

UniLink



Department of Computer Science

COSC 6352 - Advanced Operating Systems

Instructor - Dr. Yongzhi Wang

Team Members:

NARESH VEMULA (A04332886)

KOUSHIK REDDY KAMBHAM (A04316083)

SAI AVINASH VAGICHERLA (A04312707)

JASWANTH SIRIGIRI (A04332429)

ABSTRACT

Navigating the complexities of college communication can be daunting for students due to limited channels and generic media platforms. To bridge this gap, UniLink emerges as a tailored solution, designed exclusively for college students. UniLink serves as a virtual nexus, enabling seamless connection, collaboration, and content sharing within the college community. Its intuitive interface facilitates creating, updating, and deleting posts, establishing connections, and engaging with shared content. Powered by cutting-edge technologies like Fast API, Next.js, Redis, Cassandra, and Kafka, UniLink ensures reliability, security, consistency, and fault tolerance. Despite challenges like database denormalization and resource-intensive Docker deployments, UniLink remains committed to enhancing student communication and fostering a vibrant college community. In essence, UniLink is a catalyst for meaningful connections and collaborative endeavors among college students.

Keywords: UniLink, collaboration, tailored platform, Fast API, Next.js, Redis, Cassandra, and Kafka, Reliability, Security, Consistency, Fault Tolerance.

INTRODUCTION

Using this application, various users can connect with each other, fostering communication and collaboration. Additionally, organizations can utilize this platform to disseminate information about events and keep their community updated on the latest news and developments. For instance, universities can leverage this application to announce upcoming elections, coordinate group gatherings, or promote cultural festivals.

The UniLink project represents a successful fusion of modern technologies and established practices, resulting in the creation of a robust and scalable distributed system. Leveraging technologies such as FastAPI, Next.js, Apache Cassandra, and Redis cache, a platform has been developed that prioritizes user experience while ensuring fault tolerance, security, and scalability. Throughout the project, the focus has been on delivering a high-quality user experience and showcasing core distributed operating system features. The integration of Redis cache further enhances performance and scalability, ensuring optimal functionality even during periods of high demand.

Overall, the UniLink project exemplifies dedication to innovation, excellence, and user satisfaction. It provides a solution that meets the needs of modern users and offers organizations a reliable platform for communication and engagement.

SYSTEM ARCHITECTURE

The major functionalities of the project are listed below:

- Creating, Delete, and Update a post
- Connect and disconnect with people
- Interact with posts that are posted by others
- Search for a user
- Sign in/out and Log in/out of the account

They are explained in detail through the following use case diagram.

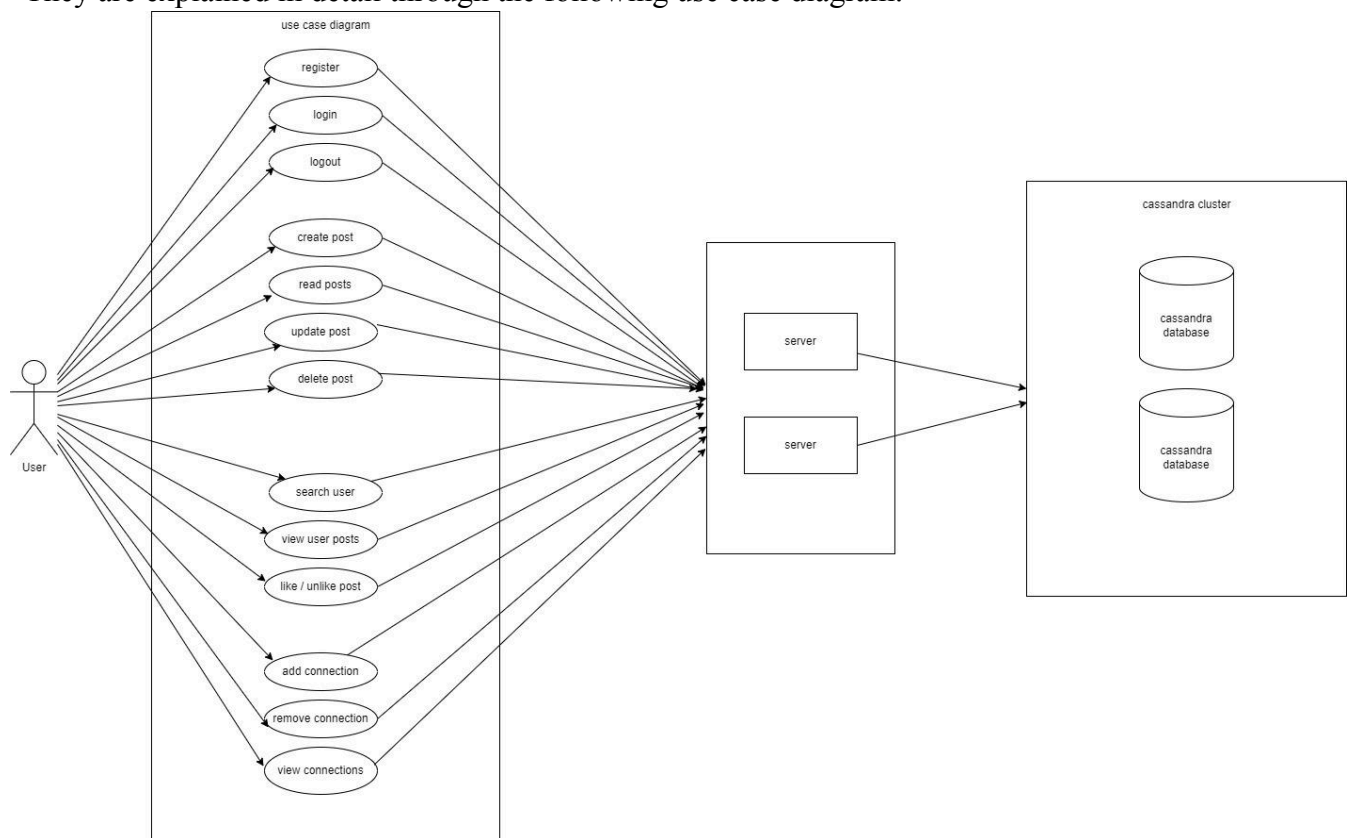


Fig 1: Use a case diagram describing user functionalities.

The use case diagram shows the different ways that users can interact with the system, as well as the system's interactions with a Cassandra database cluster.

Actors: The diagram shows one actor, labeled "User".

Use cases: The use cases are represented by ellipses. The use cases in this diagram are:

Register, Login, Logout, Read posts, Create posts, Update posts, Delete posts, Search user, View user posts, Like/unlike posts, Add connection, Remove connection, View connections

Relationships: The relationships between the actors and the use cases are shown by lines. The lines are labeled with arrows to show the direction of the interaction.

System: The system is represented by a rectangle labeled "Server". The server interacts with all of the use cases.

Database: The database is represented by a rectangle labeled "Cassandra database". The server interacts with the database to perform many of the use cases. For example, the server interacts with the database to read posts, create posts, and update posts.

Architecture Of Each Service

Various services will be implemented for the users, based on the features required.

User Service: The user can create an account, log in, and log out of the account. This follows authentication management.



Fig 2: Authentication Management.

Connection service: A user can connect, view, or remove connections with other users.

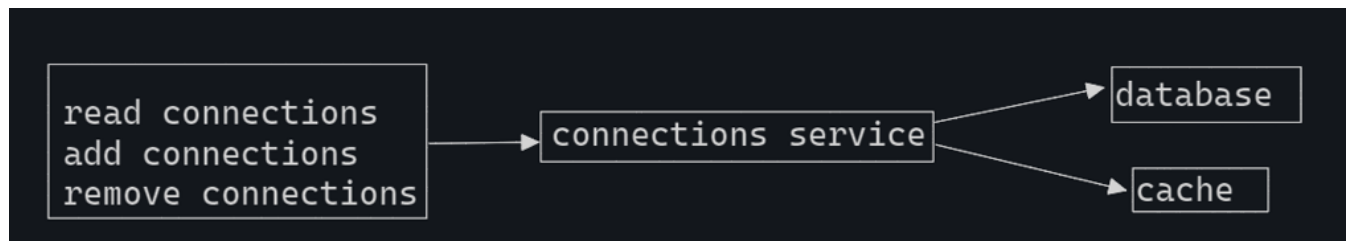


Fig 3: Connection service.

