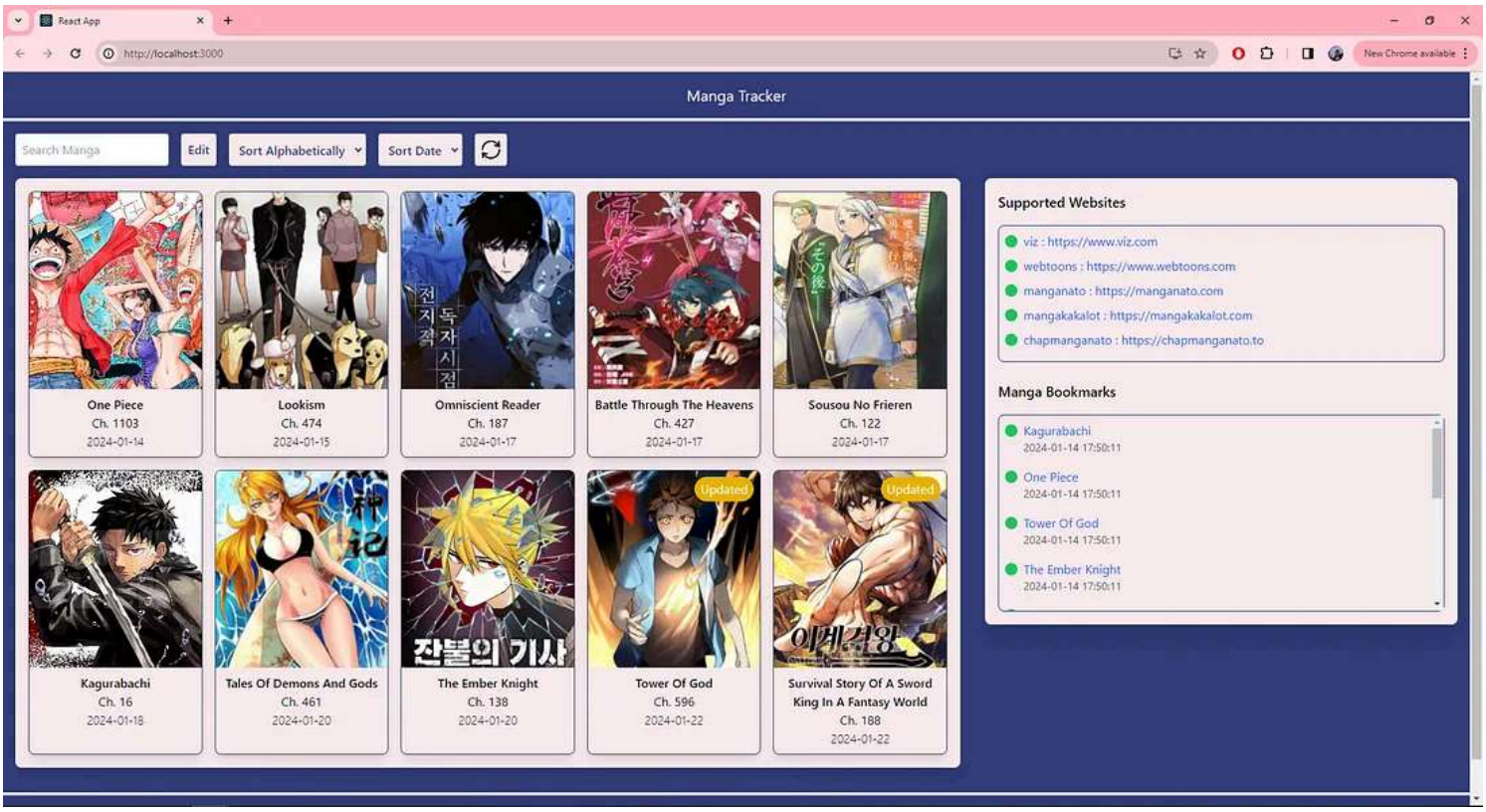
Building a Manga Tracking Application: Journey and Design

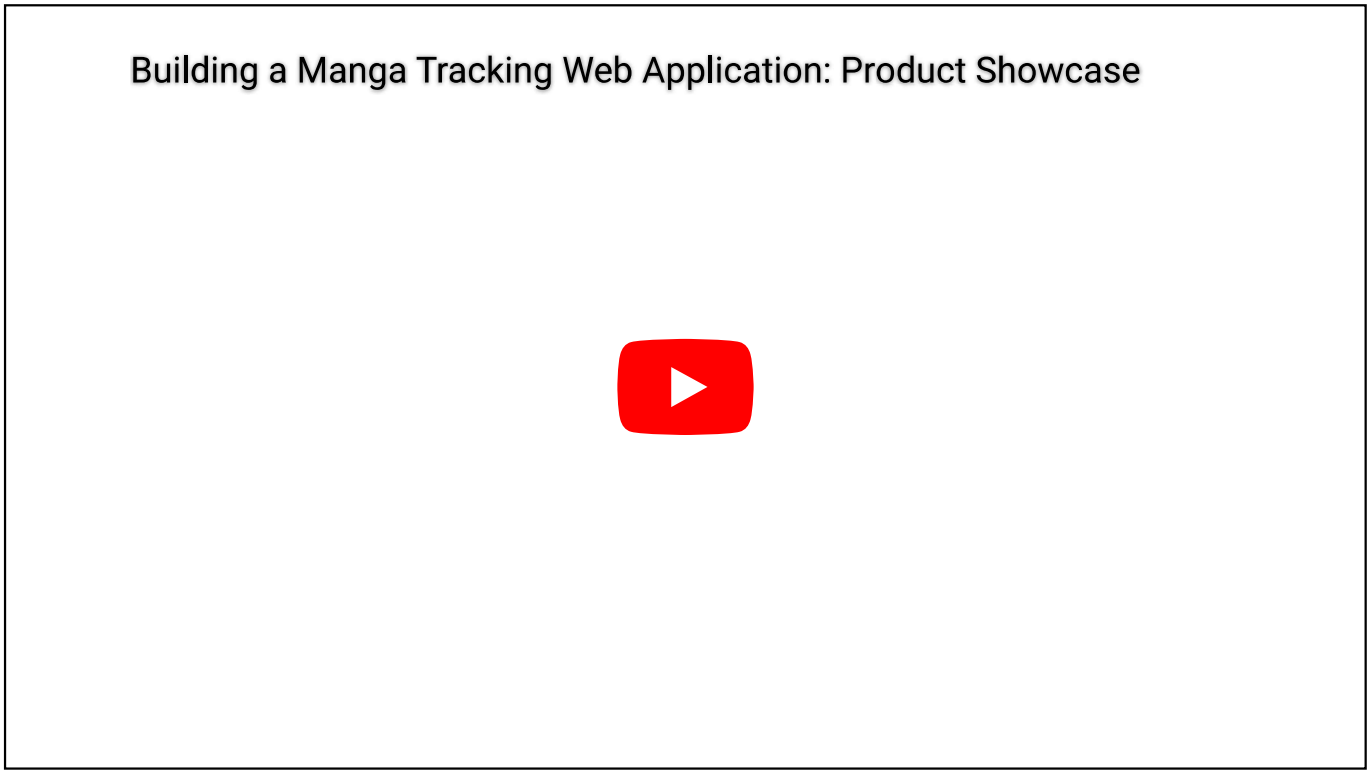
CluuCodes · Follow
22 min read · Jan 29, 2024

In this article, I demonstrate a web application I built, designed for tracking manga updates across websites all within a single interface.



Landing page for application with data pre-populated. (Image by author)

Video Demonstration



DISCLAIMER:

This content of this article explains only the key components I made along with some code snippets for understanding and does not delve in-depth to completely build this. This post will talk about concepts and technologies with little explanation, where I urge the reader to explore further on their own to get a grasp of what is going on.

Code to follow along can be found at my repo here:

It is recommended to read this article in conjunction with the code. Links to specific files in the code will also be hyperlinked throughout this article.

GitHub - christopher-luu97/manga_bookmarker: A bookmarking tool to maintain an updated list of your...

A bookmarking tool to maintain an updated list of your favourite manga across websites! - GitHub ...

github.com

Table of Contents

Introduction

Having been reading manga online for several years, I had built up a large list of manga that I had read and was reading all in my head, without being able to properly track them. Opening multiple tabs, navigating through official sites, translator groups, and other manga aggregators, I found myself spending more time searching for updates for the content I enjoy, wasting time in my day searching for manga I wanted updates about (which are far too many). This frustration ignited the spark for a solution — a personalized manga tracking application.

The Frustration

Opening countless tabs, sifting through irrelevant content, and dealing with the inefficiency of my current methods led me to embark on the journey of building this application. The primary goal was the streamline the process of checking for updates and provide a focused view of the manga I wanted updates on. No more wasted time navigating through numerous tabs and pages to find a single update — the motivation was born out of the need for efficiency.

Why spend time doing manual tasks daily when I can spend many hours more building an automated solution to do just that

The Motivation

“Why not just search for it?” you may ask. The answer lies in the growing complexity of managin an expanding manga list. Searching becomes a tedious task, especially across several websites with varying issues like dropped translations, poor quality translations or images on other websites and more. By creating a tool that simplifies tracking, I ensure a seamless experience even as the list of titles grows.

The Solution

This application was created to address the inconvenience of using multiple websites and to provide a centralized platform that could not only be used by me, but could be leveraged by others that face similar pains to me in the inefficiency of following up on manga updates. By leveraging web scrpaing techniques on selected websites (my personal few I use on a daily basis), the application allows users to build a personalized collection, tracking only the titles they are interested in. Recognizing the challenges posed by the diversity in website structures and data presentation, this project focused on understanding what would be most useful for tracking to tackle my needs. The approach involved inspecting the underlying HTML, identifying key trags for scraping, and extracting essential information such as manga names, chapter links, chapte numbers and manga thumbnails. On a more technical side, this involved creating the data models, tables and relationships to ensure that the data relationships are appropriate, along with developing the frontend approach to render the data.

