PHASE III

SOFE 3700U
Group 6
CRN 43512 002

Name	Student ID
Johnathan Howe	100785128
Okikioluwa Ojo	100790236
Sehar Ahmed	100808249
Zainab Nomani	100784761

Github: *https://github.com/okikio-school/dbms-final*

Abstract

This report discusses the implementation of fundamental database concepts CarlPost - a Headless Content Management System, similar to a Database Management System, designed for small companies, as well as large organizations.The CMS aims to allow for handling of content, enabling access by many users, enabling collaboration and administration of content, management of media, supporting integrations with various third-party applications for marketing, scheduling, and more with the help of an Application Programming Interface. The support and accessibility of users are taken into account which is why we use Typescript, Next.js, and PostgreSQL to provide the best results. We explore various aspects of the project, including but not limited to - its relationship with the course and other work, design and implementation strategies, results and analysis, as well as future enhancements.

Introduction

The primary purpose of a content management system (CMS) is to provide users with a user-friendly interface, so that individuals without extensive technical knowledge, can collaboratively create, edit, organize and publish content on the internet. As technology continues to evolve, effective management and secure publication of information is vital for the success of organizations.

With an increasing demand for versatile and user-friendly content management systems, CarlPost is designed to provide a holistic approach to content governance, user collaboration, and adaptive scalability. Many Content Management System platforms available in the current market, are closed source, unintuitive and up being very inflexible due to enforcement of strict database requirements. CarlPost aims to reinvent content creation, publication and distribution, while overcoming these issues in an efficient manner.

Security in the digital world continues to remain a paramount concern, therefore CarlPost is designed in compliance with industry standards, with robust data protection mechanisms and access controls.

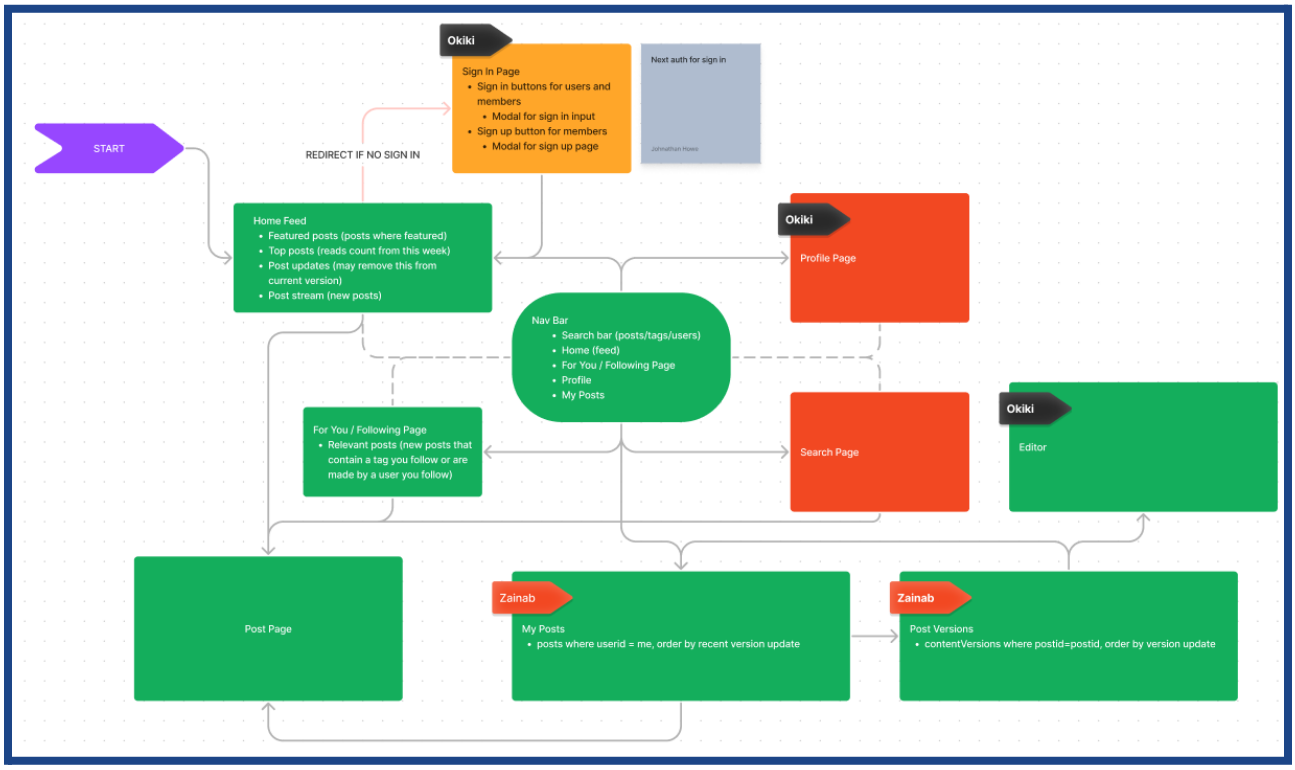
Project Goal

Our project has two key focal points: first and foremost, we aim to apply the foundational database concepts taught in class to enhance CarlPost. This involves implementing and integrating concepts we've learned as well as creating a solid foundation for our database.

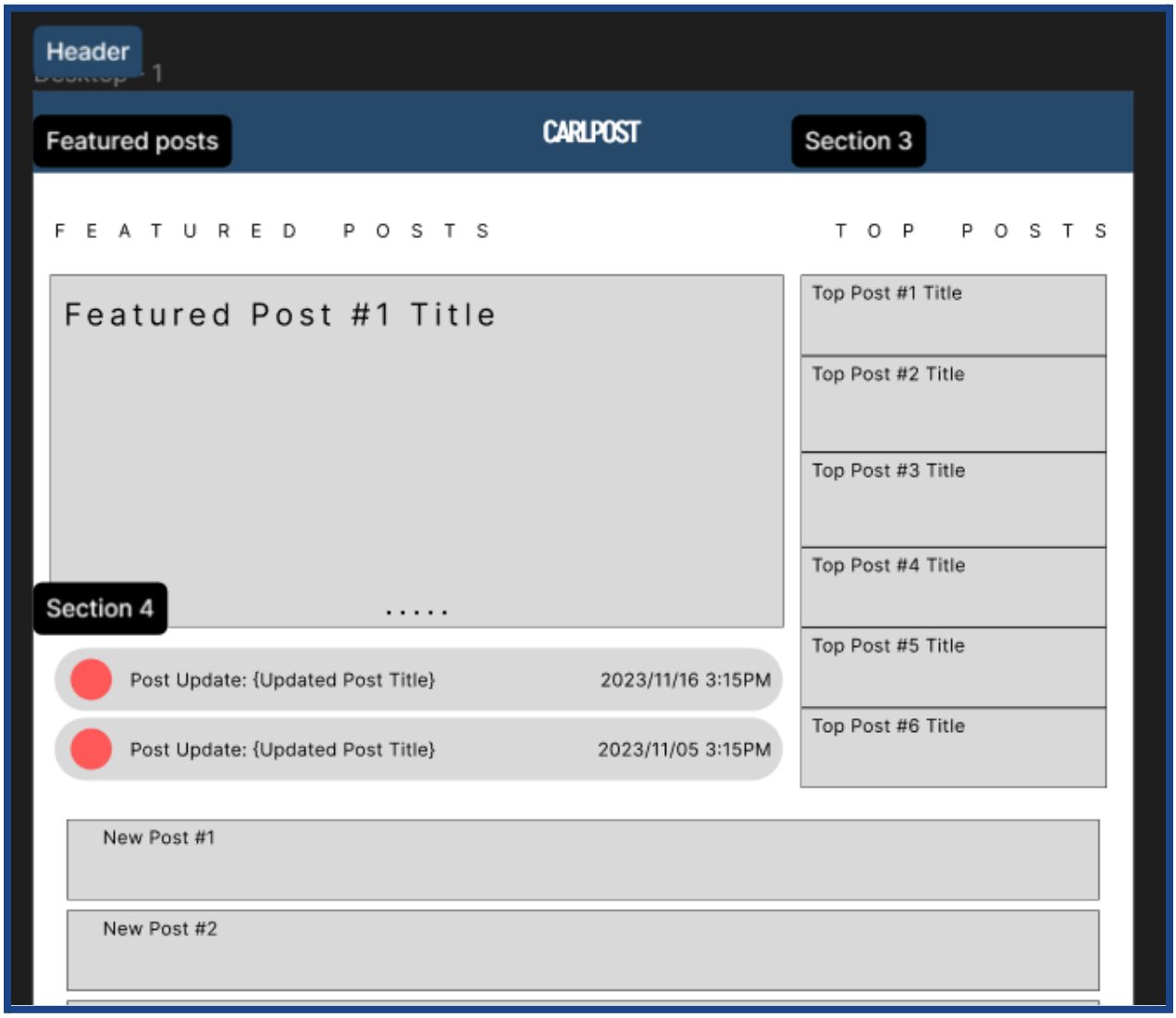
Additionally, we have set a secondary objective to leverage familiar technologies like Next.js and Typescript, which we've used in previous coursework. We also explored the potential of new, open-source database management systems, specifically Supabase and DrizzleORM.

