EE 547 – Applied and Cloud Computing

Project Report

Yuqi Chen, Jian Dong, Bei Ming

Dec 13, 2022

Blog Application

Section 1. Objectives

Blog is a very common Internet application, many people like to share their life through blog, so we designed a blog platform for users to share their ideas, or life experience. Our blog platform is mainly divided into two pages, the home page and the blog page, and it has various functions. Users can log in to our blog page after completing registration. Here, this page is divided into three sections, namely post, profile, and blog, and the blog section has two parts: private and public. profile is used to display user personal information. The post section provides a platform for users to edit blogs. After editing, users can choose to post in the private window or public window in the blog section. It is worth mentioning that private means that only editors can see the corresponding blog, and public means that the blog will be displayed to all users. In addition, we have also implemented the function that different users can edit blogs together at the same time. In addition to the main functions mentioned above, we also implemented some interesting functions through the call API, including displaying when and where users edited blogs, translation and automatic joke generation, etc.

Section 2. Technologies and Architecture

Section 2.1 Technologies

We used the angular framework to build the front end, and then combined Graph QL and node.js to complete the server construction. Then, our database used mongoDB. In addition, we deployed the client to GCP and the server to AWS.

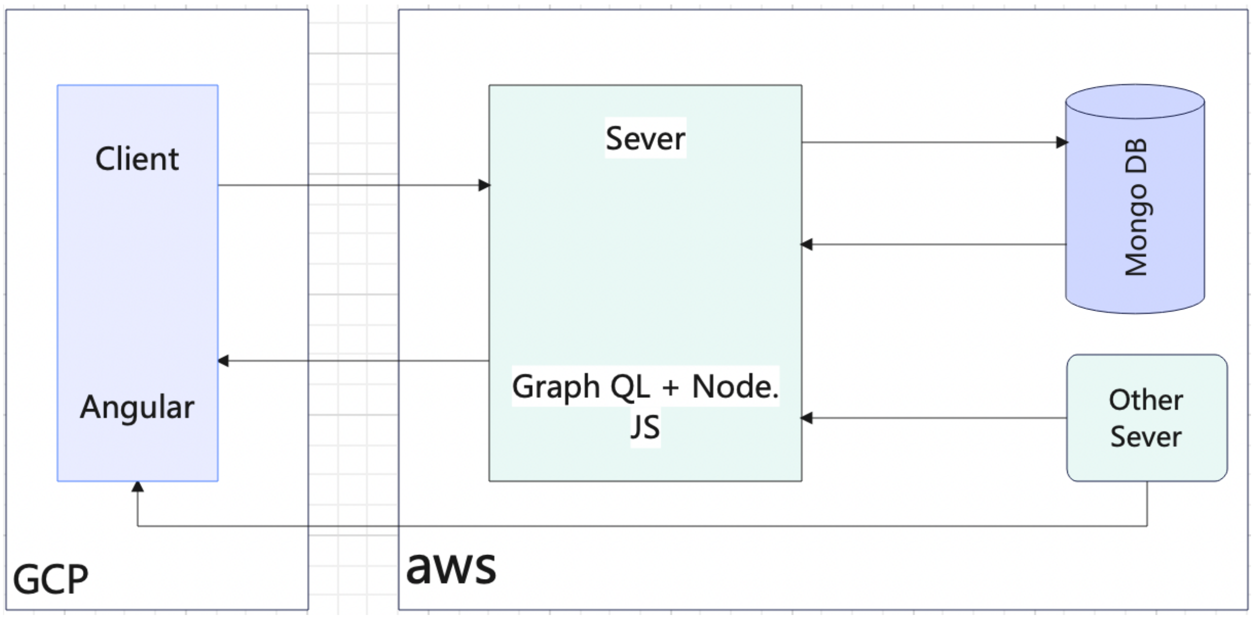


Figure1: Technology

Section 2.1.1 Graph QL & Node.js

Node.js is a framework, and a runtime (runtime) environment that runs JavaScript (JS) code in the background (or outside) of a browser. Node.js has the advantages of fast development speed and the ability to save data in local JSON in back-end development. GraphQL server is the interface between backend and application front-end, which allows developers to construct corresponding requests to pull data from multiple data sources with a single API call.

Section 2.1.2 Angular

Angular is an open source web application framework. We chose Angular because of its popularity (Angular is used by HBO, Apple, Microsoft, etc.), whose template is powerful and rich, and comes with extremely rich angular directives.

Section 2.1.3 GCP & AWS

Google Cloud Platform is a set of public cloud computing services provided by Google. Amazon Web Services offers reliable, scalable, and inexpensive cloud computing services. We choose these two cloud service platforms because they are the most widely used, most stable and comprehensive in the market.