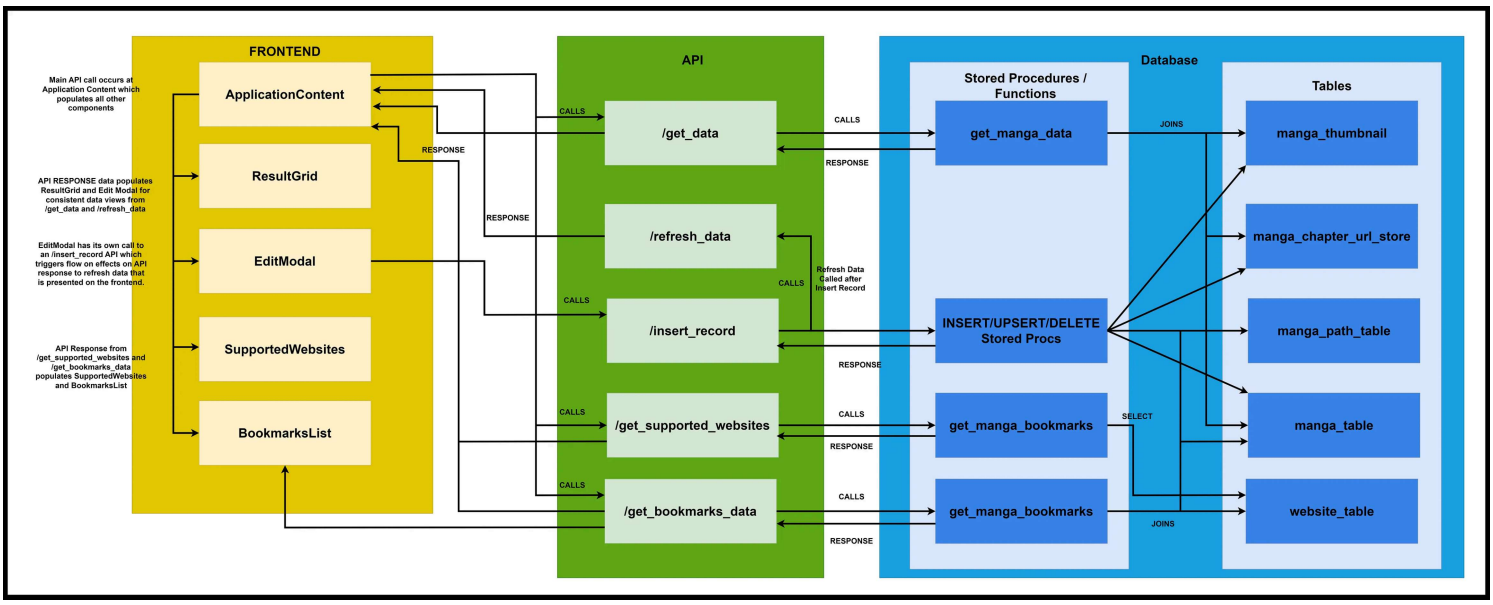
The follow sections describe my journey of designing and implementing this manga tracker, exploring the intricacies of web scraping, database management, and user interface considerations.

Project Overview

The core of this application is rooted in CRUD (Create, Read, Update, Delete) operations. Users can effortlessly manage their manga collection through a variety of features, allowing users to create new records by adding direct manga links, delete records that they are no longer interested in, and get latest chapters for tracked manga with a simple refresh button. The frontend is designed to efficiently display all the data stored in the database, providing a streamlined user experience.

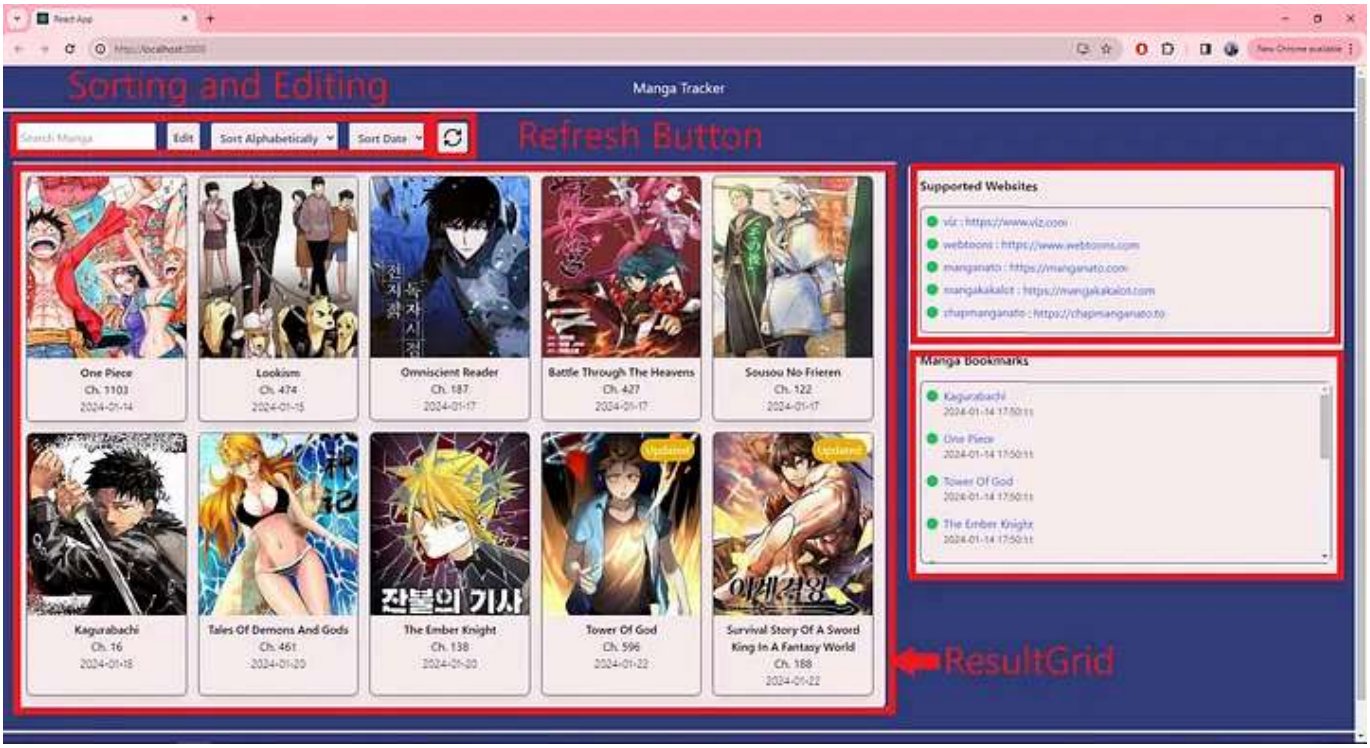
The project utilises React and TailwindCSS for an responsive styling that can be rapidly prototyped, and employing TypeScript for the frontend development. The backend is powered by Python with FastAPI for the API solution. PostgreSQL is the chosen database, utilising a range of SQL scripts, stored procedures and functions for efficient data handling. This combination of technologies ensures a seamless and efficient user experience.

System Architecture



System Architecture. (Image by Author)

The system architecture of this CRUD-based application is designed for performance and user experience. At the frontend, the application displays data retrieved from the backend database. Various APIs facilitate the interaction between the frontend and the backend. A key feature is the main component that showcases images, which not only fetches data, but is also used in the Edit window of the application. This design ensures that the results of CRUD operations are simultaneously reflected in multiple components through usages of state.



Highlighted sections from sorting and editing, refreshing, and side panel containing supported websites and bookmarks component. (Image by author)

The bookmarks list and supported mangas list on the right side panel of the main page operate in a similar fashion, retrieving data from the backend to a dynamic and up-to-date user interface. An important functionality is the ‘refresh data’ button, which initiates a backend API call. This call interacts with the PostgreSQL database, invoking Python scripts to scrape and udpate

records. Post update, the frontend fetches the revised data, thereby refreshing the displayed information and keeping the user informed with the latest updates.

This architecture not only ensures a smooth and interactive user experience, but also maintains the integrity and consistency of data across the application.

Design Decisions

In the development of any application, design decisions play a crucial role in determining the functionality, user experience, and overal success of the project. This section delves into the key choices I made during the development of my manga tracking app, covering aspects such as the integration of frontend and backed technologies, user interface design, data handling, scalability, performance, accessibility, and responsiveness.

Frontend and Backend Integration

For the frontend, I chose React, Typescript and TailwindCSS. React's component-based architecture allows for reusable and maintanable code. TypeScript offers strong typing, enhancing code quality and reducing runtime errors, a valuable assets as projects grow in complexity. TailwindCSS provides utility-first styling, enabling rapid and responsive design without sacrificing aesthetics.

The backend is powered by Python and FastAPI, which is a personal preference for my backend of choice as I have experience in programming with Python (it is a personal favourite) and FastAPI for rapid API development. FastAPI and it's asynchronous capabilities and automatic validation optimize API development, allowing me to integrate the frontend and backend appropriately, haivng them interact with each other well.

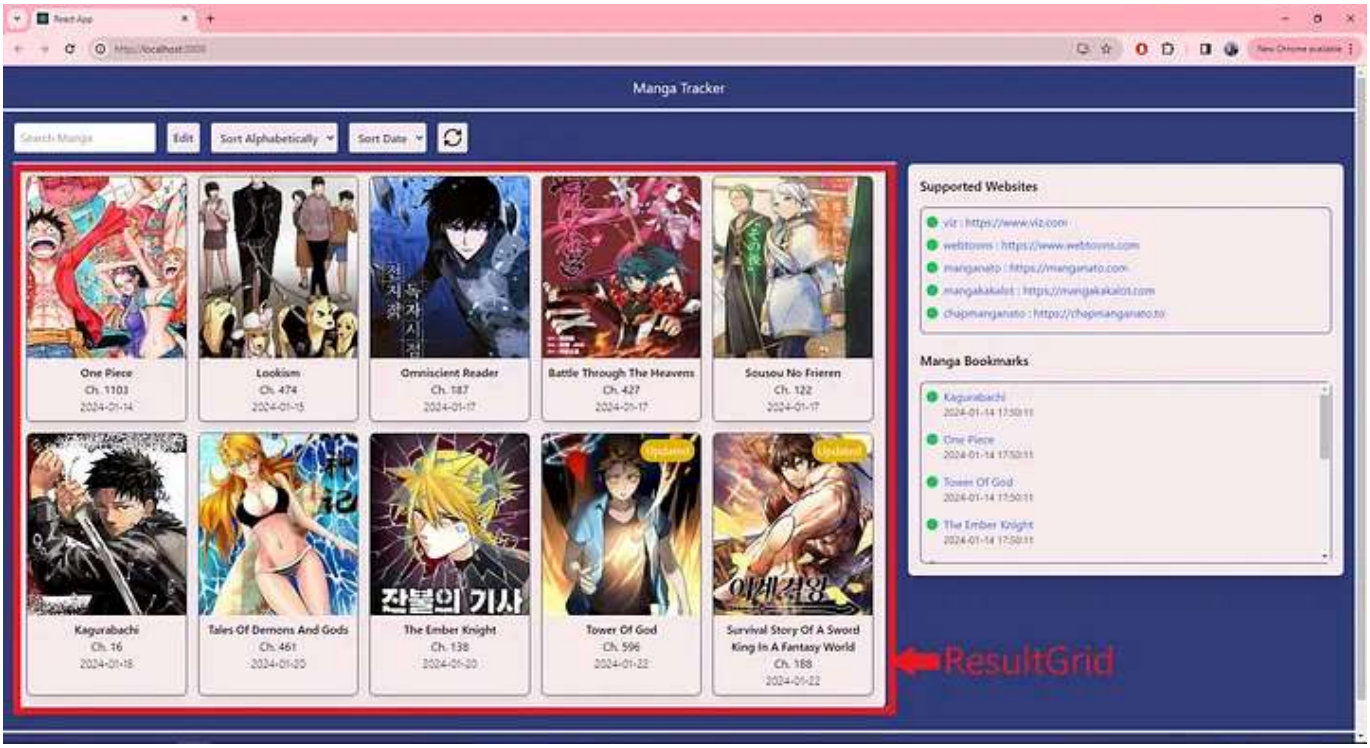
Thee frontend communicates with the backend through RESTful API endpoints created with FastAPI. These endpoints return JSON responses, facilitating smooth data exchange between the frontend and backend. My decoupled architecture through the separation of my frontend and backend, enhances maintainability and scalability.

