OpenGalaxy

System Architecture Design

Created by Alfie Atkinson

March 12, 2025

Contents

1 Introduction

1.1 Purpose

This document defines the system architecture and design specifications for OpenGalaxy, a web platform for browsing and managing open-license media such as images, audio, and videos. The app will primarily source content from the Openverse API and is intended for a wide range of users, including content creators, educators, and marketers.

The goal of this document is to provide a clear description of the architecture, design choices, and technologies used in OpenGalaxy, ensuring that all stakeholders have a consistent understanding. It will also guide the development, testing, and deployment phases of the platform.

The document will cover the following areas:

- **UI/UX Design**: It will outline the design principles behind the UI and UX, ensuring that the app is intuitive and easy to navigate for users across a variety of devices.
- **Integration with External APIs**: Specifically, how the app will interact with the Openverse API to retrieve and display open-license media content.
- **Data Management and Security**: How user data, including search preferences and history, will be managed, stored securely, and protected against potential vulnerabilities.
- **System Architecture and Technologies**: A description of the key technologies and practices such as automated testing, containerisation, and CI/CD pipelines.
- **Scalability and Performance**: How the system is designed to scale as the user base and media content grow, ensuring minimal latency and a consistent user experience across different devices and internet speeds.
- **Testing and Quality Assurance**: Guidelines for automated testing and continuous integration to ensure the application is reliable and secure.

This document will serve as a foundation for the software development process, aligning the project's goals and ensuring successful delivery of OpenGalaxy.

1.2 Scope

OpenGalaxy is designed to provide a platform for accessing open-license media. Key features include:

- **Media Search and Filtering**: Users can search for and view media through the Openverse API with filtering options for media and license type.
- **User Account Management**: Account creation, preferences saving, search history access, and favouriting capabilities.
- **Secure Data Handling**: Ensuring user data security and privacy compliance.
- **Responsive Design**: A responsive and intuivite user experience across various devices.
- **Scalable Architecture**: A modular system with modern software practices like automated testing, Docker, and CI/CD.

The app will be developed using React, Django, and PostgreSQL, with integration of the Openverse API for media content.

2 System Overview

2.1 High-Level Architecture

The system follows a three-tier architecture, consisting of distinct frontend, backend, and database layers, communicating through REST APIs and external interfaces.

- **Frontend**: Developed using TypeScript and the React framework with Redux, the frontend will provide an interactive user interface, handling user interactions and rendering media content retrieved from the backend.
- **Backend**: Implemented in Python using Django with Redis, the backend will handle business logic, authentication, data processing, and interaction with external services.
- **Database:** PostgreSQL will serve as the primary database, managing user accounts, preferences, and search history while ensuring data integrity and security.

The frontend will communicate with the backend via REST APIs, while the backend will interact with the database and external services. Key external integrations include:

- **Openverse API:** Used for retrieving open-license media content.
- **Elasticsearch:** Integrated for optimised media search and filtering capabilities, enabling efficient full-text search and indexing.

The system will be designed with scalability in mind, leveraging containerisation (e.g., Docker) and CI/CD pipelines for streamlined deployment and maintainability.

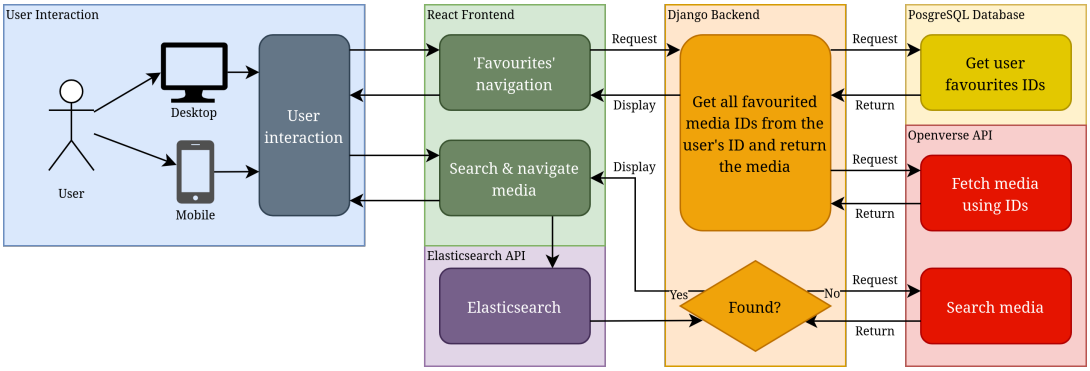

Figure 1: High-Level System Architecture of OpenGalaxy.

