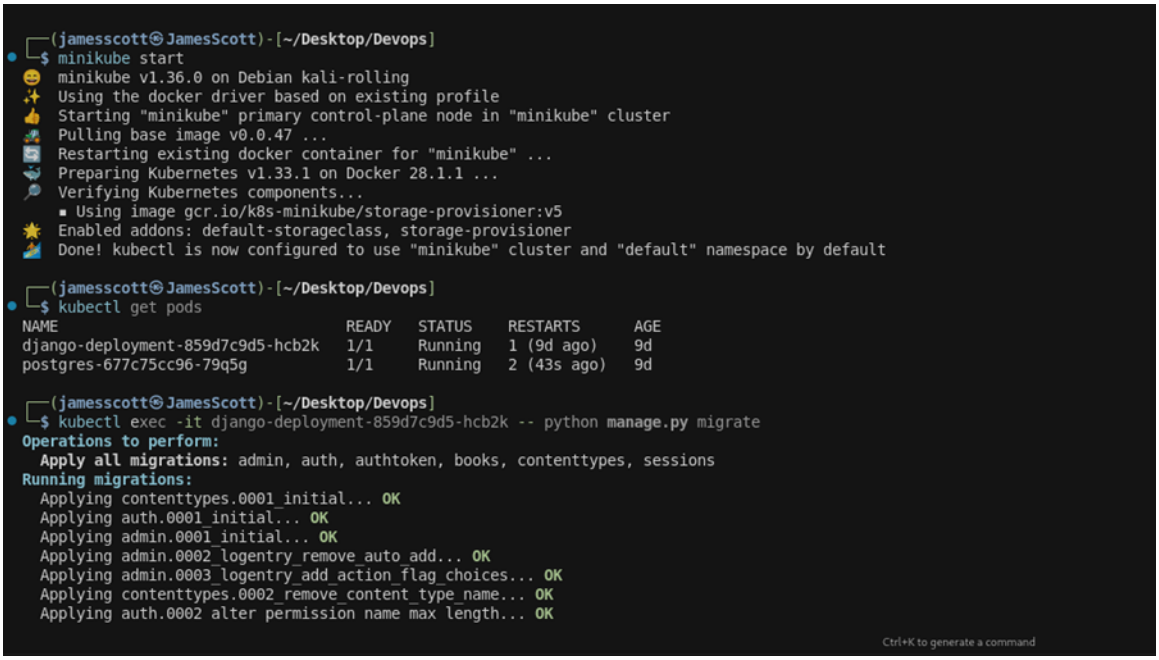


Figure 13 *kubectl get services,*

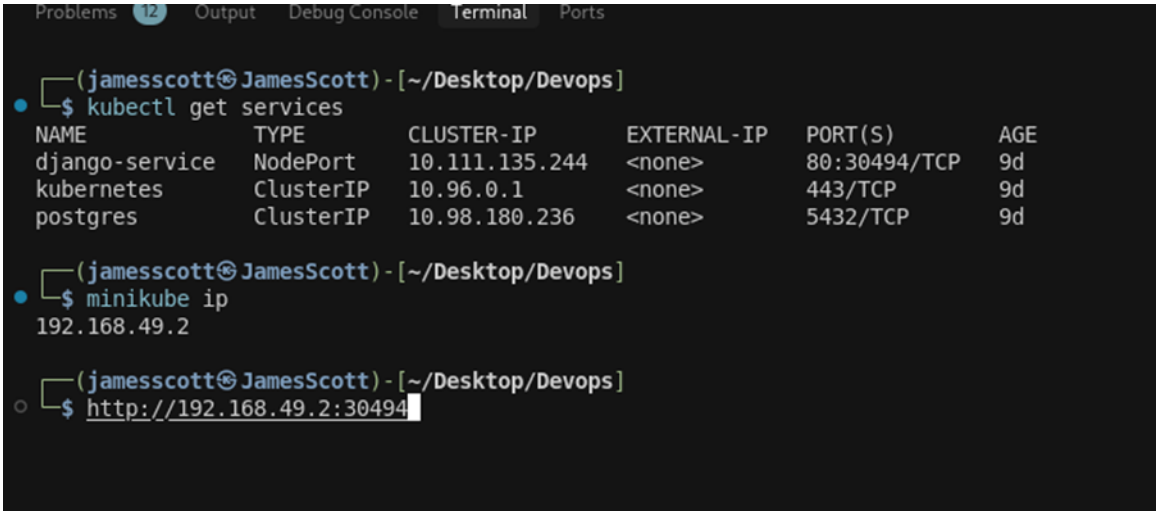


Figure 14 *Helm install/upgrade CLI output*

Figure 15 *values.yaml file*

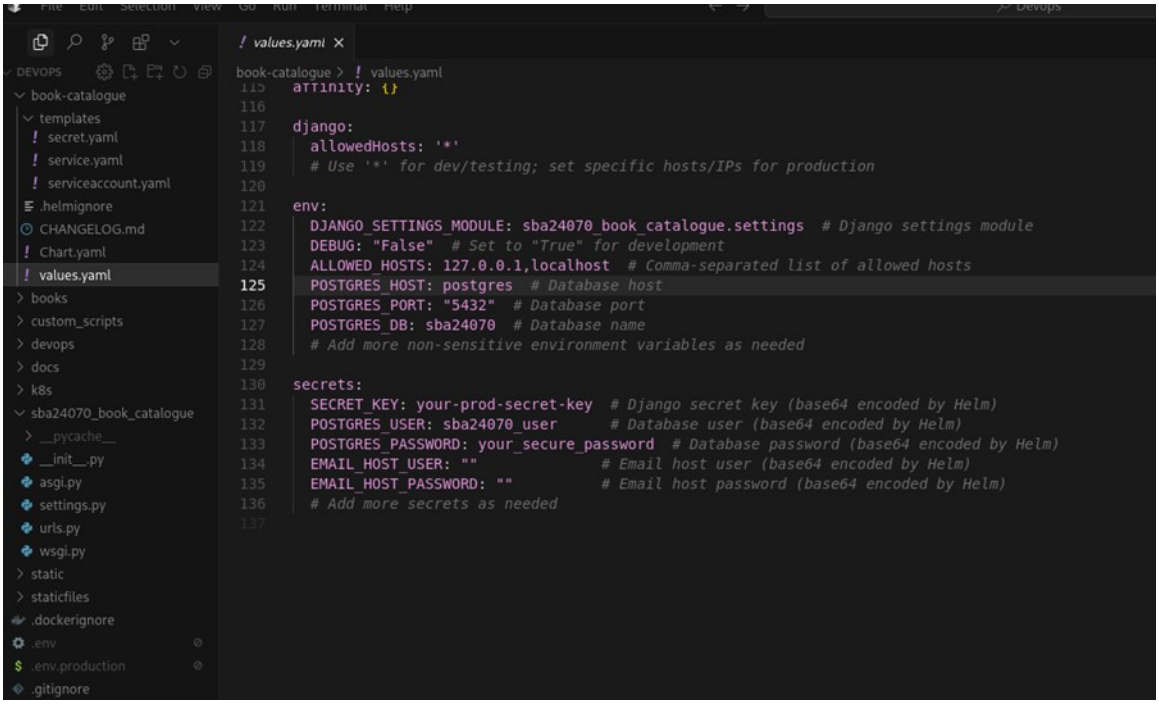
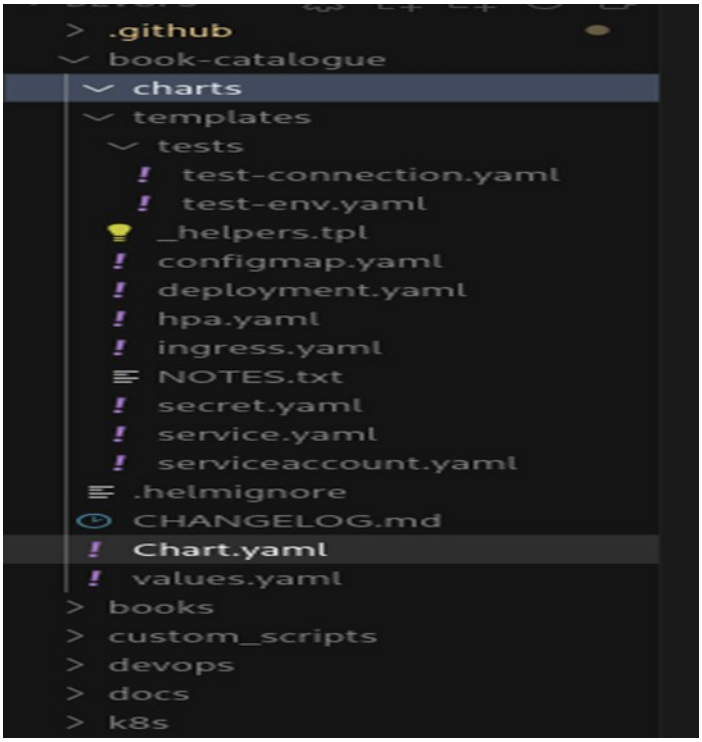


Figure 16 *chart directory.*



7. CI/CD PIPELINE

One of the most valuable aspects of this project was designing and implementing a complete CI/CD pipeline using GitHub Actions. From the very beginning, I wanted the process of building, testing, and deploying the application to be as automated as possible just like it would be in a real-world DevOps team setting. I didn't want to manually test or deploy every time I made a change; instead, I wanted to commit my code, push it to GitHub, and know with confidence that everything would be taken care of automatically.

My pipeline is configured through a YAML file inside the .github/workflows/ directory. It's structured into separate jobs that each handle a part of the process from installing dependencies, checking for database migrations, running tests and coverage reports, to finally building a Docker image and deploying it to Kubernetes using Helm.

The process kicks off whenever I push to the main branch or open a pull request targeting it. The first stage of the pipeline installs all required Python packages using the requirements.txt file. Then, it runs unit tests using Django's built-in test runner. These tests are designed to cover key functionalities in the app, such as creating and retrieving books, validating user login, and checking the notification system. I also configured coverage reporting so I could track how much of my code is tested, which helped highlight areas I could improve.

If all the tests pass, the pipeline moves on to building the Docker image of my application. I'm using the GitHub Actions docker/build-push-action, which not only builds the image but also pushes it directly to my DockerHub repository (jamesdeanscott/devops-book-app). This ensures that I always have an up-to-date, versioned image that can be deployed from anywhere.