Beyond the technical aspects, the primary aim of CarlPost is to transition from a prototype to a fully realized application that can serve the public and be commercialized. We see this project as a dynamic prototype, subject to continuous refinement and updates as we work tirelessly to transform our vision into a tangible reality.

Project Distribution



Projected Front Page



Database System Hierarchy

CarlPost's features as a content management system includes the ability to view posts, create posts, search for posts, authors and tags as well as signing in with an email and password. Users come from all over the world and in order for other users to experience CarlPost we need to save posts in a database. This is also the case for the user login and password, as these need to be stored in a secure database to prevent hacking as these are sensitive information. The database connection allows us to connect our website to servers which helps carry out tasks needed to produce the essence of CarlPost. Actions such as insertion, deletion, updating, and retrieval can all be done without interfering with the overall design and aesthetic as well as other users accounts.

Design and Implementation

Languages and APIs

In order to implement the scope of the project, we took advantage of PostgreSQL, or Postgres. Postgres is an open source relational database management system that we used to supplement our database. PostgreSQL offers reliability, transaction support, and has the ability to handle complex queries as well as being backed up by a highly stable database. The few features of Postgres that can be highlighted is the fact that it is mainly used for web applications. It can also support Next.js, a non relational server that connects the front end and the database in order for them to communicate with each other [1]. REACT works with Next.js to create the HTML pages along with shadcn/ui that contains the library of components. JSON is the post content storage method used to connect Postgres to our front end. JSON is also used for REST API, which is used to call databases or external APIs and can only do so by using JSON to communicate back and forth.

Another application that we used was Typescript, which is an extension of Javascript, which helps us use Next.js [2] as well as Tailwind CSS, which gives us classes to easily configure our CSS
. Supabase is a web posted database server that leverages Postgres and provides a full database which essentially hosts our database. Lastly we used DrizzleORM, which is an Object-Related Mapping (ORM) library which uses Javascript to manipulate the database instead of using the query language [3][4].can

Views/Prepared Statements

In Phase II we created ten views, which all are used for retrieving queries from other tables stored in the database. As we built our application we came across issues that resulted in our views not aligning with the outcome we wanted to achieve. Additionally, Drizzle ORM does not currently support traditional PostgreSQL views. Instead, we used multiple prepared statements, which are the next best implementation that Drizzle offers. Just like views, prepared statements pre-compile the sql queries and use cached query results to provide faster response times. Only one of our prepared statements was based on one of our original views; specifically, we used *View 5: Uses nested queries and of the set operations UNION, EXCEPT, or INTERSECT.* This is not to say we didn't use any other views or queries, we created 17 tables that require different and new prepared statements in order to create our application.

Prepared Statement # 1

This statement returns a list of relevant posts to a given user. Specifically, it returns any posts which were made by a user or contain a tag which the user follows. The statement uses UNION to combine the relevant posts based on user follows with the relevant posts based on tag follows. This statement is used for the "For You" page. The Drizzle implementation creates two queries and combines them with a UNION function.

Original View

View 5: Uses nested queries with and of the set operations UNION, EXCEPT, or INTERSECT

```
-- creates a view displaying posts that are relevant to the follows of each member
-- (i.e. lists posts with tags or made by users whom the member follows)
CREATE VIEW RelevantPosts AS
SELECT f.follower_id, p.post_id, p.title, t.tag_id AS followed_id,
t.name AS followed_name, f.type AS follow_type
FROM posts p, tagstoposts tp, tags t
JOIN follows f ON f.entity_id = t.tag_id
WHERE p.post_id = tp.post_id AND tp.tag_id = t.tag_id AND f.type = 'tag'
UNION SELECT f.follower_id, p.post_id, p.title, u.user_id AS followed_id,
u.name AS followed_name, f.type AS follow_type
FROM posts p, users u
JOIN follows f ON f.entity_id = u.user_id
WHERE p.userid = u.user_id AND f.type = 'user'
```

Updated SQL Statement

```
DECLARE userID varchar(36) := 'ab2157e5-5552-4f6c-a830-62842627769a';
SELECT f.follower_id, p.post_id, p.title, t.tag_id AS followed_id,
    t.name AS followed_name, f.type AS follow_type
    FROM posts p, tagstoposts tp, tags t
    JOIN follows f ON f.entity_id = t.tag_id
    WHERE p.post_id = tp.post_id AND tp.tag_id = t.tag_id AND f.type = 'tag' AND f.follower_id = userID
    UNION
    SELECT f.follower_id, p.post_id, p.title, u.user_id AS followed_id,
    u.name AS followed_name, f.type AS follow_type
    FROM posts p, users u
    JOIN follows f ON f.entity_id = u.user_id
    WHERE p.userid = u.user_id AND f.type = 'user' AND f.follower_id = userID;
```

DrizzleORM Implementation

```
export const getRelevantPosts = cache(async function getRelevantPosts(userId : string) {

  /*SELECT f.follower_id, p.post_id, p.title, t.tag_id AS followed_id, ···

  const tagfollowsprep = db.select({
    follower: follows.followerId,
    postID: posts.postId,
    title: posts.title,
    followed: tags.tagId,
    name: tags.name,
    type: follows.type,
    author: users.name,
    version: posts.version,
    date: posts.publishedDate,
  }).from(posts)
    .innerJoin(tagsToPosts, eq(posts.postId, tagsToPosts.postId))
    .innerJoin(tags, eq(tags.tagId, tagsToPosts.tagId))
    .innerJoin(follows, eq(follows.entityId, tags.tagId))
    .innerJoin(users, eq(users.userId, posts.userId))
    .where(and(
      eq(follows.type, 'tag'),
      eq(follows.followerId, userId),
  ));
```

```
  const userfollowsprep = db.select({
    follower: follows.followerId,
    postID: posts.postId,
    title: posts.title,
    followed: users.userId,
    name: users.name,
    type: follows.type,
    author: users.name,
    version: posts.version,
    date: posts.publishedDate,
  }).from(posts)
    .innerJoin(users, eq(users.userId, posts.userId))
    .innerJoin(follows, eq(follows.entityId, users.userId))
    .where(and(
      eq(follows.type, 'user'),
      eq(follows.followerId, userId),
  ));

  return await union(tagfollowsprep, userfollowsprep);
})
```