2.2 Key Technologies & Tools

OpenGalaxy utilises modern web technologies to ensure scalability, maintainability, and performance. The key technologies and their rationale are as follows:

- **Frontend: TypeScript, React, Next.js, Zustand** – Chosen for its component-based architecture, server-side rendering capabilities, and strong developer ecosystem. TypeScript enhances maintainability by enforcing type safety, while Zustand provides a lightweight and flexible state management solution.
- **Backend: Django, DRF, Redis** – Selected for its robustness, built-in security features, and scalability. Django's ORM simplifies database interactions and accelerates development while Redis enables in-memory caching for better performance.
- **Database: PostgreSQL** – A relational database chosen for its reliability, strong support for complex queries, and scalability.

- **Search: Elasticsearch** – Provides fast and efficient full-text search capabilities.
- **API Integration: Openverse API** – Used to fetch open-license media content.
- **Containerisation: Docker** – Enables consistent deployment across development, testing, and production environments.
- **CI/CD: GitHub Actions** – Automates testing, building, and deployment processes for consistent code quality and streamlined releases.

2.3 Deployment Overview

The system is designed to be cloud-hosted and containerised for efficient deployment and scaling. The deployment strategy is as follows:

- **Frontend:** Hosted on Vercel for fast deployments, automatic scaling, and built in analytics.
- **Backend:** Deployed on Heroku for a secure, managed environment with built-in scaling.
- **Database:** Managed PostgreSQL instance, hosted on Heroku Postgres for high availability.
- **CI/CD Pipeline:** GitHub Actions automates testing, building, and deployment for robust code and continuous integration.
- **Containerisation:** Docker is used for consistent deployments so that the development and production environments remain identical.
- **Security Considerations:** API keys and credentials are stored securely using environment variables or a secrets manager.

3 System Components

3.1 Frontend with React and Next.js

The frontend is developed using React with TypeScript, following a modular, component-based architecture. Key features include:

- **State Management:** Zustand is used for lightweight and flexible global state management, ensuring efficient data flow without unnecessary complexity.
- **Routing:** Next.js provides built-in routing, enabling both server-side rendering (SSR) and static site generation (SSG) for improved performance.
- **API Interaction:** The frontend communicates with the backend via REST APIs for authentication, media retrieval, and user preferences.
- **Performance Optimisation:** Next.js leverages automatic server-side rendering (SSR) and static generation (SSG) for faster load times. Additionally, lazy loading is used for images and dynamically imported components to improve responsiveness.

3.2 Backend with Django

The backend is built using Django and DRF, handling business logic and API services. Key aspects include:

- **Authentication:** Supports both traditional username-password login and OAuth authentication (Google, GitHub, etc.).
- **API Endpoints:** Implements RESTful APIs for user management, media search, and metadata retrieval.
- **Data Processing:** Handles search queries, user preferences, and caching strategies to optimise performance.
- **Security:** Implements JWT-based authentication and follows best practices for secure API access.

3.3 Database with PostgreSQL

PostgreSQL serves as the primary relational database, ensuring data integrity and scalability. Key design considerations include:

- **Schema Design:** Tables for users, media, licenses, and search history are structured for efficient queries.
- **Django ORM:** The Django ORM is used to simplify database operations and enforce relationships between tables.
- **Indexing:** Indexed columns optimise query performance, particularly for frequently accessed datasets.

3.4 Caching with Redis

Redis is used for in-memory caching to improve application performance. Its primary use cases include:

- **Session Management:** Stores user session data to reduce authentication overhead.
- **API Response Caching:** Frequently accessed API responses are cached to minimise redundant database queries.

- **Rate Limiting:** Helps prevent abuse by managing request rates to backend services.

3.5 Search with Elasticsearch

Elasticsearch is integrated to enhance search functionality and provide efficient full-text search capabilities. Key features include:

- **Media Indexing:** Open-license media is indexed to enable fast and accurate search results.
- **Query Optimisation:** Supports advanced filtering, autocomplete, and relevance-based ranking.
- **Scalability:** Handles large datasets efficiently, ensuring quick response times as content grows.

3.6 External Integrations

The system integrates with external services, primarily the Openverse API, to source open-license media. The integration strategy includes:

- **Data Flow:** The backend fetches media metadata from Openverse and processes it before sending responses to the frontend.
- **Error Handling:** Implements robust error handling and retry mechanisms for API requests.
- **Rate Limiting and Caching:** Reduces API request overhead by caching frequently accessed media entries in Redis.

4 Detailed System Design

4.1 Data Flow Diagram

In this section, we provide a detailed breakdown of the data flow in the application, starting with high-level interactions and progressing into more specific processes. The Data Flow Diagrams (DFDs) provide a visual representation of the system's data interactions and processes. Below, we have included Level 0, Level 1, and Level 2 DFDs to show the system's data flow at different levels of granularity.

