QuizApp - documentation

Team:
- Filip Juza
- Bartosz Włodarski
- Karol Zając

Description:

App allows to:
- create account
- add own questions and share them among classes,
- join class,
- create and participate quizzes,
- compare your results with others thanks to highscores

Tech stack:

- Database: MySql
- Frontend: Angular.js
- Backend: Node.js with Express.js
- Docker

How to run:

The easiest way to run the app is using docker. If it's installed on the machine, simply execute:
docker-compose up --build -d
in the main folder, that is *BD*.
Then wait for docker to create the whole structure. After a while the app will be available at http://localhost:4200/.
To check if everything was set up correctly use command :
docker-compose logs
App is ready if all of these logs are there :
- front-container | ** Angular Live Development Server is listening on 0.0.0.0:4200, open your browser on http://localhost:4200/ **
- front-container | ✔ Compiled successfully.
- back-container | wait-for-it.sh: waiting 40 seconds for database:3306
- back-container | wait-for-it.sh: database:3306 is available after X seconds
- back-container | App is listening on 3000.

To shut down the app, simply use :
docker-compose down

The other option is to run angular app and express.js server separately.
To do so, do the following :

1. run ***ng serve*** command in ‘quiz-app’ folder.
2. on the other console, run ***node app.js*** in ‘Backend’ folder.

NOTE : to run the app like this you need to change the data used to connect with database in /Backend/dbConnection.js file

Note that the app is not complete yet. To explore how it works login with user data:
Email-address *: adres@email.com*
Password *: password*

After logging you can do one of the things described below:
-create a question for chosen category,
-create a quiz with a given number of questions,
-join the quiz, if there’s any available,
-view statistics of quizes that you cannot take part in

Structure:

Project is divided into 3 main parts :

- quiz-app - This folder contains all necessery files to run angular app. Each component has its own, appropriate name. There are also 2 user-defined services :
    - DbService - service that is used to call all http request with the usage of HttpClient
    - DataService - service that is used to send data between unrelated components

- Backend - This folder contains all .js files that create the REST API. Files that describe particular routes are in the routes folder. Each route-routine has its own comment that describes what it does and how it’s used. The server is launched via *app.js* file. File dbConnection.js is used to establish connection with database. File *run-for-it.sh* is additional - it’s used to setup app in docker environment correctly.

- Database - This folder contains the code for database files:
    - *stored_procedures.sql* contains every procedure that database uses(more complicated procedures are described with a comment),
    - *tables.sql* contains code for creation of each table in the database.

  Folder named ‘init_scripts’ is used for deployment process - it contains a file *init.sql* that is used to create and fill up the database using Docker.

How it works:

The basic flow of each part of the application looks like this:
show GUI -> (some event occured) -> notify DbService with appropriate data -> call httpRequest -> SELECT/INSERT/ CALL PROCEDURE executed by routes -> get and send results -> show GUI

Each route-routine gets the data from the app in one of three different ways:

1. By parameters that were past in the URL, for example :
   http://localhost:4200/users/:email/:password
   email and *password* are the parameters that are used by the route.

2. By the object passed as a body of the request. Below is an example of code that uses that data passed by the body to execute given query on database :
   ```
   connection.query('CALL joinClass(?,?)', [req.body.userID, req.body.classCode]
   ```
   Data is then sent in the following way :
   ```
   this.http.post(this.urlClasses, {userID : this.getUser().UserID, classCode : code})
   ```

3. By the query provided in the url. For example :
   ```
   connection.query('CALL quizesInfo(?,?)',[parseInt(req.query.id), parseInt(req.query.count) ]
   ```
   Data is then sent in the following way:
   ```
   this.http.get(`${this.urlQuizes}/?id=${classID}&count=${count}`);
   ```

Complete SELECT statement is sometimes used instead of just a CALL to a database procedure. For example :

```
connection.query('SELECT * FROM User WHERE Email = ? AND Password = ?',[email,password]
```

Each question mark '?' is a place where data should be set. In the above query result query looks like this :

```
SELECT * FROM User WHERE Email = email AND Password = password
```

Where *email* and *password* are the parameters.
If there's not any error, each route-routine returns a json containing the data fetched from database as follows :

