ZuTan

VMware Hackathon

website: zutan.netlify.app

Problem Statement

1. Our project aims to build a database as a service platform that provides a constant endpoint to users for basic CRUD operations to either SQL or NoSQL databases using our platform, with the operations running as serverless functions so they can update/fetch information from the frontend without setting up a backend and API, or managing the database on their own.

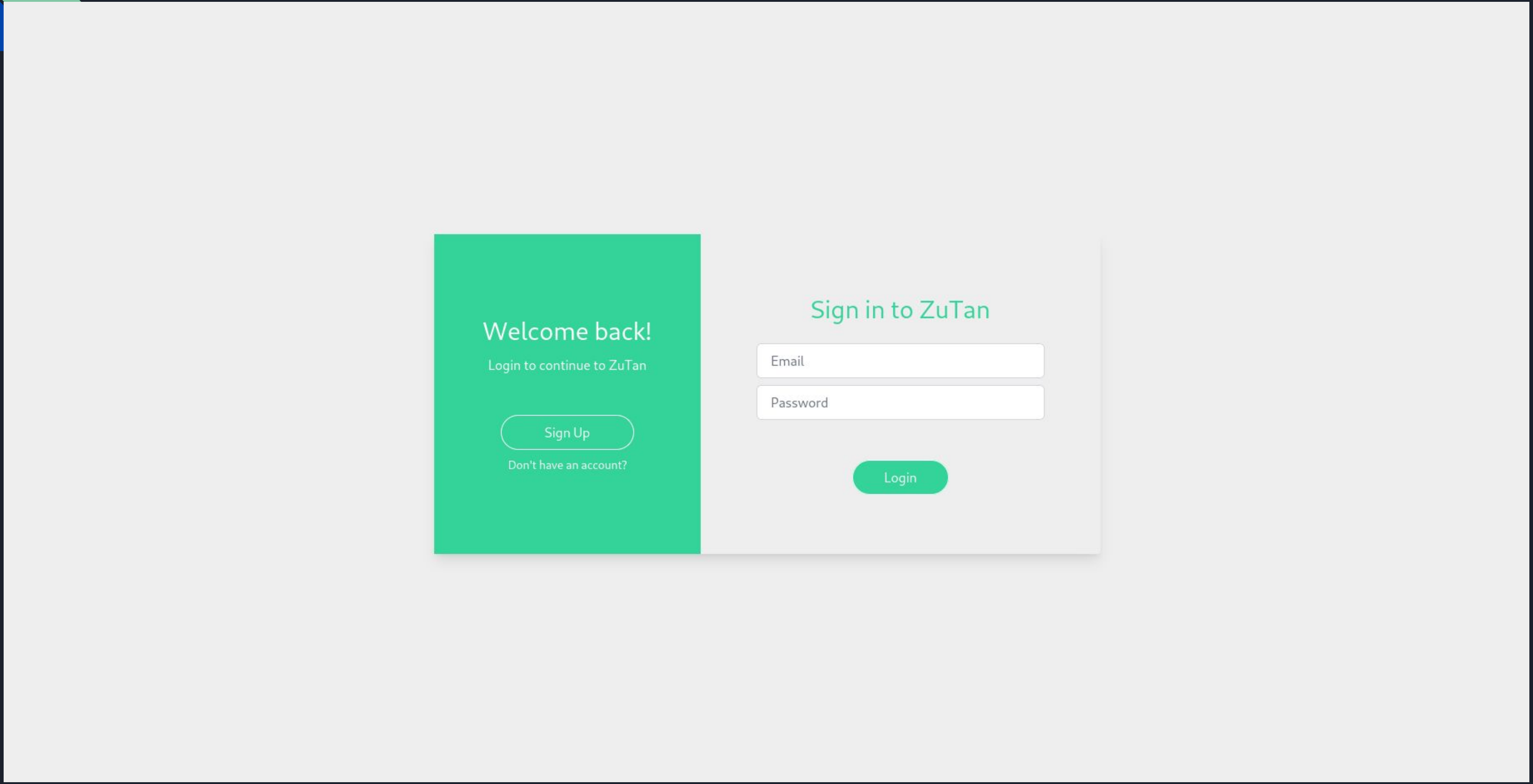
2. Additionally, we also provide a Unified Query mechanism which is database agnostic. This will be helpful for people who have no backend experience but need simple database functions like INSERT, SELECT, and more.

Approach/Implementation

The implementation involves a few separate pieces as mentioned below:

1. Docker containers are used to spawn databases on demand. Each container runs on a different port. The host machine's URL and the port number is then mapped to the user.
2. A basic uniform query system is also provided which utilises a JSON body sent as a POST request. The uniform query is then parsed and converted to the target database's native query system and executed.
3. The endpoints for these database operations are hosted on a serverless system (AWS Lambda) which ensures scalability and load handling for database operations.

Login Page



Welcome back!

Login to continue to ZuTan

Don't have an account?