4.1.1 Level 0 Data Flow Diagram

The Level 0 DFD provides a high-level overview of the main data interactions in the system, focusing on major processes and the flow of data between them. It highlights the essential components, such as user authentication and media retrieval, without delving into specific sub-processes.

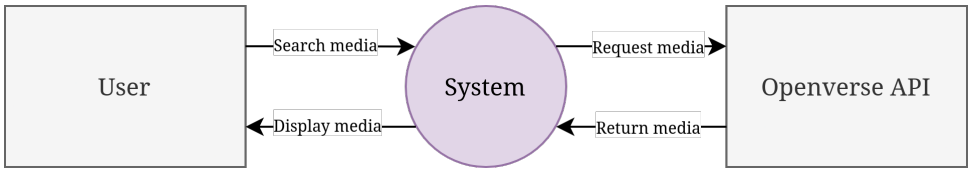

Figure 2: Level 0 Data Flow Diagram of OpenGalaxy.

4.1.2 Level 1 Data Flow Diagram

The Level 1 DFD provides a more detailed view of the system, breaking down the high-level processes into their main sub-processes. This diagram shows the flow of data between the user, authentication service, media search, and other essential components.

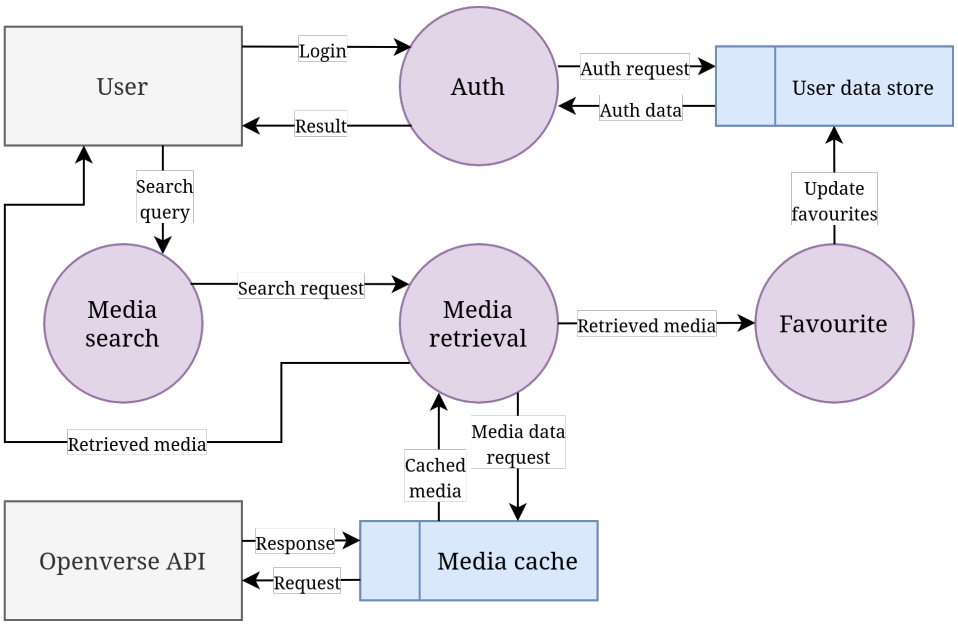

Figure 3: Level 1 Data Flow Diagram of OpenGalaxy.

4.1.3 Level 2 Data Flow Diagram

The Level 2 DFD goes even further into the details of key processes. Here, we break down the authentication process, media search, and media retrieval into their individual sub-processes, showing specific interactions and how data flows between components such as cache lookups, session management, and the interaction with the Openverse API.