Section 2.2 Architecture

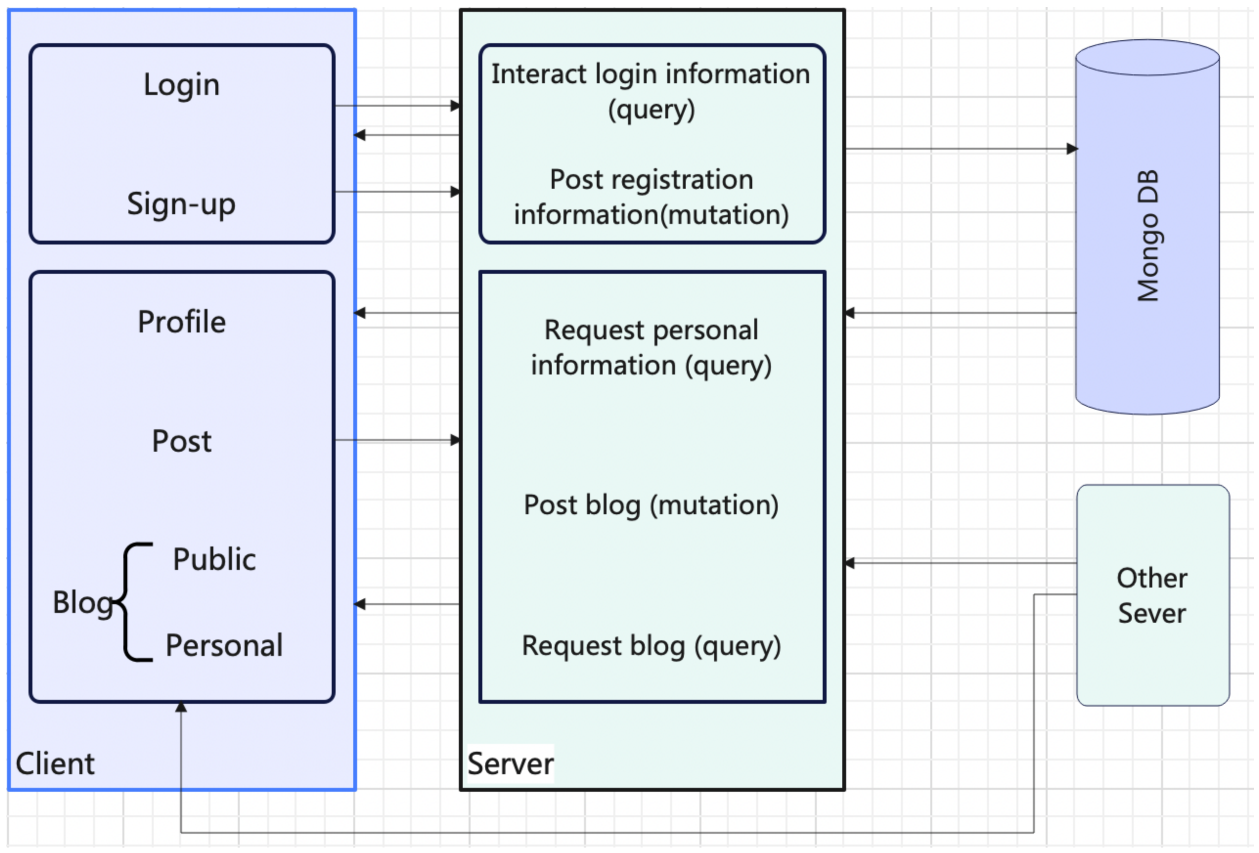


Figure 2: Architecture

Section 2.3 Compute Needs

Section 2.3.1 Backend Compute Needs

The Backend uses AWS as the server. We use an AWS EC2 instance. It is Ubuntu-focal-20.04 base. We use port 3000. The public ipv4 is 34.213.46.21. The public Ipv4 DNS is http://ec2-34-213-46-21.us-west-2.compute.amazonaws.com

Section 2.3.2 Frontend Compute Needs

Google shell is needed for an angular project to deploy on GCP. It is an interactive shell environment where we can manage our projects and resources.

Section 3. Implementation

Section 3.1 Backend Implementation

The backend could be divided into three parts, MongoDB, node.js and graphql. These would be introduced in this section. The deployment of the backend would also be introduced in section 3.1.4.

Section 3.1.1 MongoDB Implementation

Because of functional needs, we use two collections in this project, one is users and the other one is blogs.

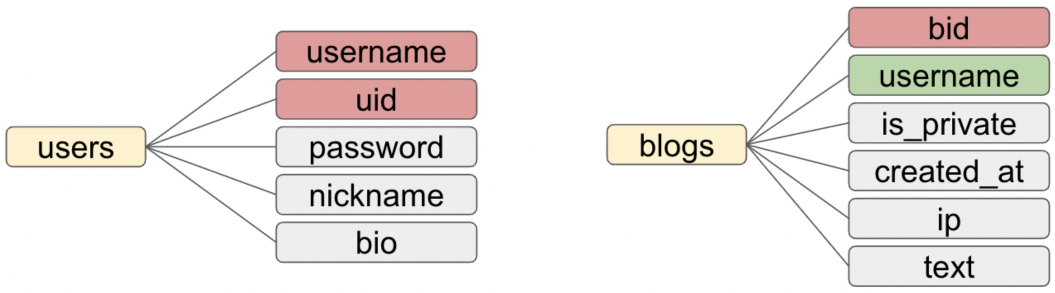


Figure 3. The structure of MongoDB

The collections and attributes are shown in figure 1. For the ‘users’ collection, it has five attributes which are username, uid, password, nickname and bio. Except uid is ObjectID, the other attributes are all strings. Username and uid in users are unique. For the ‘blogs’ collection, it has six attributes. Except bid is ObjectID, the other attributes are all strings. Bid in blogs is unique, and username is the foreign key.

Section 3.1.2 Graphql Implementation

We also use Graphql in the backend. Our graphql uses apollo server. Dataloader is used to cache data. Websocket is used for subscriptions. The graphql has six type elements. First, we would introduce three basic types.

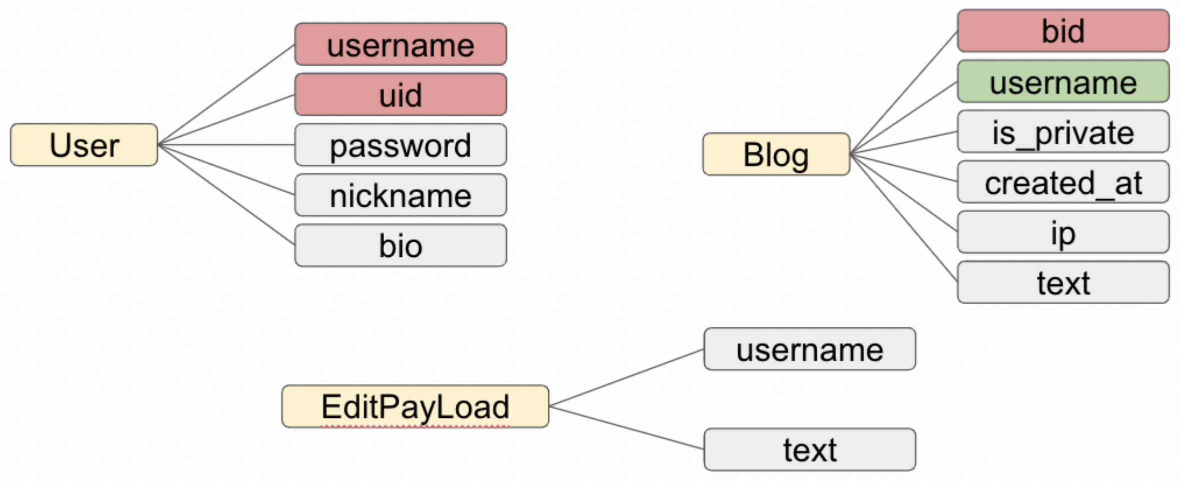


Figure 4. The part of graphql structure

As shown in figure 3. There are ‘User’, ‘Blog’ and ‘EditPayLoad’. For User and Blog, the attributes are the same as the attributes introduced in MongoDB. For EditPayLoad, it is used in blogEdit and the attributes are username and text. This part of the structure is shown in figure 3.