Interaction service, Timeline service, Search service:

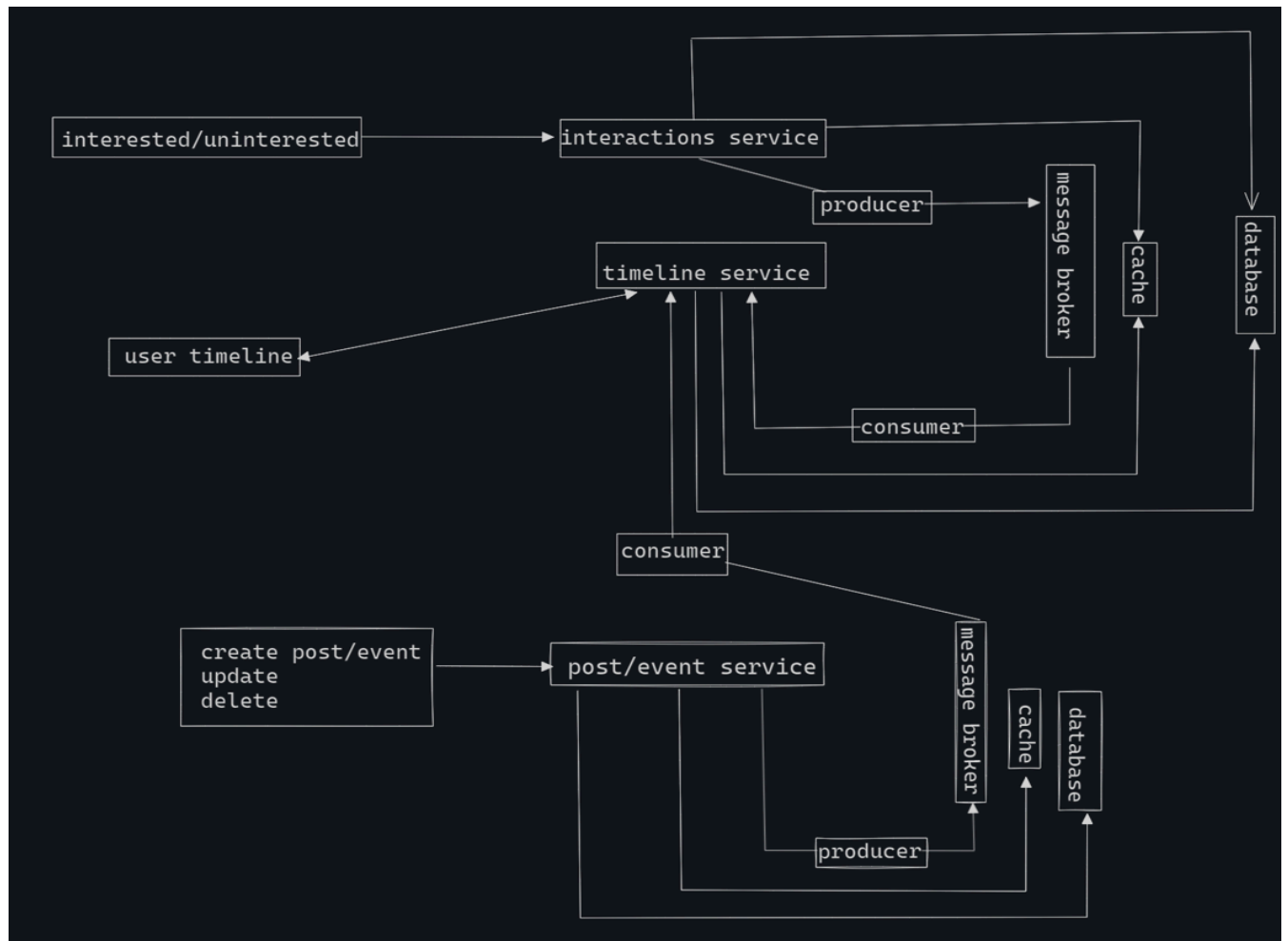


Fig 4: Timeline and Interaction Service

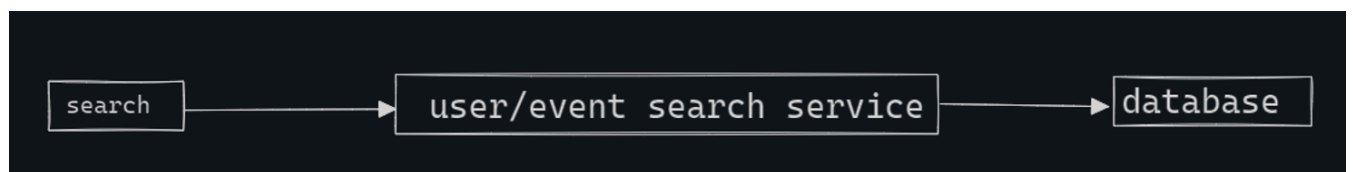


Fig 5: Search Service

Post/Event Service: This service remains the central component, responsible for:

- Create post/event: When a new post/event is requested, the service creates a new entry in the Cassandra database and caches relevant data in Redis.
- Update post/event: If an existing post/event needs modification, the service updates the corresponding entry in Cassandra and refreshes the cached data in Redis.
- Delete post/event: To remove a post/event, the service deletes its entry from Cassandra and removes the cached data from Redis.

Database and Cache:

- Cassandra: This NoSQL database serves as the persistent storage for all post/event data.
- Redis: This in-memory data structure acts as a cache, holding frequently accessed post/event information to improve performance.

Message Broker (Apache Kafka):

- Producer: The post/event service acts as a Kafka producer, publishing messages about changes to posts/events.
- Consumer: Other services, like:

Timeline service: Consumes messages to update user timelines whenever a post/event is created or updated.

Search service: Consumes messages to keep its search index synchronized with the latest post/event information.

Interested/Uninterested: This section likely reflects user interactions with posts/events. Users can choose to be "interested" in specific events, potentially triggering additional notifications or actions.

DESIGN AND IMPLEMENTATION

The project uses the different aspects of distributed systems which are described below.

Fault Tolerance:

The project is built in a way that does not support a centralized database. We are going to use a single database on every server. For using databases at each server the Cassandra database is used. Since Cassandra is not compatible to use in the Windows environment it is deployed as a container in Docker. The changes made in one server are synchronized in all the other server databases. In case a database at the server gets crashed it is going to connect to another database server. In this way fault tolerance is achieved without any data loss when the database is crashed.


















<input type="checkbox"/>	Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
<input type="checkbox"/>	 cassandra_cluster		Running (1/2)	186.63%		6 minutes ago	  
<input type="checkbox"/>	 container-1-cassandr	 e28e12879456  cassandra:jammy	Running	3.76%	9042:9042 	6 minutes ago	  
<input type="checkbox"/>	 container-2-cassandr	 a6f709211879  cassandra:jammy	Exited (143)	182.87%	9043:9042	52 minutes ago	  

Fig 6: Cassandra Cluster.

In the above figure, one of the databases has ceased functioning, yet the distributed system continues operating without interruption using the other database. This demonstrates the system's fault tolerance, enabling users to carry out operations seamlessly even in the event of a database failure. By swiftly connecting to alternative clusters when needed, the system ensures uninterrupted execution, safeguarding against disruptions.

Security:

- JSON Web Tokens (JWT) is used by the application to securely authenticate and authorize users. When a user successfully registers and logs in, the JWT approach generates an access token. This access token is delivered with every request made to the server after it has been saved in a cookie on the client side.
- The JWT approach is used to produce an access token during user registration and login. The inbuilt Python module algorithm called HS256 is used and also the secret key (ACCESS TOKEN SECRET) is used to sign the user. Additionally, the token has a predetermined time limit (TOKEN DURATION SECONDS).
- On the client side, the produced access token is kept in a cookie. The access token cookie is deleted after the user logs out to make sure they can't access any protected content again.
- The access token is given to the server with each request made by a user. Prior to handling the request, the server uses the OAuth2PasswordBearer protocol to verify the access token. The request will only be processed further and an appropriate response will be given if the token is valid.