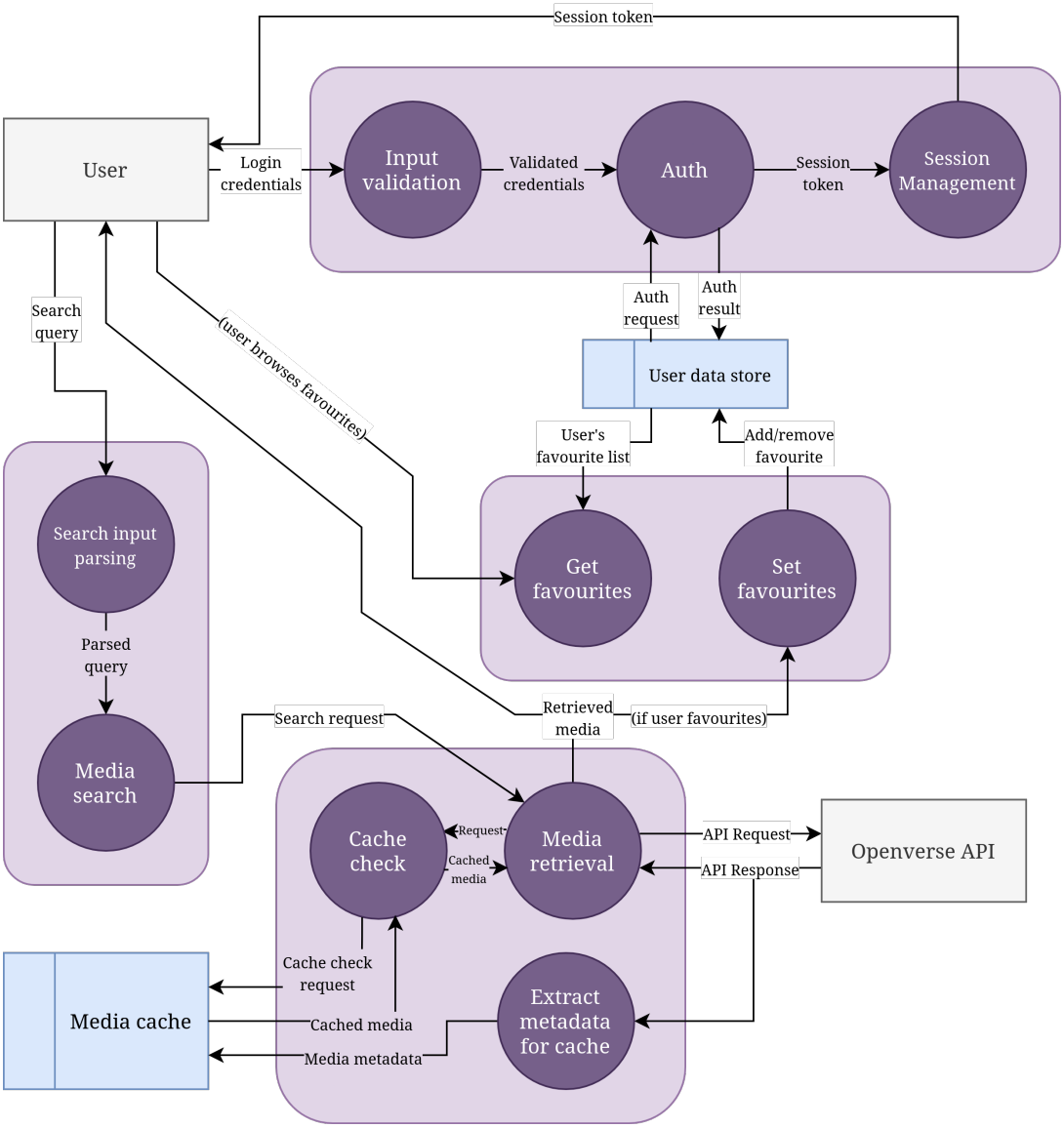

Figure 4: Level 2 Data Flow Diagram of OpenGalaxy.

4.2 Entity-Relationship Diagram

The Entity-Relationship Diagram (ERD) provides a visual representation of the key entities and their relationships within the OpenGalaxy database. This diagram illustrates how user data, media content, search history, and other system components interact with each other. It serves as a foundational blueprint for understanding how the data will be stored, retrieved, and manipulated within the PostgreSQL database.