The next three parts are ‘query’, ‘mutation’, and subscription. The structure of these three types are shown in figure 4.

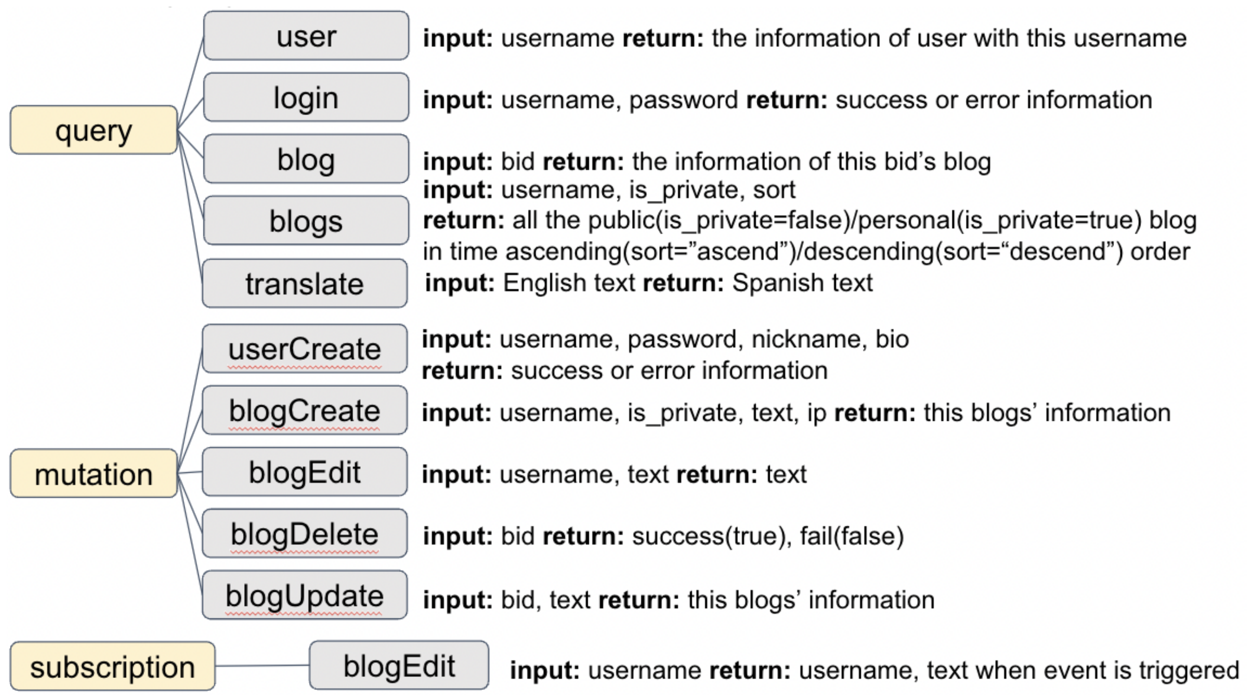

query
user
input: username return: the information of user with this username
login
input: username, password return: success or error information
blog
input: bid return: the information of this bid's blog
blogs
input: username, is_private, sort
return: all the public(is_private=false)/personal(is_private=true) blog in time ascending(sort="ascend")/descending(sort="descend") order
translate
input: English text return: Spanish text
mutation
userCreate
input: username, password, nickname, bio
return: success or error information
blogCreate
input: username, is_private, text, ip return: this blogs' information
blogEdit
input: username, text return: text
blogDelete
input: bid return: success(true), fail(false)
blogUpdate
input: bid, text return: this blogs' information
subscription
blogEdit
input: username return: username, text when event is triggered

Figure 5. The part of graphql structure

For 'query', there are five functions.

user is used to get user information using username. It would use username as key and find the element in users collection in MongoDB. This function is used by the getting user profile function of the client.

login is used to check whether a username matches the password. This function would find the element in users collection in MongoDB using username. If the password matches the username, it would return 'success'. It also returns error information when the username does not exist or the password does not match the username.

blog is used to get blogs' information using bids. It would use the data loader to get the information from the bid.

blogs are used to get blogs to display. It has multiple inputs. When 'is_private' is false, it would use is_private equal false to query the blogs collection in the database. When 'is_private' is true, it would use the username to query the blogs collection in the database. When sort is ‘ascend’, it would sort all the blogs in time ascending order. When sort is ‘descend’, it would sort all the blogs in descending order. It would be called when showing the public posts or personal posts in the client server.

Translate is used to translate English to Spanish. It would get text from the client and call an external api to get the Spanish translation. Finally pass it to the client.

For ‘mutation’, there are five functions.

userCreate can get sufficient information from the client and then make it a user structure, then insert it into the users collection of the database. Before that, it would find the user using username in users collection of the database. If a user exits, it would return the error information. If created successfully, it would return success.

blogCreate can get sufficient information from the client and then make it a blog structure, then insert it into the blogs collection of the database. It would return a blog block.

blogEdit is used to trigger the subscription, if it is called, the subscription would get the EditPayLoad block.

blogDelete is used to delete a post. It could get the bid from the client, using bid find and delete it from the blogs collection of the database. If deleting success it would return true, else it would return false.

blogUpdate is used to update a post. It could get sufficient information from the client, and update the blog in blogs collections of the database using bid.

For ‘subscription’, there is a function called blogEdit. The input of this function is username. When it is called, the server would ‘listen’ to the signal. When mutation blogEdit is called, it would get the EditPayLoad information. It would compare the input username and the username in EditPayLoad, if they are the same, it would return the EditPayLoad to the client and continue listening. If they are different, it would not return anything and continue listening.

Section 3.1.3 Nodejs Implementation

We mainly use express to route the address. We use app.use to get and post './graphql'. But we use Apollo server to listen to the port, because we use websocket protocol, so we need to use server to listen, not the app. When calling the external api, we also used 'axios' to get the response from the end point.