So, In this way, the Security feature is implemented in the application using JSON Token.


```

ACCESS_TOKEN_SECRET = "70b159270abc185e853b00bc01e7f34640fa1f4a20aa51334e6db29afe19c119"
JWT_ALGORITHM = "HS256"
TOKEN_DURATION_SECONDS = 60*60*60
oauth2scheme = OAuth2PasswordBearer(tokenUrl="/auth/login")
token_duration = timedelta(seconds=int(TOKEN_DURATION_SECONDS))
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def hash_password(password):
    hashed_password = pwd_context.hash(password)
    return hashed_password

def isPasswordsMatch(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

```

Fig 7 : JWT Token Generation And Hashing Technique(using HS256 hashing algorithm)

Consistency:

Utilizing Kafka as a message broker, we achieved data consistency across user activities. For instance, actions such as following/unfollowing users or adding/editing/deleting posts are instantly propagated to other users. This ensures that all users have real-time visibility into user activities.

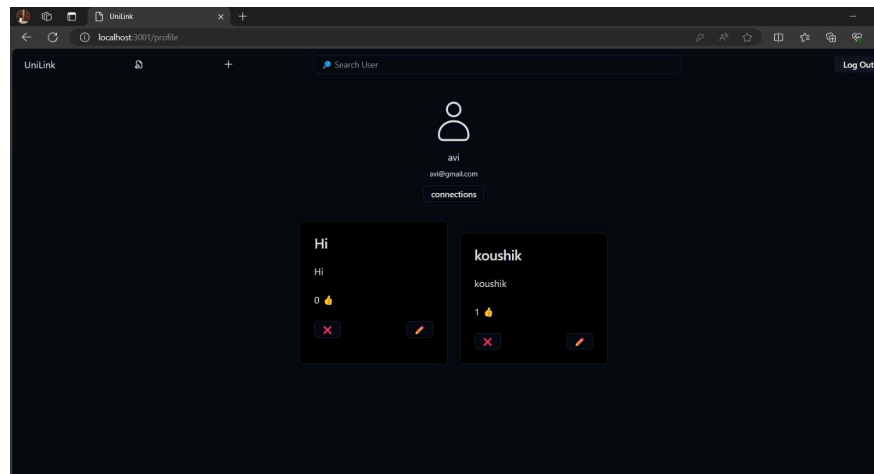


Fig 8: Local host 3001 (avi user profile)

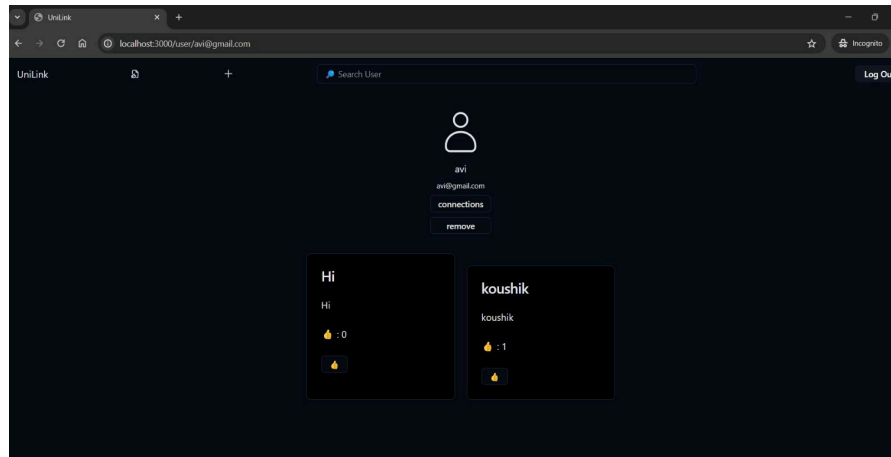


Fig 9: Localhost 3000 (searching avi from koushik log in details)

In the pictures above, we can see there are 2 local hosts that are working concurrently, In one server(in port 3001) Avi's profile is logged in and on another server(in port 3000) koushik profile is logged in. When a new post is created in port 3001, named 'Hi', it will concurrently reflect the post named 'Hi' in port 3000. This is how consistency is achieved.

Replication:

The system uses replication in order to improve scalability and resource management to allocate resources for optimal performance. Multithreading is used for this purpose. The data would be stored in a cluster and its respective replicas would be stored in different clusters. Such an approach makes data retrieval easy and enables us to build a distributed and decentralized system.

A replication factor (RF), which controls how many copies of each data row are kept in the cluster, is used to configure replication. Replication factors of two are commonly used in two-node clusters to store one copy of the data on each node.

In CQL (Cassandra Query Language), use the CREATE KEYSPACE command to set the replication factor while creating a keyspace. For instance, the following command uses the SimpleStrategy for a single datacenter to create a keyspace called "unilinkspace" with a replication factor of two:

```
CREATE KEYSPACE unilinkspace WITH replication = {'class': 'SimpleStrategy',
'replication_factor': 2};
```

```
container-1-cassandra
cassandra:jammy
e28e12879456
9042.9042

Logs Inspect Bind mounts Exec Files Stats
# cqlsh;
Connected to Test Cluster at 127.0.0.1:9042
[cqlsh 6.1.0 | Cassandra 4.1.4 | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh> use unlinkspace;
cqlsh:unlinkspace> select * from user_credentials;

email | hashed_password
-----|-----
avi@gmail.com | $2b$12$N5yriHuaR7N699E7JE3uq.he0L4.0wQJISInpi7NX/j6SILXT/00y
abd@gmail.com | $2b$12$R0J.GJofMF9YtGLVW.Z9LuGmHvadL0/2io7SHAEoSREKsmk6P..le
jake@gmail.com | $2b$12$GyNaKfBNRZ9aLuBKPdD6.0FLxluqMU4DSrvXDrYa.y95QLYK1DZqK
koushik@gmail.com | $2b$12$1V55S8ZGtoVgkNdGJ/har.wIPGob0QLnj7p9Y4Ur.I3BJLFUMLWta
abc@gmail.com | $2b$12$0d5xGZ3KGUwMBD1QKEvLAeqLAFF568.xjvL0nBxmCdgaNnTfyMlwq
steve@gmail.com | $2b$12$71NcaQzBnrBbL41CEGHhE.qt06r/yll21lh9GIKNSz1CtIppvF.UG
jsirigiri@gmail.com | $2b$12$GqvJf/ZBg6z0LN0eMQxmAu7XoI9IM942NakjuoMBgZYUlpknmSJTK
peralta@gmail.com | $2b$12$V481ZKklvqtb634qpvgFpez/.oVct3KqRJMMJa6e0uzLTg28B5d0i

(8 rows)
cqlsh:unlinkspace> select * from posts;

email | xtimestamp | id | description | interested | title
-----|-----|-----|-----|-----|-----
avi@gmail.com | 2024-05-04 22:59:07.851000+0000 | 8c3d4617-13a5-4b22-b0dd-2e4bba3c4214 | Hi | 0 | Hi
avi@gmail.com | 2024-04-29 07:57:00.181000+0000 | 525fe1c8-6188-48f8-ab10-c414046b44d1 | koushik | 1 | koushik
jake@gmail.com | 2024-04-29 16:30:15.691000+0000 | c432acf9-182b-4da7-8131-91ee510c80a8 | ds | 0 | rg
jake@gmail.com | 2024-04-29 03:12:21.779000+0000 | 23362eea-5389-45ac-ae0c-067410c0f7d2 | EditDescOne | 0 | checkOneEdit
koushik@gmail.com | 2024-05-04 22:48:58.450000+0000 | 036f9cc7-9e40-43ec-b4c4-ea190363f70c | Hello, I am a new user | 0 | Hello
peralta@gmail.com | 2024-04-29 03:20:59.417000+0000 | d6a42702-2805-46f8-9f42-e9ce959b8e07 | fromperaltafaultasdf | 1 | fromperaltafault

(6 rows)
```