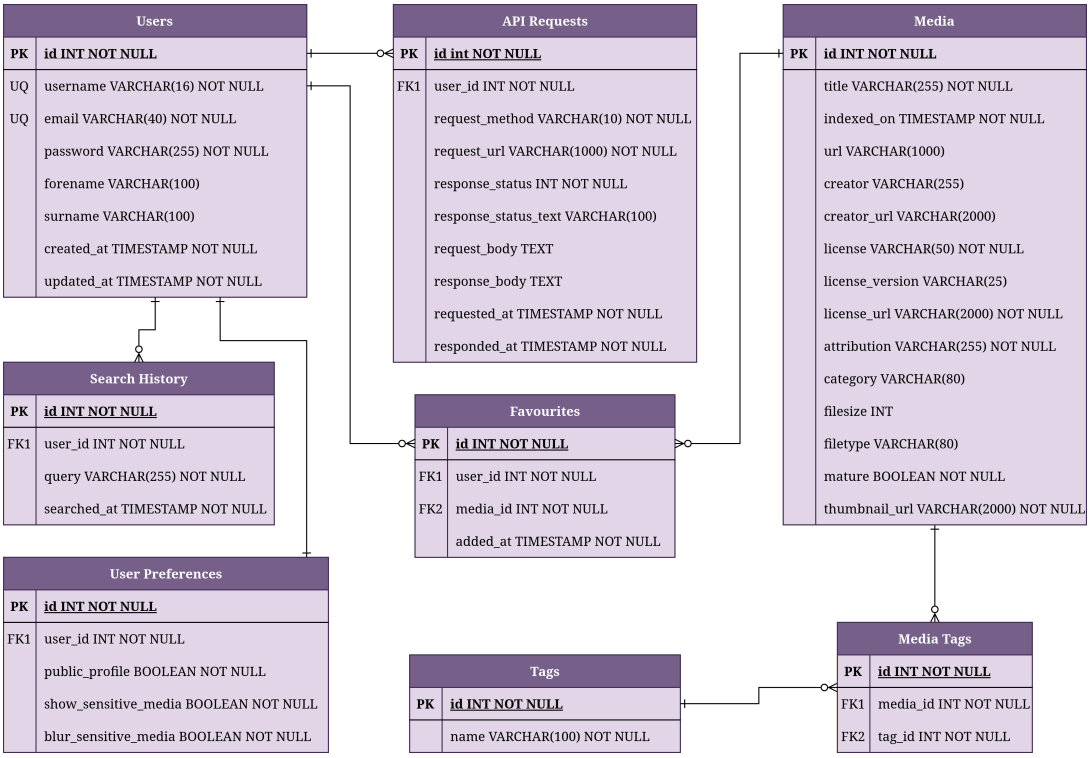

Figure 5: Entity Relationship Diagram for OpenGalaxy Database.

The OpenGalaxy database is normalised to the third normal form (3NF) to ensure efficient data management, minimise redundancy, and enhance data integrity. Normalisation is the process of organising data into separate tables to eliminate repetitive data and ensure that the database schema adheres to certain rules that reduce the potential for data anomalies (such as update, insert, or delete anomalies). The main goals of normalisation are:

- **Data Integrity:** By separating the data into distinct, related tables, normalisation ensures that each piece of information is stored only once, which prevents inconsistency and ensures that updates are done in a single place.
- **Elimination of Redundancy:** Normalisation helps in removing the need for repetitive storage of data, improving both the efficiency of storage and data consistency across the system.
- **Optimised Query Performance:** By splitting data into tables based on their logical relationships, normalisation enables more efficient queries. It reduces the load on the database and helps in fetching only relevant data, thereby improving performance.

4.3 Sequence Diagram

The sequence diagram illustrates the core user interactions within OpenGalaxy, covering account management, media discovery, and personalisation. It depicts the process of signing up, editing preferences, searching for media, viewing media details, and favouriting media. The diagram highlights how user actions trigger system processes, including database interactions and API requests, ensuring a seamless experience.