Section 3.1.4 Backend Deployment

We use Amazon Web Service(AWS) EC2 instance. So we follow the instructions of AWS. Creating an SSH connection, moving the local project to server and installing the dependency for it. Finally install MongoDB and open it. Then start the server and add the safety group. Create port 3000 as our safe inbound rules.
The public Ipv4 DNS is
'http://ec2-34-213-46-21.us-west-2.compute.amazonaws.com'.
The graphql api is
'http://ec2-34-213-46-21.us-west-2.compute.amazonaws.com:3000/graphql'

Section 3.2 Frontend Implementation

Section 3.2.1 Angular Implementation

As for the frontend, we used Angular framework to design the pages and interact with the backend. The primary purpose of Angular is to develop single-page applications so that it avoids refreshing the page frequently and effectively reduces the response time to give a better user experience. We used HTML/CSS to design the structure and styling of the pages aimed at rendering fairly good-looking pages to users. And worth to mention, Apollo-angular model and HttpClient module are two models that we used to interact with the server. Apollo-angular model is used to send query, mutation and subscription directives to the GraphQL server. And Http-client is used to request and retrieve data from external URLs.

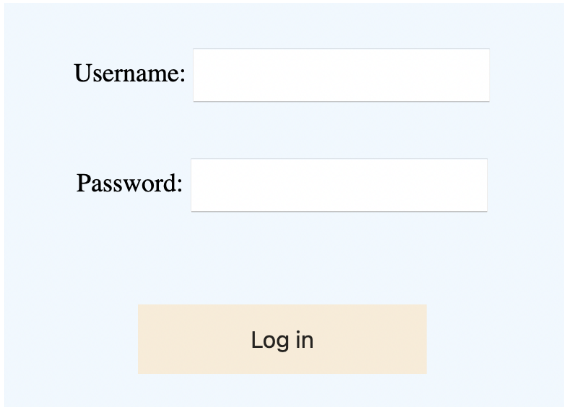


Figure 6: Login page

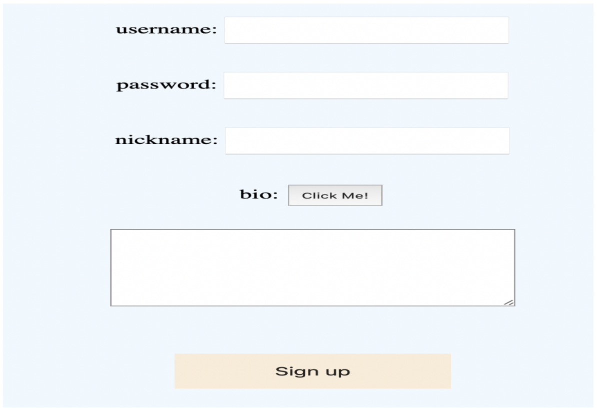


Figure 7: Signup page

At first, the user will see the main page where you can login and signup. On creating an account, the user needs to enter the username, password and nickname. And for the bio, the user can either write it on their own or simply click “Click Me!” button to automatically generate a joke. After clicking this button the HttpClient will take over and send a request to an external API and retrieve the response. Then after clicking the sign up button, the Apollo-client will send a query to GraphQL server to check if the username already exists. If the username exists, there’ll be a pop up alerting the user. If not, a new account has been created.

On the Login page, the user needs to enter the username and password, after clicking the login button, Apollo-client will send a query to GraphQL server to check if username exists and if the password matches the username. Then the user will successfully enter the application. First comes with the blogs page where the user can see the public blogs posted by all of the users.

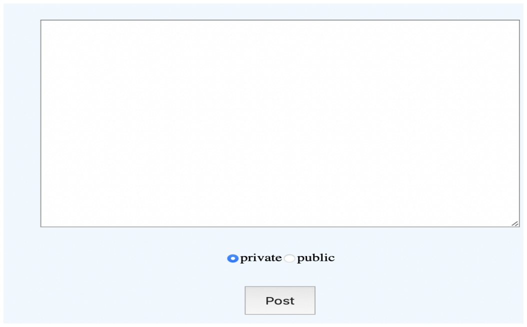
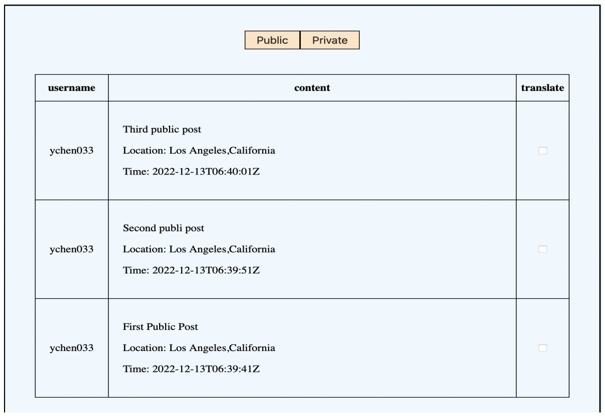


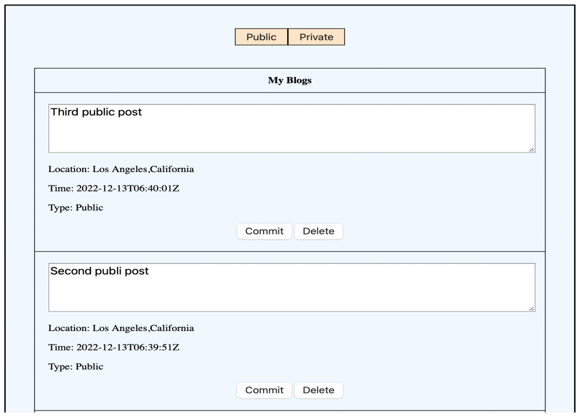
Figure 8: Post page

Let’s first make some posts to see how this will work. Now we are on the POST page, where the user can write their own blogs. One of the features is that we support co-edition of the blog by multiple users at the same time. In order to do that, the user can share their account with others and when multiple users are on the same page, they can start co-editing the blog. The way we do this is that when multiple users enter this page, the Apollo Client will send a subscription to the server to initialize a web socket session. Any change of the blog will notify the server to broadcast the up-to-date content to all of those who are in this page with the same account. And the user can either choose to make the blog public or private. Private means the user can only see this blog by their own account. After clicking the “POST” button the client will first detect the IP address and implicitly bind the current address to the blog. Then the Apollo Client will send a mutation to the server. For now, a blog has been successfully created.



username	content	translate
ychen033	Third public post Location: Los Angeles,California Time: 2022-12-13T06:40:01Z	☐
ychen033	Second publi post Location: Los Angeles,California Time: 2022-12-13T06:39:51Z	☐
ychen033	First Public Post Location: Los Angeles,California Time: 2022-12-13T06:39:41Z	☐