Output

follower_id	post_id	title	followed_id	followed_name	follow_type
"ab2157e5-5552-4	"2eb6c629-9a2b-4	"ipsum reiciend	"0ca71ecb-bd27-4	"Douglas Abbott'	"user"
"ab2157e5-5552-4	"4d97ea7c-722d-4	"conscendo comi	"bb199305-0816-4	"Mr. Cedric Brue	"user"
"ab2157e5-5552-4	"4de4637c-3172-4	"Introducting Ou	"bb199305-0816-4	"Mr. Cedric Brue	"user"
"ab2157e5-5552-4	"9377f04a-25cd-4	"test post"	"bb199305-0816-4	"Mr. Cedric Brue	"user"
"ab2157e5-5552-4	"a113b2d9-680a-4	"What IS the dea	"bb199305-0816-4	"Mr. Cedric Brue	"user"

Prepared Statement # 2

This statement returns important data about a given post version, including the content of the post, represented as JSON. This statement is used for any pages which display the contents of a post.

New SQL Statement

```
SELECT
  contentVersions.content AS content,
  posts.title AS title,
  users.name AS author,
  posts.published_date AS published_date,
  contentVersions.version_id AS version
FROM
  posts
LEFT JOIN
  contentVersions ON contentVersions.post_id = posts.post_id
LEFT JOIN
  users ON users.user_id = posts.userid
WHERE
  posts.version = contentVersions.version_id AND
  posts.post_id = postID AND
  contentVersions.version_id = versionID;
```

DrizzleORM Implementation

```ts
//post content
export const getPostContent = cache(async function getPostContent(postID : string, versionID : number) {
  const postcontentprep = db.select({
    content: contentVersions.content,
    title: posts.title,
    author: users.name,
    published_date: posts.publishedDate,
    version: contentVersions.versionId,
  }).from(posts)
    .leftJoin(contentVersions, eq(contentVersions.postId, posts.postId))
    .leftJoin(users, eq(users.userId, posts.userId))
    .where(
      and(
        eq(posts.version, contentVersions.versionId),
        eq(posts.postId, postID),
        eq(contentVersions.versionId, versionID)
        )
    );
  return await postcontentprep.execute();
})
```

Output

content	title	author	published_date	version
"{\"markdown\":\	"Lorem Ipsuim"	"Okiki Ojo"	"2023-11-23 03:(	28

Prepared Statement # 3

This statement returns a list of the top posts (by post reads) on the site. This statement is used in the top posts section of the Home Feed.

New SQL Statement

```sql
SELECT
  posts.post_id AS id,
  posts.title AS title,
  users.name AS author,
  posts.published_date AS date,
  COUNT(postreads.post_id) AS reads,
  posts.version AS version
FROM
  posts
LEFT JOIN
  postReads ON posts.post_id = postReads.post_id
LEFT JOIN
  users ON users.user_id = posts.userId
GROUP BY
  posts.post_id, posts.title, users.name, posts.published_date
HAVING
  COUNT(postreads.post_id) > 0
ORDER BY
  reads DESC;
```

DrizzleORM Implementation

```
//top posts
const reads = sql<number>`cast(count(${postReads.postId}) as int)`;
const toppostsprep = db.select({
  id: posts.postId,
  title: posts.title,
  author: users.name,
  date: posts.publishedDate,
  reads: reads,
  version: posts.version
}).from(posts)
  .leftJoin(postReads, eq(posts.postId, postReads.postId))
  .leftJoin(users, eq(users.userId, posts.userId))
  .groupBy(posts.postId, posts.title, users.name, posts.publishedDate)
  .having(gt(reads, 0))
  .orderBy(desc(reads))
  .prepare("list_top_posts");
```

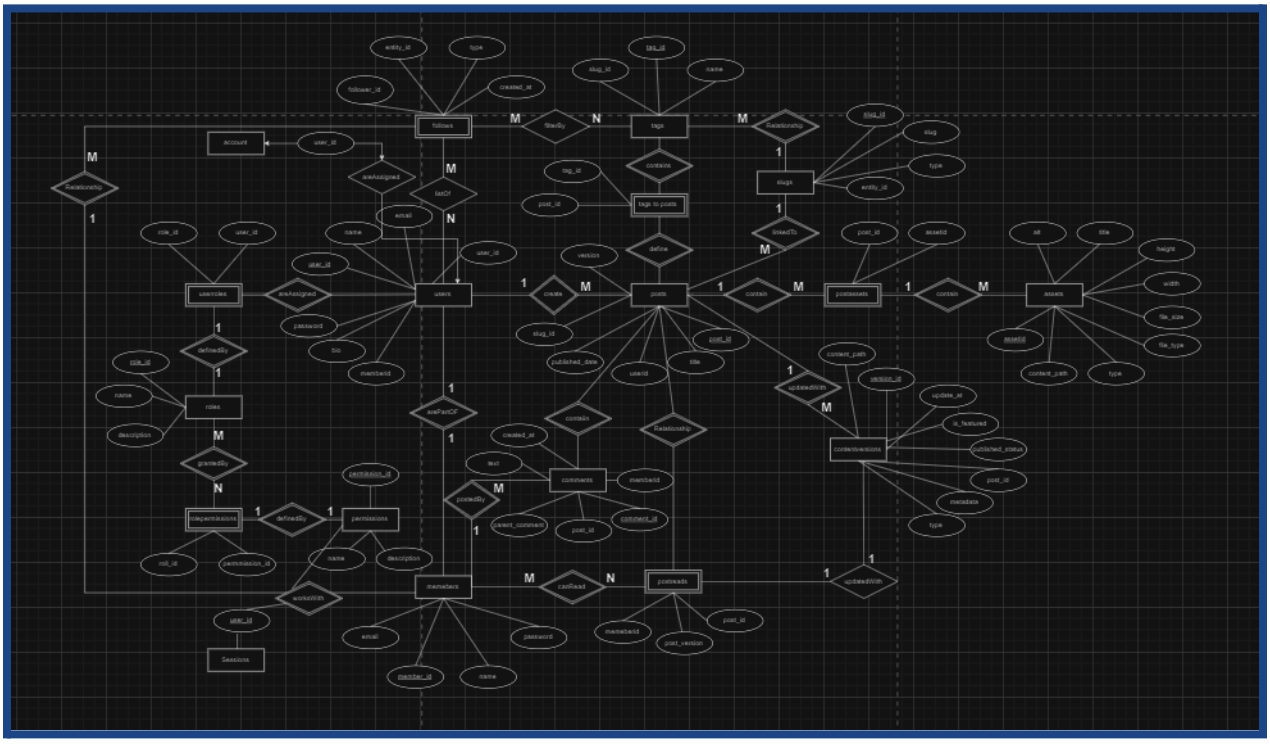
Output

id	title	author	date	reads	version
"4d97ea7c-722d-4	"conscendo comit	"Mr. Cedric Brue	"2023-11-17 17:1	2	14
"6a00d1fa-cc29-4	"villa argumentu	"Marcia Bechtela	"2023-11-17 17:1	2	20
"9b2e1fda-0f1a-4	"contego acsi cc	"Alberta Rath-Al	"2023-11-17 17:1	1	15
"d6cdacad-a8cc-4	"surculus callic	"Salvatore Mayer	"2023-11-17 17:1	1	11
"01884e98-016e-4	"denego virga cc	"Thelma Blick"	"2023-11-17 17:1	1	12
"2eb6c629-9a2b-4	"ipsum reiciend	"Douglas Abbott'	"2023-11-17 17:1	1	19
"fabfdf47-9ef7-4	"deinde aggredic	"Jaime Wolf"	"2023-11-17 17:1	1	16
"5214d669-f733-4	"vindico"	"Carmen Cassin"	"2023-11-17 17:1	1	13