Once the image is pushed, another job takes care of deployment. This is where Kubernetes and Helm come into play. GitHub Actions uses my Kubernetes config file (securely stored and injected via GitHub Secrets) to connect to my Minikube cluster. Then it runs a Helm upgrade command to redeploy the app with the new image version. This step is important because it ensures the live version of the app always reflects the latest code without needing to SSH into a server or manually trigger updates.

To make sure the pipeline runs securely, I use GitHub Secrets to store all sensitive data, including my DockerHub credentials, database passwords, Kubernetes config, and Django secret key. None of this is exposed in the codebase or logs. This is a core DevOps best practice, and it gave me a deeper appreciation for how real teams protect their credentials during automated deployments.

What I really like about this pipeline is how hands-off it is. I can write code, commit it, and focus entirely on development knowing that GitHub Actions will take care of the rest. If something breaks, I get immediate feedback in the Actions tab, along with logs that help me troubleshoot. If everything works, the changes are live within a few minutes, deployed cleanly and consistently.

This setup mirrors how professional CI/CD systems work in companies today, and building it taught me a lot about automation, scripting, secret management, and cloud-native deployment workflows.

Figure 17 *GitHub Actions*

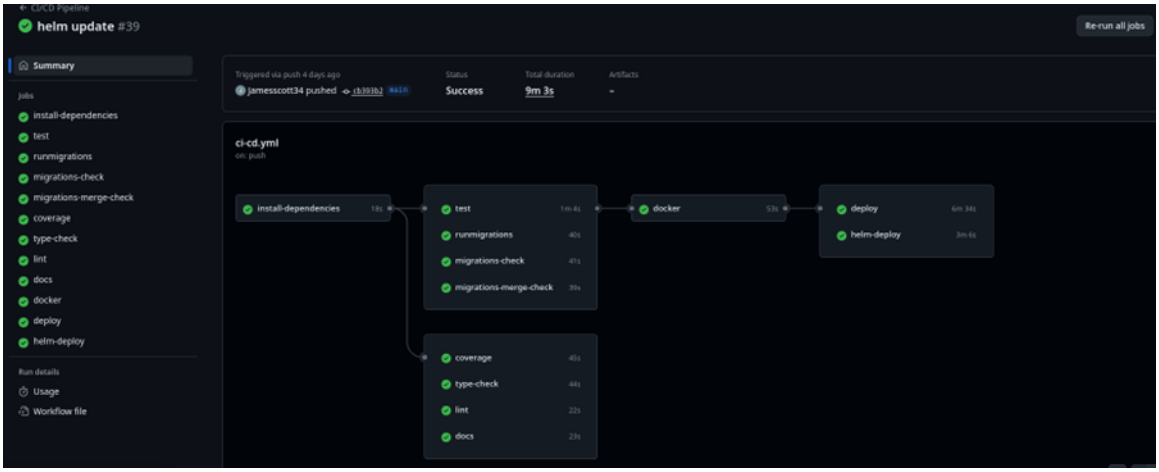


Figure 18 *Workflow YAML file,*

```
hub > workflows > ci-cd.yml
name: CI/CD Pipeline

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
  workflow_dispatch:

jobs:
  install-dependencies:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.13'
      - name: Cache pip
        uses: actions/cache@v4
        with:
          path: ~/.cache/pip
          key: ${{ runner.os }}-pip-${{ hashFiles('requirements.txt') }}
          restore-keys: |
            ${{ runner.os }}-pip-
      - name: Install dependencies
        run: pip install -r requirements.txt

  test:
    needs: install-dependencies
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres:13
        env:
          POSTGRES_DB: sba24070
```

8. TESTING & QUALITY

Testing was a major focus of this project to ensure reliability, maintainability, and confidence in the deployment process. I created a comprehensive tests.py file that targets all core components of the Book Catalogue App including models, user authentication, notifications, tags, and even real-world email delivery using environment variables.

The test suite is built using Django's TestCase framework and simulates real-world application behaviour. The Book model tests verify that books can be created and retrieved correctly, and that unique ISBN constraints are enforced. It also ensures that missing required fields like title or author raise expected errors. Additionally, it tests the view count functionality using the increment_view_count() method and verifies that statistics such as the most-read and most-viewed books are calculated correctly. Tag assignment is also tested to confirm that books can be tagged and retrieved accurately. Finally, the string representation test ensures that each book returns its title when converted to a string, which is important for admin interfaces and logging.

For the User and Notification models, the test suite checks that passwords are securely hashed using Django's check_password() and that users can be authenticated using their stored credentials. The notification tests verify that notifications are saved properly for each user and that they can be marked as read, updating the is_read flag in the database.

The email backend is tested using Django's SMTP backend. If valid environment variables are set, the test attempts to send a real email to verify that the email system works. If the necessary variables are missing, the test is gracefully skipped to avoid breaking the CI pipeline. This makes the test optional and adaptable depending on the environment.