Figure 9: Public posts



My Blogs

Third public post

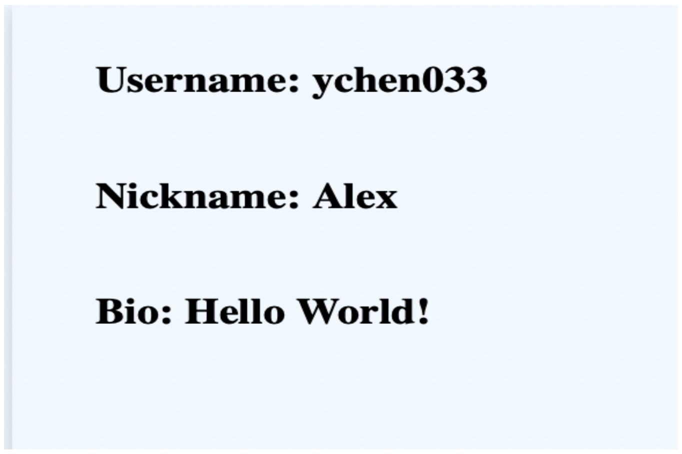
Location: Los Angeles,California
Time: 2022-12-13T06:40:01Z
Type: Public

Second publi post

Location: Los Angeles,California
Time: 2022-12-13T06:39:51Z
Type: Public

Figure 10: Private posts

On the home page, each row shows the related information of each blog like username, the location and the time. We also added a feature that the user can translate the content from English to Spanish. After clicking the checkbox, Apollo Client will send a query to the GraphQL server, and the server will send a GET request to an external URL to request and retrieve the translation. Then forward the response to the client. The translation will be shown at the bottom of each row. The user can either check the checkbox to render the translation or uncheck it to hide the translation.


Username: ycchen033

Nickname: Alex

Bio: Hello World!

Figure 11: Profile page

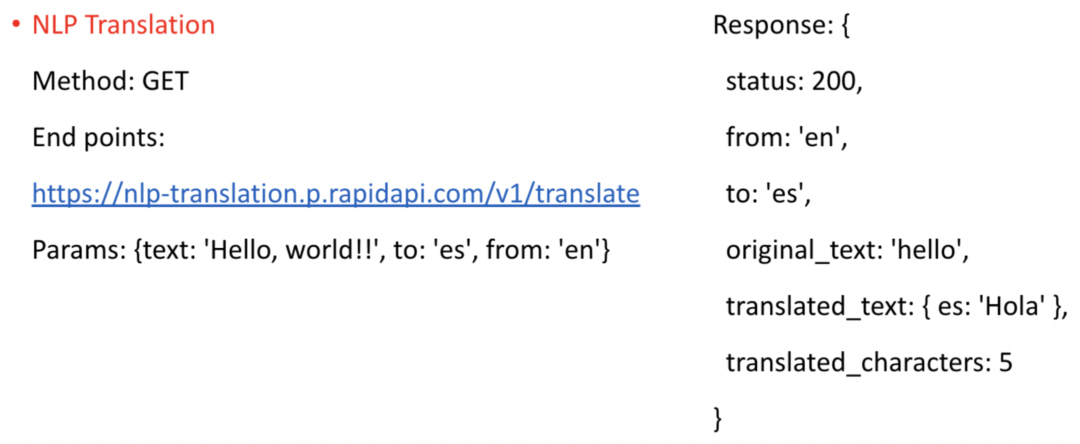
On personal blogs are all of the blogs that the user posted before. The user can edit their blogs and then click commit to make changes to any of the blogs. The user can also click the delete button to delete the blog. Both actions will trigger Apollo Client to send a mutation to the GraphQL server. On the personal information page, the user can see the information that they entered on the signup page.

Section 3.2.2 Frontend Deployment

We deployed our Frontend angular application on GCP. First create a project on the GCP console and then create a bucket, where we upload a folder generated by ng build and an app.yaml file. Our website can be found at http://blogapp-371504.wl.r.appspot.com/login.

Section 4. Datasources and Additional APIs

We use three external api, there are NLP Translation, Official Joke API and IP info API. The details are shown in figure 12, figure 13 and figure 14.



• NLP Translation

Method: GET

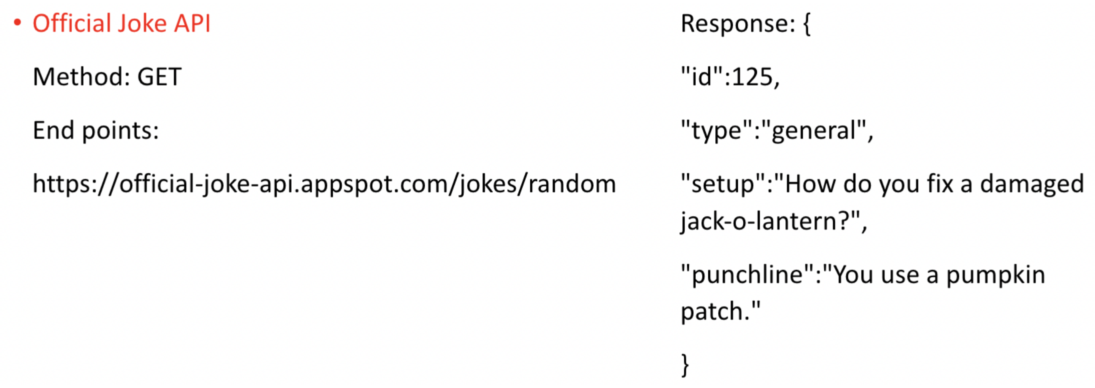
End points:

https://nlp-translation.p.rapidapi.com/v1/translate

Params: {text: 'Hello, world!!', to: 'es', from: 'en'}

```
Response: {
  status: 200,
  from: 'en',
  to: 'es',
  original_text: 'hello',
  translated_text: { es: 'Hola' },
  translated_characters: 5
}
```