Fig 10: Database-1(Cassandra container -1)

```
container-2-cassandra
cassandra:jammy
a6f709211879
9043.9042

Logs Inspect Bind mounts Exec Files Stats
# cqlsh;
Connected to Test Cluster at 127.0.0.1:9042
[cqlsh 6.1.0 | Cassandra 4.1.4 | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh> use unlinkspace;
cqlsh:unlinkspace> select * from user_credentials;

email | hashed_password
-----|-----
avi@gmail.com | $2b$12$N5yriHuaR7N699E7JE3uq.he0L4.0wQJISInpi7NX/j6SILXT/00y
abd@gmail.com | $2b$12$R0J.GJofMF9YtGLVW.Z9LuGmHvadL0/2io7SHAEoSREKsmk6P..le
jake@gmail.com | $2b$12$GyNaKfBNRZ9aLuBKPdD6.0FLxluqMU4DSrvXDrYa.y95QLYK1DZqK
koushik@gmail.com | $2b$12$1V55S8ZGtoVgkNdGJ/har.wIPGob0QLnj7p9Y4Ur.I3BJLFUMLWta
abc@gmail.com | $2b$12$0d5xGZ3KGUwMBD1QKEvLAeqLAFF568.xjvL0nBxmCdgaNnTfyMlwq
steve@gmail.com | $2b$12$71NcaQzBnrBbL41CEGHhE.qt06r/yll21lh9GIKNSz1CtIppvF.UG
jsirigiri@gmail.com | $2b$12$GqvJf/ZBg6z0LN0eMQxmAu7XoI9IM942NakjuoMBgZYUlpknmSJTK
peralta@gmail.com | $2b$12$V481ZKklvqtb634qpvgFpez/.oVct3KqRJMMJa6e0uzLTg28B5d0i

(8 rows)
cqlsh:unlinkspace> select * from posts;

email | xtimestamp | id | description | interested | title
-----|-----|-----|-----|-----|-----
avi@gmail.com | 2024-05-04 22:59:07.851000+0000 | 8c3d4617-13a5-4b22-b0dd-2e4bba3c4214 | Hi | 0 | Hi
avi@gmail.com | 2024-04-29 07:57:00.181000+0000 | 525fe1c8-6188-48f8-ab10-c414046b44d1 | koushik | 1 | koushik
jake@gmail.com | 2024-04-29 16:30:15.691000+0000 | c432acf9-182b-4da7-8131-91ee510c80a8 | ds | 0 | rg
jake@gmail.com | 2024-04-29 03:12:21.779000+0000 | 23362eea-5389-45ac-ae0c-067410c0f7d2 | EditDescOne | 0 | checkOneEdit
koushik@gmail.com | 2024-05-04 22:48:58.450000+0000 | 036f9cc7-9e40-43ec-b4c4-ea190363f70c | Hello, I am a new user | 0 | Hello
peralta@gmail.com | 2024-04-29 03:20:59.417000+0000 | d6a42702-2805-46f8-9f42-e9ce959b8e07 | fromperaltafaultasdf | 1 | fromperaltafault

(6 rows)
```

Fig 11: Database -2(Cassandra container - 2)

In the above pictures, we can see that data is replicated in both databases(clusters). so we have the same data(user_credentials and posts) in both clusters. This is how data replication is done in this distributed system using the Cassandra database.

Scalability:

It refers to ensuring that system performance is not jeopardized in the face of an increasing number of users and messages. To manage growing requests, the caching techniques using Redis are implemented.

- **Caching:** If a user creates a post and still wants to access the post that has been made. Instead of requesting the Database, the request is sent to check in the cache memory and if the requested data is found in the cache the request is further not sent into the database. By storing frequently accessed data in memory, Redis can significantly improve the performance of applications by reducing the need to retrieve data from slower data stores such as databases.

```
def get_redis_connection():  
    r = redis.Redis(  
        host='us1-smashing-louse-41806.upstash.io',  
        port=41806,  
        password='13259e87c12441009b3b5905ebe0c840',  
    )  
    return r
```

Fig 12: Code Snippet For Redis Connection

Communications And Networking:

Maintaining effective communication and networking among distributed system nodes is crucial. By ensuring seamless interaction between nodes, we facilitate efficient data exchange and system functionality using the protocol called “**Gossip Protocol**”. It refers to the communication used in the distributed systems to disseminate information efficiently among nodes. It communicates directly with one another without relying on a central coordinator or server. Each node in the network periodically selects a random set of neighbor nodes to exchange information with. The below figure explains the code snippet for connection of two databases synchronizing the data in each of their data.

```

name: cassandra_cluster
services:
  node-1-cassandra:
    image: cassandra:jammy
    container_name: container-1-cassandra
    ports:
      - "9042:9042"
    environment:
      - CASSANDRA_SEEDS=node-2-cassandra
  # Connecting to 2nd DB
  node-2-cassandra:
    image: cassandra:jammy
    container_name: container-2-cassandra
    ports:
      - "9043:9042"
    environment:
      - CASSANDRA_SEEDS=node-1-cassandra
  # Connecting to 1st DB
networks:
  cassandra-network:

```

Fig 13: Code Snippet Showing Both Cassandra Clusters Are Connected

TECHNOLOGY STACK

The following technologies have been used for the development of this project. They are given below:

Front-End:

We have used technologies like Next.js. It simplifies React application development with features like server-side rendering and static site generation. Its dynamic routing and API routes enable flexible and efficient development of dynamic web applications. Next.js automatically optimizes code splitting and image loading for improved performance. With built-in CSS and styling options, Next.js provides a comprehensive solution for building modern web applications.

Back-End:

FastAPI and Next.js together offer high performance and scalability for building modern web applications. With FastAPI's efficient backend API development and Next.js's frontend capabilities, developers can create full-stack applications seamlessly. Leveraging type safety and developer productivity features, this combination ensures robustness and rapid development. The asynchronous nature of FastAPI and optimized rendering in Next.js provide an excellent user experience with fast

page loads and efficient data handling.

Database:

The choice for the project has been Cassandra since its schema-less nature allows our system to be very flexible. Its scalability feature allows our system to handle a lot of database traffic.

```
def set_cassandra_connection():  
    cluster = Cluster()  
    session = cluster.connect("unispace")  
    connection.register_connection(name="default", session=session, default=True)  
    connection.set_default_connection("default")  
    connection.set_session(session)
```

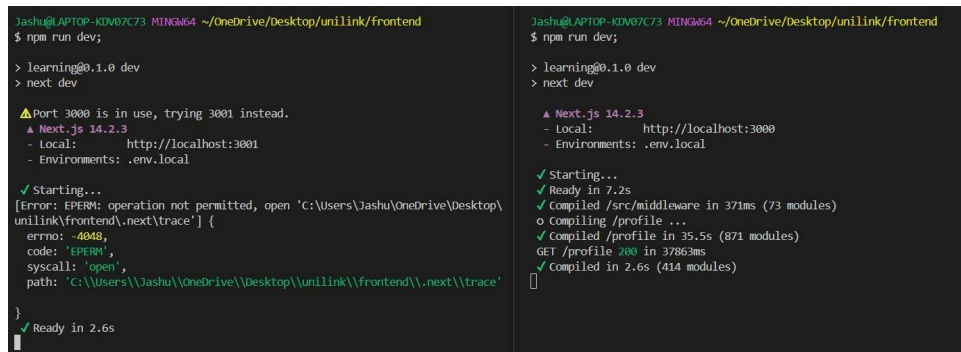
Fig 14: Database connection

This Fig:14 describes a function to connect to a Cassandra database cluster named “unispace”.

RESULTS AND EVALUATIONS

Now let us look at the execution of the project.