User Interface and Experience

Frontend Code Here.

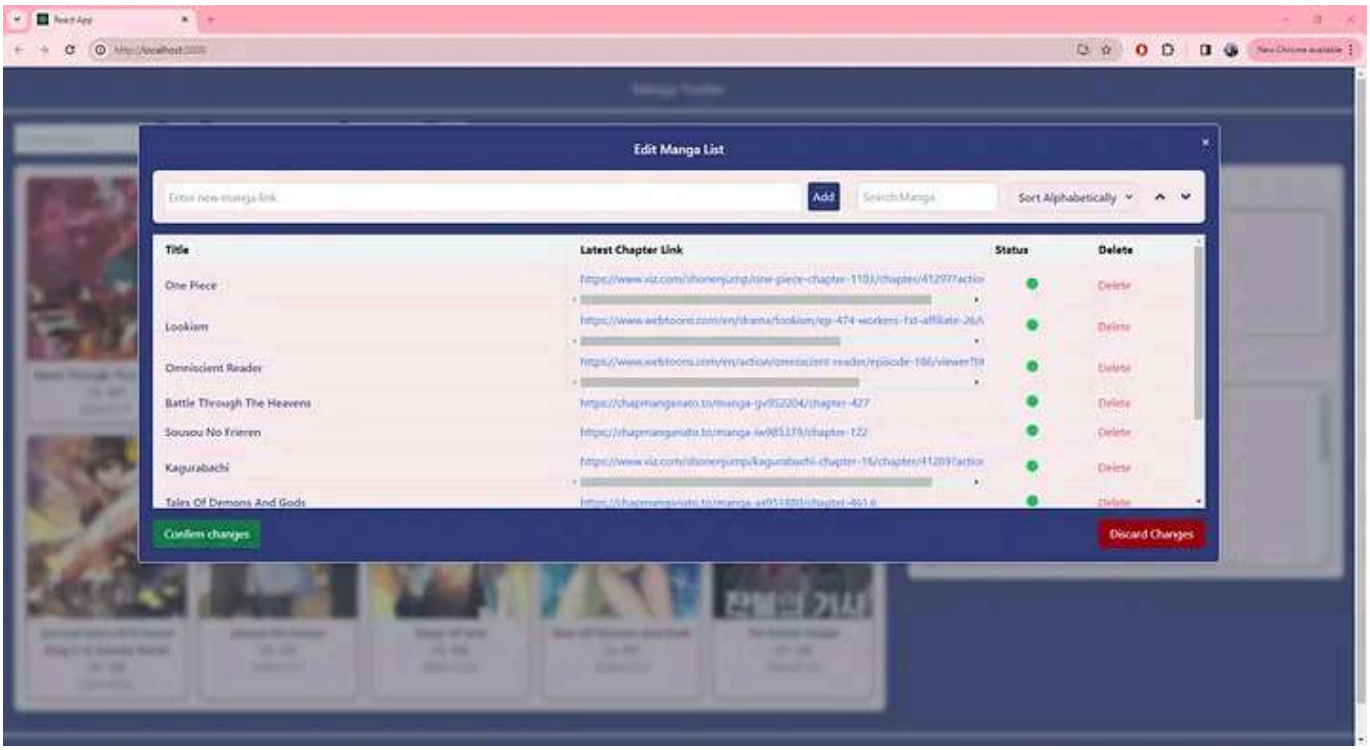
Designing an intuitive and responsive UI involves careful consideration of component interactions, state managements, and data flow. For instance:

- The **ResultsGrid** component is structured to efficiently display manga titles, considering factors like data input and loading times when other actions occur such as refreshing data and handling searching and sorting display



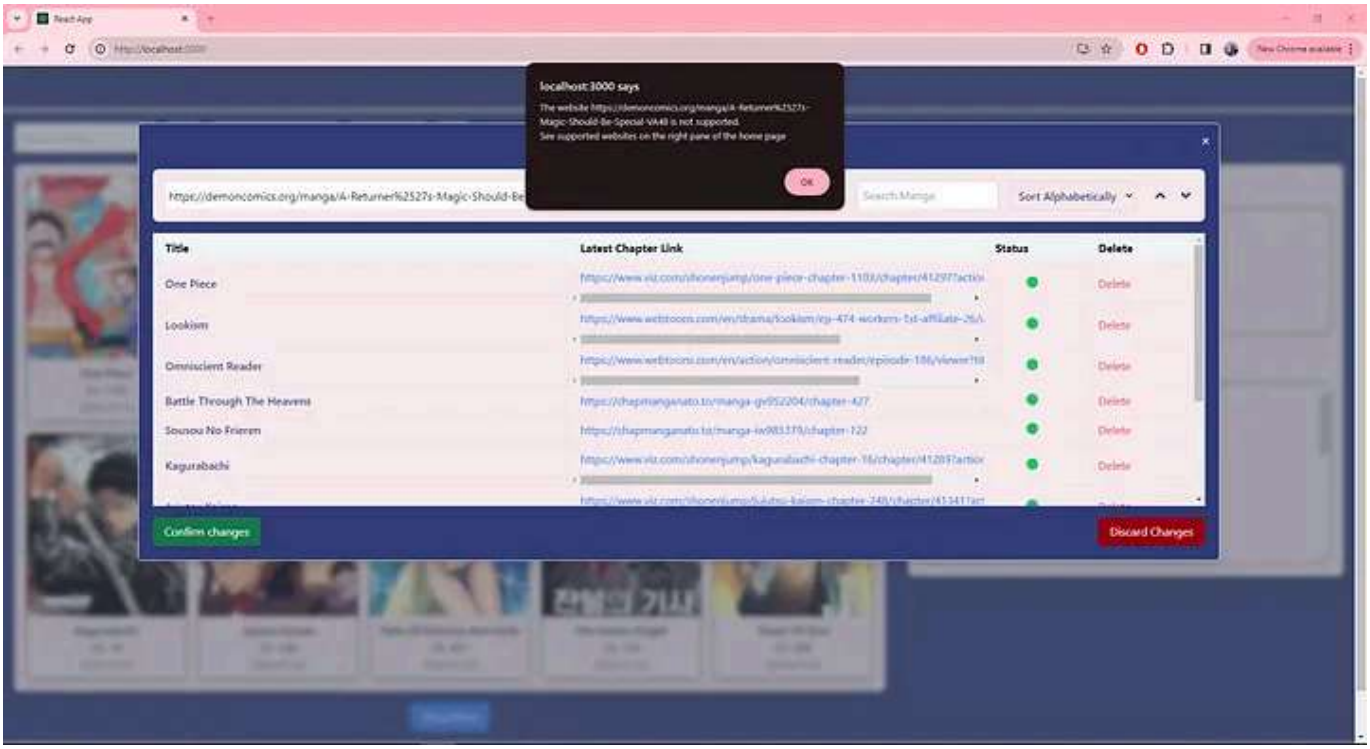
ResulGrid component highlighted on landing page. (Image by author)

- The **EditButton** is designed for ease of use, enabling users to navigate to a separate modal component that handles CRUD operations.
- The **EditModal** component, crucial for the CRUD operations, is crafted for user-friendly interaction and efficient data management, allowing the users to add and delete records, and indicating to the user when operations succeed, fail and are still processing. This component is crucial as it reflects changes immediately both within the modal and on the ResultsGrid component as it finalizes.



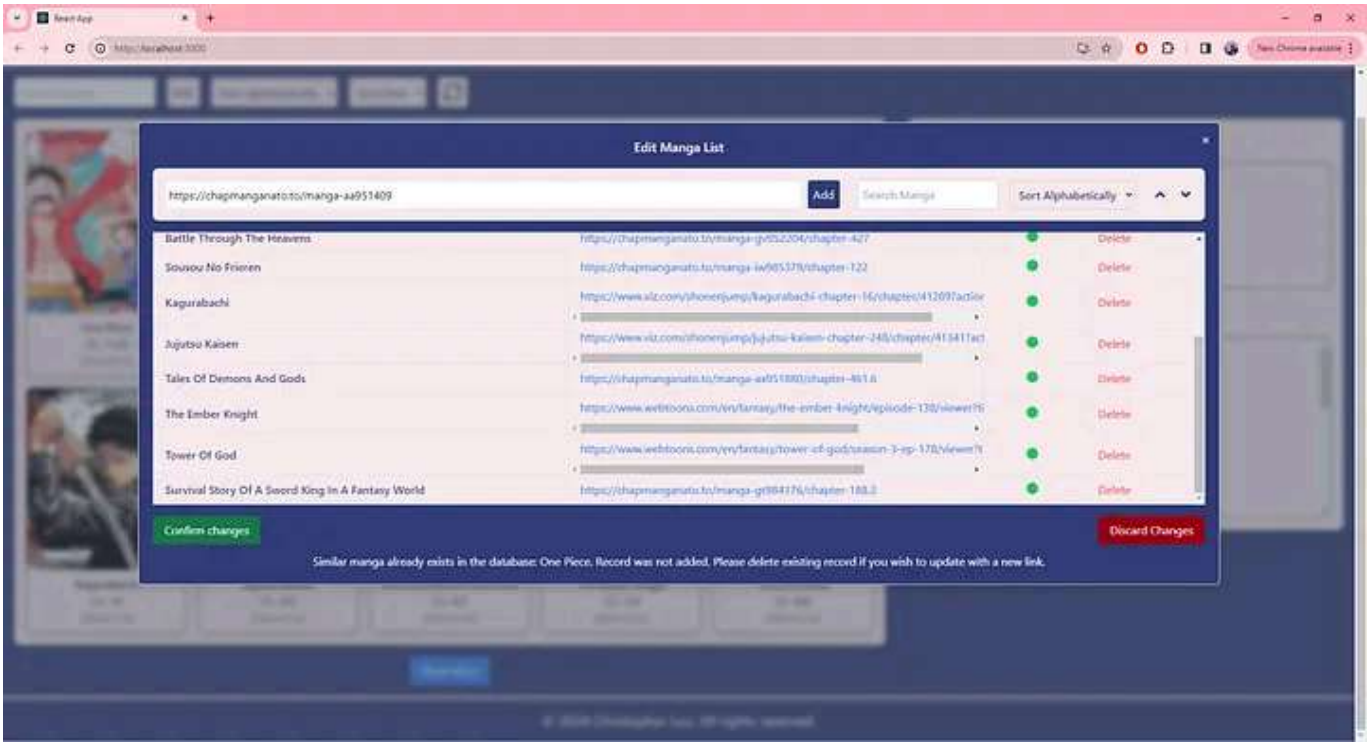
Edit modal to perform CRUD operations. (Image by author)

- **Non-supported websites** is handled by the application as well, where if any non-supported website links are passed through then the user is notified.