Schematics and Design Diagrams[1]

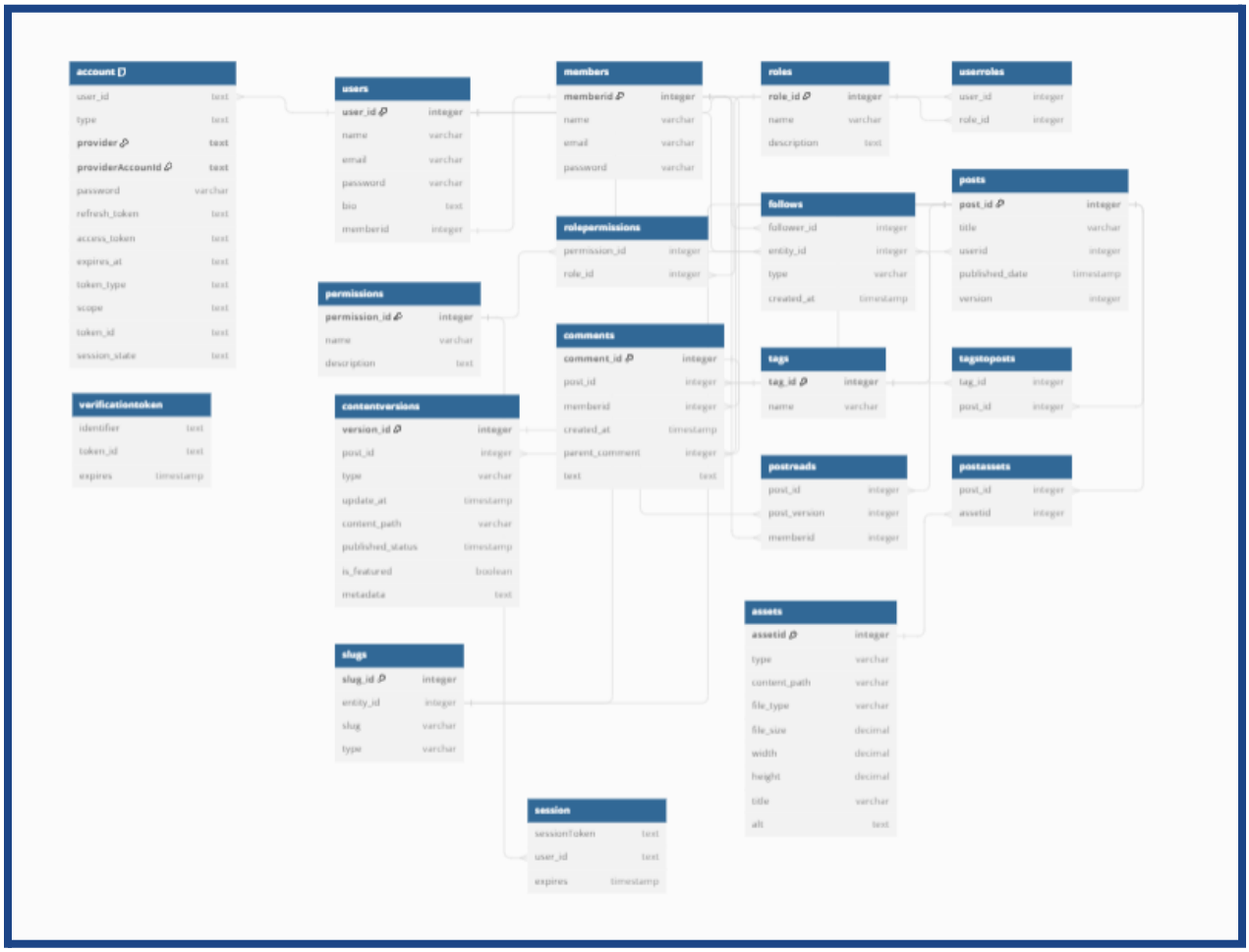
These are our schematic which are used to organize and map or entity relationship

I. Entity Relationship Diagram



[1] Higher quality images can be found on GitHub

II. Relational Schema



Relation to DBMS / Other Work

In Phase I, we drew inspiration from Okikioluwa's co-op term experience, where he meticulously evaluated various CMS platforms. It became evident that many existing solutions are characterized by being closed-source, inflexible, and not particularly user-friendly. One common drawback observed was the imposition of a specific database type, be it SQL or NoSQL, resulting in a restrictive environment. Our vision emerged from the desire to offer a more adaptable and user-centric alternative.

Our inspiration further stems from popular blog post platforms such as Tumblr and Reddit, which have become integral parts of our experiences. While harnessing the essence of these platforms, our project takes a progressive leap in server management. Our goal is to transcend the limitations observed in traditional CMS platforms, fostering flexibility, user-friendliness, and a forward-thinking approach.

Future Work

The long term vision for a project of this scale is to allow clients to be able to integrate their existing database providers, single-sign-on system and more. In the case that clients use multiple database providers, the system would allow them to mix and match as they see fit. However, our main vision is to allow the system to support Cloud Native deployment, such as docker or kubernetes, in a manner that is secure and easy to self-host.

Additionally, there are a few features that we intended to include but were not able to complete in the time provided. These features include a profile page, where the user can edit their profile info, support for following users within the user interface, tracking of post reads, which would be used to track the top posts on the site, and complete the search feature to find posts based on key search terms. We would also like to improve the user interface in the future, since the current version is a basic prototype.