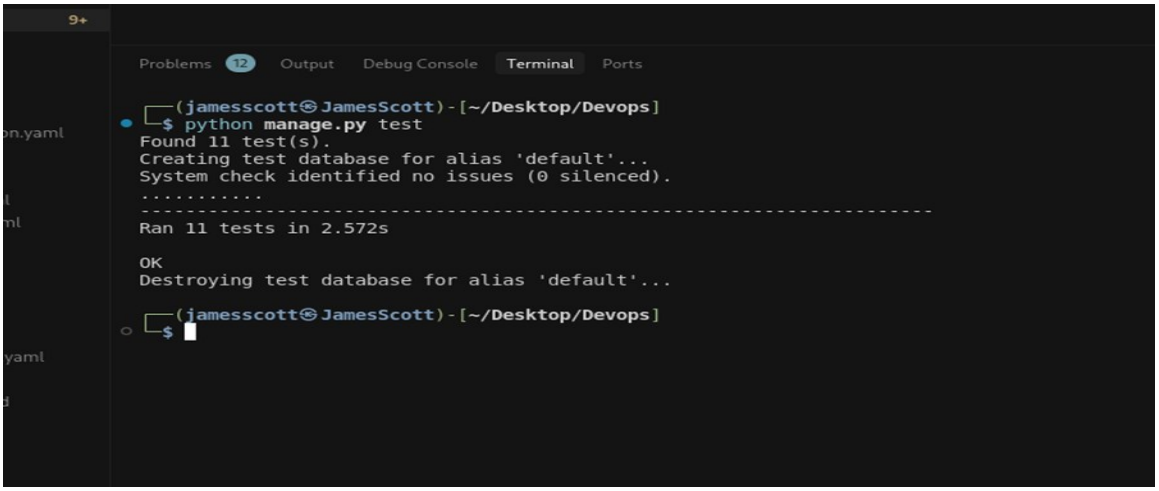
There is also a test class called EnvVarTest, which ensures that key environment variables like EMAIL_HOST_USER and EMAIL_HOST_PASSWORD are present. If they are not, a warning is printed and related tests are skipped instead of causing the pipeline to fail.

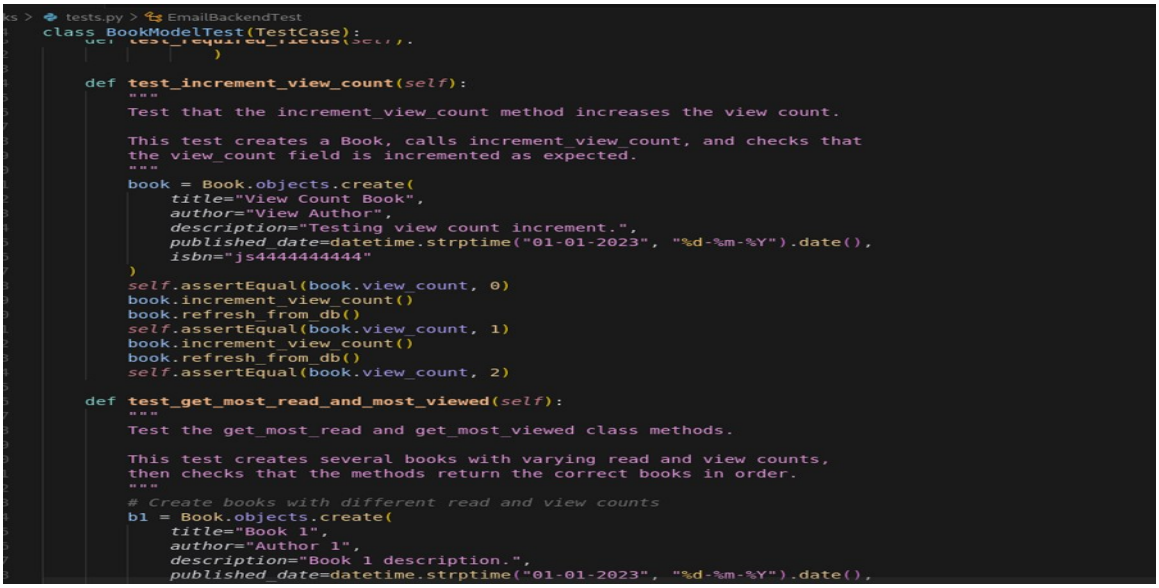
In terms of code quality, I used flake8 to enforce Python code style and linting rules. This helps catch syntax issues and formatting inconsistencies early. I also used mypy to run static type checks on annotated functions and classes. This gave me additional confidence in the correctness and maintainability of the codebase. To measure how much of the code is tested, I used the coverage tool to generate detailed reports and identify any parts of the application that lacked test coverage.

All of these tests and checks are automatically run through the GitHub Actions CI/CD pipeline. Whenever I push a change or open a pull request, the workflow installs dependencies, runs tests, checks code style, type hints, and generates a coverage report. Only if all checks pass does the pipeline proceed to build the Docker image and deploy the updated version. This workflow ensures that only high-quality, tested code reaches production, and it has helped me catch several issues early during development.

In summary, the testing strategy in this project covers both backend logic and critical integration points such as email sending. Combined with automated quality checks and CI enforcement, it has helped keep the application stable and production-ready throughout development

Figure 19 *running python manage.py test, and image of test*





9. APPLICATION FEATURES

The Book Catalogue App is more than just a book-tracking tool it is a fully featured, production-ready Django web application that supports personal library management, administrative oversight, external API integration, user notifications, and RESTful web services. The platform is designed to be user-friendly, flexible, and scalable, offering a professional-grade experience for both end-users and system administrators.

From the user perspective, the app allows seamless account creation, secure login/logout, and personalized management of a user's private book collection. Once logged in, users can perform full CRUD operations they can add new books, update existing entries, delete titles, and view their entire catalogue in an organized interface. Each book record can include a title, author, ISBN, description, cover image, published date, and custom tags for categorization. For convenience, ISBNs are validated automatically during book entry, and missing values like ISBNs can be auto-generated by the system.