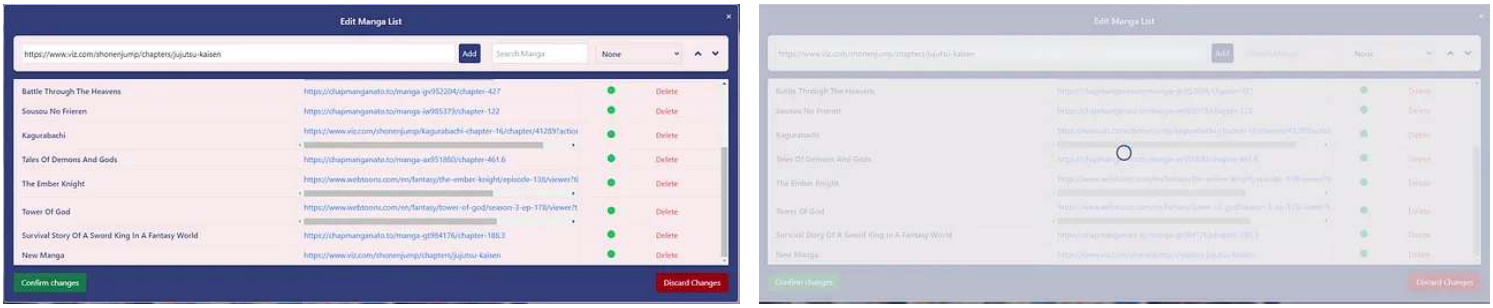
Non-supported websites alert from edit modal. (Image by author)

- **Duplicate records** are handled from inserting by also notifying the user if a record, based off string distance on their name matched to the database is beyond a threshold.

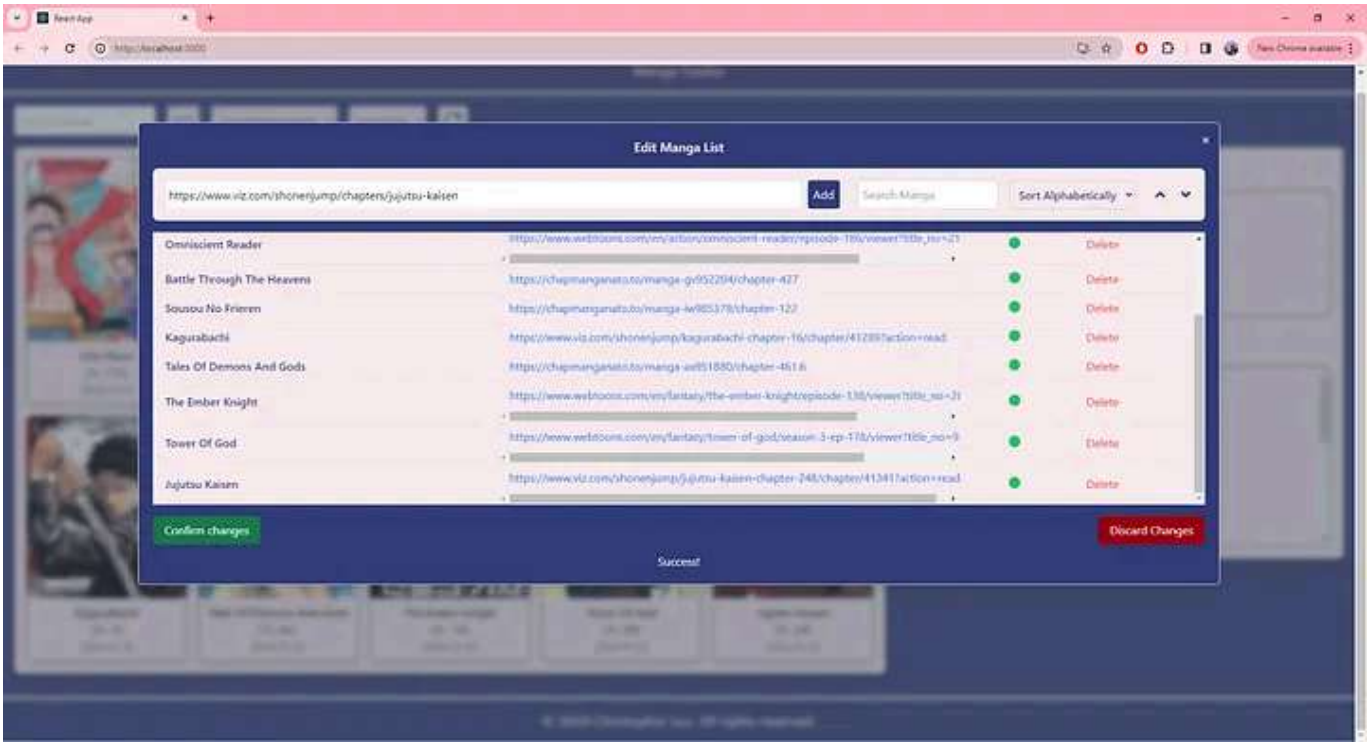


Duplicate record add with message displayed to user. (Image by author)

- Autoscroll is implemented in the edit modal to handle the case for when the list of manga tracked expands, then the user can quickly navigate to the top or the bottom of the modal. This is also implemented when the user clicks “Add” with a new record, which shows the record at the bottom of the edit modal, also scrolling the user to show them that record.

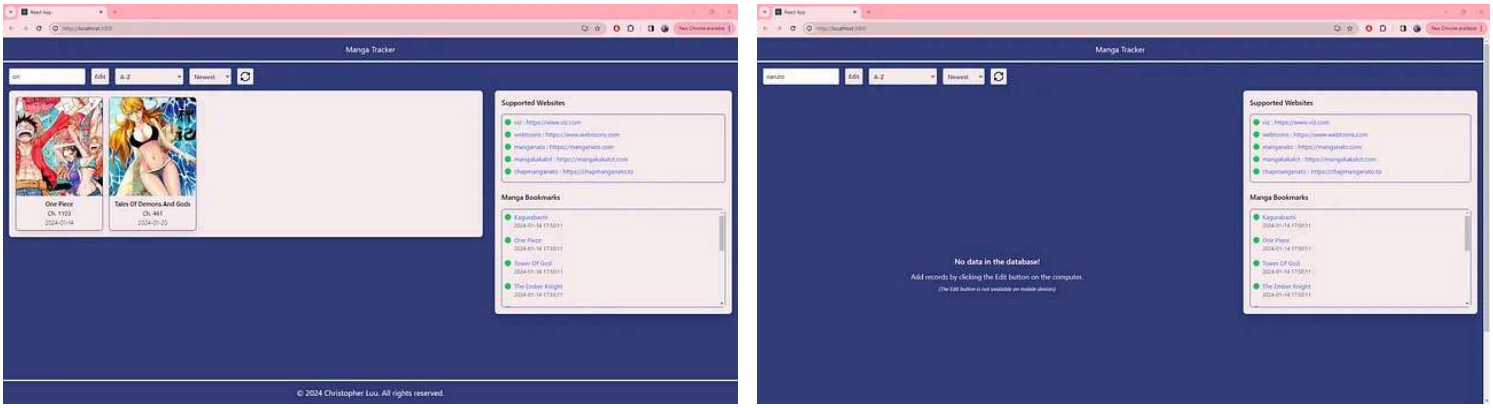


Left: Edit modal adding new record (Jujutsu Kaisen) at bottom. Right: Loading screen while record is scraped. (Images by author)

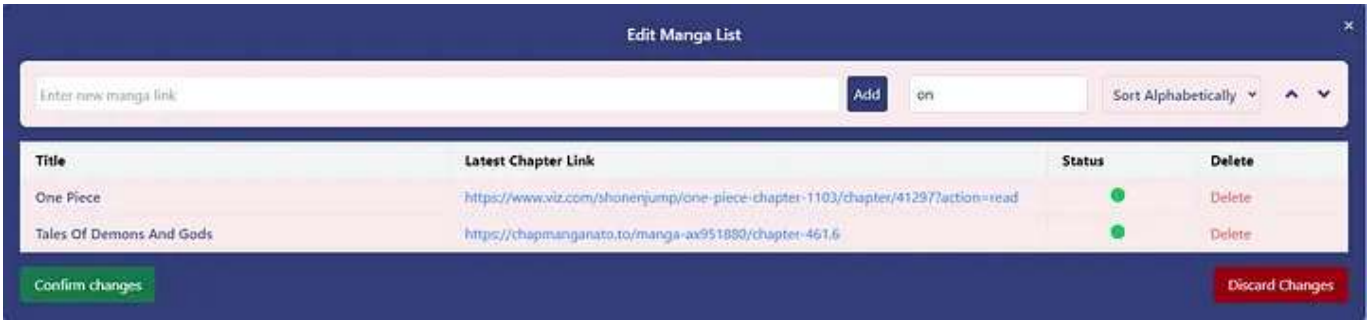


Successful scrape and adding of record (Jujutsu Kaisen) (Image by author)

- **SearchBar** displays real-time filtering based on text input on both the ResultsGrid and EditModal. This strips away the users need to hit the “enter” button to trigger a submit to search, but instead, filters as you type, allowing users to immediately see the results of their search. This component handles casing as well to ensure that results match on string without caring for case.



