Low Level Design

Contents

1. Introduction

    1. 1 What is low level design document

    1. 2 Scope

2. Technology Stack

3. Components breakdown

    3.1 Frontend

    3.2 Backend

    3.3 Database

    3.4 Dataflow

    3.5 Authenricatione

    3.6 Error handling

    3.7Security consideration

    3.8Deployment

4. conclusion

1. Introduction

1. 1 What is low level design document

The goal of LLD or a low-level design document (LLDD) is to give the internal logical design of the actual program code for Food Recommendation System. LLD describes the class diagrams with the methods and relations between classes and program specs. It describes the modules so that the programmer can directly code the program from the document.

1. 2 Scope

Low-level design (LLD) is a component-level design process that follows a step-by- step refinement process. This process can be used for designing data structures, required software architecture, source code and ultimately, performance algorithms. Overall, the data organization may be defined during requirement analysis and then refined during data design work

2. Technology Stack

- Backend: Python, Flask
- Frontend: HTML,Tailwind css,javascript, Angular
- Database: No Databace
- Version Control: Git

3. Components Breakdown

3.1. Frontend

HTML Structure

- Home Page: Displays featured Anime, Trending,Movies

- Anime Detail Page: Lists all Details of selected anime
- and similar anime .

- Anime Episode Page: Display a video player to play the
- episode and download link.

- Category Page: Displays selected category anime .

- Result Page: Displays anime related to search query

-

Tailwind CSS Styling

- Use Tailwind CSS for styling components and ensuring responsive design.

- Components: Buttons: Customizable with various states (hover, active).

  Cards: For displaying anime with image, title, and language.

  Forms: Styled inputs and buttons for user information.

JavaScript Functionality

- Use JavaScript for:

  - Handling dynamic content (e.g., updating cart totals).
  - Managing local storage for cart items.
  - Validating form inputs on the client side.