A standout feature is the integration with the **Open Library API**. This allows users to search for books online and import full metadata with a single click. When importing, the app automatically fetches the book title, author, description, ISBN, and cover image streamlining the process and eliminating the need to enter data manually. This feature alone makes the app extremely powerful for users managing large libraries or looking to explore new books without leaving the platform.

Another major highlight is the **notification system**. Admins can send personalized or system-wide notifications to users. These notifications may include book recommendations, important announcements, or usage-related messages. Each user has access to a notification panel where they can view unread and read messages, and mark them accordingly. This communication channel creates a more interactive and guided user experience, especially for educational or enterprise environments.

Administrators have access to a dedicated **Admin Dashboard** that allows them to manage users and books across the entire system. From here, admins can view analytics about reading trends, the most popular or most viewed books, and user activity. Admins can also assign books to users, manage user notes, and monitor real-time notifications. These features make the app suitable for use in schools, reading groups, or team-based library tracking scenarios.

All of this functionality is accessible through a clean, responsive HTML + Bootstrap web interface. Pages include:

- A **Login/Register** screen for authentication
- A **Home/Dashboard** displaying a user's books
- A **Book Detail View** for each entry
- A **Notification Centre**
- A **Book Search Page** with Open Library integration
- An **Admin Management Page** (for superusers only)

The web UI is designed to be usable on both desktop and mobile, with clear navigation and feedback. A future upgrade may include full mobile responsiveness with progressive web app (PWA) support.

From a backend perspective, the app also offers a fully featured **REST API**. Every major action creating, retrieving, updating, and deleting books is mirrored in the API. The API also includes endpoints for user authentication, notification handling, tag management, and statistics retrieval. This means that the system could be easily extended to mobile clients, frontend frameworks like React, or integration with external tools. Full API documentation and usage examples are available in the repository.

To support advanced use cases, the app also includes additional features such as:

- Reading Progress Tracking
- View Count Analytics
- Tag Assignment and Filtering
- Admin Referral Assignments
- Email Notification System (Gmail support, with environment-configured credentials)
- User Notes
- Secure Password Hashing and Authentication
- Test Coverage and CI Enforcement
- Support for local (SQLite), Docker (PostgreSQL), and Kubernetes deployments

The application supports multi-environment deployment and can be launched using local scripts, Docker Compose, Kubernetes manifests, or Helm charts. Each environment is tested and supported by a fully automated GitHub Actions CI/CD pipeline that runs tests, lints the code, builds Docker images, and deploys to a Kubernetes cluster.

Figure 20 *Login Screen*

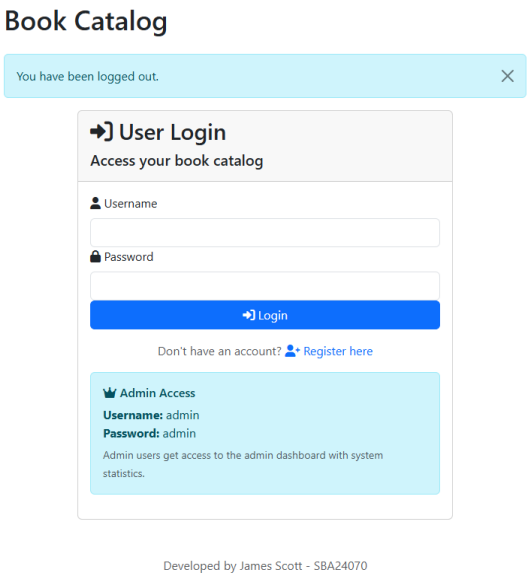


Figure 21 *Admin curl login*

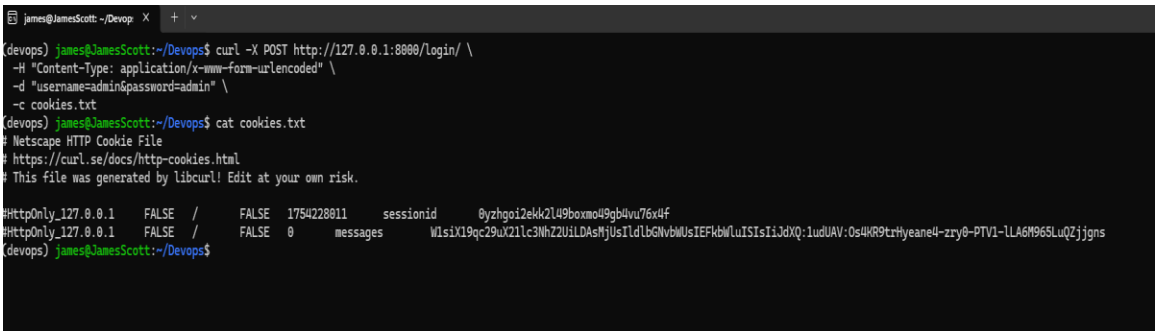
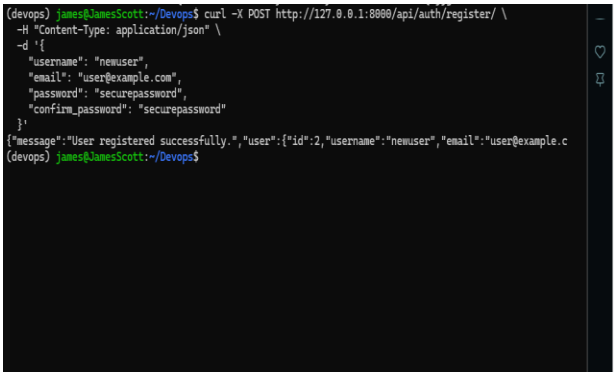