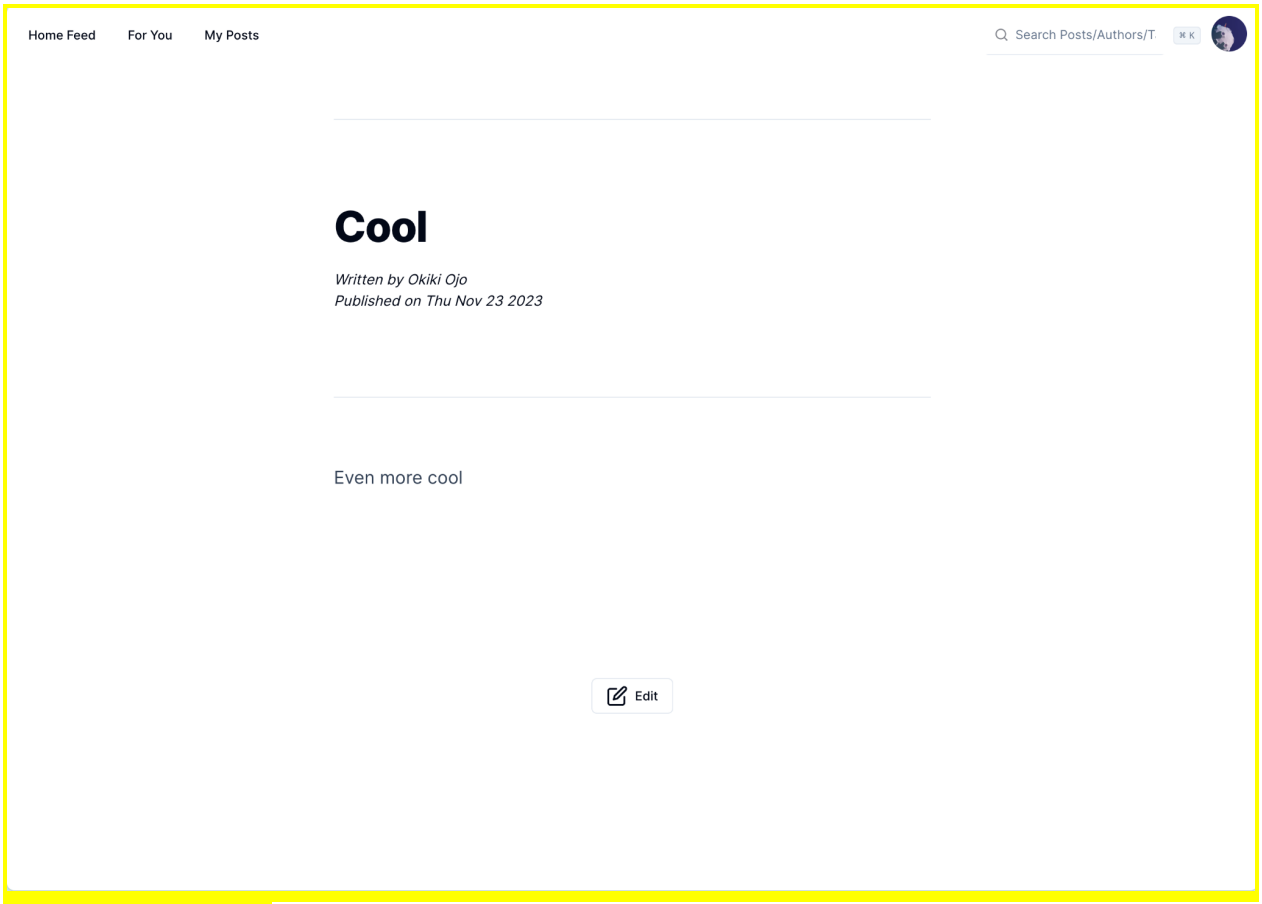
Conclusion

In conclusion, CarlPost is an innovative solution in the realm of content management systems, driven by an interface that enables users to collaboratively create, edit, organize, and publish content on the internet. Recognizing the evolving landscape of technology, the effective management and secure publication of information are identified as crucial elements for this project. CarlPost sets itself apart from existing platforms by offering a holistic approach to content governance, user collaboration, and adaptive scalability. Unlike many closed-source and inflexible alternatives, CarlPost is designed to reinvent content creation and distribution, overcoming challenges through an efficient and intuitive interface. In essence, CarlPost stands as a dynamic prototype, embodying continuous refinement and updates. It not only addresses the technical intricacies of database management but also strives to fulfill its overarching vision of becoming a tangible and impactful tool for content creation and distribution in the digital landscape.

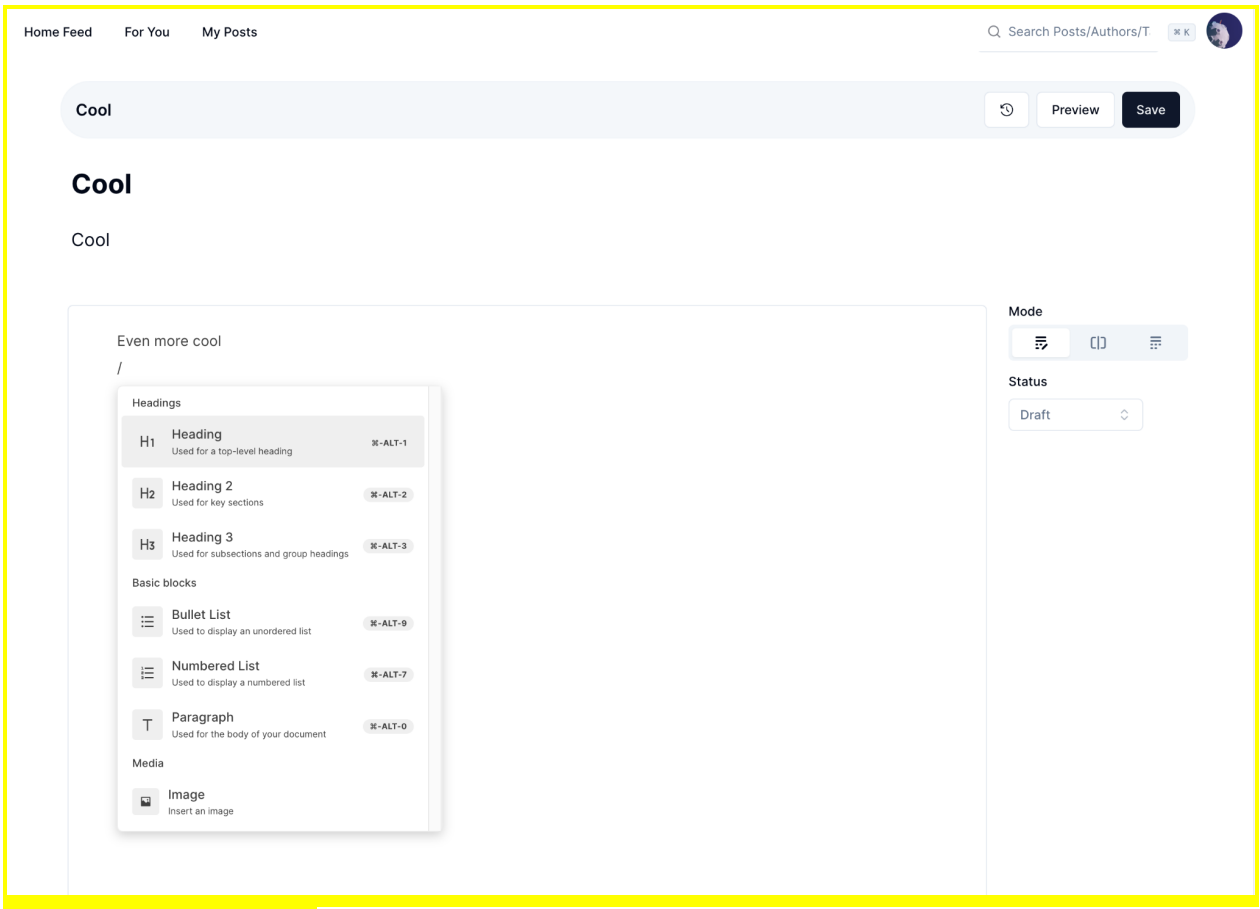
References

1. *Getting Started with PostgreSQL*. (2021, April 27). PostgreSQL Tutorial. https://www.postgresqltutorial.com/postgresql-getting-started/
2. *JavaScript With Syntax For Types*. (n.d.). https://www.typescriptlang.org/
3. *What is an ORM (Object Relational Mapper)?* (n.d.). Prisma's Data Guide. https://www.prisma.io/dataguide/types/relational/what-is-an-orm
4. Drizzle-Team. (n.d.). *GitHub - drizzle-team/drizzle-orm: TypeScript ORM that feels like writing SQL*. GitHub. https://github.com/drizzle-team/drizzle-orm

