**HumorMe**! – **A Social Media Platform**

Oct 6th, 2023

SHRISTI SHRESTHA

Contents

[Introduction 3](#_Toc147494600)

[Related Works 4](#_Toc147494601)

[System Architectural Design 5](#_Toc147494602)

[**Table: users** 6](#_Toc147494603)

[**Table: jokes** 6](#_Toc147494604)

[**Table: comments** 6](#_Toc147494605)

[**Table: ratings** 6](#_Toc147494606)

[**Table: user\_followers** 6](#_Toc147494607)

[Detailed Description of Components 8](#_Toc147494608)

[**Account Registration and Login** 8](#_Toc147494609)

[**View and Search Joke Posts** 8](#_Toc147494610)

[**Post a Joke** 8](#_Toc147494611)

[**Comment on a Joke** 8](#_Toc147494612)

[**Rate a Joke** 8](#_Toc147494613)

[**Delete a Joke** 8](#_Toc147494614)

[**Visit User Profile** 8](#_Toc147494615)

[Conclusion 9](#_Toc147494616)

# **Introduction**

HumorMe! is a fun community space for people to share their sense of humor through jokes. Like the name suggests (HumorMe!), the idea is to spread humor and offer a virtual place of belonging for people. With rise of memes and funny reels in apps like Facebook and Instagram, there is a huge spike of users who consume exclusively funny content every day. The existing applications like Facebook, Reddit, Instagram, and Twitter are becoming extensively heterogenous in their content for example, politics, health, messaging, and others. This web app, HumorMe! exclusively caters to the humor content and can be thought as a starting point for aspiring comedians. Based on this idea, the webapp showcases a proof-of-concept designed for anyone. The app is hosted at at <http://3.143.17.104:3000> and offers an easy-to-use public space to post, comment and rate jokes. Users can browse through jokes anonymously and are required to sign up to engage in activities such as create jokes, comment, and rate jokes from other users. There are profile pages set up for each user who signed up and users can follow each other for customized content in future.

HumorMe! is a three-tier web application served by three individual servers: Apache serving client app, tomcat serving backend server (serving RESTful APIs), and a Postgres data server. The Apache server serves the presentation layer to this app and is composed of both static and dynamic content delivered to the browsers. The client content is developed and build using NextJS framework which supports/compiles JSX into HTML content. The client app is hosted at <http://3.143.17.104:3000>. We used SpringBoot framework to build backend server which exposes RESTful APIs and is hosted at a separate EC2 instance (check <http://3.141.47.20:8081/swagger-ui.html>). SpringBoot is a widely popular framework to build microservices that also builds an embedded Apache Tomcat server with the Java Servlets. We used this embedded Tomcat server to deploy our backend server. Postgres server is hosted at the same EC2 instance as backend server and are separated logically. User interactions such as page load, click buttons, submit forms trigger API calls (HTTP) to the backend server. The backend server processes API requests, run data query in Postgres and returns the data in JSON format to the users’ browsers.

# **Related Works**

# **System Architectural Design**

The system architecture follows the general three-tier web application design as described earlier in the Introduction section. There are two Amazon EC2 instances launched each serving a backend server (including a Postgres server), and the client server respectively. The first instance runs the Apache Web Server hosting the static web content at its root and can be accessed at <http://3.143.17.104:3000>. The second instance runs embedded Apache Tomcat server containing the Java Servlets. A Postgres server is installed in the same instance and configured to allow access to the SpringBoot application through a Java Database Connector (JDBC) library. The backend server is hosted at <http://3.141.47.20:8081/> and its RESTful APIs can be accessed here “/swagger-ui.html”. The client app makes HTTP calls to the RESTful APIs directly by using the public URL of the EC2 instance. Deployment guidelines are provided in detail with the source codes.