Figure 22 *Curl user creation*



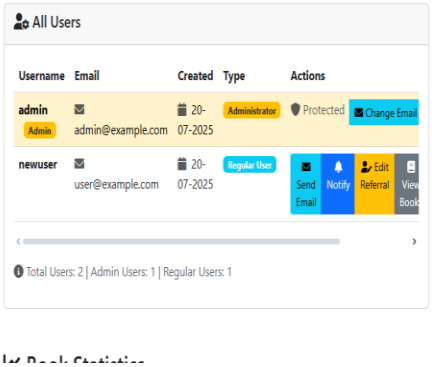


Figure 23 *Login as new user*

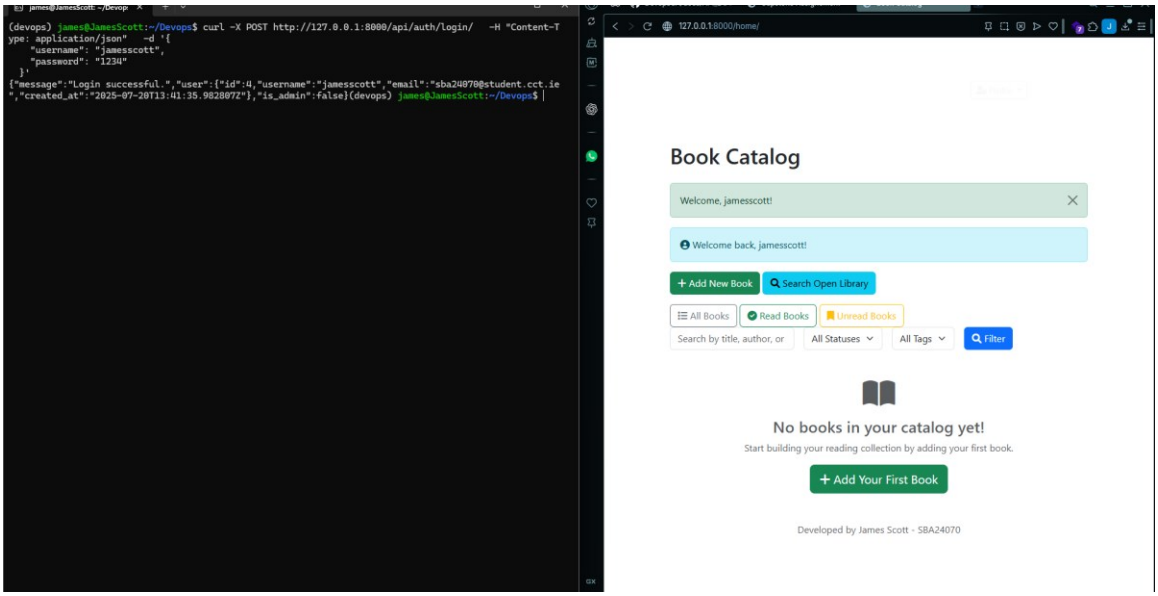


Figure 24 *Add a book*

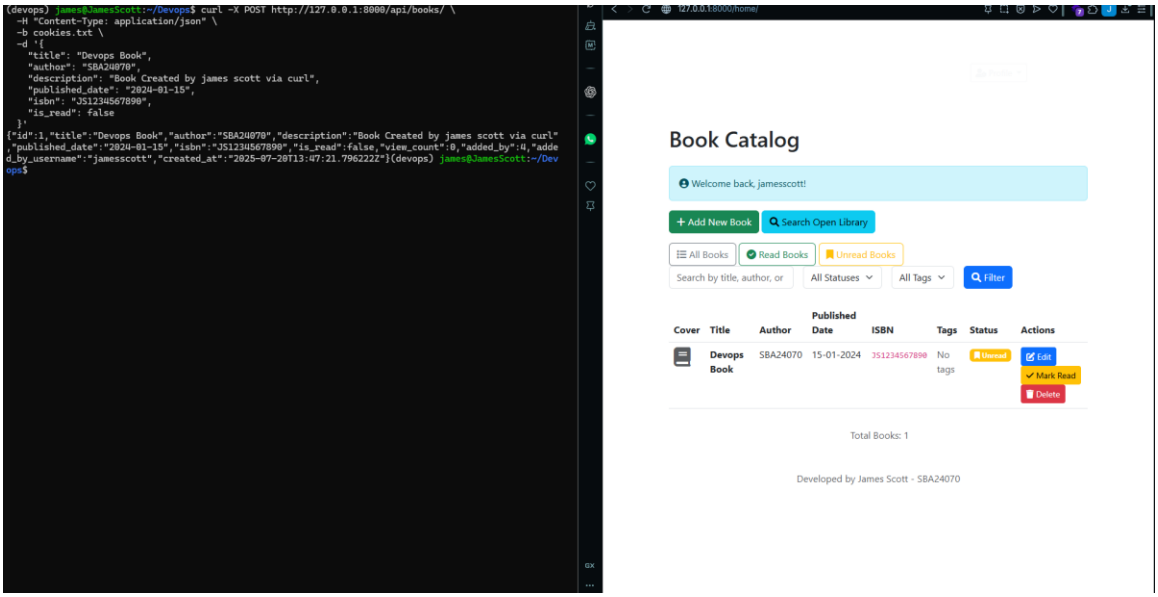


Figure 25 *View books*

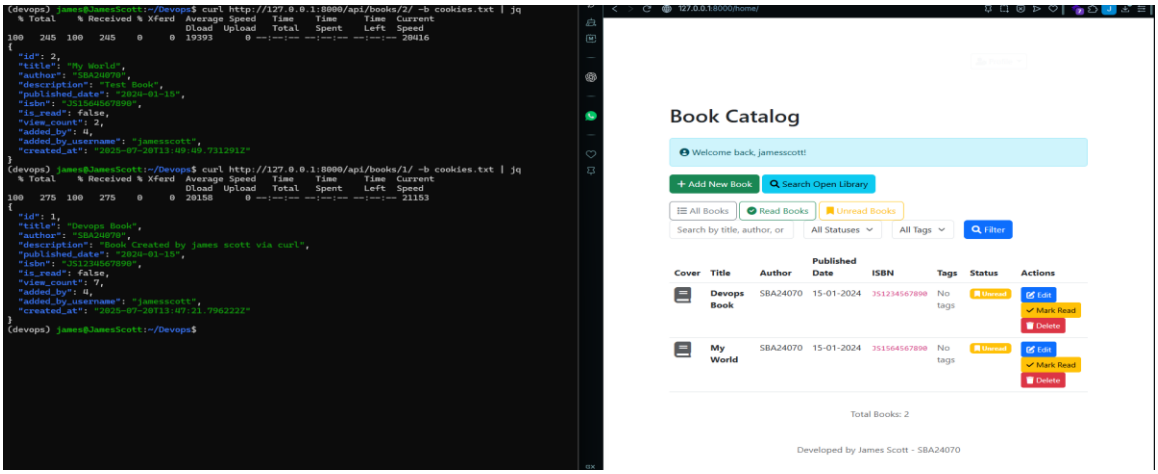


Figure 26 *Delete books*

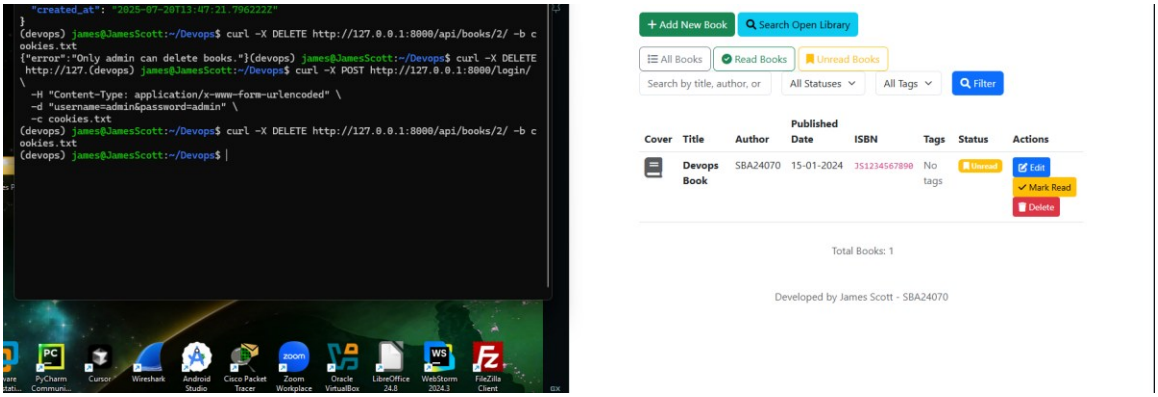


Figure 27 *Library api view*

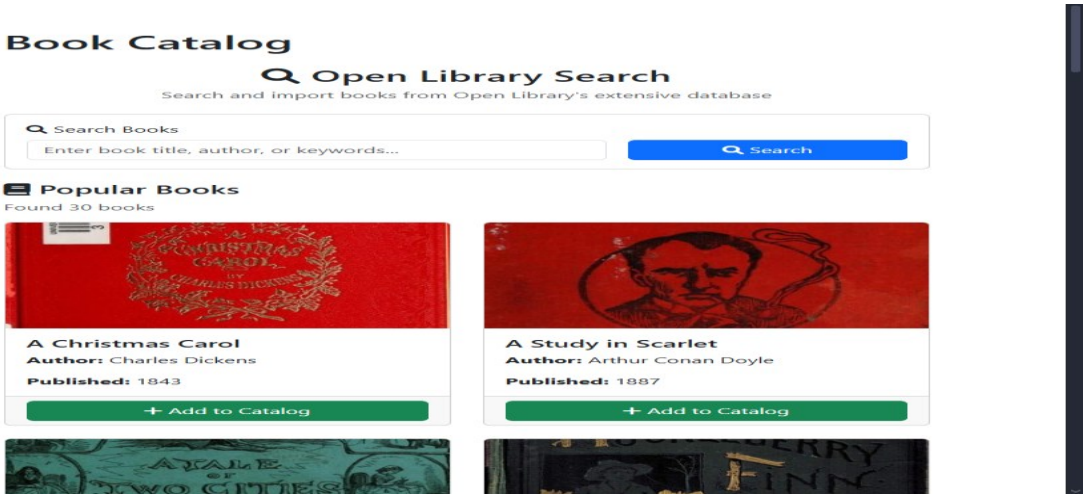


Table 2 *REST API Endpoints*

Method	Endpoint	Description	Auth Required	Example Payload
POST	/api/auth/login/	Login with username and password	No	{ "username": "admin", "password": "admin" }
POST	/api/auth/register/	Register a new user	No	{ "username": "newuser", "email": "...", "password": "..." }
POST	/api/books/	Create a new book	Yes	{ "title": "...", "author": "...", "isbn": "...", "published_date": "YYYY-MM-DD" }
GET	/api/books/	Get list of all books	Yes	
GET	/api/books/<id>/	Retrieve book by its ID	Yes	
PUT	/api/books/<id>/	Update book details by ID	Yes	Same as create
DELETE	/api/books/<id>/	Delete book by ID	Yes	

10. CHALLENGES & LEARNINGS

One of the biggest challenges in this project was getting Kubernetes and Helm to behave the way I wanted. At first, I ran into a lot of trial-and-error tiny mistakes in YAML files would break entire deployments, and figuring out why services weren't being exposed correctly took time. Sometimes, the application would build fine but wouldn't run properly in the cluster because of how environment variables or volumes were handled. I had to do a lot of digging through documentation and community threads just to get the basics running smoothly.

Working with Docker also presented its own set of problems. I remember struggling with missing dependencies in the Dockerfile, or the app refusing to serve static files correctly in production mode. It took multiple attempts to get the Dockerfile structured properly, and at one point, I had to rebuild the image from scratch just to isolate a problem.