In the figure below, we can see that 2 different servers are running concurrently in 2 different port numbers(local hosts) i.e., 3000 and 3001. Here multiple servers exchange information with each other to achieve distributed communication. The servers work independently. We can also add more servers with different port numbers.



```
Jashu@LAPTOP-KDV07C73 MINGW64 ~/OneDrive/Desktop/unilink/frontend
$ npm run dev;

> learning@0.1.0 dev
> next dev

▲ Port 3000 is in use, trying 3001 instead.
▲ Next.js 14.2.3
- Local: http://localhost:3001
- Environments: .env.local

✓ Starting...
[Error: EPERM: operation not permitted, open 'C:\Users\Jashu\OneDrive\Desktop\unilink\frontend\.next\trace'] {
  errno: -4048,
  code: 'EPERM',
  syscall: 'open',
  path: 'C:\Users\Jashu\OneDrive\Desktop\unilink\frontend\.next\trace'
}
✓ Ready in 2.6s

Jashu@LAPTOP-KDV07C73 MINGW64 ~/OneDrive/Desktop/unilink/frontend
$ npm run dev;

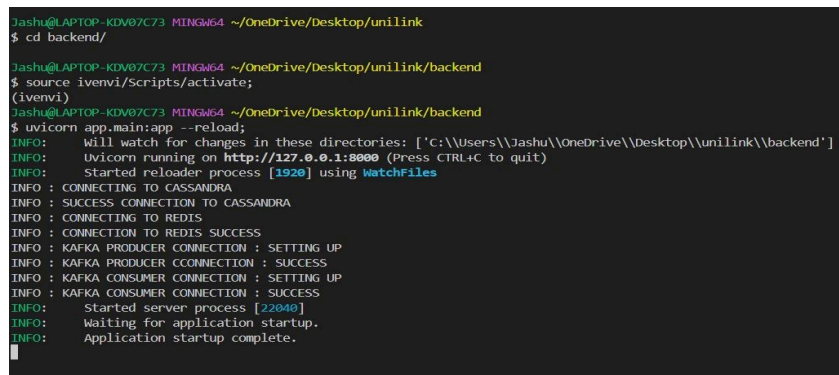
> learning@0.1.0 dev
> next dev

▲ Next.js 14.2.3
- Local: http://localhost:3000
- Environments: .env.local

✓ Starting...
✓ Ready in 7.2s
✓ Compiled /src/middleware in 371ms (73 modules)
o Compiling /profile ...
✓ Compiled /profile in 35.5s (871 modules)
GET /profile 200 in 37863ms
✓ Compiled in 2.6s (414 modules)
```