Figure 12. The details of NLP translation API



• Official Joke API

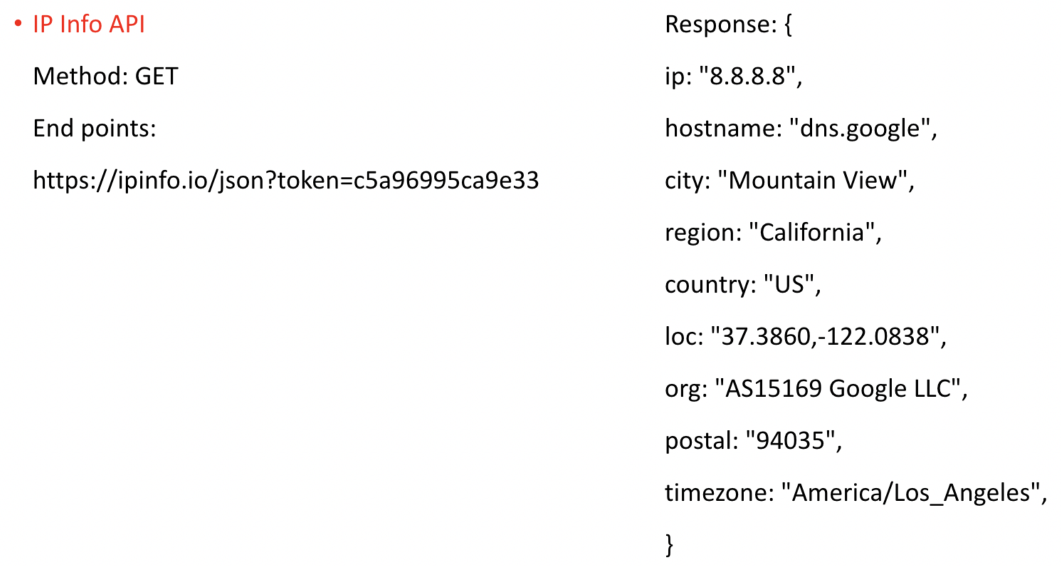
Method: GET

End points:

https://official-joke-api.appspot.com/jokes/random

```
Response: {
"id":125,
"type":"general",
"setup":"How do you fix a damaged
jack-o-lantern?",
"punchline":"You use a pumpkin
patch."
}
```

Figure 13. The details of Official Joke API



• IP Info API

Method: GET

End points:

https://ipinfo.io/json?token=c5a96995ca9e33

```
Response: {
ip: "8.8.8.8",
hostname: "dns.google",
city: "Mountain View",
region: "California",
country: "US",
loc: "37.3860,-122.0838",
org: "AS15169 Google LLC",
postal: "94035",
timezone: "America/Los_Angeles",
}
```

Figure 14. The details of IP Info API

We have called NLP translation when querying ‘translation’. The Official Joke API is called when the user wants a random bio introduction. When creating or editing the blogs, the IP Info would be called to get the address.

Section 5. Conclusions

- Successfully built backend using graphql, node.js, websocket, apollo server and complete all the backend functions
- Successfully built frontend using angular, apollo angular, websocket, rendered the right content on each page and realized proposed functions
- Successfully deployed backend on AWS and frontend on GCP

Section 6. References and Libraries

Section 6.1 Backend references and Libraries

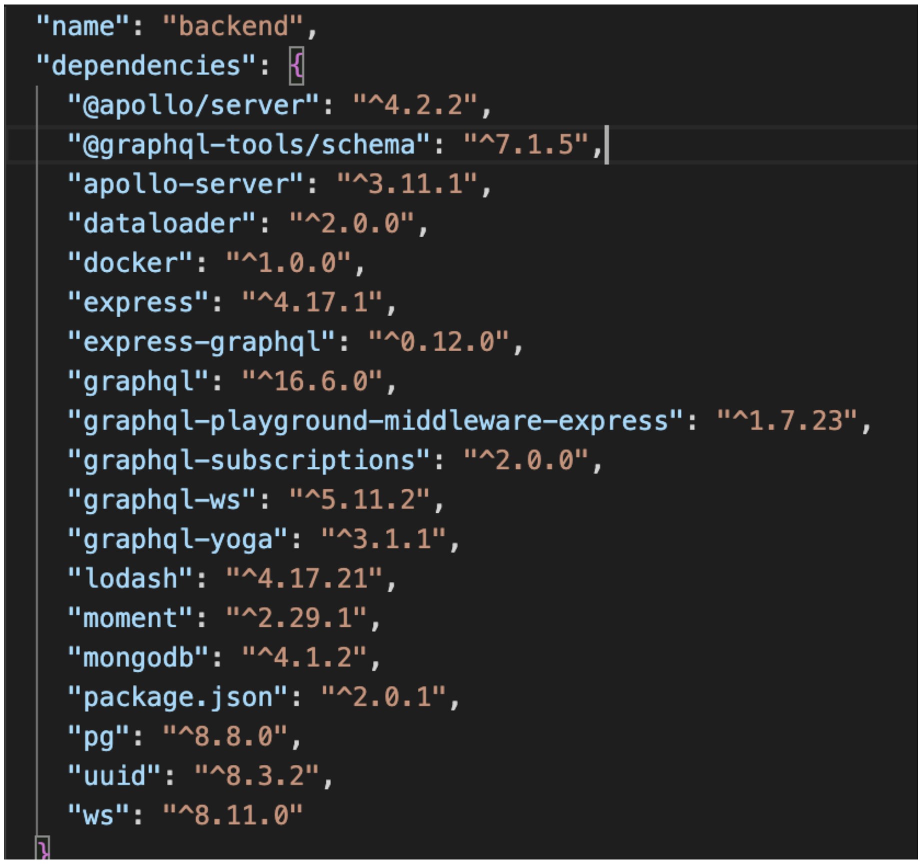
The reference and libraries we used in the backend are shown in this section.

Section 6.1.1 Backend references

- Apollo docs: Subscriptions in Apollo Server
  https://www.apollographql.com/docs/apollo-server/data/subscriptions/
- Apollo docs: API Reference: Apollo Server
  https://www.apollographql.com/docs/apollo-server/api/apollo-server/#includestacktraceinerrorresponses
- NLP translation
  https://rapidapi.com/gofitech/api/nlp-translation
- Graphql docs
  https://graphql.org/learn/
- Nodejs docs
  https://nodejs.org/en/docs/