3.2. Backend

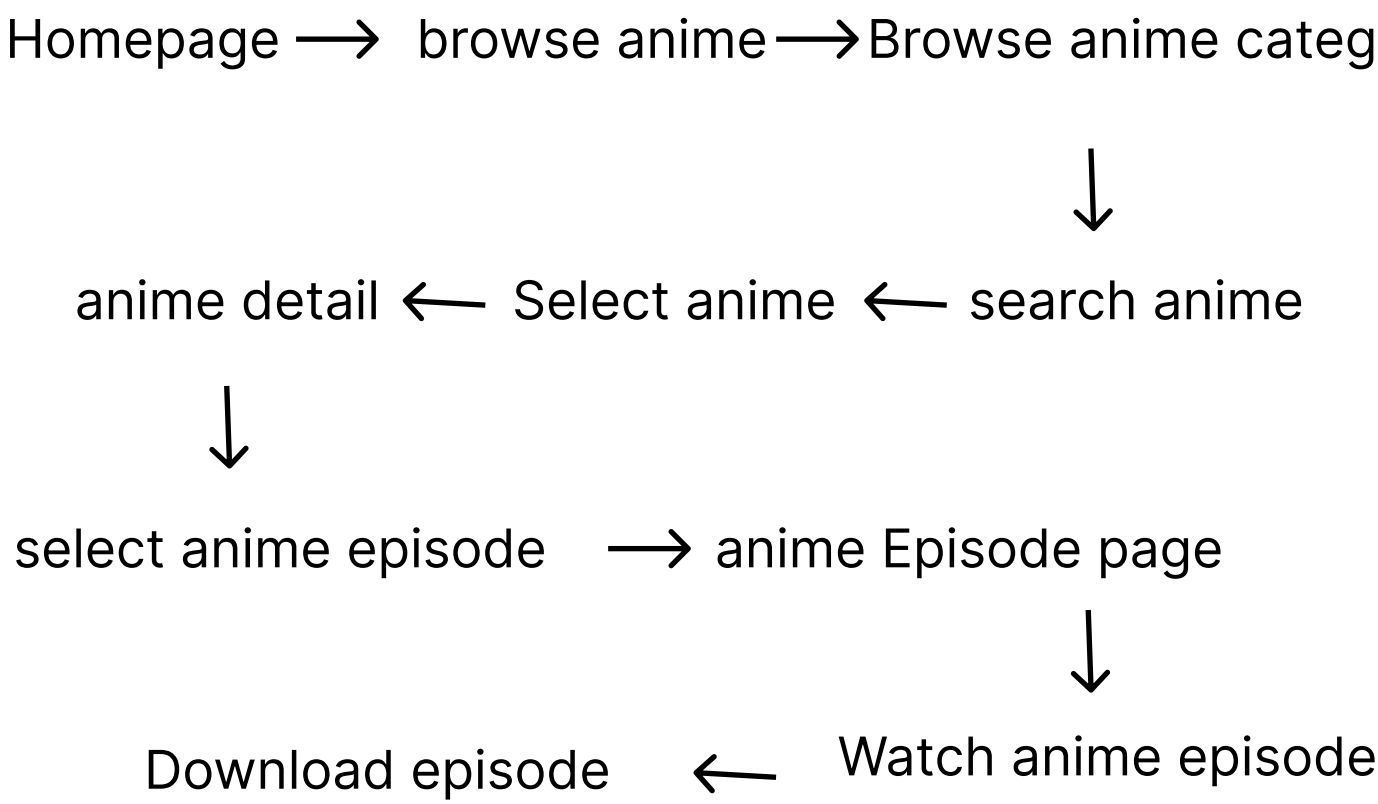
Frontend Routes

- React Routes
    - GET /home: Display homepage.
    - GET /anime_detail: Display anime detail page
    - GET /anime_episode: Display anime episode page
    - GET /home_cat/id:category :Display category page.
    - GET /search:Display searchResult page
- Flask API
    - GET /http://127.0.0.1:5000/home: Fetch anime data of homepage.
    - GET /http://127.0.0.1:5000/home/anime_detail?url=${category}`: Fetch anime detail data of specific anime .
- GET /http://127.0.0.1:5000/home/anime_episode?url=${url} : Fetch episode data of anime.
- GET /http://127.0.0.1:5000/home: Fetch homepage anime data according to category.
- GET /http://127.0.0.1:5000/home/search?name=${category} :Fetch anime related to searched query

Database

In this website it does not have any database of its own instead it used database of of the parent website as it runs on a principle of web scraping as it only has json files in which the data is being stored which are scraped

3.4 Data flow


Homepage
browse anime
Browse anime category
search anime
Select anime
anime detail
select anime episode
anime Episode page
Watch anime episode
Download episode

3.5. Authentication

No authentication is required as it run on web scraping

3.6. Error Handling

- Backend Errors: Implement try-catch blocks and error logging on the backend to capture scraping or API failures and send meaningful error messages.
- Frontend Errors: Use Angular's built-in error handling mechanisms (e.g., ErrorHandler class) to catch and display frontend errors gracefully.
- Graceful Degradation: Provide users with fallback messages or retry options if the API or web scraping service encounters an issue.
- 404 Handling: If a requested anime or page doesn't exist, display a custom 404 page, guiding users to continue browsing.
- Global Error Logging: Utilize services like Sentry or LogRocket to log frontend and backend errors for better diagnostics and monitoring.

3.7. Security Considerations

- Data Validation: Ensure that user input (e.g., search queries) is sanitized to prevent SQL injection and cross-site scripting (XSS) attacks.
- Authentication & Authorization: Implement proper user authentication (e.g., JWT or OAuth) to secure access to personalized features like watchlists and profiles.
- HTTPS Encryption: Use HTTPS to encrypt data transmitted between the client and server, preventing man-in-the-middle attacks.
- API Rate Limiting: Implement rate limiting on the web scraping API to prevent abuse and excessive load on external websites.
- CORS Security: Configure CORS settings carefully to allow only trusted domains to interact with the backend, preventing unauthorized access.

3.8. Testing

- Unit Testing: Use Jasmine and Karma to write unit tests for Angular components, ensuring correct functionality for features like search, navigation, and the video player.
- Integration Testing: Test the integration between Angular frontend and backend APIs to ensure data flows correctly and the scraping logic works as expected.
- End-to-End Testing: Use tools like Protractor or Cypress to simulate real user interactions and test the complete workflow, from searching to streaming.
- Backend Testing: Implement unit tests for the scraping logic using frameworks like pytest or Mocha to ensure the backend correctly scrapes and serves anime data.
- Load Testing: Use tools like Apache JMeter or Locust to simulate high traffic on the website, ensuring the platform can handle large user volumes.

3.9. Deployment

1. Frontend Deployment: Use Render’s static site hosting to deploy the Angular application, providing fast and scalable access to the frontend.
2. Backend Deployment: Deploy the backend API (e.g., Flask or Node.js) on Render’s cloud platform, ensuring it’s accessible via HTTPS and securely communicates with the frontend.
3. Environment Variables: Store sensitive keys (e.g., third-party API keys or database credentials) securely in Render's environment variables for both frontend and backend.
4. Continuous Deployment: Set up automatic deployment on Render by connecting the GitHub repository, ensuring the application is automatically updated whenever changes are pushed.
5. Monitoring: Use Render's monitoring tools to keep track of performance, uptime, and errors to ensure the website is running smoothly post-deployment.

4. Conclusion

This detailed low-level design document provides a comprehensive overview of the architecture, components, data flow, and implementation strategy for the anime streaming website. It serves as a foundation for developers to follow during the implementation phase while ensuring a clear understanding of each part of the project.