Debugging the CI/CD pipeline with GitHub Actions was another learning curve. Some scripts worked locally but failed during the GitHub build process, mostly because the environments were slightly different. For example, database connectivity issues or environment variables not loading properly would cause test jobs to fail, and it wasn't always obvious why. I ended up adding logging and running smaller test pipelines until I got it stable.

Managing secrets securely was also something I had to learn quickly. Early on, I had a few hardcoded credentials in config files, which I knew wasn't safe. Eventually, I moved all secrets into .env files and used GitHub Secrets for the CI pipeline, which was a much better approach. It made the deployment process cleaner and helped me understand how to inject environment variables properly in Docker and Kubernetes contexts.

Despite the setbacks, these experiences pushed me to learn more than I expected. I became more confident using GitHub Actions for automated workflows, writing reliable Helm templates, and troubleshooting container issues. I also got a lot better at reading logs, isolating bugs, and testing one piece at a time instead of trying to fix everything at once.

Looking back, I now see those issues not as problems but as stepping stones. They helped me develop a better workflow, improve my attention to detail, and build confidence working with DevOps tools in a real-world context.

11. CONCLUSION

The Book Catalogue App stands as the culmination of hands-on experimenting, trial-and-error, and a focused effort to master real-world DevOps technologies. What began as a Django-based book tracker quickly evolved into a complete CI/CD-ready platform running in a Kubernetes environment and along the way, I came to fully appreciate the depth and power of modern DevOps workflows.

This project was not just about getting a web app to work it was about building the kind of system you would find in a real engineering environment: containerized, automated, and deployable across multiple environments with a single command. I used Docker to ensure portability and consistency, GitHub Actions to automate testing and deployment, and Helm to manage Kubernetes resources with flexibility and precision. These tools not only improved the way the app was built and deployed but also taught me how to think about infrastructure as part of the software itself.