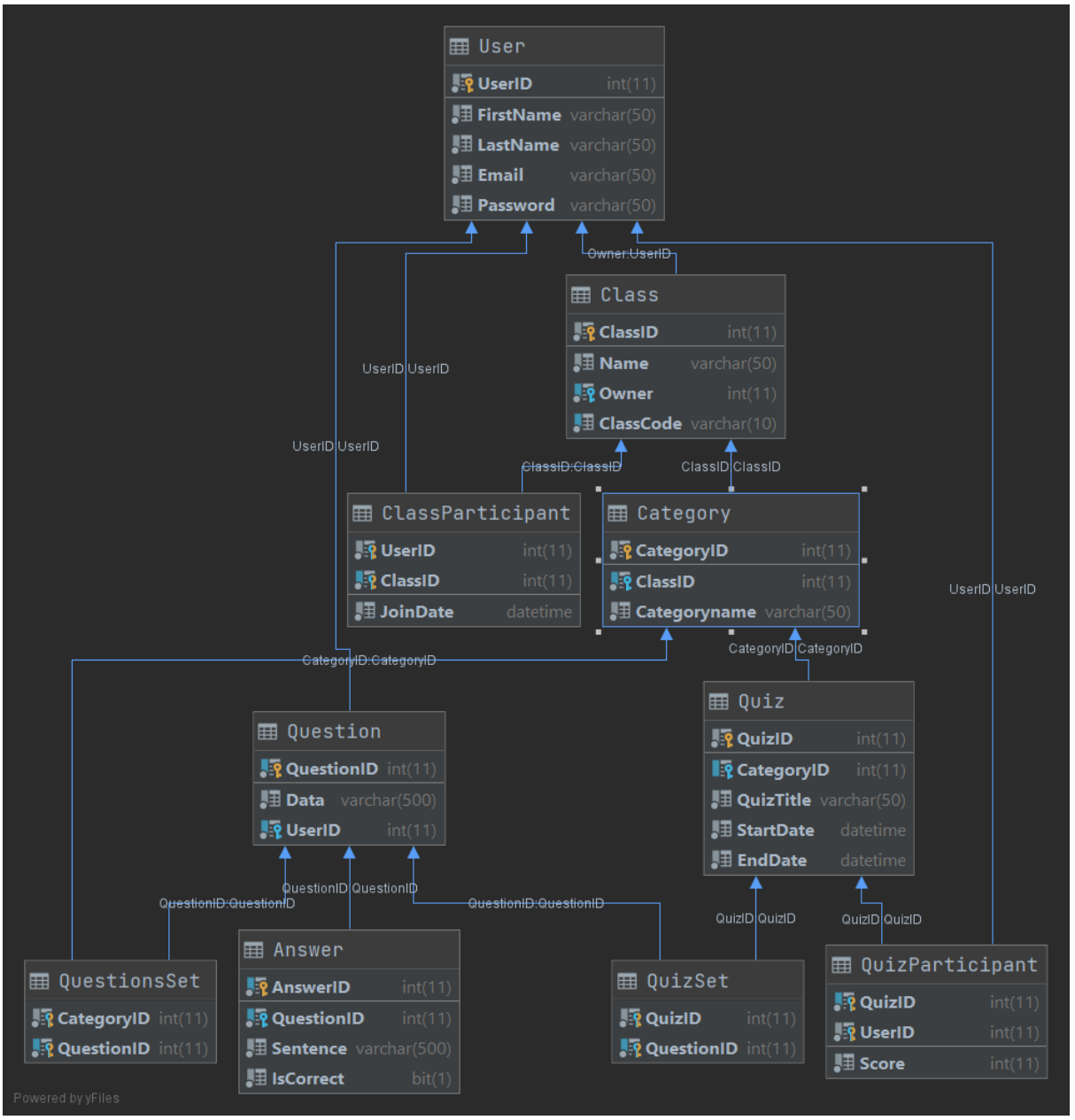
```
 res.status(200).json(results);
```

Where *results* is what database query returned.

If there isn't any data returned by the SQL statement, result status of the request is still 200, which means everything is okay. We decided that we shouldn't mark it as a server error, as it is not - server responded as it should. It just returns nothing - data wasn't found for given SQL statement.

Database:

1. Schema:



- User - basic information about user
- Class - has its name, owner and classCode which is used to join it
- Category - is owned by some class, e.g. class called Math has categories: geometry, algebra, calculus. Category has own pool of questions which can be used to create quiz
- Question - is owned by some user. Data means that question.
- Answer - is connected with question and can be correct or incorrect
- QuestionSet - connects questions with categories (many to many)
- Quiz - has its own title, date when it starts and ends and to which category belongs.
- QuizSet - connects questions with quizzes (many to many)
- QuizParticipant - contains results of quizzes solutions (used for highscores)