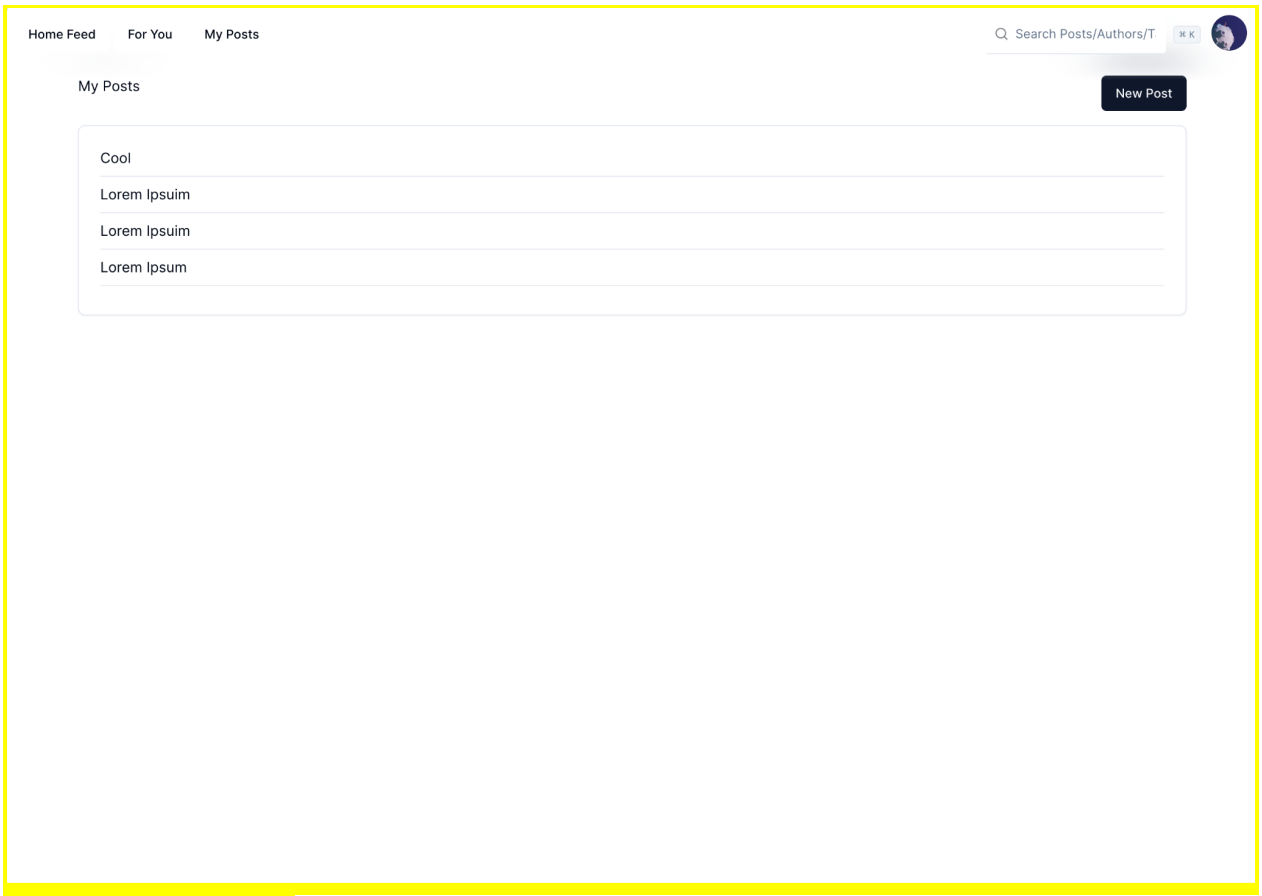
Appendix



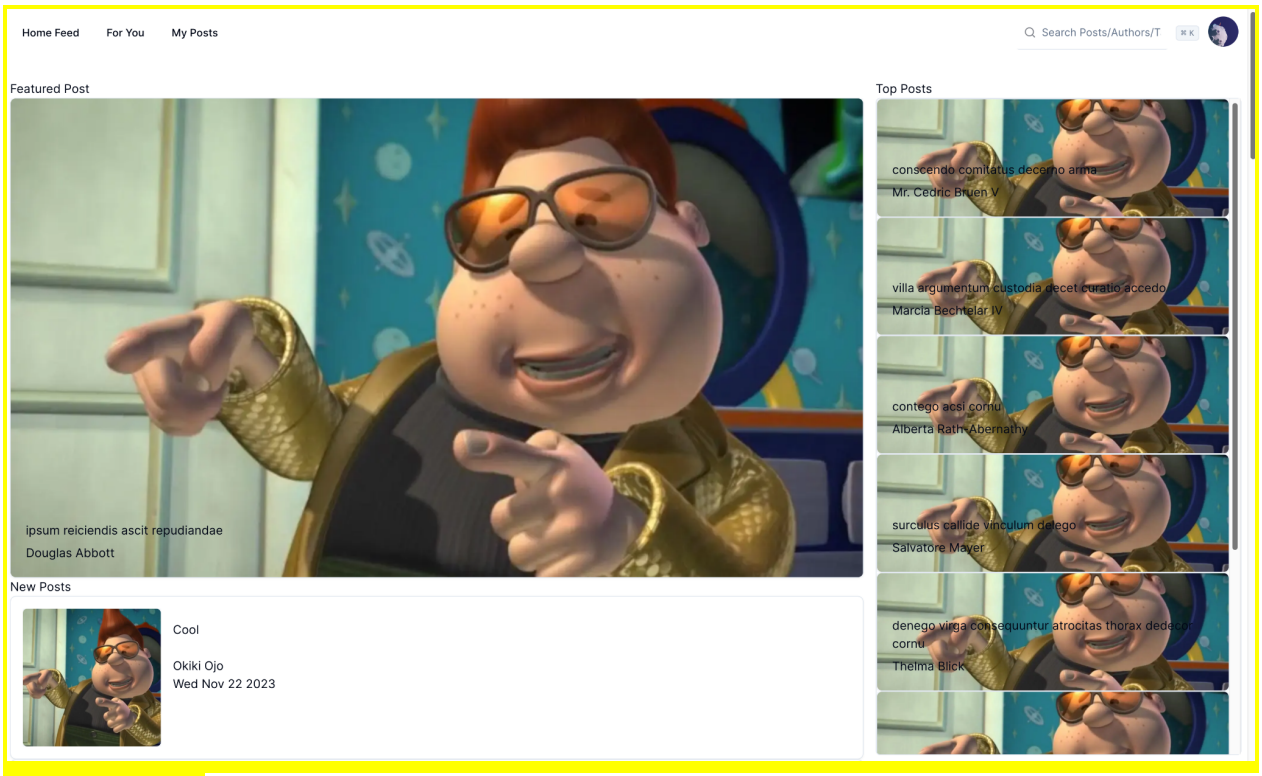
View of post



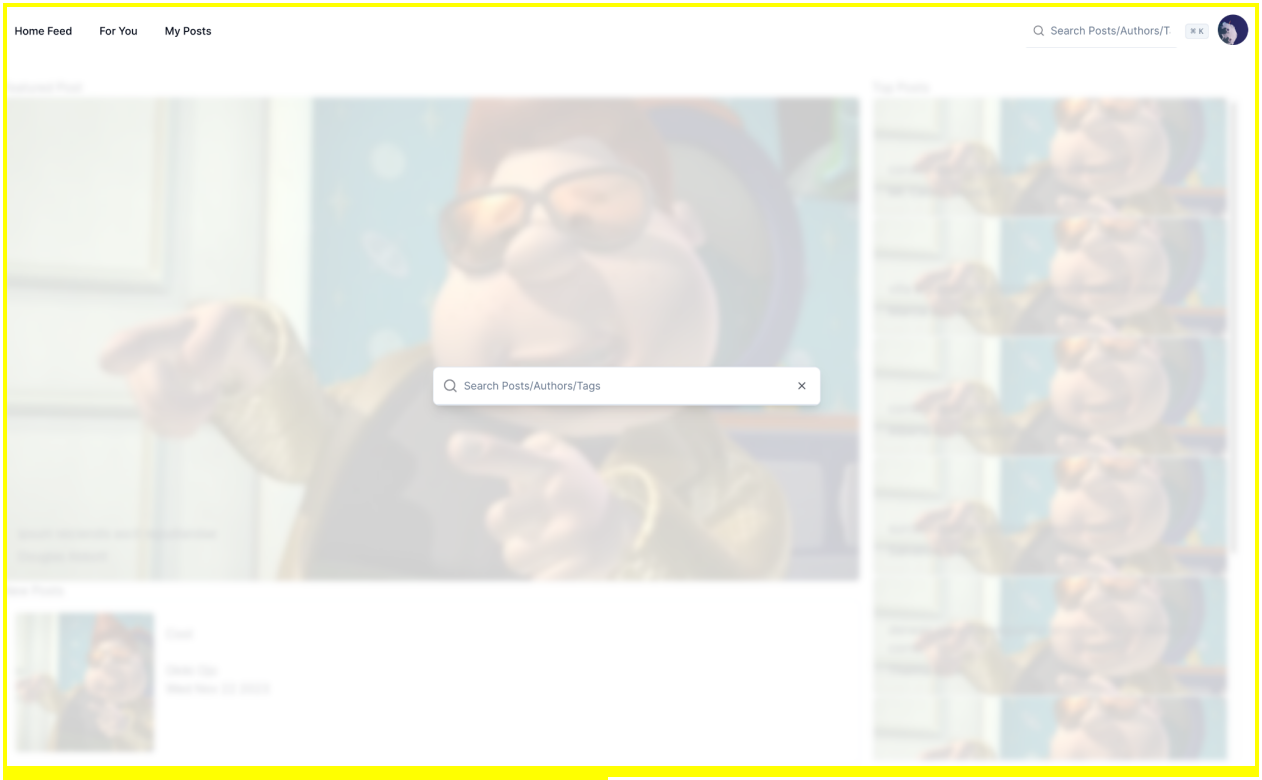
Edit post page



My Posts page

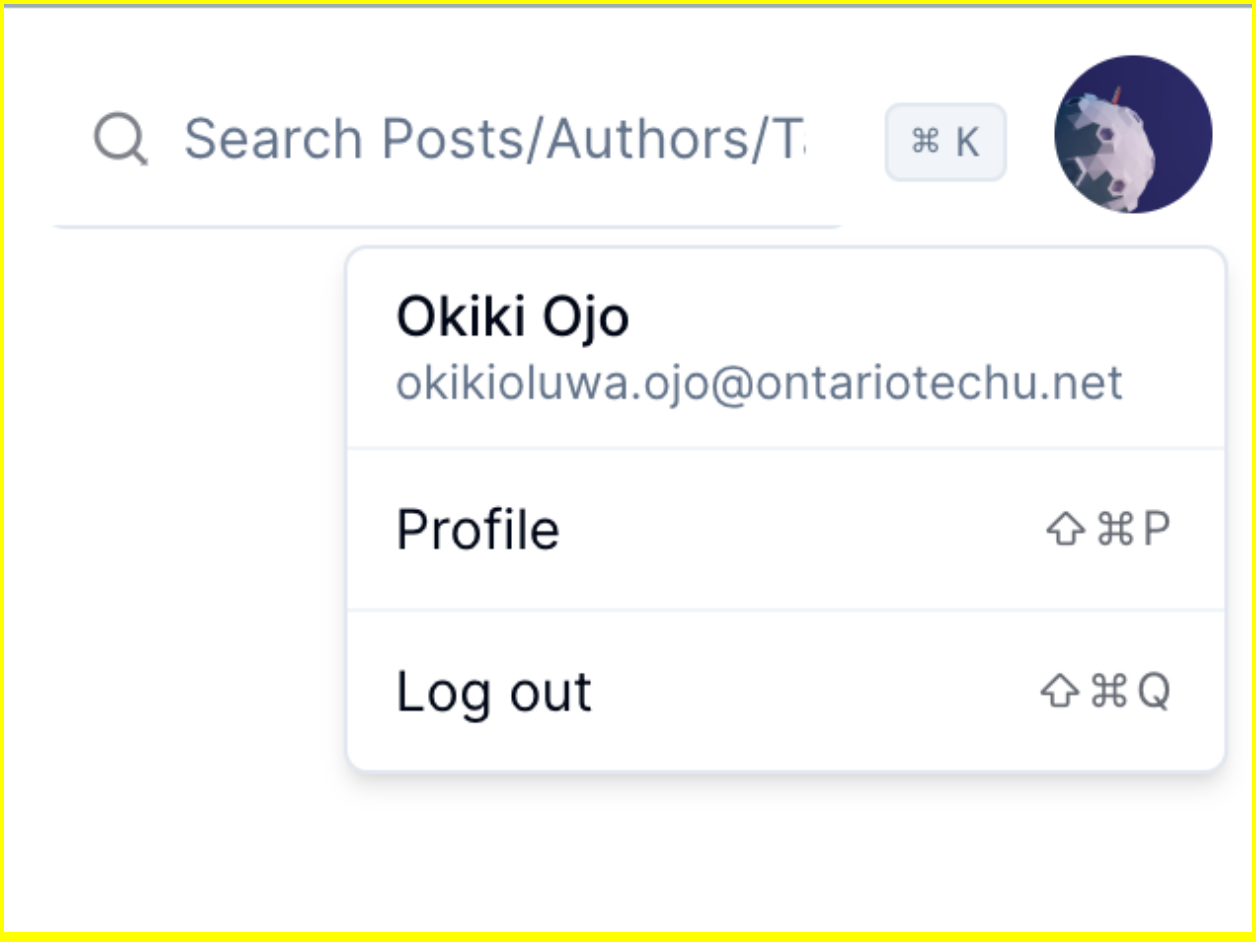


Home page