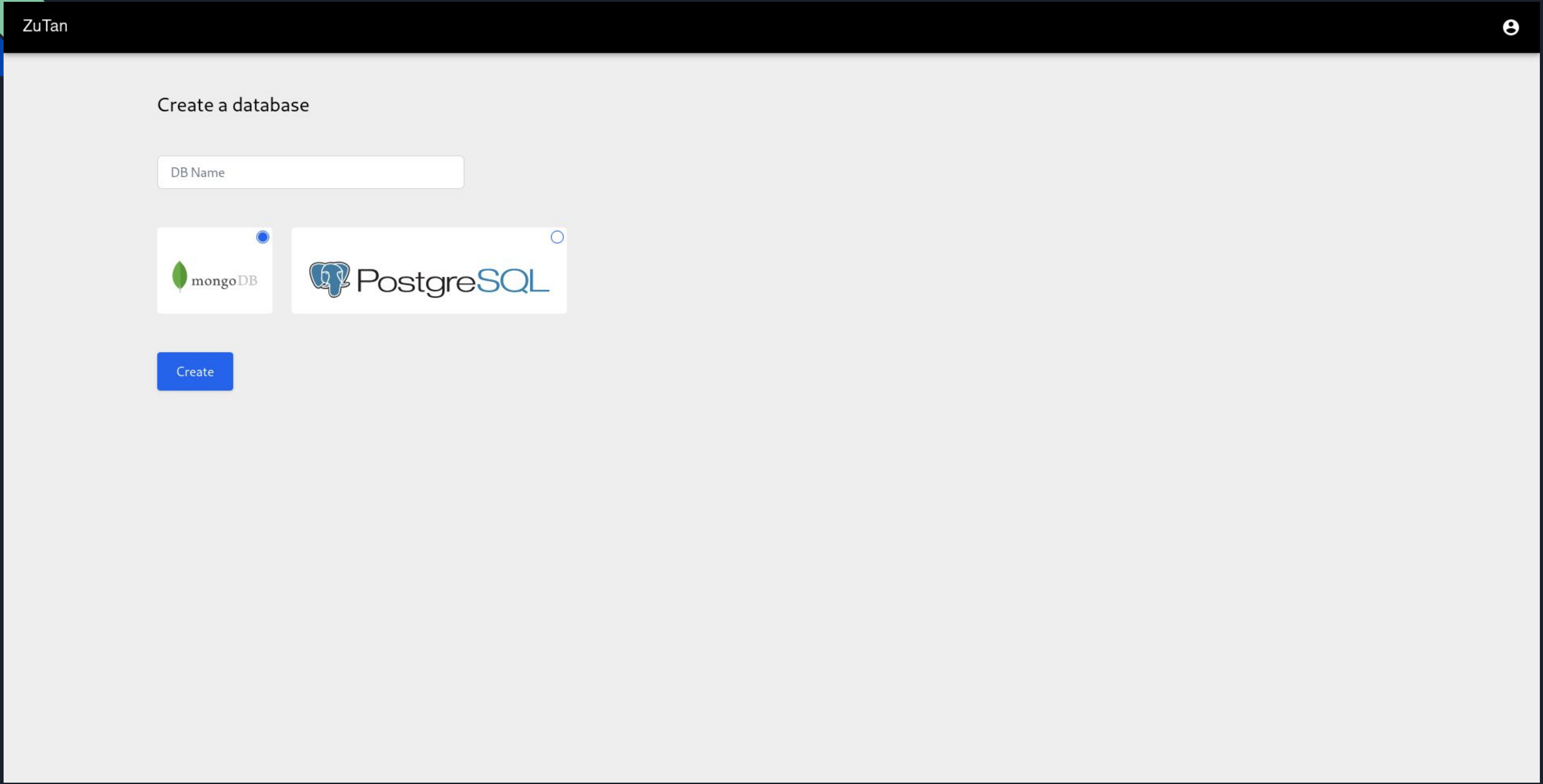
Sign in to ZuTan

Auth System

The login/signup Auth system was built by us that uses a Mongo database for storing user information and JSON Web Tokens to verify client for protected routes like creating new databases, etc.

The frontend is ReactJS with Redux state manager.