Section 6.1.2 Backend Library and Dependency

The dependency library of the backend is shown in figure 7.



```
"name": "backend",
"dependencies": {
  "@apollo/server": "^4.2.2",
  "@graphql-tools/schema": "^7.1.5",
  "apollo-server": "^3.11.1",
  "dataloader": "^2.0.0",
  "docker": "^1.0.0",
  "express": "^4.17.1",
  "express-graphql": "^0.12.0",
  "graphql": "^16.6.0",
  "graphql-playground-middleware-express": "^1.7.23",
  "graphql-subscriptions": "^2.0.0",
  "graphql-ws": "^5.11.2",
  "graphql-yoga": "^3.1.1",
  "lodash": "^4.17.21",
  "moment": "^2.29.1",
  "mongodb": "^4.1.2",
  "package.json": "^2.0.1",
  "pg": "^8.8.0",
  "uuid": "^8.3.2",
  "ws": "^8.11.0"
}
```

Figure 15. The dependency library of backend

Section 6.2 Frontend references and Libraries

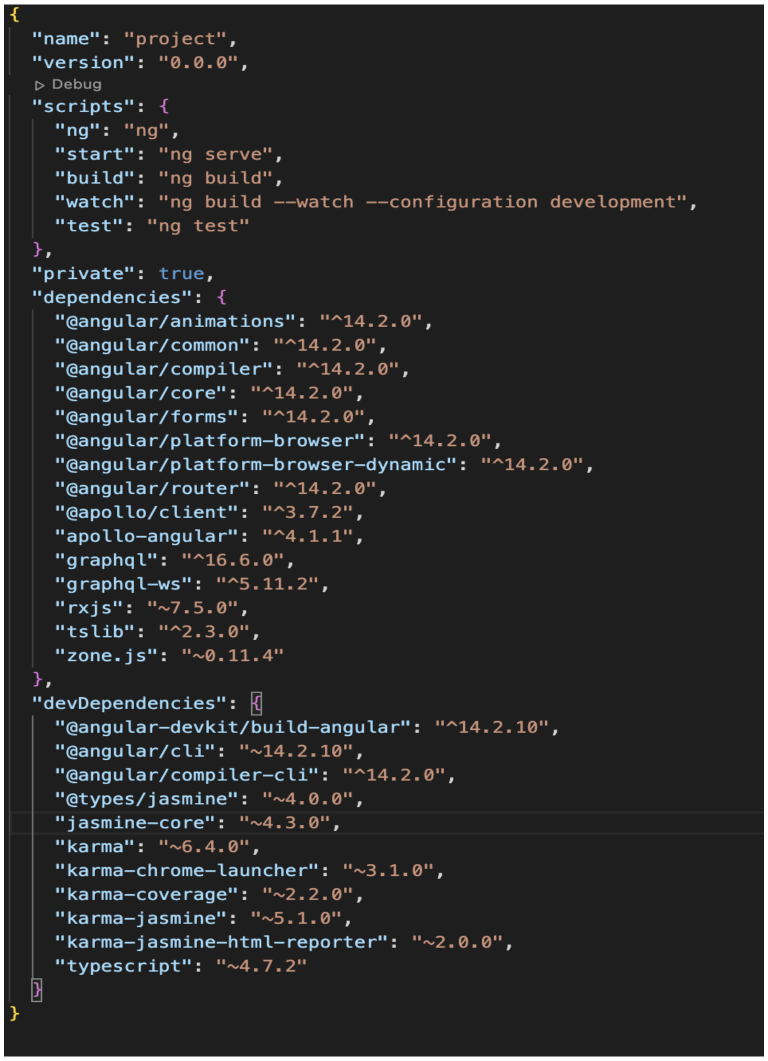
The reference and libraries we used in the frontend are shown in this section.

Section 6.2.1 Frontend references

- Angular 13
  https://angular.io/docs
- Apollo-angular
  https://the-guild.dev/graphql/apollo-angular/docs

Section 6.2.2 Frontend Library and Dependency

The dependency library of the frontend is shown in figure 16.



```
{
  "name": "project",
  "version": "0.0.0",
  ▷ Debug
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "watch": "ng build --watch --configuration development",
    "test": "ng test"
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "^14.2.0",
    "@angular/common": "^14.2.0",
    "@angular/compiler": "^14.2.0",
    "@angular/core": "^14.2.0",
    "@angular/forms": "^14.2.0",
    "@angular/platform-browser": "^14.2.0",
    "@angular/platform-browser-dynamic": "^14.2.0",
    "@angular/router": "^14.2.0",
    "@apollo/client": "^3.7.2",
    "apollo-angular": "^4.1.1",
    "graphql": "^16.6.0",
    "graphql-ws": "^5.11.2",
    "rxjs": "~7.5.0",
    "tslib": "^2.3.0",
    "zone.js": "~0.11.4"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "^14.2.10",
    "@angular/cli": "~14.2.10",
    "@angular/compiler-cli": "^14.2.0",
    "@types/jasmine": "~4.0.0",
    "jasmine-core": "~4.3.0",
    "karma": "~6.4.0",
    "karma-chrome-launcher": "~3.1.0",
    "karma-coverage": "~2.2.0",
    "karma-jasmine": "~5.1.0",
    "karma-jasmine-html-reporter": "~2.0.0",
    "typescript": "~4.7.2"
  }
}
```

Figure 16. The dependency library of backend

Appendix

- Graphql schema

```
type Query {
   user(username: String!): User

   login(username:String!, password: String!): String

   blog(bid: ID!): Blog

   blogs
       (
       is_private: Boolean
       sort: String
       username: String
       ): [Blog]!

   translate(text: String): String
}

type Mutation {
   userCreate(
       username: String!
       password: String!
       nickname: String!
       bio: String!
   ): String
   blogCreate(
       blogInput: BlogInput!
   ):Blog
   blogEdit(
       username: String
       text: String
   ):String
   blogUpdate(
       bid: ID!
       text: String!
   ):Blog
   blogDelete(
       bid: ID!
   ):Boolean
}
```

```
type Subscription{
   blogCreate: BlogPayLoad
   blogEdit(username: String): EidtPayLoad
   blogDelete: BlogPayLoad
}

type Blog{
   bid: ID!
   username: String!
   created_at: String
   text: String
   is_private: Boolean
   ip: String
}

input BlogInput{
   username: String!
   is_private: Boolean!
   text: String!
   ip: String
}

type BlogPayLoad{
   bid: ID
   username: String
   text: String
}

type User {
   username: String!
   password: String!
   uid: ID!
   nickname: String!
   bio: String!
}

type EidtPayLoad{
   username: String
   text: String
}
```