Fig 15: Concurrent server operation

The code snippet demonstrates launching a backend server where the server is listening on port 8000. The messages provide information on successful connection establishment to the Cassandra database, Redis cache, Kafka producer, and consumer.



```
Jashu@LAPTOP-KDV07C73 MINGW64 ~/OneDrive/Desktop/unilink
$ cd backend/

Jashu@LAPTOP-KDV07C73 MINGW64 ~/OneDrive/Desktop/unilink/backend
$ source ienvi/scripts/activate;
(iveni)
Jashu@LAPTOP-KDV07C73 MINGW64 ~/OneDrive/Desktop/unilink/backend
$ uvicorn app.main:app --reload;
INFO: Will watch for changes in these directories: ['C:\Users\Jashu\OneDrive\Desktop\unilink\backend']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [1920] using WatchFiles
INFO: CONNECTING TO CASSANDRA
INFO: SUCCESS CONNECTION TO CASSANDRA
INFO: CONNECTING TO REDIS
INFO: CONNECTION TO REDIS SUCCESS
INFO: KAFKA PRODUCER CONNECTION : SETTING UP
INFO: KAFKA PRODUCER CONNECTION : SUCCESS
INFO: KAFKA CONSUMER CONNECTION : SETTING UP
INFO: KAFKA CONSUMER CONNECTION : SUCCESS
INFO: Started server process [22040]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Fig 16: Backend (Cassandra, Redis, Kafka connection)

Since Cassandra is not supported in Windows we are using Docker containers for using Cassandra. The figure shows the two databases running in Docker concurrently.



















<input type="checkbox"/>	Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
<input type="checkbox"/>	 cassandra_cluster		Running (2/2)	6.31%		43 minutes ago	  
<input type="checkbox"/>	 container-1-cassandr	 e28e12879456  cassandra:jammy	Running	3.32%	9042:9042 	43 minutes ago	  
<input type="checkbox"/>	 container-2-cassandr	 a6f709211879  cassandra:jammy	Running	2.99%	9043:9042 	43 minutes ago	  

Fig 17: Cassandra cluster

In this distributed system, a new user can sign up on the register page which is shown in Fig: 18

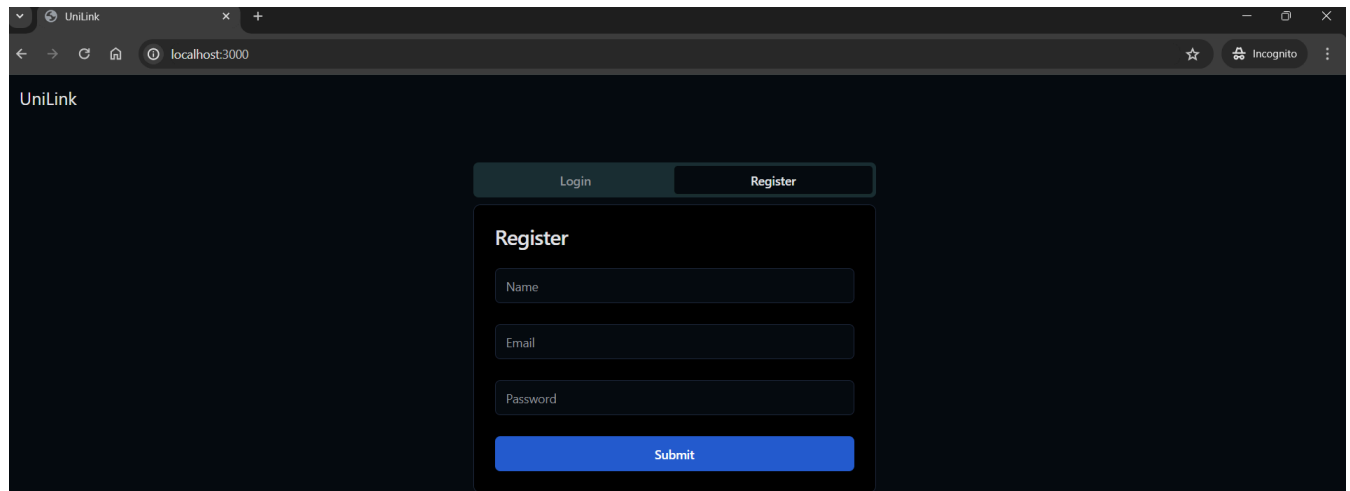


Fig 18: Register page for the new users.

Let us create an account in UniLink with the following credentials(the credentials can be changed as per the user's choice):

Username: [koushik](#)

Email: [koushik@gmail.com](#)

Password: [koushik@gmail.com](#)

The user enters all his credentials and clicks on “Submit”.

If the user enters an incorrect format for the mail ID there will be a message displayed which is “enter valid mail ID”.

For the existing users, they can log in directly on the login page

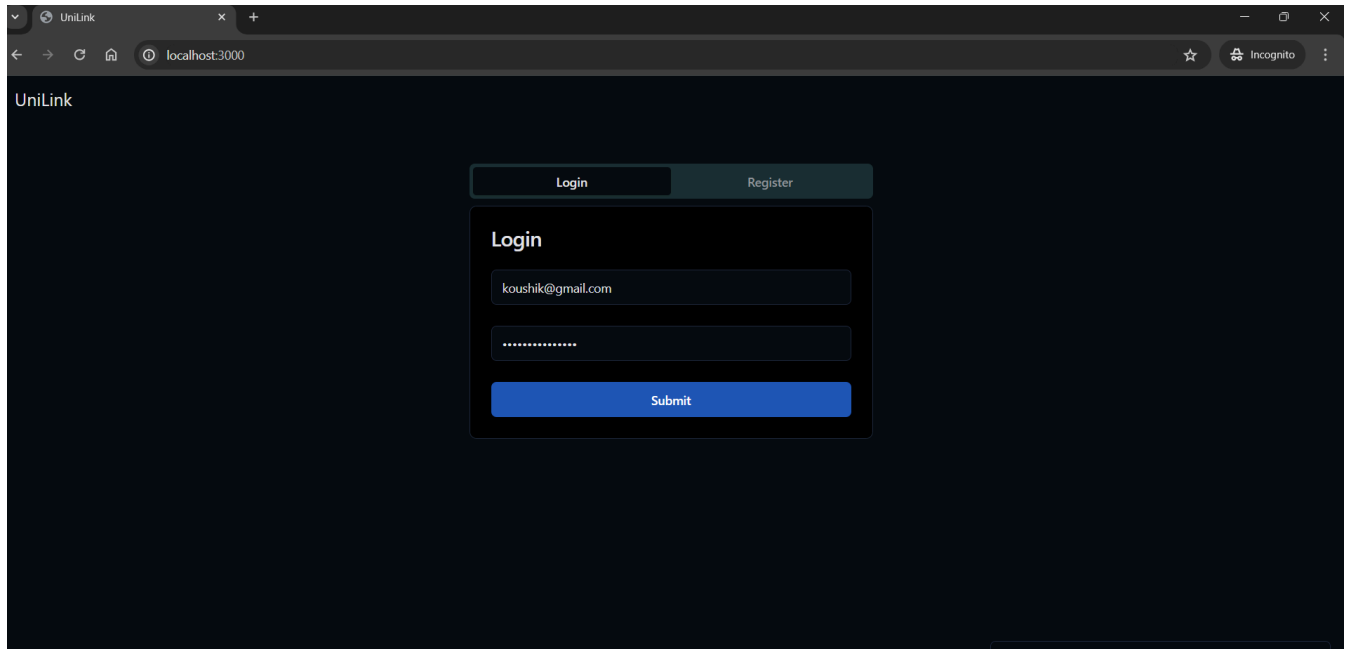


Fig 19: Login page for existing users.

After logging in, the user profile looks as shown in the below figure. There are a lot of options here like create a new post, search for a user, connect with them, and logged out.

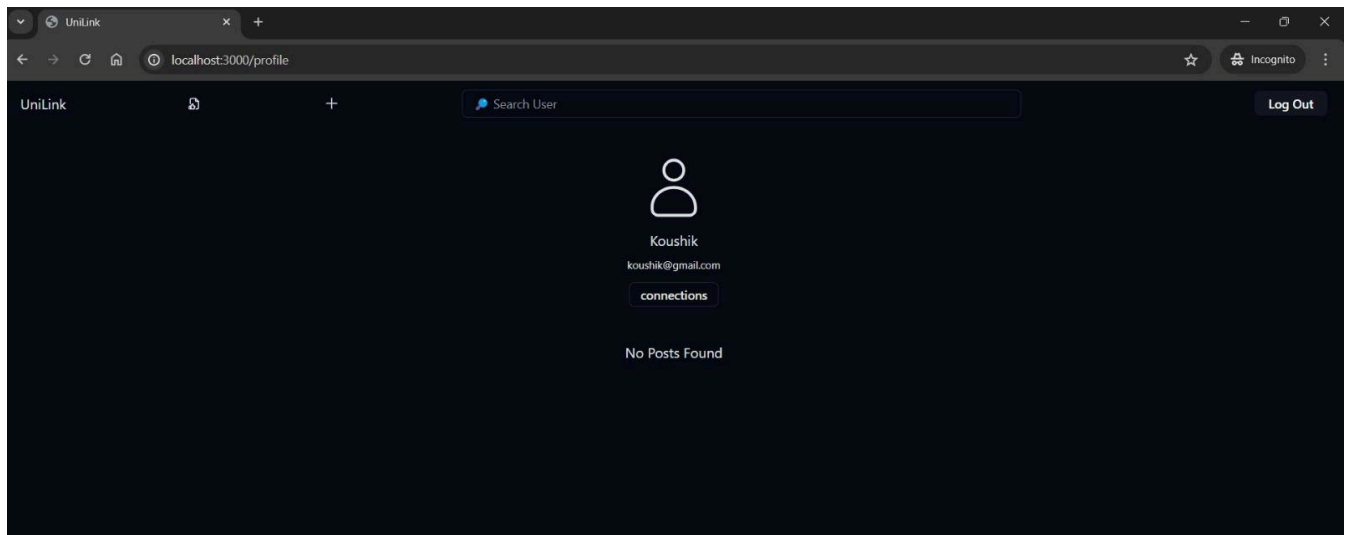


Fig 20: User profile.

In the below figure, we can see the user is creating a post by clicking on the '+' sign, then entering the subject and message, and then clicking on submit.

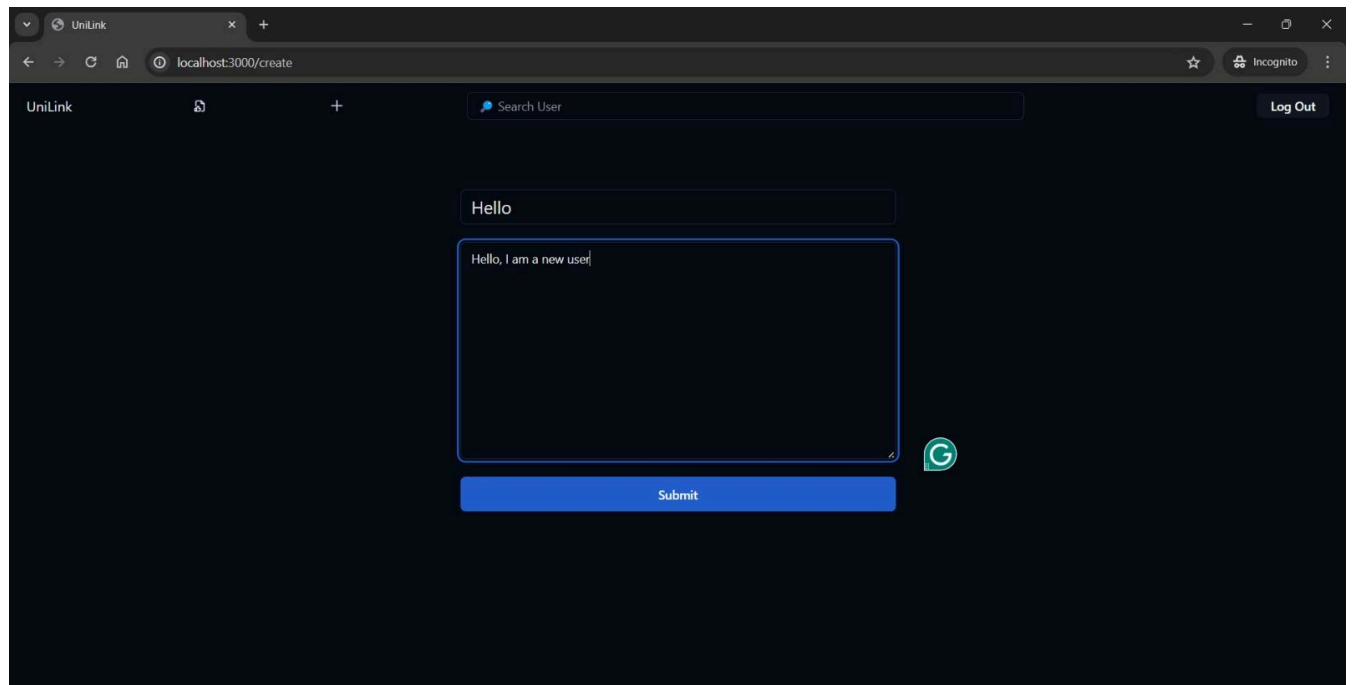


Fig 21: Post Creation.

Now there will be a change in the user profile page, as a new post is created. The post can be edited, or deleted. This post will be seen by all followers who follow the user.

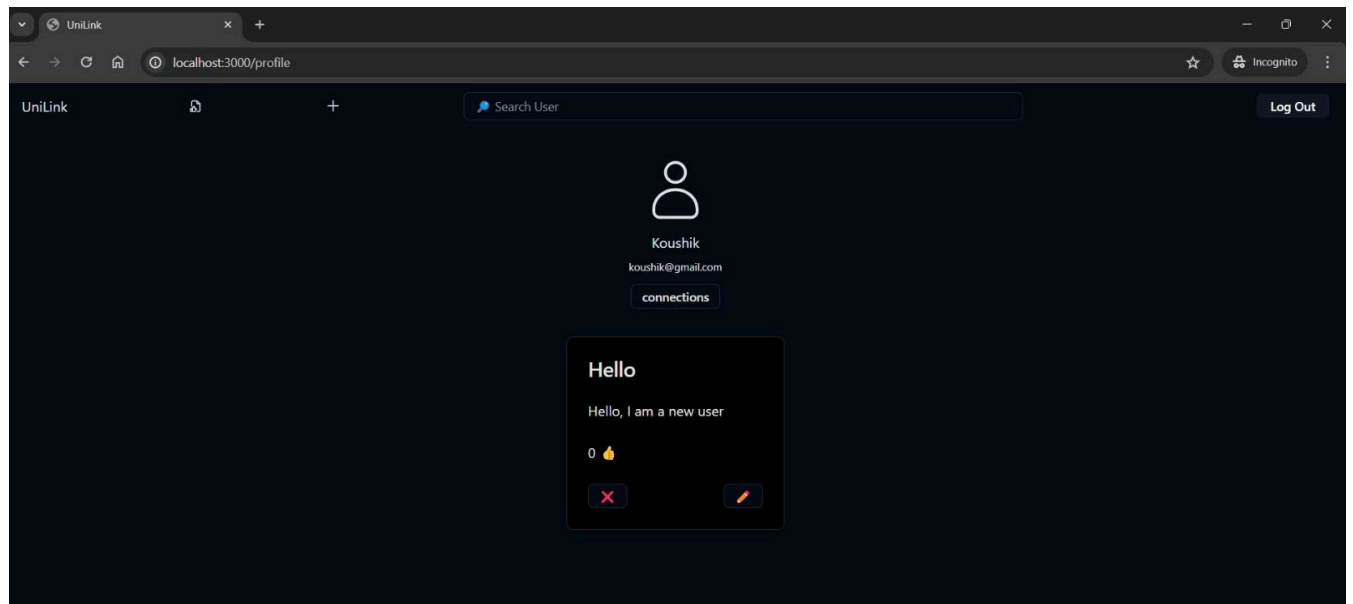


Fig 22: User profile after post creation.

There is a search event, where we can search for a user using the search option. After searching for a user, the page looks like as shown in the picture below,

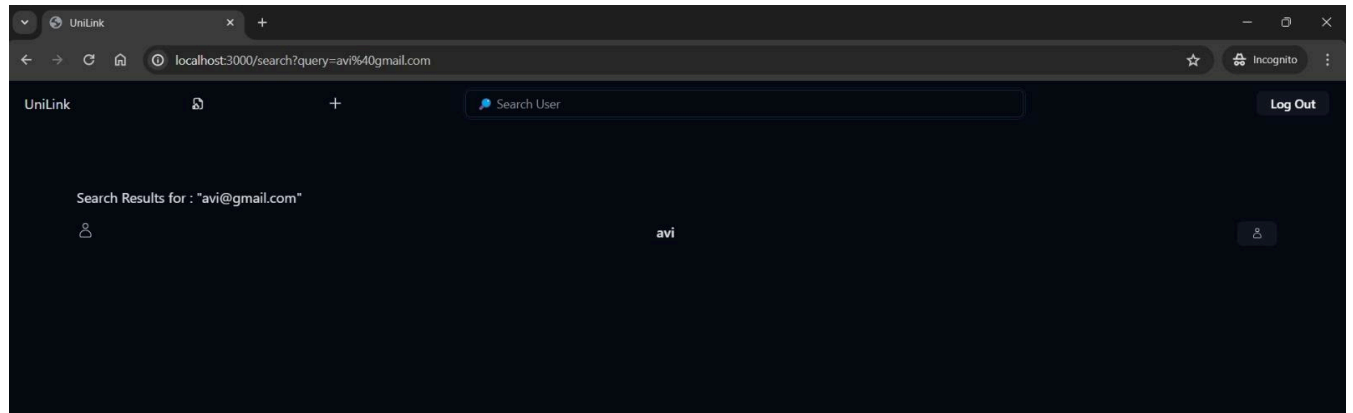


Fig 23: User search Result.

After clicking on that user, the page redirects to the avi user profile and we can see the posts posted by that particular user, see the connections of the user, add the user, and remove the user.