Create Database Page



ZuTan

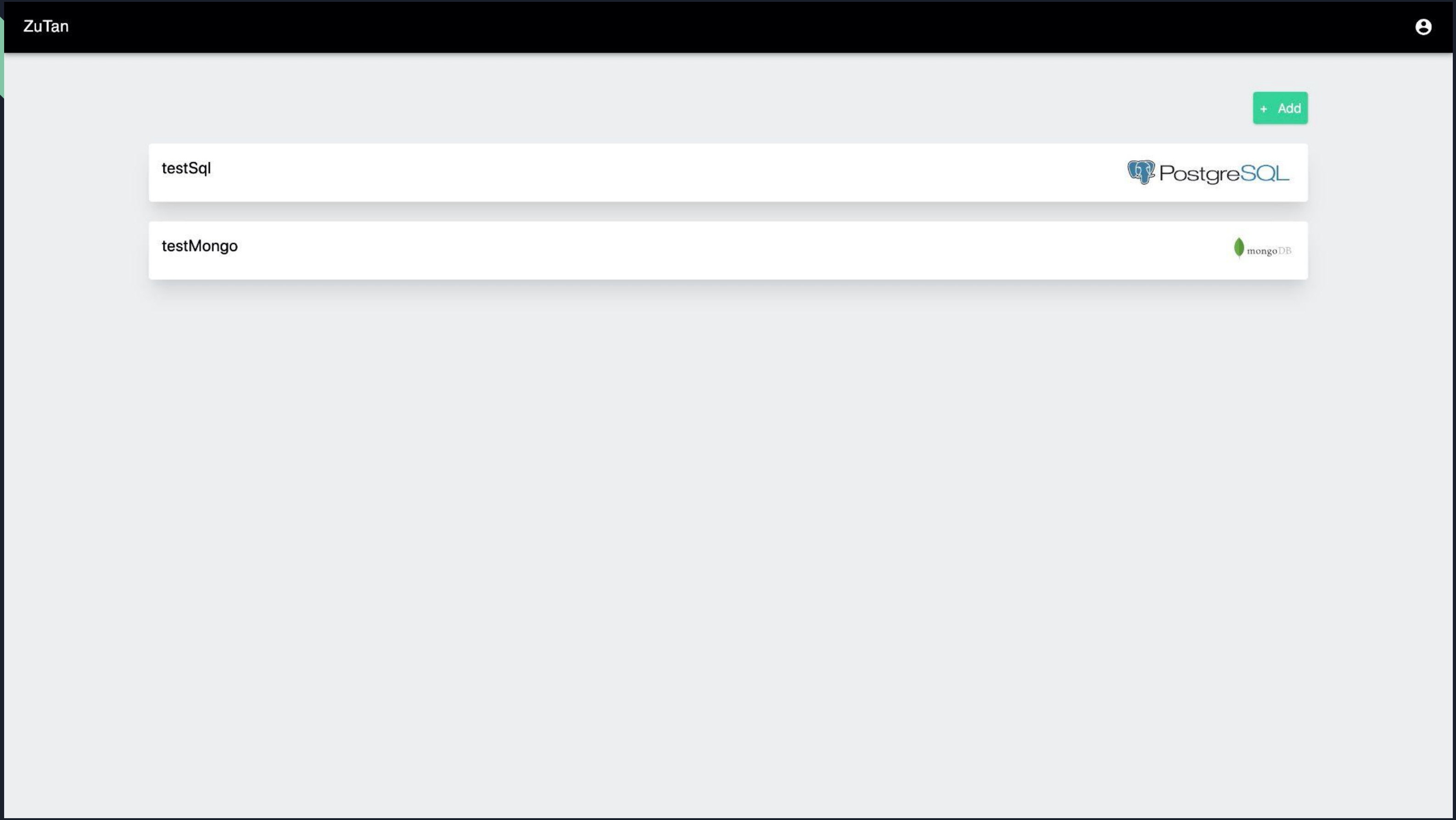
Create a database

DB Name

[✓ mongoDB] PostgreSQL

Create

View Databases Page



ZuTan

+ Add

testSql

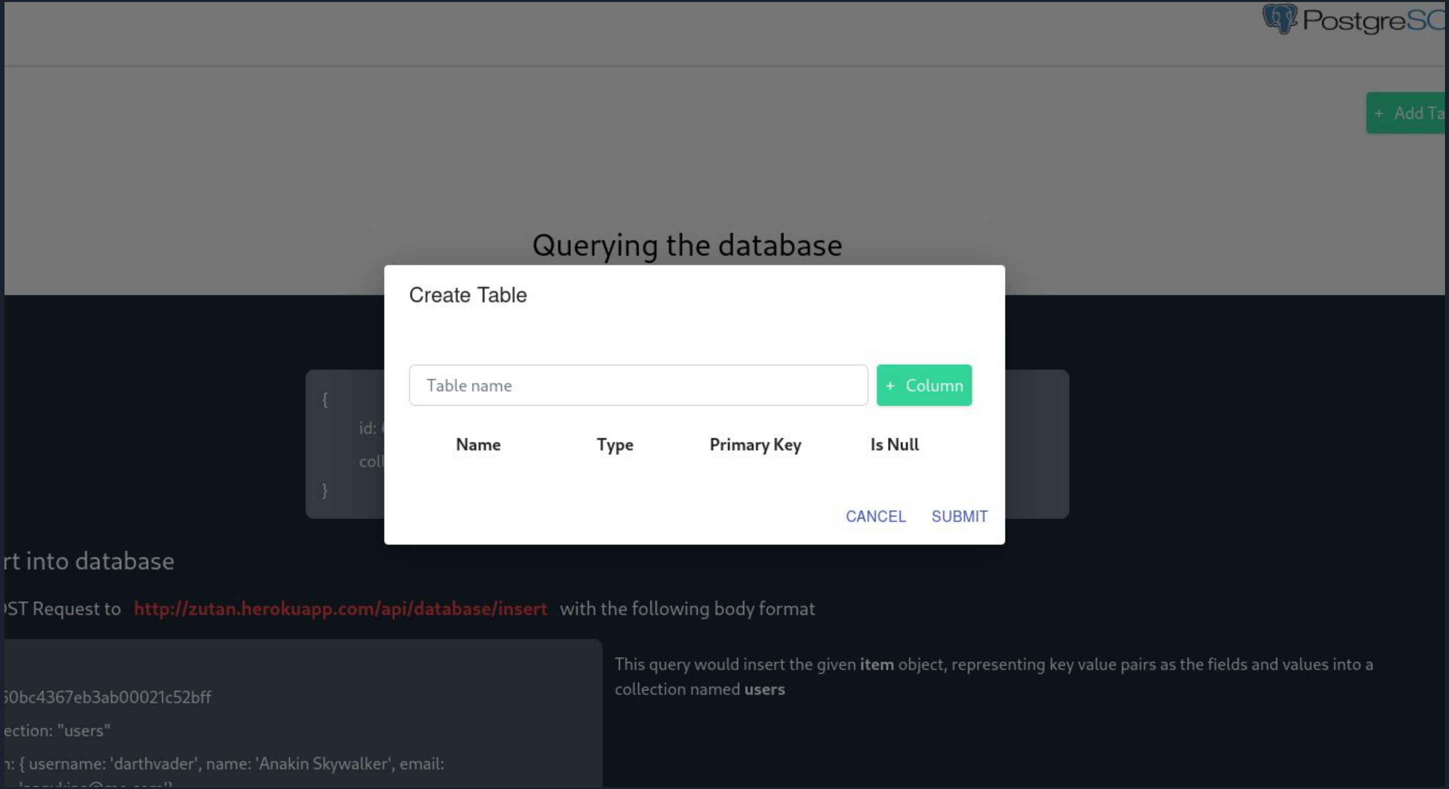
testMongo

Creating Database

The users have a choice between PostgreSQL or Mongo Database. When the user creates a new Database, we send a request to create the database to our DB Server which spawns the database as a Docker container and