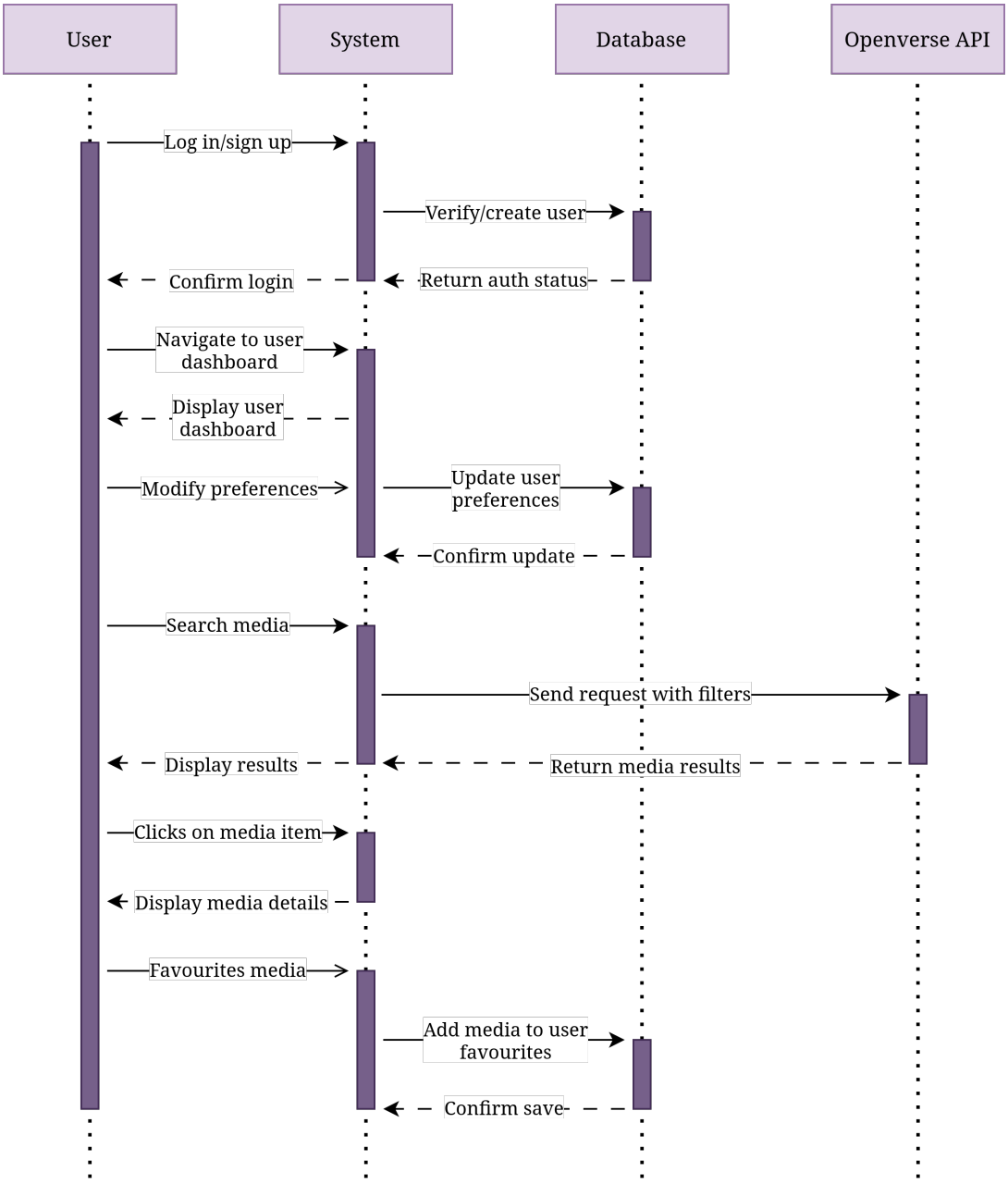

Figure 6: Sequence Diagram of a Typical User Journey in OpenGalaxy.

4.4 UI/UX Wireframes

4.4.1 Landing Page

The landing page includes the website name, a hero, a prominent search bar for discovering media, and links to the user dashboard (or prompts to log in/sign up).

Figure 7: Wireframe for the Landing Page of OpenGalaxy.

4.4.2 Search Results

The search results page includes filters and sorts to refine results by media type and license, allowing users to quickly find content that meets their needs. Media items are displayed as thumbnails with basic metadata (title, license, etc.).

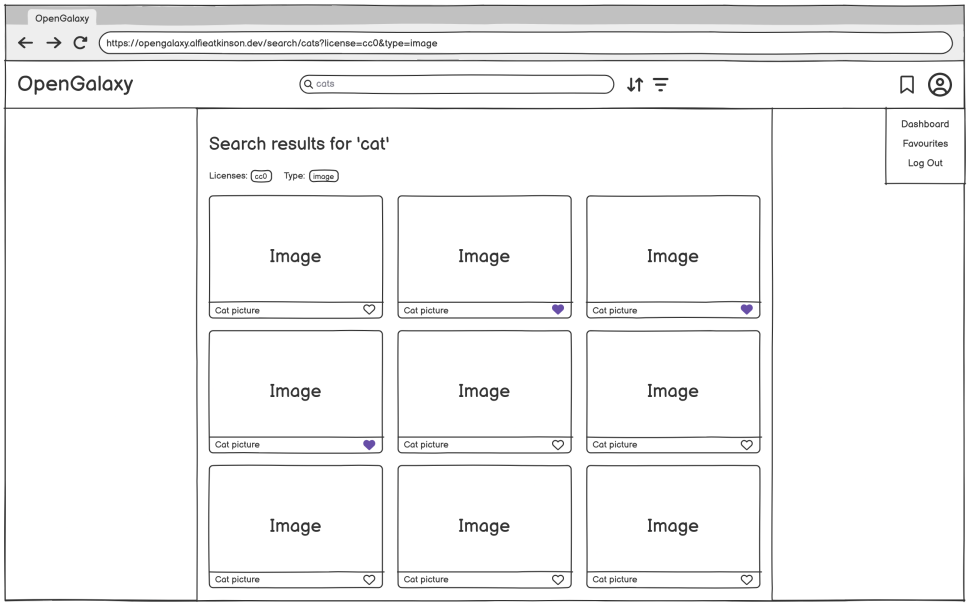

Figure 8: Wireframe for the Search Results Page of OpenGalaxy.