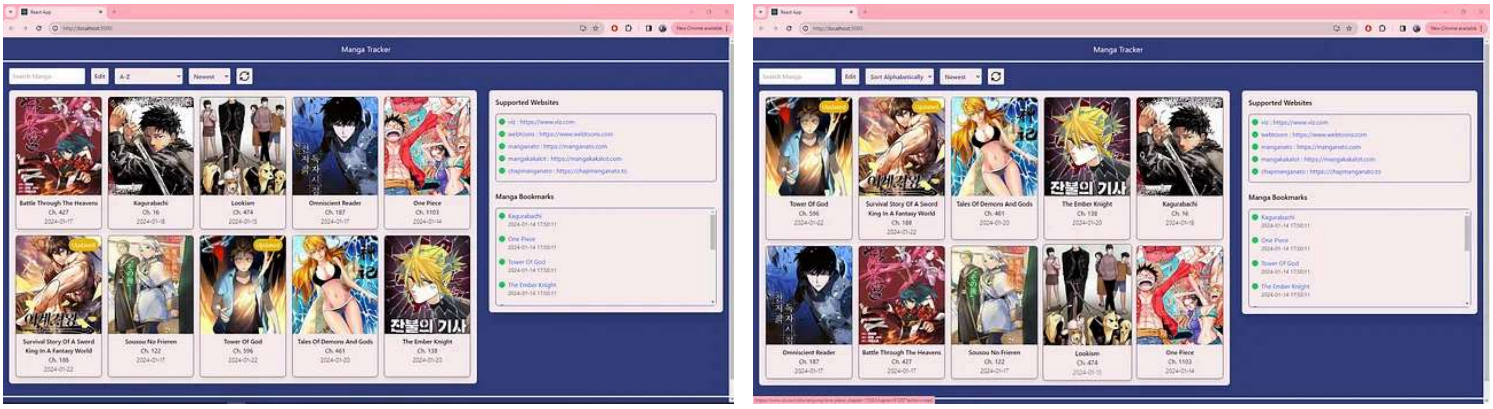
Left: Result grid search when user input’s text “on”. Right: If seach does not exist. (Image by author)



Modal search displays similar results when user searches for text “on”. (Image by author)

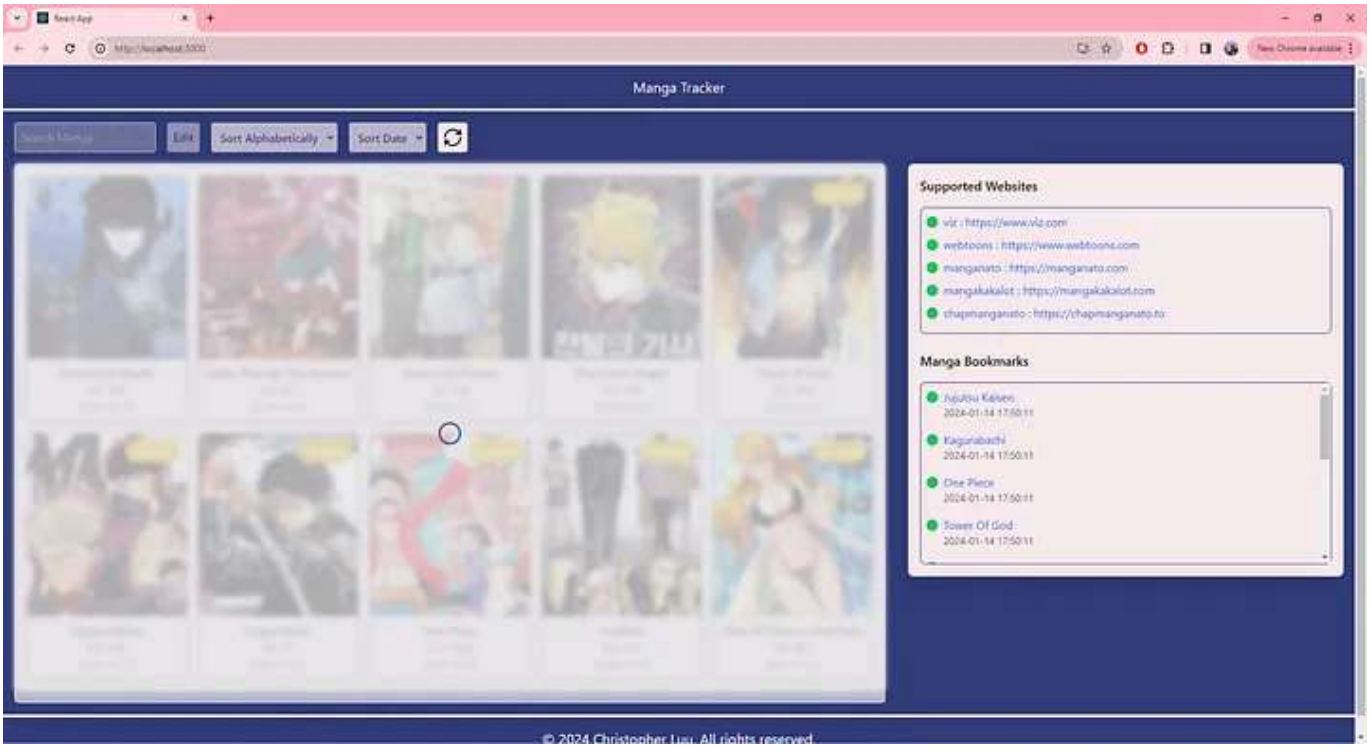
- **AlphabetSort and DateSort** are sorting capabilities on the data for boththe ResultsGrid and EditModal that allow the user to sort data from A-Z (and vice versa) as well as by the latest date the manga chapter record was updated. In combination, the data is sorted by alphabet first and then has their dates sorted within alphabet groups. This ensures that for

each alphabetically sorted group, we understand what the most recently updated record was.



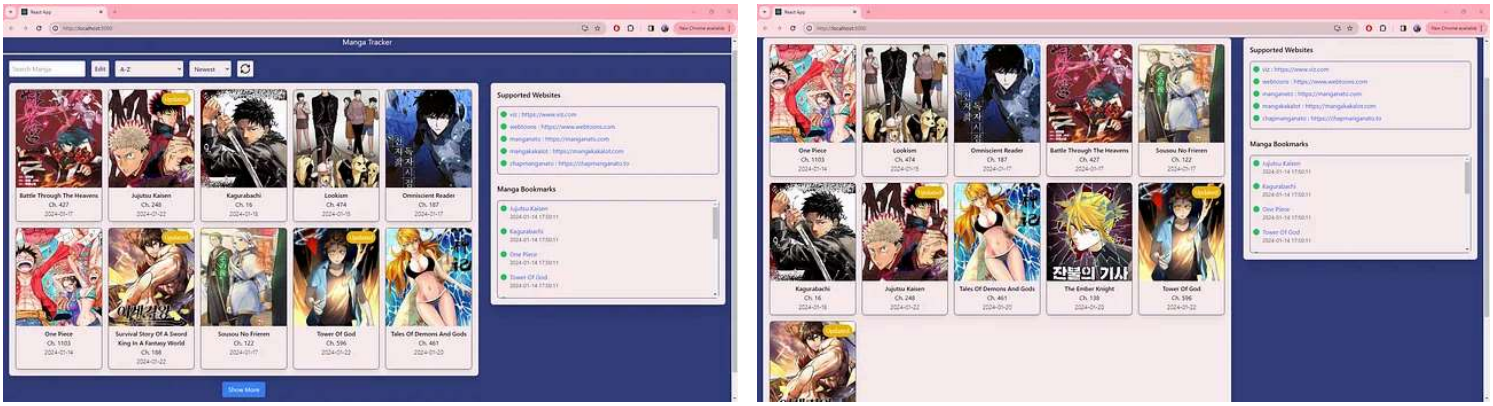
Left: Sort alphabetically and by date. Right: Sort by newest only (Image by author)

- The refresh button and search functionality are optimized for user engagement and efficient navigation through handling of frontend display of data as well as enabling and disabling of certain features when another is in operation to reduce the potential for misuse, leading to potential errors.



Greyed out buttons and search to reduce issues. (Image by author)

- **Show more records** is a button that appears when the resultgrid exceeds it's defined size. This button changes to "Show less" once the result grid has been expanded allowing the user to minimize the number of items displayed.

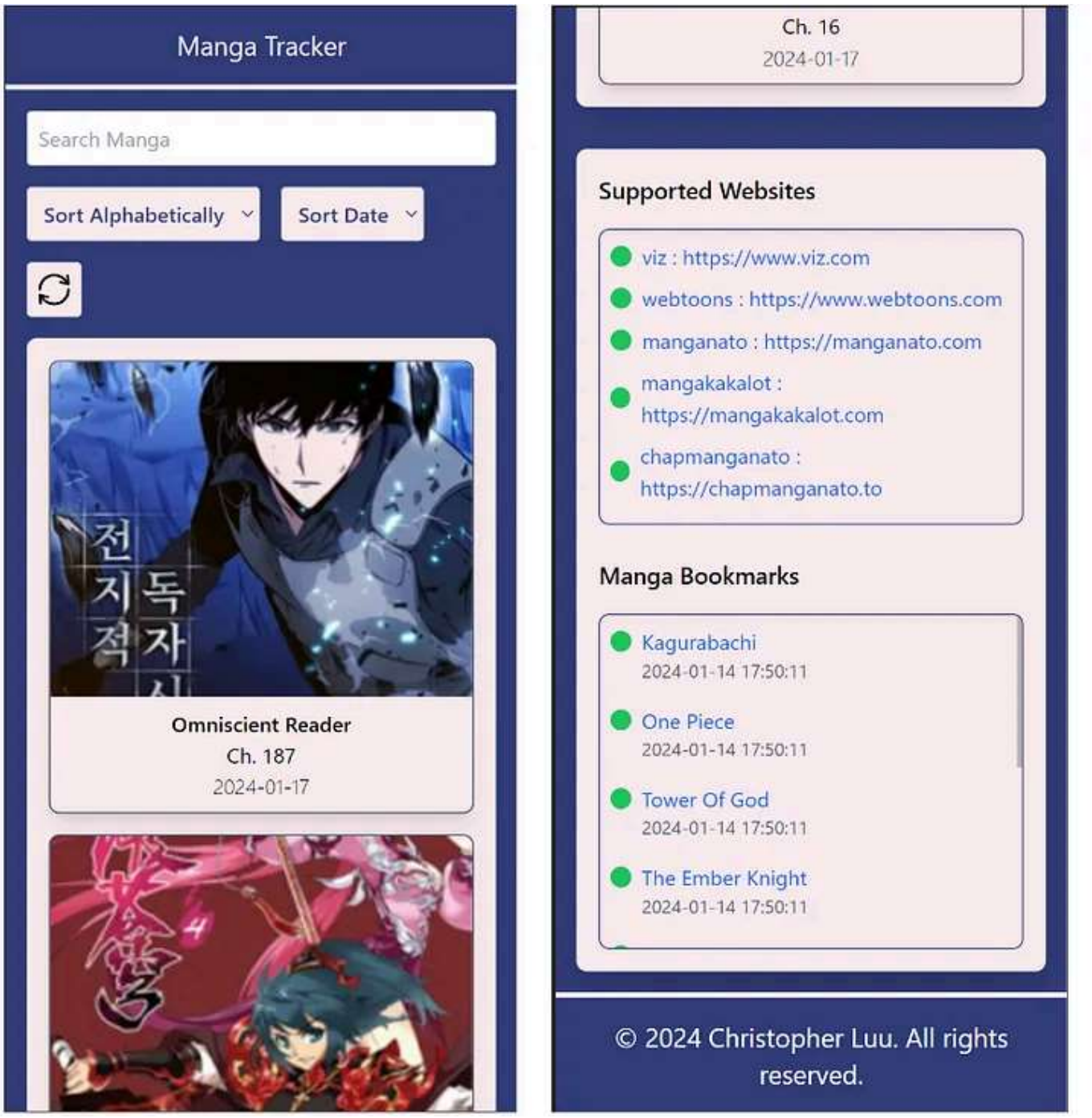


Left: ResultGrid with Show more. Right: Result grid expanded (Image by author)

To maintain a user-friendly experience during data fetches and updates, I implemented loading screens and disabled certain interactions during critical operations. This strategy minimizes complexity and enhances usability, ensuring users are aware of ongoing processes and updated data. Through addition of searching and sorting, enhanced user experience is provided to allow the user to manage their records as they see fit, especially as their data collection expands further and searching for records to be removed or checking if they exist already becomes difficult to manage without such functionality.