keeps track of database and it’s port, address, login credentials etc.

These metadata are used later when we need to query these databases.

Creating a table (1)



```
{
    id: (
    coll
}
```

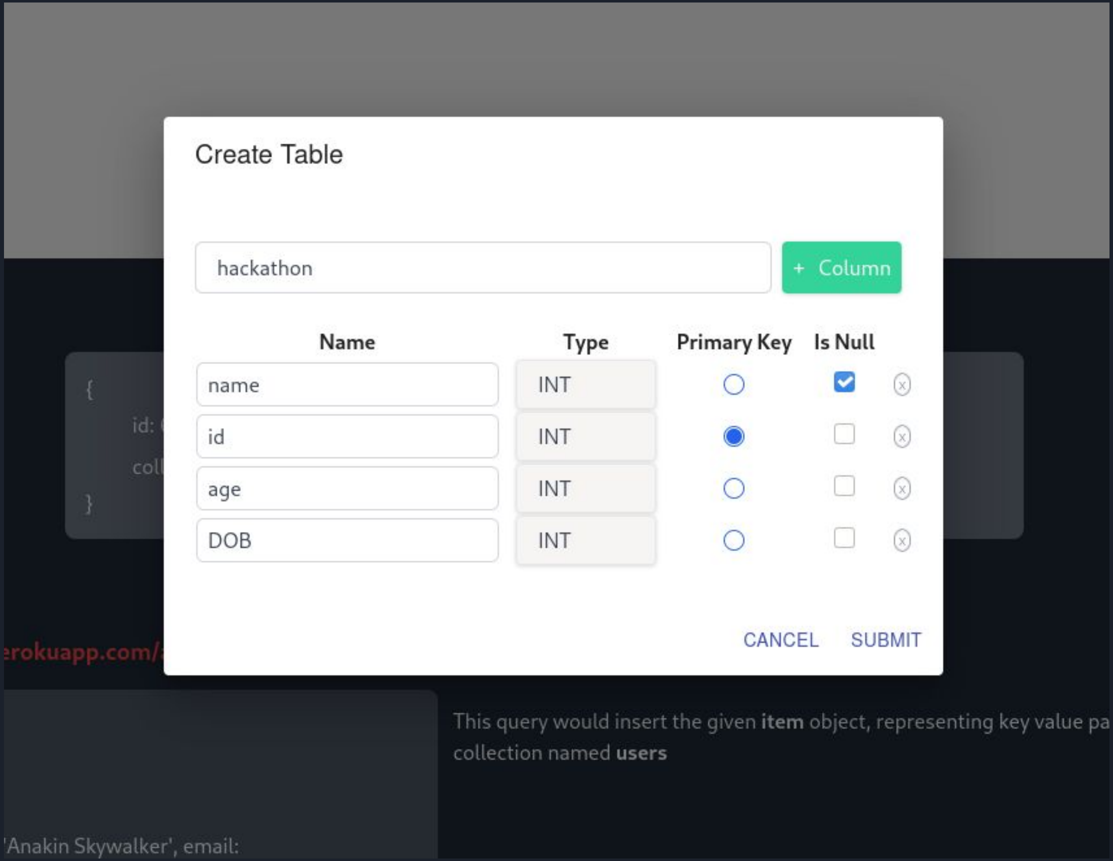
rt into database

ST Request to http://zutan.herokuapp.com/api/database/insert with the following body format

```
i0bc4367eb3ab00021c52bff
ection: "users"
n: { username: 'darthvader', name: 'Anakin Skywalker', email:
```