4.4.3 Media Details

The media details page allows users to view a larger preview of the image, along with detailed metadata (creator, license, file size, etc.). The page also includes options to favourite and download the media. Clear attribution and licensing details are shown to ensure compliance with open-license standards.

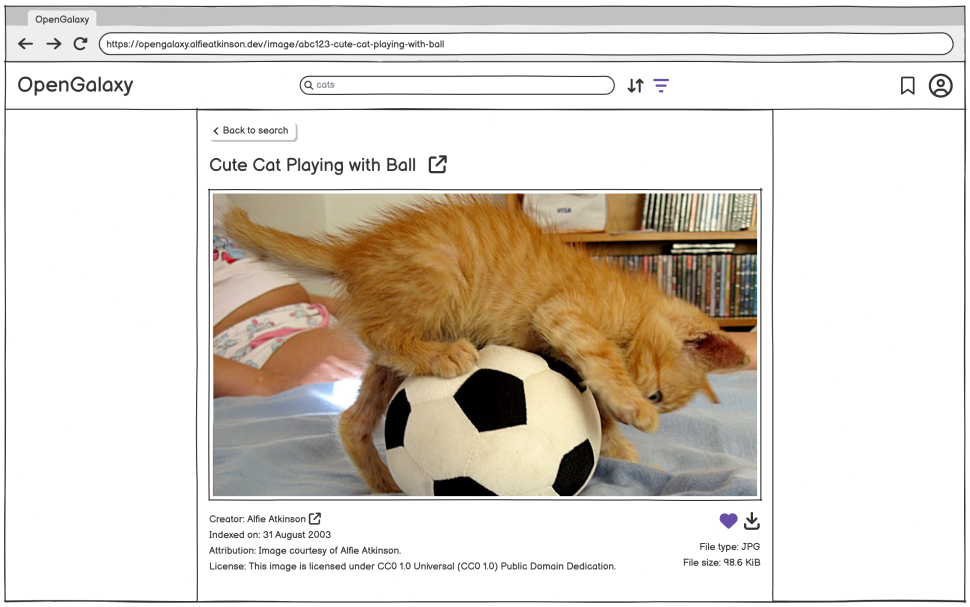

Figure 9: Wireframe for the Media Details Page of OpenGalaxy.

4.4.4 User Dashboard

The user dashboard page allows users to manage their account, preferences, and content interactions. Users can change preferences, favourited media, and update account settings.

Figure 10: Wireframe for the User Dashboard Page of OpenGalaxy.

5 System Deployment

5.1 Hosting & Infrastructure

The OpenGalaxy platform will be hosted using a cloud-based infrastructure to ensure scalability, reliability, and performance. The hosting strategy leverages modern tools and platforms to facilitate a seamless deployment pipeline.

- **Cloud Provider:** The platform will be hosted on Heroku for backend services and Vercel for frontend hosting. Both platforms offer easy-to-use deployment options, automatic scaling, and built-in CI/CD integrations.
  - Heroku will manage the backend infrastructure, including the Django application, PostgreSQL database, and Redis caching system. Heroku's auto-scaling feature ensures the platform can handle increased traffic and usage.
  - Vercel will be used for frontend hosting, providing static file delivery with optimised CDN integration. Vercel is ideal for Next.js applications due to its support for server-side rendering (SSR) and static site generation (SSG), which ensures fast performance and improved SEO.
- **Containerisation:** To ensure consistency between development, staging, and production environments, the platform will be containerised using Docker. This approach ensures that all services are isolated and run in the same environment, eliminating potential discrepancies between environments.
- **Content Delivery Network (CDN):** Vercel will automatically integrate a global CDN, enabling quick content delivery to users worldwide, especially for static assets like images, videos, and media files.
- **Security:** All sensitive configurations, such as database credentials, API keys, and environment variables, will be securely stored using Heroku Config Vars and Vercel Secrets Manager. These services ensure that credentials are not exposed and that only authorized services have access to them.