The design ensures accessibiltiy and responsiveness across various devices. This is achieved through adaptive layouts, scalable components, and user-friendly navigation tools like auto-scrolling for tables and "scroll to top" and "scroll to bottom" buttons. Furthermore, the editing feature is a computer

browser feature only, removing it's usage on mobile devices, as I deemed performing crud on mobile a bad experience.



Mobile view of the website. Left: Landing page top. Right: Bottom of page. (Image by Author)

These features enable a consistent and enjoyable experience regardless of device used.

Data Handling and State Management

State management is crucial for ensuring a seamless user experience. React hooks like **useState**, and **useEffect** are employed for managing data and tracking changes. This approach ensures that components sharing data are synchronized and updated prompty.

React's **useState** hook is a cornerstone for state management in functional components. It enables components to maintain local state, which can change over time in response to user interactions or other events. In our manga tracking application, **useState** is extensively used to keep track of various pieces of data

The **useEffect** hook is used for side effects, such as data fetching. In the application, it fetches manga data, bookmarks, and supported websites from a backend API. The **useEffect** hook is configured to re-fetch data when the **refreshData** state changes, ensuring that the UI stays up-to-date with the latest data.

These can be seen in my code called **ApplicationContent**. Alternatively, the snippet of useEffect and useState usage can be found below.

```
const [mangaData, setMangaData] = useState<mangaDataInterface[]>([]);
const [refreshData, setRefreshData] = useState(false);
const [bookmarksData, setBookmarksData] = useState([]);
const [supportedWebsitesData, setSupportedWebsitesData] = useState<
  WebsiteData[]
>([]);

useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get("<Some API URL>/get_data");
        setMangaData(response.data);
```

```
        const bookmarksResponse = await axios.get(
          "<Some API URL>0/get_bookmarks_data"
        );
        setBookmarksData(bookmarksResponse.data);

        const supportedWebsiteResponse = await axios.get(
          "<Some API URL>/get_supported_websites"
        );
        setSupportedWebsitesData(supportedWebsiteResponse.data);
      } catch (error) {
        console.error("Error fetching data:", error);
      }
    };

    fetchData();
  }, [refreshData]);
```

PyDantic Models

PyDantic Models are leveraged to ensure that data read from and inserted into the database conforms to a predefined structure. This structured approach to data management is vital for maintaing consistency, reliability, and the integrity of the data throughout the application.

- **Data Validation:** PyDantic automatically valides data against the model's structure, ensuring that only valid data is processed, reducing the risk of data-related errors and inconsistencies.
- **Type Safety:** The explicity typing in PyDantic models ensures that each field contains data of the expected type. This is crucial for maintaining data integrity and preventing type-related issues. (Hence why I also like Typescript)
- **Clarity and Maintenance:** The clear definition of data structures makes the codebase easier to understand and maintain. It also simplifies the process of modifying the data model as the application evolves.

```
from pydantic import BaseModel
from typing import Optional

class MangaRecord(BaseModel):
    chapter_number: Optional[int] = None
    id: str
    imageUrl: Optional[str] = None
    lastUpdated: Optional[str] = None
    link: str
    status: str
    title: str

class MangaList(BaseModel):
    manga_records: list[MangaRecord]
```

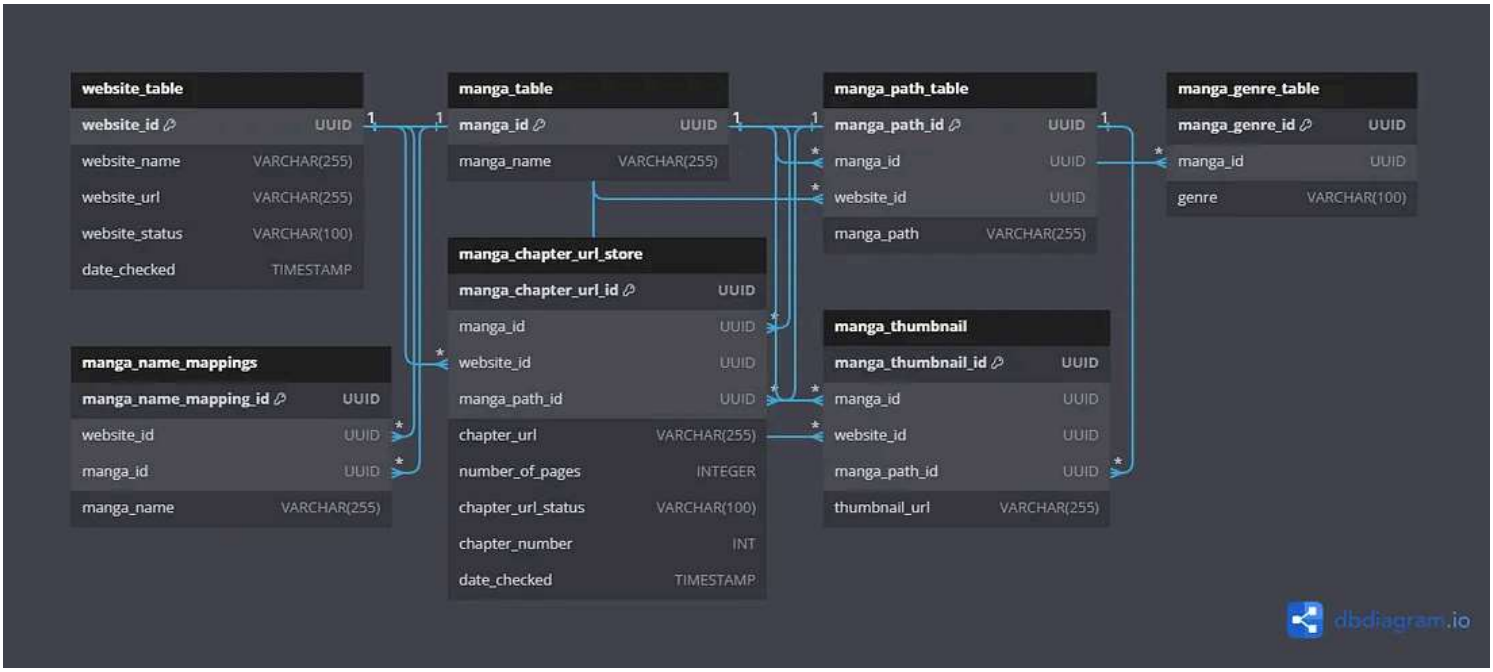
In the context of database design (Section 5) and backend processing (Section 6), these models facilitate seamless data flow between the web scraping logic, backend processing, and database operaitons. They provide a consistent interface for handling manga data, ensuring that every component of the application interacts with well structured and validated data. This model forms the bridge between the frontend, backend, and database, aligning the entire application with a unified and robust data structure.

Database

Database code here

The database design and management played a key role in ensuring efficient data handling and scalability. My approach focused on designing well-structured tables, using stored procedures and functions to handle CRUD operations from the frontend actions. The database schema was crafted to handle not just current requirements but also future expansions such as

incorporating user tables for when the application expands beyond a single user, allowing for multiple users to have their own personalized content.



Entity Relationship Diagram for my Database Schema. Made with dbdiagram.io (Image by author)

Stored Procedures and Functions are used to manage interactions with the database tables. These database routines facilitate data retrieval, updates, and deletions, ensuring that the applications backend remains agile and responsive. The use of stored procedures also enhances security by encapsulating business logic within the database, reducing exposure to SQL injection attacks. Below shows a stored procedure to insert and/or update records for new manga chapters. Whereas the use of stored functions in this application was for data retrieval only.

```
CREATE OR REPLACE PROCEDURE insert_manga_chapter_url_store(
    p_manga_chapter_url_id UUID,
    p_manga_id UUID,
    p_website_id UUID,
    p_manga_path_id UUID,
    p_chapter_url VARCHAR,
```

```
    p_number_of_pages INTEGER,
    p_chapter_url_status VARCHAR,
    p_chapter_number INTEGER,
    p_date_checked TIMESTAMP
)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Check if the record already exists
    IF EXISTS (
        SELECT 1 FROM manga_chapter_url_store
        WHERE manga_id = p_manga_id
          AND website_id = p_website_id
          AND manga_path_id = p_manga_path_id
    ) THEN
        -- Update the existing record
        UPDATE manga_chapter_url_store
        SET
            chapter_url = p_chapter_url,
            number_of_pages = p_number_of_pages,
            chapter_url_status = p_chapter_url_status,
            chapter_number = p_chapter_number,
            date_checked = p_date_checked
        WHERE
            manga_id = p_manga_id
            AND website_id = p_website_id
            AND manga_path_id = p_manga_path_id;
    ELSE
        -- Insert a new record
        INSERT INTO manga_chapter_url_store (
            manga_chapter_url_id,
            manga_id,
            website_id,
            manga_path_id,
            chapter_url,
            number_of_pages,
            chapter_url_status,
            chapter_number,
            date_checked
        ) VALUES (
            p_manga_chapter_url_id,
            p_manga_id,
            p_website_id,
            p_manga_path_id,
            p_chapter_url,
            p_number_of_pages,
            p_chapter_url_status,
            p_chapter_number,
            p_date_checked
        );
```

```
    END IF;
END;
$$;
```

Below is a function which is used to get data to populate the frontend.

```
CREATE OR REPLACE FUNCTION get_manga_data()
RETURNS TABLE(
    id UUID,
    title VARCHAR,
    chapter VARCHAR,
    lastUpdated TIMESTAMP,
    imageUrl VARCHAR,
    status VARCHAR,
    chapter_number INTEGER
) AS $$
BEGIN
    RETURN QUERY
    SELECT
        m.manga_id AS id,
        m.manga_name AS title,
        mc.chapter_url AS chapterLink,
        mc.date_checked AS lastUpdated,
        mt.thumbnail_url AS imageUrl,
        mc.chapter_url_status AS status,
        mc.chapter_number as chapter_number
    FROM
        manga_table m
        JOIN manga_chapter_url_store mc ON m.manga_id = mc.manga_id
        LEFT JOIN manga_thumbnail mt ON m.manga_id = mt.manga_id;
END;
$$ LANGUAGE plpgsql;
```

Data Handling and Future Considerations: The database design emphasizes proper data relationships and integrity, ensuring that manga information is accurately represented and easily accessible. For future enhancements I am considering expanding the user-related features, such as implementing

more sophisticated user profiling, with the potential for recommendation algorithms.