This query would insert the given **item** object, representing key value pairs as the fields and values into a collection named **users**

Creating a table (2)



Create Table

hackathon

+ Column

Name	Type	Primary Key	Is Null	
name	INT	○	✓	ⓧ
id	INT	●	☐	ⓧ
age	INT	○	☐	ⓧ
DOB	INT	○	☐	ⓧ

CANCEL SUBMIT

This query would insert the given **item** object, representing key value pa
collection named **users**

'Anakin Skywalker', email:

Creating Tables

For PSQL we require the users create the Table schema. This schema is then sent to the database server which creates the tables in the appropriate server.

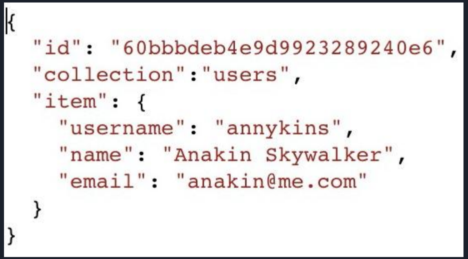
For Mongo, we don’t require any schema since it’s NoSQL.

Uniform Query System

1. As of now, the database’s (mongo and psql) create, read and delete methods are supported by our uniform abstraction.
2. Querying the database involves sending a POST request to one of the 3 endpoints, “/insert”, “/select”, “/delete” based on the required operation. A sample query for each is show below.
3. The body of the POST request is then parsed and converted to the target DB’s query language, executed and the results returned back to the user. Errors are handled on our end and an appropriate error message is sent back to the user.
4. Additionally, for SQL, item-table validation is also carried for every query to validate the schema provided. This however isn’t required for Mongo as it allows schema-less entries.

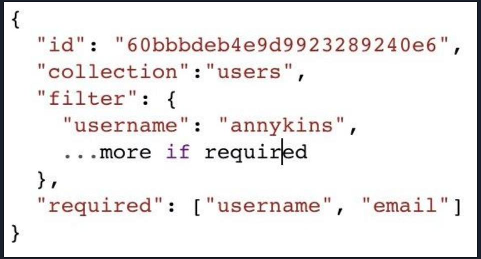
Sample queries are mentioned in the next slide.

Uniform Query System (2)



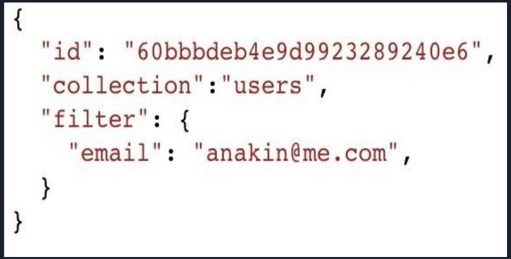
```
{
  "id": "60bbbdeb4e9d9923289240e6",
  "collection":"users",
  "item": {
    "username": "annykins",
    "name": "Anakin Skywalker",
    "email": "anakin@me.com"
  }
}
```

A sample insert query. Allows insertion of the object mentioned in the "item" field.



```
{
  "id": "60bbbdeb4e9d9923289240e6",
  "collection":"users",
  "filter": {
    "username": "annykins",
    ...more if required
  },
  "required": ["username", "email"]
}
```

A sample search query. Allows the usage of a filter, and a required field which returns only the required columns/fields. Returns all items if there's no filter/required.



```
{
  "id": "60bbbdeb4e9d9923289240e6",
  "collection":"users",
  "filter": {
    "email": "anakin@me.com",
  }
}
```

A sample delete query. Deletes all items that match the filter, or the entire collection if a filter is not provided