More importantly, the challenges I faced YAML misconfigurations, debugging CI pipelines, managing secrets, and orchestrating services became the most valuable parts of the journey. They forced me to read deeper, experiment more, and adopt better practices. As a result, I not only learned how to solve technical problems, but also how to prevent them through automation, clear structure, and a well-defined workflow.

The Book Catalogue App is a working, full-featured platform with real users, admin control, RESTful API integration, Open Library support, and notification systems. But it's also a learning artifact proof that I can take a DevOps concept and bring it to life with tools that are actually used in industry today.

Above all, this capstone experience has helped me gain the confidence and technical foundation to work on real DevOps teams to collaborate on code, build pipelines, solve deployment issues, and continuously deliver working software.

12. REFERENCES

Scott, J. (2025) *DevOps Capstone: SBA24070 Book Catalog App*. [GitHub Repository] Available at: https://github.com/Jamesscott34/DevopsCA

Docker Inc. (2024) *Docker Overview*. [Online] Available at: https://docs.docker.com/get-started/overview/

Kubernetes Authors (2024) *Kubernetes Documentation*. [Online] Available at: https://kubernetes.io/docs/home/

Helm Authors (2024) *Helm Charts - The Kubernetes Package Manager*. [Online] Available at: https://helm.sh/docs/intro/using_helm/

Django Software Foundation (2024) *Django Documentation*. [Online] Available at: https://docs.djangoproject.com/en/stable/

GitHub Inc. (2024) *GitHub Actions Documentation*. [Online] Available at: https://docs.github.com/en/actions

Open Library (2024) *Open Library API Documentation*. [Online] Available at: https://openlibrary.org/developers/api

PostgreSQL Global Development Group (2024) *PostgreSQL Documentation*. [Online] Available at: https://www.postgresql.org/docs/

Python Software Foundation (2024) *Python 3.10 Documentation*. [Online] Available at: https://docs.python.org/3.10