Backend Processing

Backend code here.

The backend processing of my manga tracking application is a orchestration of web scraping, data management, and structuring information to conform to my predifined PyDantic model and database structures. This process is crucial for ensuring that data from various scraped websites are accurately captured, standardized, and stored for efficient retrieval and display on the frontend.

Web Sraping and Data Handling: The core of the backend processing involves scraping data from different manga websites. Each website has its own unique structure and content presentation, requiring tailored scraping strategies. To manage this, I develop specific scraper classes, with a parent class for different child classes that handle peculiarities of individual websites to inherit from, especially for methods that should be implemented across all. Below shows the first few sections of this manga scraping class. Code Here.

```
class MangaScraper:
    def __init__(self, manga_list: List[dict]):
        """
        Initializes the MangaScraper with a list of manga.

        Args:
            manga_list (List[dict]): List of manga with their details.
        """
        self.manga_list = manga_list
        self.soup: Optional[bs4.BeautifulSoup] = None
```

```
    def scrape_manga(self, website_name: str, manga_name: str) -> Tuple[Optional
        """
        Scrapes manga data from a given website.

        Args:
            website_name (str): The name of the website to scrape.
            manga_name (str): The name of the manga to scrape.

        Returns:
            Tuple[Optional[str], Optional[str], str]: Returns the most recent UR
        """
        website_url, manga_url = self.get_urls(website_name, manga_name)
        complete_url = self.normalize_url(website_url + manga_url)
        response = requests.get(complete_url)

        if response.status_code == 200:
            soup = bs4.BeautifulSoup(response.content, 'html.parser')
            self.soup = soup
            return self.parse_html(soup, website_url, complete_url)
        else:
            return None, None, f'Failed to retrieve webpage, status code: {respo
```

These scraper classes are responsible for fetching manga data, including chapter URLs (link to the latest chapters), thumbnails, and manga names. The scraping logic is encapsulated within individual methods used for each class. This approach ensures modularatiy and ease of maintenance, allowing for addition of new website sources of the modificaiton of existing ones with minimal impact on the overall system.

Below shows the first few methods for my MangaScraperService class which is utilized by my APIs. Here we see how different websites are scraped differently, but we ensure that the data returned is all the same format for the database and frontend to work as expected. **Code Here.**

```
class MangaScraperService:
    def __init__(self):
```

```
        self.ms_db = MangaScraperDB()

    def scrape_record(self, manga_list: MangaList) -> Tuple[List[Dict[str, Any]]
        """
        Runner function to scrape and do all the backend stuff

        Args:
            manga_list (_type_): List of manga

        Returns:
            tuple(db_data, error_list): Returns the db object to be upserted int
        """
        error_list = [] # List to store websites that are not supported
        output_list = []
        manga_list = self.get_new_record(manga_list)
        ms = MangaScraper(manga_list)
        mk_scraper = MangaKakalotScraper(manga_list)
        for item in ms.manga_list:
            item_base_url = ms.get_base_url(item.link)
            if "viz" in item_base_url:
                db_data = self.viz_scrape(item, manga_list)
                output_list.append(db_data)
            elif "webtoons" in item_base_url:
                db_data = self.webtoon_scrape(item, manga_list, mk_scraper)
                output_list.append(db_data)
            elif "chapmanganato" in item_base_url:
                db_data = self.mangakakalot_scrape(item, manga_list)
                output_list.append(db_data)
            else:
                error_list.append(item.link)

        return (output_list, error_list)

    def get_new_record(self, manga_list: MangaList) -> List[MangaRecord]:
        """
        Create the new records into a new list that we can iterate over and scra
        Potentially scrape concurrently for N records.

        Args:
            manga_list (MangaList): List of manga
        """
        # From the provided manga list
        # Need to identify all the new ones can create new data structure for th
        # These are item's with a "new_" prefix in the ID

        new_list = [item for item in manga_list.manga_records if "new_" in item.
        return new_list

    def viz_scrape(self, item: Dict[str, Any], manga_list: List[MangaRecord]) ->
        """
```

```
        Scrape data for a manga from the Viz website.

        Args:
            item (Dict[str, Any]): A dictionary containing the manga item detail
            manga_list (list): The list of manga records.

        Returns:
            Dict[str, Any]: The scraped data for the manga.
        """
        vs = vizScraper(manga_list)
        return vs.create_record(item.link)

    def webtoon_scrape(self, item: Dict[str, Any], manga_list: List[MangaRecord]
        """
        Scrape data for a manga from the Webtoons website and leverage MangaKaka

        Args:
            item (Dict[str, Any]): A dictionary containing the manga item detail
            manga_list (list): The list of manga records.
            mk_scraper (MangaKakalotScraper): The MangaKakalot scraper instance.

        Returns:
            Dict[str, Any]: The scraped data for the manga.
        """
        ws = webtoonScraper(manga_list)
        db_data = ws.create_record(item.link)

        # Find manga link in MangaKakalot and extract the thumbnail URL
        search_url = mk_scraper.find_manga_link(search_query=db_data["manga_name
        if search_url:
            db_data["manga_thumbnail_url"] = mk_scraper.extract_thumbnail(search
        else:
            db_data["manga_thumbnail_url"] = "https://NONE"
        return db_data

    def mangakakalot_scrape(self, item: Dict[str, Any], manga_list: List[MangaRe
        """
        Scrape data for a manga from the MangaKakalot website.

        Args:
            item (Dict[str, Any]): A dictionary containing the manga item detail
            manga_list (list): The list of manga records.

        Returns:
            Dict[str, Any]: The scraped data for the manga.
        """
        mk = MangaKakalotScraper(manga_list)
        base_url = mk.get_base_url(item.link)
        return mk.create_record(item.link, base_url)
```

Data Structuring with PyDantic Models: To ensure the consistency and validity of scraped data, I utilized PyDantic models for structuring and validating data before it is processed for database insertion. PyDantic models provide a layer of abstraction, allowing the backend to handle complex data transformations and validations efficiently. This ensures that only well-structured and valid data is persisted in the database, reducing the likelihood of data-related errors and inconsistencies.

Database Interactions: The MangaScraperService class manages the overall workflow of scraping, structuring, and storing data. It includes methods to scrape records, insert records to the database and deleting records from the databaselike that orchestrate the various steps involved in processing manga data. The class interacts with the database through the MangaScraperDB class, executing operations such as inserts, updates, and deletes.

Below shows my MangaScraperDB class with it's init and the select method to insert manga chapters and their url (also shown above in the SQL script) by calling the stored procedure above and inserting in parameters.

Code Here.

```
class MangaScraperDB:
    """
    Class that connects and performs insertions to the database hosted on Postgr
    """
    def __init__(self):
        try:
            creds_path = os.path.join(os.getcwd(),"src/secrets/db_config.json")
            credentials = self.read_db_credentials(creds_path) ## To change into
```

```
            self.conn = psycopg2.connect(
                host= credentials["host"],
                database= credentials["database"],
                user= credentials["user"],
                password= credentials["password"]
            )
        except OperationalError as e:
            print(f"Error connecting to PostgreSQL database: {e}")
        except (FileNotFoundError, KeyError) as e:
            print(f"Error reading database configuration: {e}")

    def insert_manga_chapter_url_store(self, record: Dict, manga_id:str, website_
        """
        Insert the full link to the latest manga chapter

        Args:
            record (Dict): Dictionary containig all relevant information
            manga_id (str): Relevant manga id
            website_id (str): Relevant Website ID
            manga_path_id (str): Relevant manga path id
        """
        manga_chapter_url_id = str(uuid.uuid4())
        try:
            with self.conn.cursor() as cur:
                cur.execute("CALL insert_manga_chapter_url_store(%s, %s, %s, %s,
                        (manga_chapter_url_id,
                         manga_id,
                         website_id,
                         manga_path_id,
                         record["chapter_url"],
                         int(record["number_of_pages"]),  # Ensure this is an in
                         str(record["chapter_url_status"]),
                         int(record['chapter_number']),
                         datetime.strptime(record["date_checked"], '%Y-%m-%d %H:
                        ))
                self.conn.commit()
        except Exception as e:
            print(f"Error in insert_manga_chapter_url_store: {e}")
            self.conn.rollback()
        return manga_chapter_url_id
```

The backend processing of the manga tracking application is a complex yet well-organized system that efficiently handles data from multiple sources. The combination of targeted web scraping, structured data models, and

robust database interactions forms the backbone of the application, enabling it to provide users with up-to-date and accurate manga information. This backend architecture not only ensures the seamless operation of the application but also lays a strong foundation for future enhancements and scalability.

Scalability and Performance

Don't read things all into memory. Scaling up will slow calculations and processing down as well as handling data base I/O.

useMemo alleviates some of this, such as in my searchbar where calculations are saved in memory and we don't have to re-render the site and calculations every time.

React's **useMemo** hook is utilized to optimize performance by memoizing expensive calculations. In our application, sortedAndFilteredMangaData is computed using useMemo, which ensures that sorting and filtering operations are only re-computed when necessary (i.e., when mangaData, searchTerm, sortAlphabetOption, or sortDataOption change). This optimization is crucial for maintaining smooth performance, especially when dealing with large datasets. This is shown in the function below which performs and does many calculations within useMemo to perform sorting by alphabet, date as well as filtering via search.

```
const [mangaData, setMangaData] = useState<mangaDataInterface[]>([]);
const [searchTerm, setSearchTerm] = useState("");
const [sortAlphabetOption, setSortAlpabetOption] = useState("none");
const [sortDateOption, setSortDateOption] = useState("none");


const sortedAndFilteredMangaData = useMemo(() => {
    /**
```

```
 * Memoized function to sort and filter manga data based on specified option
 *
 * This function performs combined sorting, grouping by alphabet, and orderi
 * It first sorts the data alphabetically if the option is selected, then gr
 * After that, it sorts each group by date (newest to oldest) if the date op
 *
 * @returns {mangaDataInterface[]} - Sorted and filtered manga data.
 */
const performSortingAndFiltering = () => {
  // Initial sorted data is a copy of the mangaData array
  let sortedData: mangaDataInterface[] = [...mangaData];

  // Sort alphabetically if the option is selected
  if (sortAlphabetOption !== "none") {
    sortedData = [...mangaData].sort((a, b) =>
      sortAlphabetOption === "alphabetical"
        ? a.title.localeCompare(b.title)
        : b.title.localeCompare(a.title)
    );
  }

  // Group data by alphabet and sort within groups based on date if the date
  if (sortDateOption !== "none") {
    const groupedData: mangaDataInterface[][] = sortedData.reduce(
      (acc: mangaDataInterface[][], manga) => {
        const lastGroup = acc[acc.length - 1];

        if (
          !lastGroup ||
          (sortAlphabetOption === "alphabetical" &&
            manga.title.charAt(0).toLowerCase() !==
              lastGroup[lastGroup.length - 1].title
                .charAt(0)
                .toLowerCase()) ||
          (sortAlphabetOption === "reverse-alphabetical" &&
            manga.title.charAt(0).toLowerCase() !==
              lastGroup[lastGroup.length - 1].title.charAt(0).toLowerCase())
        ) {
          // If it's a new group, create a new group with the current manga
          acc.push([manga]);
        } else {
          // If it's the same group, add the manga to the existing group
          lastGroup.push(manga);
        }

        return acc;
      },
      []
    );
```

```
    // Sort each group based on date and flatten the array
    sortedData = groupedData
      .map((group) =>
        group.sort((a, b) => {
          const dateA = new Date(a.lastUpdated).getTime();
          const dateB = new Date(b.lastUpdated).getTime();

          return sortDateOption === "newest-date"
            ? dateB - dateA
            : dateA - dateB;
        })
      )
      .flat();
  }

  // Filter the sorted data based on the search term
  return sortedData.filter((manga) =>
    manga.title.toLowerCase().includes(searchTerm.toLowerCase())
  );
};
```

Where this code and updated data is used to dynamically update the result grid and is simply used by inserting it within the HTML as data.

```
<ResultsGrid mangaData={sortedAndFilteredMangaData} />
```

Future Improvements

There are several improvements to the application that could be made, from a UI perspective, backend design of the system and coding improvements.