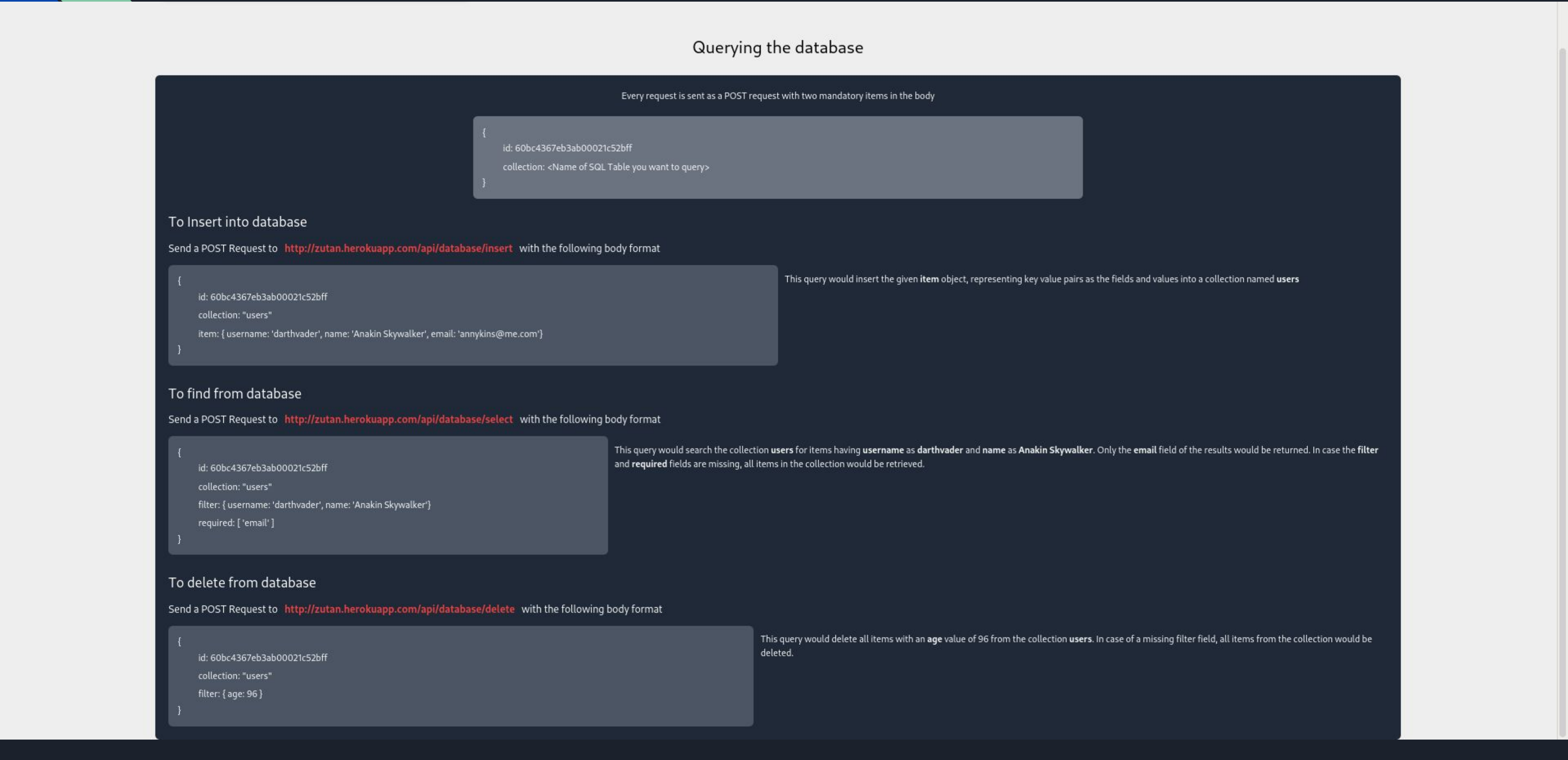
Serverless Functions

The queries from the users are run as Serverless functions on AWS Lambda. When the user sends a request to SELECT, INSERT or DELETE to the endpoint, it spawns a new serverless function that converts the user input to SQL Query or Mongo Command based on database, connects to database, executes command and returns any data is required like in case of SELECT. The Lambda Functions are behind an API Gateway which the users use to send their queries.

Why Serverless Functions?

With the the approach of serverless functions, there’s no centralised bottleneck if there are a large number of requests from different sites/apps coming in to make database queries. Each query is run independently with its own resources without having to setup and manually allocate server for these.

Documentation for UQL on the Platform



Querying the database

Every request is sent as a POST request with two mandatory items in the body

```
{
    id: 60bc4367eb3ab00021c52bff
    collection: <Name of SQL Table you want to query>
}
```

To Insert into database

Send a POST Request to http://zutan.herokuapp.com/api/database/insert with the following body format

```
{
    id: 60bc4367eb3ab00021c52bff
    collection: "users"
    item: { username: 'darthvader', name: 'Anakin Skywalker', email: 'annykins@me.com'}
}
```

This query would insert the given **item** object, representing key value pairs as the fields and values into a collection named **users**

To find from database

Send a POST Request to http://zutan.herokuapp.com/api/database/select with the following body format

```
{
    id: 60bc4367eb3ab00021c52bff
    collection: "users"
    filter: { username: 'darthvader', name: 'Anakin Skywalker'}
    required: [ 'email' ]
}
```

This query would search the collection **users** for items having **username** as **darthvader** and **name** as **Anakin Skywalker**. Only the **email** field of the results would be returned. In case the **filter** and **required** fields are missing, all items in the collection would be retrieved.

To delete from database

Send a POST Request to http://zutan.herokuapp.com/api/database/delete with the following body format

```
{
    id: 60bc4367eb3ab00021c52bff
    collection: "users"
    filter: { age: 96 }
}
```

This query would delete all items with an **age** value of 96 from the collection **users**. In case of a missing filter field, all items from the collection would be deleted.