5.2 Deployment Diagram

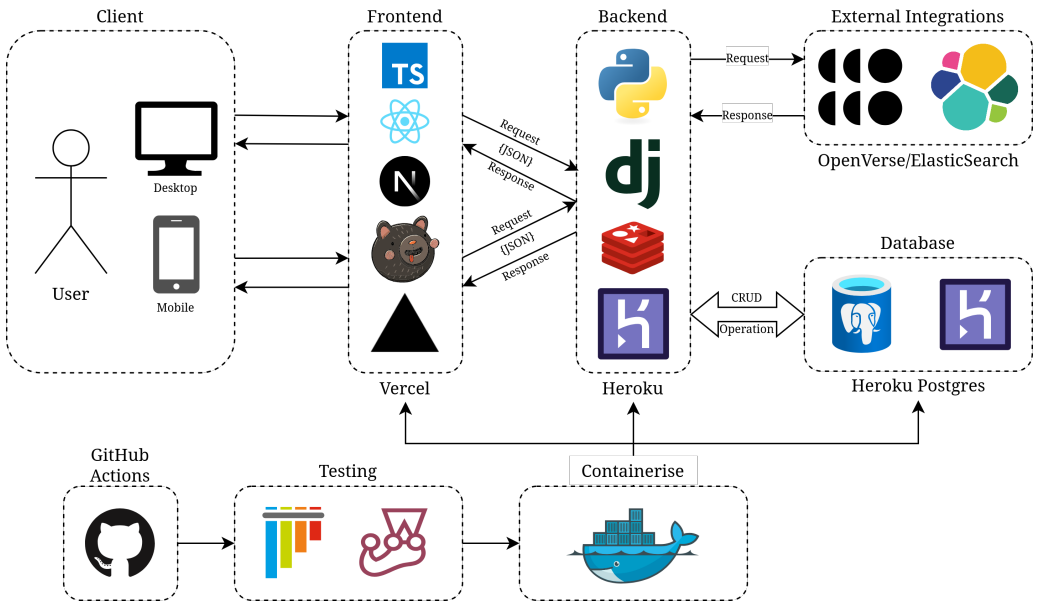

Figure 11: Deployment Diagram for OpenGalaxy

The deployment diagram provides a high-level overview of the deployment architecture, showing where each system component is hosted and how they interact. This includes the following components:

- **Frontend:** The frontend will be hosted on Vercel. Vercel's serverless architecture allows for efficient deployment and automatic scaling, ensuring optimal performance for users accessing the media browsing platform from various devices. Static files, including HTML, CSS, JavaScript, and images, will be served from Vercel's CDN for faster load times and a seamless user experience.
- **Backend:** The backend will be deployed on Heroku, which provides a managed platform for hosting the Django application. Heroku's auto-scaling capabilities ensure that backend services can scale horizontally to handle increased traffic. It will manage REST API requests, user authentication, media search, and data processing.
- **Database:** The database will be managed by Heroku Postgres, a fully managed relational database service that supports high availability and automatic scaling. The PostgreSQL database will store user data, media metadata, search history, and preferences.
- **Caching Layer:** Redis will be used for in-memory caching to store frequently accessed data, reducing the load on the backend and improving response times. Redis will be deployed alongside the backend on Heroku.
- **External Services:** The system will interact with Openverse API to retrieve open-license media and Elasticsearch to provide advanced search functionality. Both services will be marked as external and integrated with the backend via API calls.
- **CI/CD Pipeline:** GitHub Actions will be used for continuous integration and continuous deployment (CI/CD). The pipeline will automate testing, building, and deployment of both frontend and backend applications. It will trigger deployments to Heroku and Vercel upon successful code merges or updates.
- **Security Measures:** All communication between the client and server will be encrypted using HTTPS, ensuring the confidentiality of data. JWT will be used for secure user authentication, and sensitive data will be stored securely in the database using encryption techniques.

6 Security & Performance Considerations

6.1 Security Measures

To ensure the security of the OpenGalaxy platform, several best practices and technologies will be employed:

- **Authentication Mechanisms:** User authentication will be managed using JWT (JSON Web Tokens) for secure, stateless sessions. OAuth will also be integrated to allow third-party authentication (e.g., Google, GitHub).
- **Data Encryption:** All sensitive data will be transmitted over HTTPS, ensuring that data remains encrypted during transit. Database encryption will be implemented to protect stored user information, including personal data and search history.
- **API Security:** Rate limiting will be enforced for the Openverse API integration to prevent overuse and ensure fair access. API keys will be securely managed through environment variables or a secrets manager to mitigate the risk of unauthorized access.

6.2 Performance Considerations

To ensure that OpenGalaxy performs efficiently and scales effectively, the following strategies will be employed:

- **Caching:** Redis will be used for in-memory caching, significantly improving response times by storing frequently accessed media and API results, thus reducing database load and external API calls.
- **Database Indexing:** PostgreSQL will be optimised by indexing key fields, such as media type, license type, and user search history, to enhance query performance and minimise latency.
- **Frontend Optimisation:** Next.js will facilitate server-side rendering (SSR) and static site generation (SSG), improving load times and SEO. Lazy loading of media and components will also be implemented to optimise page load speeds.
- **Scalability:** OpenGalaxy will be designed for horizontal scalability. Auto-scaling features on Heroku and Vercel will manage increased traffic and demand. Docker containers will ensure environment consistency, while Kubernetes may be employed for orchestration in the future, ensuring smooth scaling.