The clientside application is developed using NextJS framework which supports React (JavaScript XML, JSX) and provides configuration for building static content from JSX. For design, we used Ant Design (Antd) Library which provides high level components such as button, form, menu, dropdown, layout, and so on (<https://ant.design>). We also used “axios” to make API calls to the backend server (<https://www.npmjs.com/package/axios>). The libraries are installed using Node Package Manager (NPM) which is supported by NextJS. Any dynamic content required to populate the web pages undergo the process of making API calls to the server and inserting the responded data into the appropriate JSX elements. Since the backend and the client servers are hosted at separate domains, we also enabled Cross Origin Resource Sharing (CORS) functionality in the backend to allow the requests from the clientside. In other words, before making API calls using methods like POST, GET, PATCH, PUT AND DELETE, the browser makes an additional call “OPTIONS” to the server to check whether the server resource can be served in different origins (server and client hosted at two different host origins). The server responds with a header indicating that the requested resource can be served in different origin. Once this communication is established between the server and the client, the browser makes the APIs calls to access the resources and server responds accordingly. In the clientside, we have used three types of storage: local storage, redux and state variables to store data from the server. Local storage holds data in the browser and lives even after the browser is closed. Redux stores the data throughout the session and is removed/reloaded after page reload. State variables store values within each component level and are removed once the DOM removes the component from the browser.

Before we dive into the details of our RESTful APIs, let’s investigate the data models used in our Postgres server. Postgres Server is a relational database management system for persistent storage and ease of integration with Java Servlets using JDBC library. Figure 1 shows the Entity Relationship model used in our HumorMe! server.

Assumptions and constraints besides the cardinalities specified in the diagram are:

* All columns in tables are nullable except the primary keys
* All primary keys hold unique constraints and throw error when violated

The models presented in the Figure 1 above are represented by entities in SpringBoot. We used @Entity annotation in each model class that implicitly transforms the Java Classes into relational tables without requiring us to write the SQL for table creation (unless we are dealing with version control at schema level, which is not used in this implementation). We also used Java Persistence API (JPA) library, a widely used Object Relational Mapping (ORM) specification, to make queries such as insert, update, delete and select into the database. In what follows, we describe the relational tables generated using our data model classes in Java.

## **Table: users**

The table stores the information of users once they sign up in the platform. A unique identified of type Integer (Long in Java) is created automatically when an insert query in this table is issued. A unique constraint is added to the column “email” so that users with same email address are prevented from registering as users in the platform. The password column stores the hash value (using BCryptPasswordEncoder provided by SpringBoot Security library) of the raw password users used in their registration process.

## **Table: jokes**

The table stores information details of joke related posts from users. Here, text and labels are directly supplied by the users, and each joke is associated with a single user (one to one mapping). The table stores the summary of each label ratings (lame, dark, funny, and hilarious) inside column “ratings” as a string value and are transformed into key values when returning the data to the client. The column “totalComments” stores the total number of text reviews commented for the joke and is dependent on the number comments for the joke.

## **Table: comments**

The table stores detail information related to a text review commented on a particular joke. It has a one-to-one mapping with a user and a joke. In other words, a comment is only made by a single user for a single joke.

## **Table: ratings**

The table stores label rating information by a user to a joke. Therefore, it also has a one-to-one mapping with a user and a joke. A label rating replaces the existing “like or thumbs up or star-based” ratings we have seen in other existing platforms like Facebook and Reddit. Here, a user can rate a joke using one of the four rating labels “lame, funny, hilarious and dark”. This label is not the same thing as labels when posting a joke. In creating a joke, we add custom labels which is equivalent to saying “search tags” for the joke.

## **Table: user\_followers**

The table stores information about who is following whom. In other words, it stores two columns: “user\_id” and “follower\_id”. Here, “follower\_id” indicates the users who are following the given “user\_id”. This way, we can keep a track of the number of followers and the number of followings for each user. Followers indicate other users who are following a given user. Following indicates other users to whom a given user is following. Since this is a many-to-many relationship between the users, the same “user\_id” is allowed to be repetitive in the table.

# **Detailed Description of Components**

## **Account Registration and Login**

The table stores

## **View and Search Joke Posts**

The table stores

## **Post a Joke**

The table stores

## **Comment on a Joke**

The table stores

## **Rate a Joke**

The table stores

## **Delete a Joke**

The table stores

## **Visit User Profile**

The table stores

# **Conclusion**