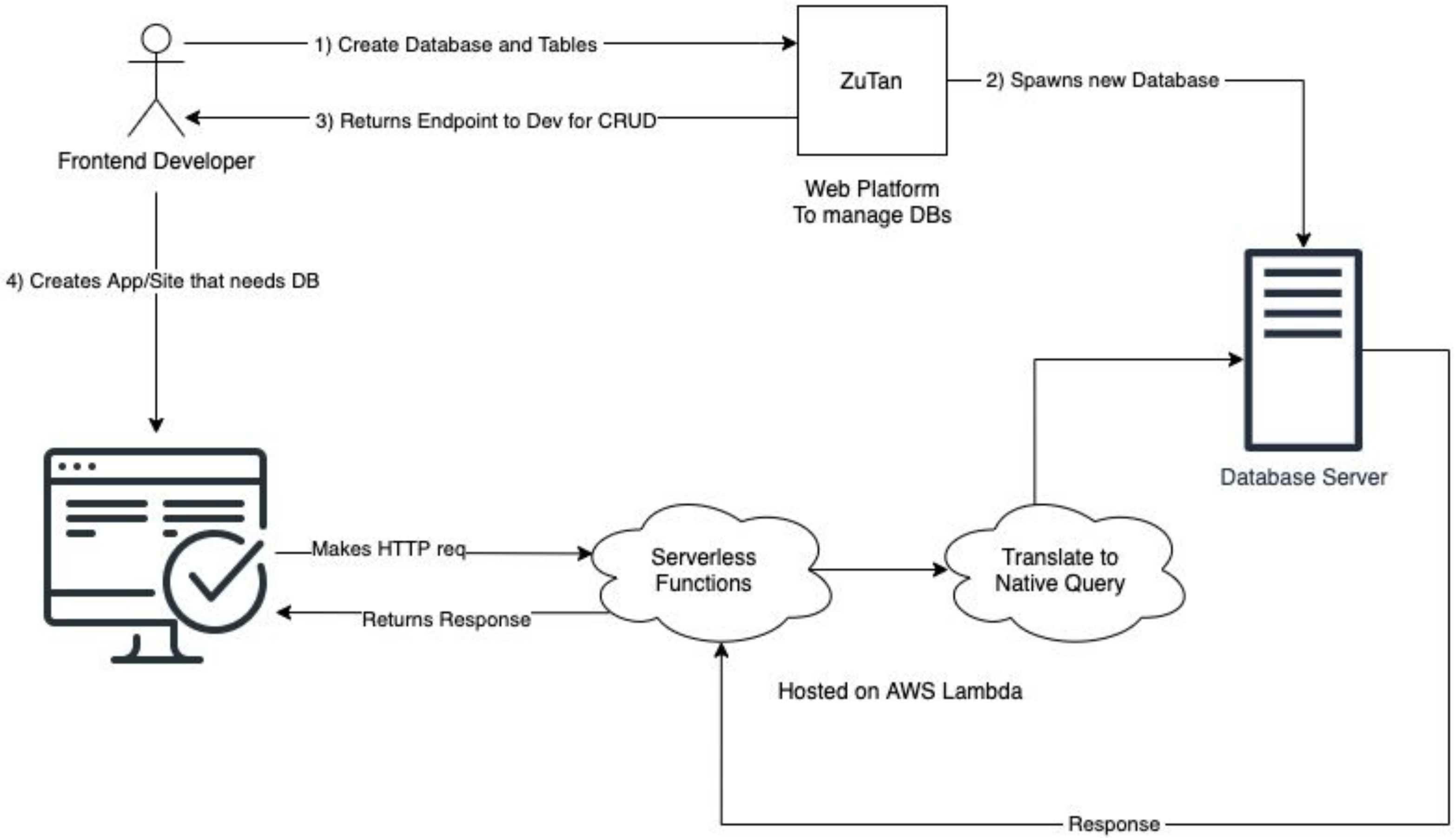


Figure: Control Flow

Architecture and Working

Once a user sets up the database on the platform, it sends a request to DB server which spawns the databases as containers and stores metadata like connection URI, Port, ID etc.

The user is now provided with a fixed endpoint to which they make requests for CRUD operations. The requests follow our Unified Query Format so the inputs from the user are DB Agnostic.

Each request from the user for a DB Query spawns a Lambda function that gets database information based on the ID in the request, converts the inputs to a Native SQL or Mongo command, connects to the database instance and executes the command.