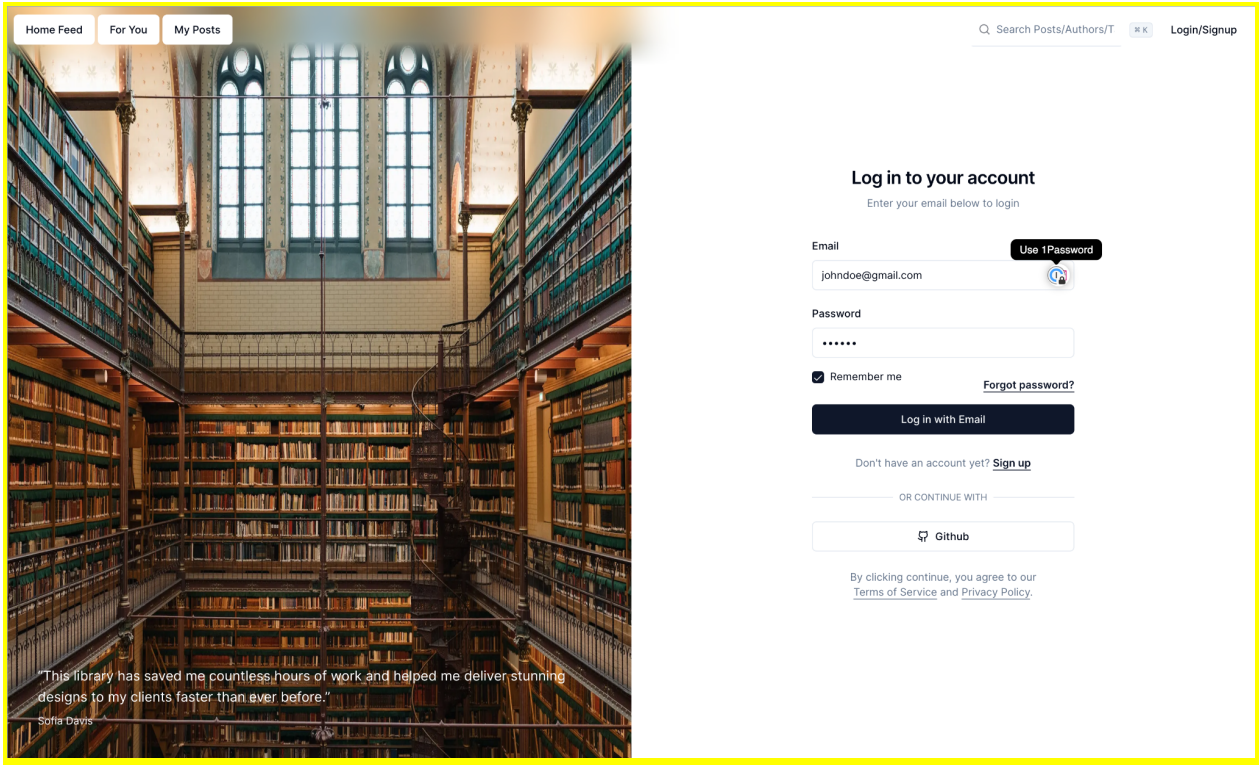


Search section (Incomplete)

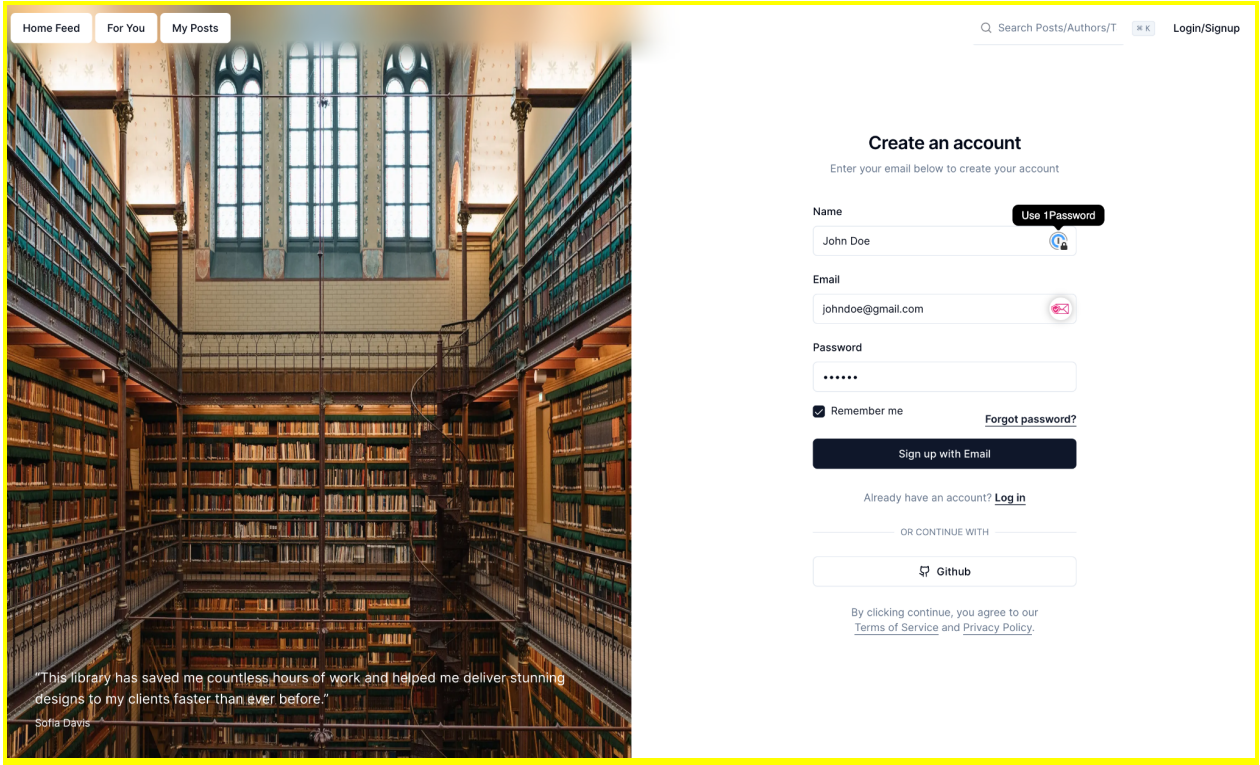
Profile Page (Incomplete)



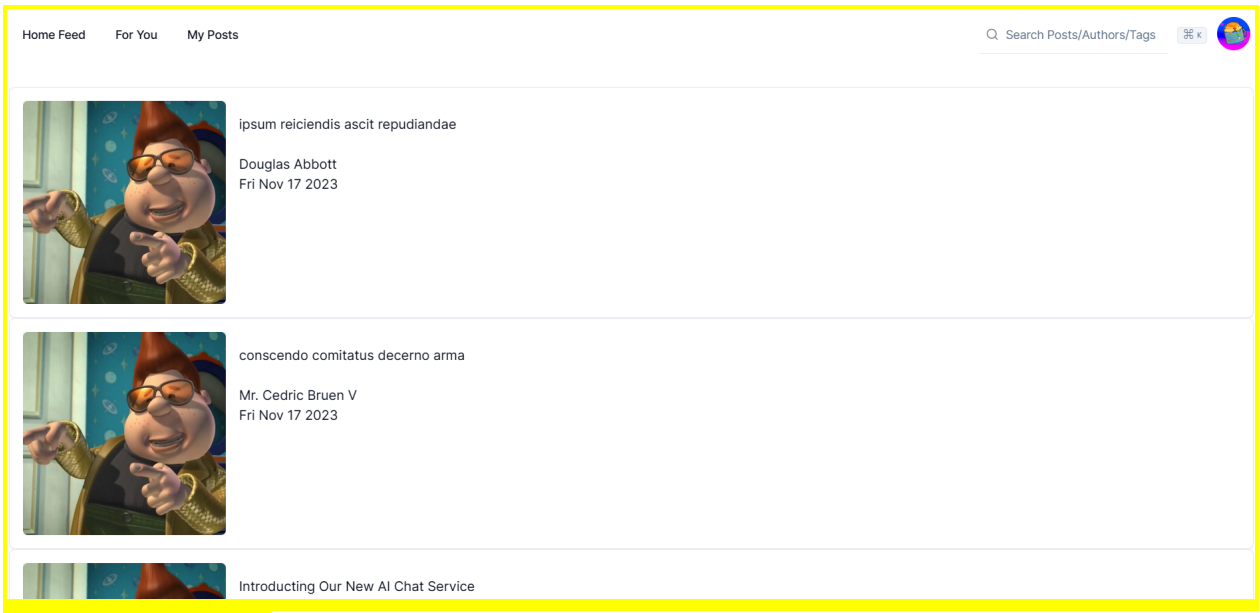
Profile Dropdown



Login Page (OAuth works but there are some bugs with sign)



Signup Page


Home Feed
For You
My Posts
Search Posts/Authors/Tags
⌘ K
ipsum reiciendis ascit repudiandae
Douglas Abbott
Fri Nov 17 2023
conscendo comitatus decerno arma
Mr. Cedric Bruen V
Fri Nov 17 2023
Introducting Our New AI Chat Service

For You Page