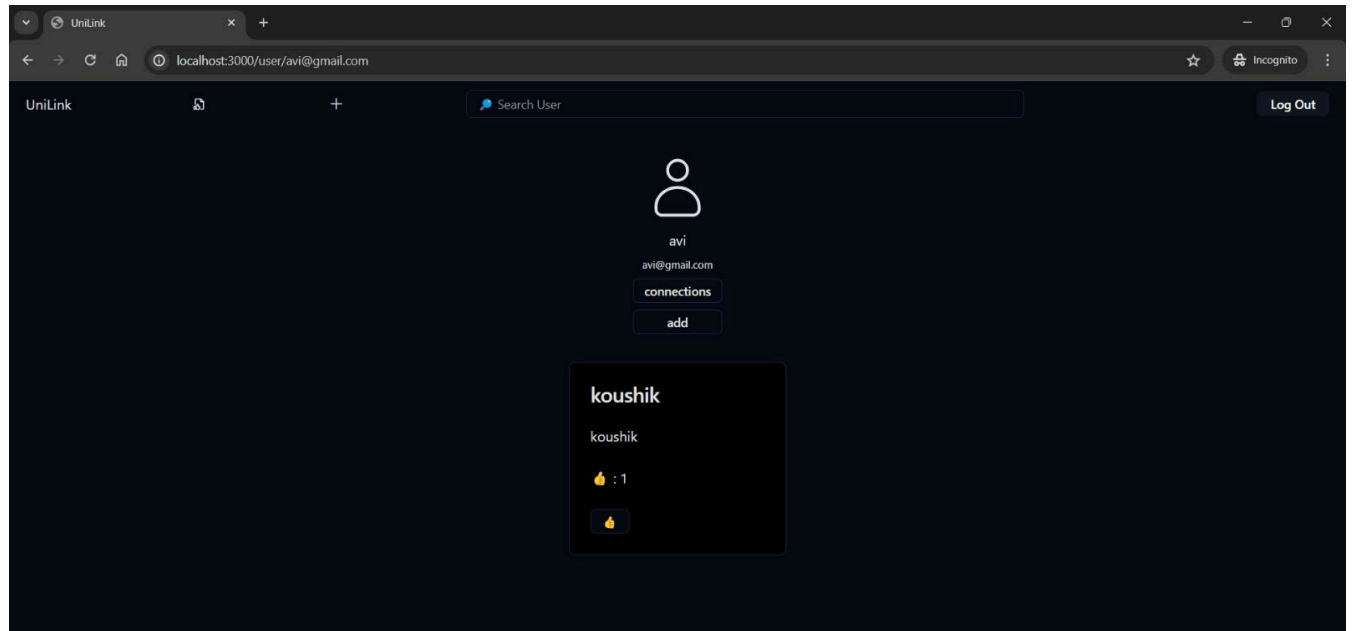


Fig 24: Other user's profile.

We can also see the connections made by the user by clicking on the connections button.

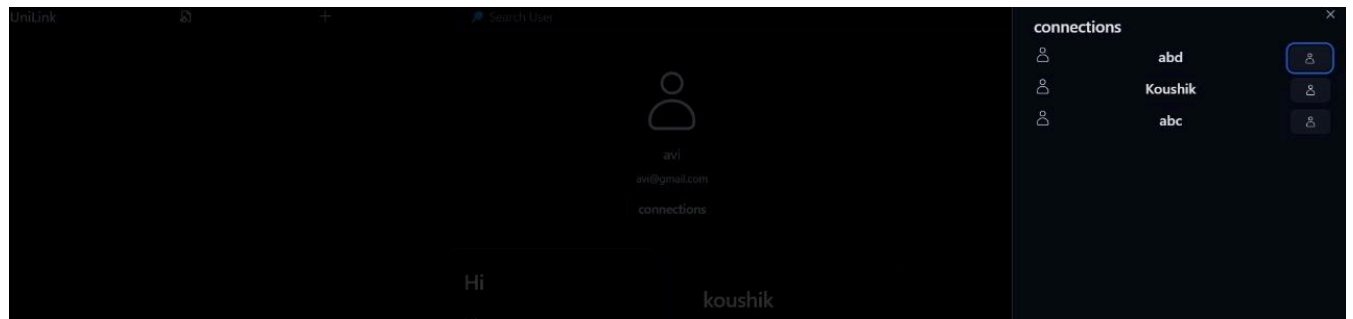


Fig 25: Connections made by a particular user

In Cassandra's database, we can see the details of all users who signed in. The email id and the encrypted password are stored and the password is encrypted using the hashing algorithm. the command used for this is given below.

`SELECT * FROM user_credentials;`

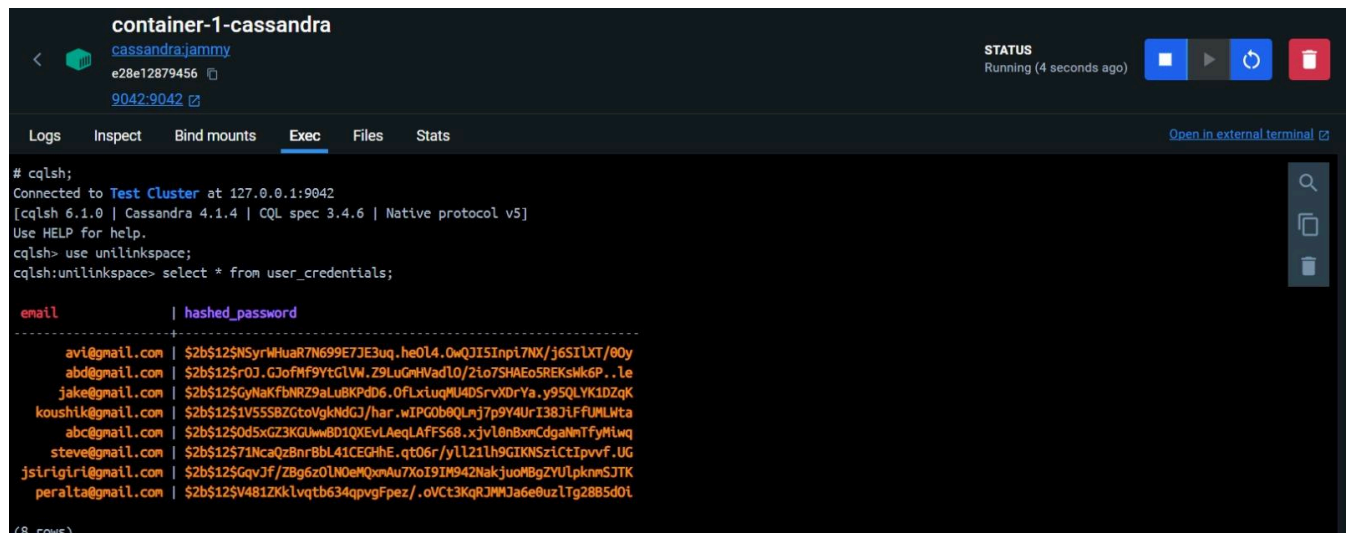


Fig 26: Cassandra Database: User Credentials.

We can also see the posts posted by all the users using the command.

`SELECT * FROM posts;`

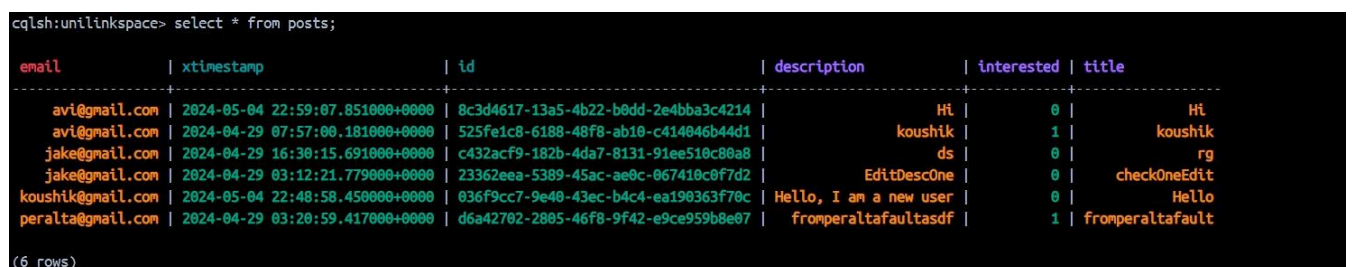


Fig 27: Posts by all users

REFLECTION

During the implementation of this project, we got a chance to learn various technologies and apply them practically in our project. As well as we have also encountered various challenges while developing this project which are as follows:

Learning Points:

- **Implementing Cache Using Redis:** Redis is one of the main technologies utilized in this application. Gaining knowledge about cache eviction regulations, cache-aside techniques for scalability, and Redis Python documentation allowed for the effective usage of Redis as a cache layer, improving the performance of the application.
- **Managing Of Cassandra Instances With Docker Containers:** The Cassandra cluster was created and maintained using Docker because the Cassandra Database was not compatible for Windows Operating system. Acquiring an understanding of Docker and Integrating the Database in Docker with UI is crucial and challenging in the application development.
- **Using Cassandra For Replication, Fault Tolerance:** A thorough understanding of fault tolerance and replication in a Cassandra cluster was necessary for the application to work correctly. This required understanding about the Gossip protocol's significance for preserving cluster state, the default consistency level of ONE, and hinting's role in guaranteeing data consistency. Gaining an understanding of these ideas made it possible to deploy a reliable and expandable database system.

Difficulties Overcome:

- **Synchronization Issues:** Ensuring synchronization between frontend and backend. Since we are using Docker it is very important to implement the API calls to the backend without any data loss.
- **Stack Variability:** Integrating multiple technologies across the frontend, backend, and database layers can be complex, especially when dealing with different programming languages, frameworks, and libraries in the application development. The microservices are used for the integration.
- **Error Debugging:** During the implementation the errors are occurring and to overcome these errors the various sites like stackoverflow, official documentations are referred.

LIMITATIONS AND CHALLENGES

We encountered some challenges and limitations associated with this project. They are:

- Denormalization in the UniLink project's database design can lead to a data redundancy problem, resulting in increased storage requirements. This duplication of data across multiple tables may consume more storage space than necessary, potentially impacting the overall efficiency and scalability of the application.
- One potential limitation could be the resource-intensive nature of Docker, the technology used for deploying the application. Docker requires substantial system resources, including CPU, memory, and storage, which could potentially limit the scalability of the application, particularly in resource-constrained environments.