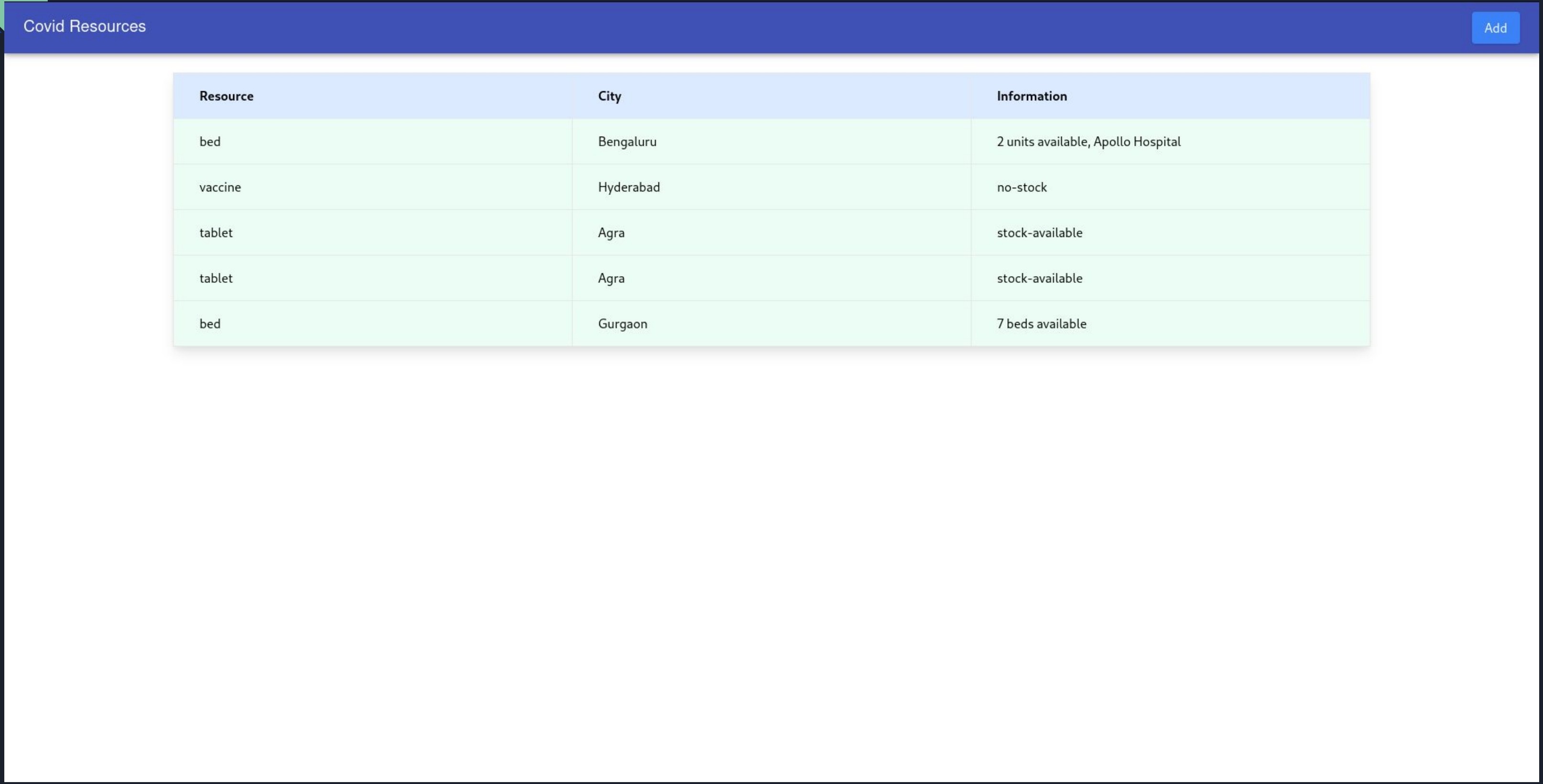
Thus, we have a platform that provides a powerful backend with database to the frontend developers without having to do any of the manual setup or hosting.

Sample use-case implementation and Demo

- We made a mock website that uses our platform as a backend service
- This site uses the insert query internally to add new information and sends the request with the object to the database server
- The new updated table can then be viewed, and this is achieved by the Select query being sent internally.

A covid resource website creating using our DB service as backend

website: https://zutan-sample.netlify.app/



Resource	City	Information
bed	Bengaluru	2 units available, Apollo Hospital
vaccine	Hyderabad	no-stock
tablet	Agra	stock-available
tablet	Agra	stock-available
bed	Gurgaon	7 beds available

Technologies Used:

- MERN Stack
  - Mongodb
  - Express JS
  - React JS
  - Node JS
- PostgreSQL
- Docker
- AWS Lambda
- AWS EC2 instance
- Deploying services: Heroku and Netlify

Future Implementations

1. The queries currently supported can be extended to include more complex queries. The uniform query system has to be rewritten as the current parsing system is quite basic and doesn't scale for more complex queries.
2. More databases can be supported.
3. A frontend can be provided, additionally to the API already provided on the platform to manage database entries without the user making API calls. This would allow our platform to double up as a Content Management System (CMS).
4. The current project uses AWS Lambda for serverless functionalities, this can be transferred to use VMWare Tanzu service for the same.

Video Demo

Drive Link to Video:

https://drive.google.com/drive/folders/169hlzyosNX1IJAdMBxbc-mRZ2zkqtQEH?usp=sharing

Youtube link to video:

https://youtu.be/O1lIBgTsG58

Members List

- Ashwin Alaparthi
- Raghav Roy
- Ritik Hariani
- Sourav Tekken

PES University

Thank you!