Being more of a backend and data dev rather than frontend, I recognise my frontend code has a lot of improvement to be made. These include separating out large blocks of code into modular components and not

hardcoding in API paths, but to instead read these paths from a file. This way, edits to the API paths can be made much more easier.

For scalability and performance, the existing format always fetches ALL data from the database to render on the frontend and sends a request with ALL data back to the backend to process, which then gets processed sequentially via a for loop. As the system is currently small, these wait times for processing and manga scraping are within reason, but as the system expands this would be unacceptable and would have to be addressed. This can be addressed through multiple approaches, and will be discussed in a future article when I have had approached this need for speed.

Challenges and Learning Experiences

During the development of the manga tracking application, I encountered several challenges, each providing valuable learning experiences.

One of the primary challenges I faced was in the realm of web scraping thumbnails that could be rendered on the frontend. Although my initial strategy worked when I started off with scraping Viz, scraping from Webtoons was a hurdle to overcome as their images were hosted on servers beyond my access. This limitation led me to find an alternative which is to scrape from Mangakakalot, which assumes that Mangakakalot includes the Webtoons in their database. This experience taught me the critical importance of choosing data sources wisely and adapting to technical constraints to ensure a comprehensive dataset for my application.

Another key challenge was designing the database. My goal was to create a structure that not only catered to my immediate needs for tracking manga but also had the flexibility to support future feature development. This forward-looking approach in database design underscored the significance

of scalability and adaptability, essential qualities for building robust and extendable systems.

The biggest challenge was on integrating the various components of the application — from the frontend and API to data processing and the database — was a complex endeavor. Each element needed to work in harmony, and even minor inconsistencies could lead to issues that were difficult to debug quickly. This integration process highlighted the complexity of managing state and API interactions in the frontend, particularly as the project scope grew.

Through tackling these challenges, I gained valuable insights into the intricacies of web scraping. Understanding each website's structure and the data available was crucial for developing an efficient scraping strategy and informed my approach to database design. Planning for a full-stack application required careful consideration of numerous design decisions, ultimately enriching my knowledge and experience in system design. This project highlighted the importance of flexibility, strategic planning, and understanding both frontend and backend technologies. Such experiences are crucial for any developer, as they enrich our skill set and prepare us for future challenges in the dynamic world of technology.

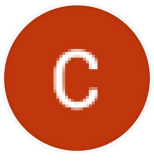
Conclusion

Reflecting on the journey of building this manga tracking application, the key takeaway is the complexity and effort required in web scraping, especially when aggregating data from multiple sources which have their own structure and data format that needs to be considered in conjunction with the defined/planned structure you may have in mind. It reinforced the importance of planning to minimize rework and technical debt.

Looking ahead, I plan to deploy this application on my Raspberry Pi for website hosting, incorporating user login and authentication features, and separating out the database to it's own server. My ultimate goal is to develop a feature akin to Spotify Wrapped, but for manga readers and this all starts with being able to track what manga is read for a user first (and that user is me to start with). This vision drives my ongoing commitment to enhancing the application, ensuring it remainds a tool I can use and potentially for others too if I deploy it to the wider internet.

Stay tuned for future updates and new features as I continue to refine and expand this project!

Manga | Software Development | Web Development | Python | React



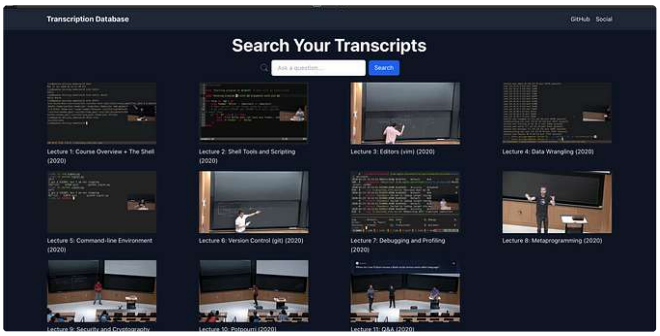
Written by CluuCodes

Follow

5 Followers

Software engineer, data analyst, consultant by trade and super passionate about building software products. Follow along to see more projects built!

More from CluuCodes



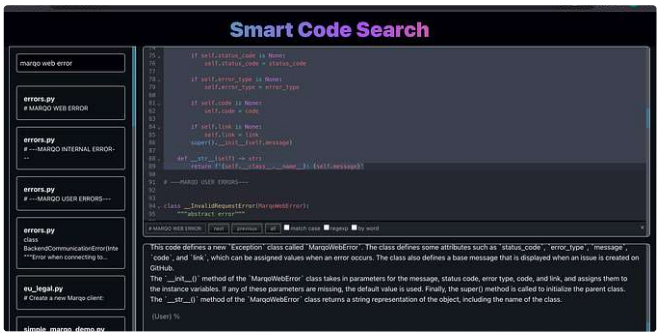
CluuCodes

How I Made a Complete Vector Search Web Application with...

In this article, I demonstrate how I built a web application designed for quick natural...

12 min read · Aug 7, 2023

66



CluuCodes

Code Search and Question Answering using Marqo and a...

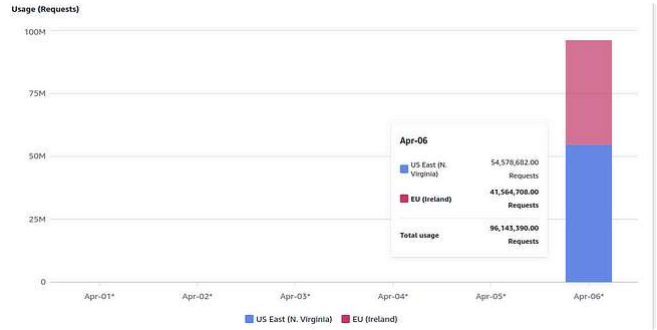
In this article, I demonstrate how I built a web application designed for offline search and...

21 min read · Oct 29, 2023

2

See all from CluuCodes

Recommended from Medium



Maciej Pocwierz

How an empty S3 bucket can make your AWS bill explode

Imagine you create an empty, private AWS S3 bucket in a region of your preference. What...

4 min read · 6 days ago

8.1K 106

Matthias Dittrich in UX Planet

Stop designing chat-based AI tools.

It is time to evolve AI tools beyond prompt-based interfaces and consider new mental...

5 min read · Apr 21, 2024

953 20

Lists

Coding & Development

11 stories · 593 saves

General Coding Knowledge

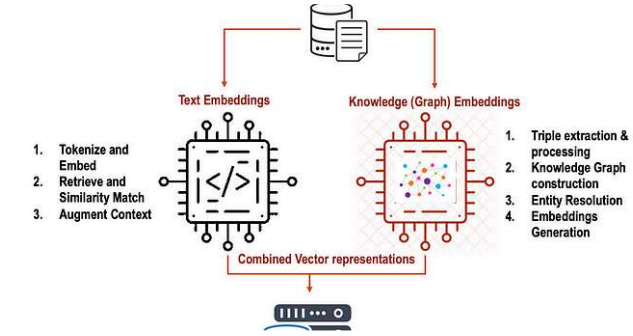
20 stories · 1186 saves

Stories to Help You Grow as a Software Developer

19 stories · 1030 saves

Predictive Modeling w/ Python

20 stories · 1151 saves



Pinjari Rehan in JavaScript in Plain English

15 Time-Saving Websites Every Developer Needs

Ever thought that there aren't enough hours in the day for all your never-ending...

5 min read · 6 days ago

361 4

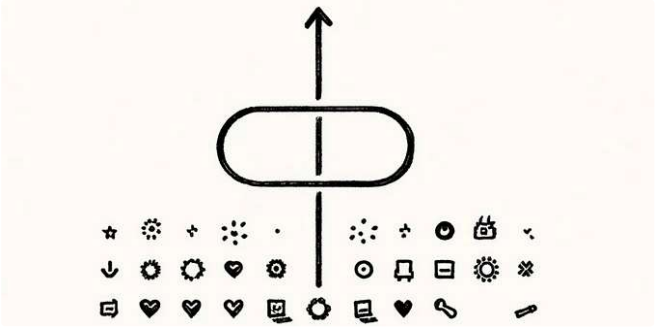
Sunila Gollapudi in Towards Data Science

Combine Text Embeddings and Knowledge (Graph) Embeddings i...

In my previous articles, I wrote about using Knowledge Graphs in conjunction with RAGs...

· 15 min read · 6 days ago

414 5



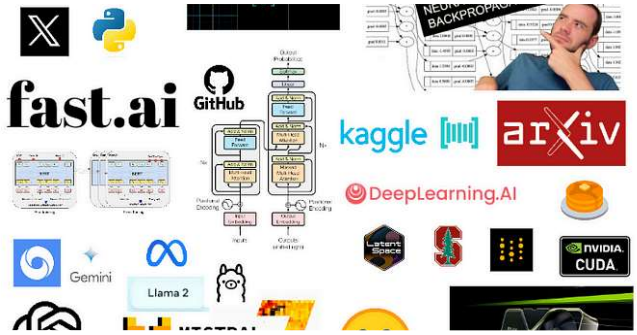
Luke Anthony Firth in UX Collective

Beyond "done"— prioritizing quality in product experiences

When quantity of features is the default for many organizations, how can UX-minded...

6 min read · 3 days ago

195



Benedict Neo in bitgrit Data Science Publication

Roadmap to Learn AI in 2024

A free curriculum for hackers and programmers to learn AI

11 min read · Mar 10, 2024

10.8K 119

See more recommendations