FUTURE ENHANCEMENTS

The UniLink project presents several opportunities for enhancements to improve user experience and functionality. Integrating chat and group chat features would enable real-time communication and collaboration among students. A bookmark facility would allow users to save and easily access important posts or resources. Calendar synchronization for events would help students stay organized and informed about upcoming activities. A share feature would enable users to easily distribute relevant content with their peers. Additionally, creating a separate space for storing notes would provide students with a convenient and organized way to manage their academic and personal thoughts and reminders. These enhancements, if implemented, would contribute to a more comprehensive and engaging platform for college students to connect, communicate, and collaborate effectively.

TEAM CONTRIBUTION

These are the contributions made by each of the team members.

S.No	Group Members	Contributions
1	NARESH VEMULA	<ul style="list-style-type: none">• Designed the UI – Front end using Next JS made it user-friendly• Documentation• 25% - contribution
2	KOUSHIK REDDY KAMBHAM	<ul style="list-style-type: none">• Cassandra Database is implemented as models and tables for the distributed features• Replication and Fault Tolerance• 25% - contribution
3	SAI AVINASH VAGICHERLA	<ul style="list-style-type: none">• Redis cache is implemented for Lightning-Fast Data Access• Implemented JWT (Security)• 25% - contribution
4	JASWANTH SIRIGIRI	<ul style="list-style-type: none">• Implemented Fast API which is used for the concurrency and high performance for the back end• Architecture design• 25% - contribution

CONCLUSION

Using this application, various users can connect with each other, fostering communication and collaboration. Additionally, organizations can utilize this platform to disseminate information about events and keep their community updated on the latest news and developments. For instance, universities can leverage this application to announce upcoming elections, coordinate group gatherings, or promote cultural festivals.

The UniLink project represents a successful fusion of modern technologies and established practices, resulting in the creation of a robust and scalable distributed system. Leveraging technologies such as FastAPI, Next.js, Apache Cassandra, and Redis cache, a platform has been

developed that prioritizes user experience while ensuring fault tolerance, security, and scalability. Throughout the project, the focus has been on delivering a high-quality user experience and showcasing core distributed operating system features. The integration of Redis cache further enhances performance and scalability, ensuring optimal functionality.

Overall, the UniLink project exemplifies dedication to innovation, excellence, and user satisfaction. It provides a solution that meets the needs of modern users and offers organizations a reliable platform for communication and engagement.

REFERENCES

We would like to credit the following references which had been helpful during the course of our project.

- [1] <https://github.com/emmannweb/blog-mern-app>
- [2] <https://medium.com/building-the-open-data-stack/build-an-event-driven-architecture-with-apache-Kafka-apache-spark-and-apache-cassandra-6f0fc0c87e42>
- [3] <https://upstash.com/>
- [4] <https://github.com/pyropy/fastapi-socketio>
- [5] <https://www.w3schools.com/>