- Graphql resolvers

```
Query: {
     user: async (_, { username }, context) => {
       let user = await context.db._mongoDb.collection('users').find({username:
username}).sort().toArray();
       if(user.length != 0){
         return context.loaders.user.load(user[0]._id.toString());
       }else{
         throw new GraphQLError("username is not exist!!");
       }
     },
     login: async (_, { username, password }, context) => {
       let user = await context.db._mongoDb.collection('users').find({username:
username}).sort().toArray();
       if(user.length != 0){
         if(user[0].password === password){
           return "success";
         }else{
           return "password";
         }
       }else{
         return "username";
       }
     },
     blog: (_, {bid}, context) => {
         return context.loaders.blogs.load(bid);
     },
     blogs: async (_, { is_private=false, sort='descend', username=null},
context) => {
         let blogs = new Array();
         let sorted_type = -1;
         if(sort === 'ascend'){
             sorted_type = 1;
         }
         if(is_private){
             blogs = await
context.db._mongoDb.collection('blogs').find({username:
username}).sort({created_at: sorted_type}).toArray();
         }else{
             blogs = await
context.db._mongoDb.collection('blogs').find({is_private:
false}).sort({created_at: sorted_type}).toArray();
         }
         let n = blogs.length;
         let i = 0;
         let write_data = new Array();
         while(i < n){

write_data.push(context.loaders.blogs.load((blogs[i]._id.toString())));
             i++;
         }
         return write_data;
     },
     translate: async (_, {text}, context) =>{
       let options = {
         method: 'GET',
         url: 'https://nlp-translation.p.rapidapi.com/v1/translate',
         params: {text: text, to: 'es', from: 'en'},
         headers: {
           'X-RapidAPI-Key':
'eadb410edcmshca6c29a90a093dap153f4fjsn59e59b67d8f1',
           'X-RapidAPI-Host': 'nlp-translation.p.rapidapi.com'
         }
       };
       let return_value = "";
       await axios.request(options).then(function (response) {
         return_value = response.data.translated_text.es;
       }).catch(function (error) {
         console.error(error);
       });
       return return_value;
     },
   },
   Mutation:{
     userCreate: async (_, userInput, context) => {
       let username = userInput.username;
       let password = userInput.password;
       let bio = userInput.bio;
       let nickname = userInput.nickname;
       let newObjectId = ObjectId(global_uid);
       global_uid ++;
       let single_user = {
           _id: newObjectId,
           username: username,
           password: password,
           bio: bio,
           nickname: nickname
       };
       let check = await
context.db._mongoDb.collection('users').find({username:
username}).sort().toArray();
       if(check.length === 0){
         await context.db._mongoDb.collection("users").insertOne(single_user);
         return "success";
       }else{
         return "username";
       }
     },
     blogCreate: async (_, {blogInput}, context) => {
         let text = blogInput.text;
         let username = blogInput.username;
         let is_private = blogInput.is_private;
         let newObjectId = ObjectId(global_bid);
         let ip = blogInput.ip;
         global_bid ++;
         let time = new Date().toISOString();
         let single_blog = {
             _id: newObjectId,
             username: username,
             created_at: time,
             is_private: is_private,
             text: text,
             ip:ip
         };
         await context.db._mongoDb.collection("blogs").insertOne(single_blog);
         pubsub.publish('Blog_Create', {
           blogCreate: {
               bid: newObjectId.toString(),
               username: username.toString(),
               text: text
           }
         });
         return context.loaders.blogs.load(newObjectId.toString());
     },
     blogEdit: async (_, {username, text}, context) => {
       pubsub.publish('Blog_Edit', {
           blogEdit: {
               username: username.toString(),
               text: text.toString()
           }
       })
       return text;
     },
     blogUpdate: async (_, {bid, text}, context) => {
       let id = new ObjectId(bid);
       let new_text = text;
       let blog = await context.db._mongoDb.collection('blogs').find({_id:
id}).toArray();
       let time = new Date().toISOString();
       blog[0].text = new_text;
       blog[0].created_at = time
       await context.db._mongoDb.collection('blogs').updateOne({_id: id},
{$set:blog[0]});
       context.loaders.blogs.clear(id.toString());
       return context.loaders.blogs.load(id.toString());
     },
     blogDelete: async (_, {bid}, context) => {
         let id = new ObjectId(bid);
         let data = await context.db._mongoDb.collection('blogs').find({_id:
id}).toArray();
         let ans = await
context.db._mongoDb.collection('blogs').deleteOne({_id: id});
         if (ans.deletedCount != 0){
             context.loaders.blogs.clear(id.toString());
             return true;
         }
         else{
             return false;
         }
     }
   },
   Subscription: {
     blogCreate: {
       subscribe: () => pubsub.asyncIterator(['Blog_Create']),
     },
     blogEdit: {
       subscribe: withFilter(() => pubsub.asyncIterator(['Blog_Edit']),
       (EditPayLoad, {username}) => {
         return EditPayLoad.blogEdit.username === username;
       }
```

```
    ),
  },
  blogDelete: {
    subscribe: () => pubsub.asyncIterator(['Blog_Delete']),
  }
},
Blog: {
  bid: async ({bid}, _, context) => {
      return await context.loaders.blogs.load(bid)
      .then(({bid}) => bid);
  },
  username: async ({bid}, _, context) => {
      return await context.loaders.blogs.load(bid)
      .then(({username}) => username);
  },
  is_private: async ({bid}, _, context) => {
      return await context.loaders.blogs.load(bid)
      .then(({is_private}) => is_private);
  },
  text: async ({bid}, _, context) => {
      return await context.loaders.blogs.load(bid)
      .then(({text}) => text);
  },
  created_at: async ({bid}, _, context) => {
      return await context.loaders.blogs.load(bid)
      .then(({created_at}) => created_at);
  },
  ip: async ({bid}, _, context) => {
    return await context.loaders.blogs.load(bid)
    .then(({ip}) => ip);
  },
},
User: {
  uid: async ({uid}, _, context) => {
      return await context.loaders.user.load(uid)
      .then(({uid}) => uid);
  },
  username: async ({uid}, _, context) => {
      return await context.loaders.user.load(uid)
      .then(({username}) => username);
  },
  password: async ({uid}, _, context) => {
      return await context.loaders.user.load(uid)
      .then(({password}) => password);
  },
  nickname: async ({uid}, _, context) => {
      return await context.loaders.user.load(uid)
      .then(({nickname}) => nickname);
  },
  bio: async ({uid}, _, context) => {
      return await context.loaders.user.load(uid)
      .then(({bio}) => bio);
  },
}
```