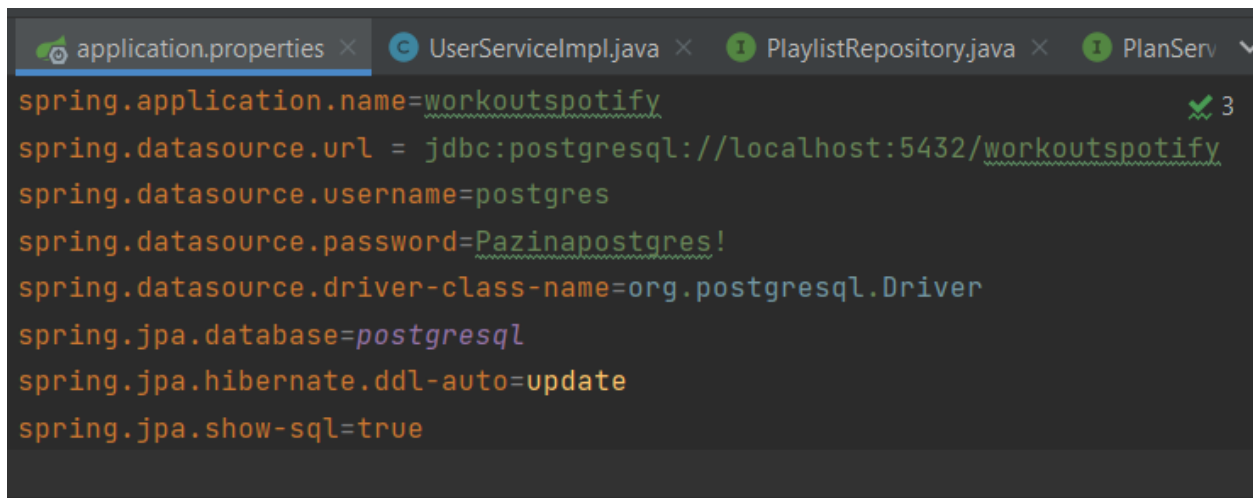


Nikola's Documentation

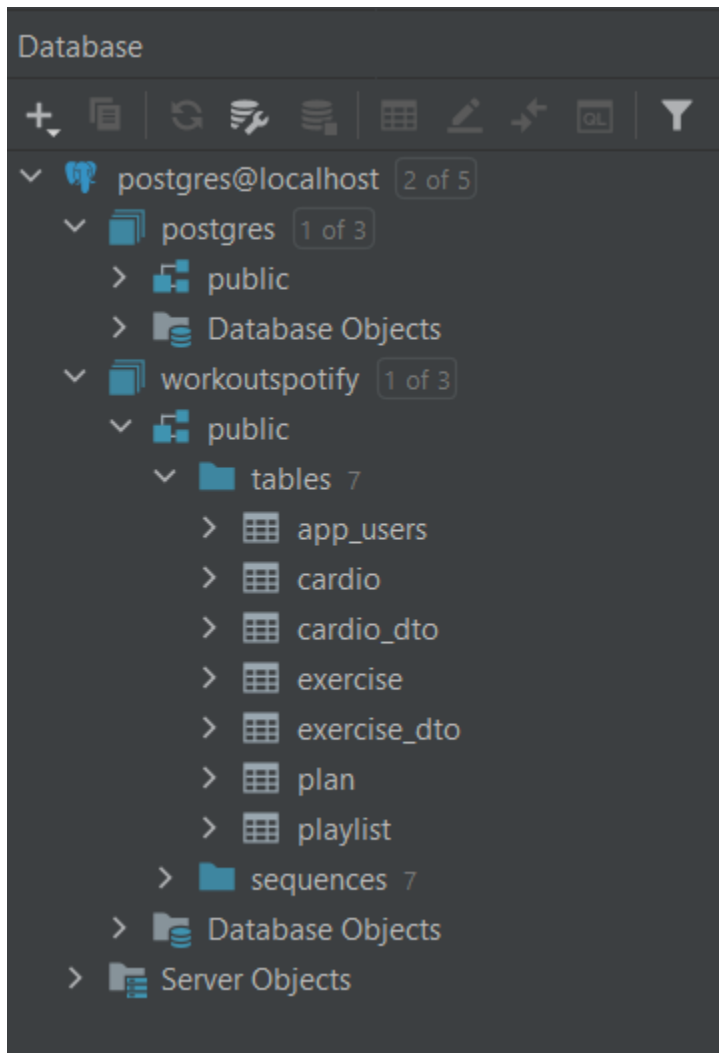
Greetings to whoever is reading this, I am Nikola Kamchev, one of the members of the team, who is responsible for creating this project. My role during this project was to create the backend application, setup the database (including all the tables) and connect the backend application with the frontend application. In this documentation, I'll explain how I setup all of the necessary components to work with, how I created the backend application and how I established the connections between the backend and frontend application.

Firstly, I want to cover how I setup the database. The database I chose to use is PostgreSQL, which was very easy to setup. I established a connection with my backend application via IntelliJ and changed the application properties of the backend app (picture 1), so whatever I created in the backend application, it directly updates in the schema "workoutspotify" (picture 2) and I can see what changes happen in the schema when I'm running the terminal.



```
application.properties x UserServiceImpl.java x PlaylistRepository.java x PlanServ
spring.application.name=workoutspotify ✓ 3
spring.datasource.url = jdbc:postgresql://localhost:5432/workoutspotify
spring.datasource.username=postgres
spring.datasource.password=Pazinapostgres!
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.database=postgresql
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Picture 1. Application properties



Picture 2. Postgresql database

goal (to determine for what purpose the exercise is used for), sets, minimum reps and maximum reps.

The exercises can be added to the database through a post method from the exercise controller (picture 3), which creates the exercise in the exercise service module and saves it in the table "exercise" through the exercise repository (picture 4). I used Postman to create the exercises myself and to use them in order to create complete workouts (picture 5).

```
@PostMapping(value="/create")
public String createExercise(
    @RequestBody Exercise exercise
){
    this.exerciseService.create(exercise.getName(),exercise.getType().toString(), exercise.getTargetMuscles(), exercise.g
    return "Success!";
}
```

Picture 3. Controller post method

Next, I want to cover how I created the backend application. The application uses the MVC pattern, excluding the views (those are the frontend pages made in React). Since the application is divided into 2 components, the first one being the workout generator and the second one being the playlist generator, I created a backend application, which can manipulate both components. But I will cover the application in 3 different segments, the workout segment, the playlist segment and the user segment, which includes the first 2 segments.

1. Workouts

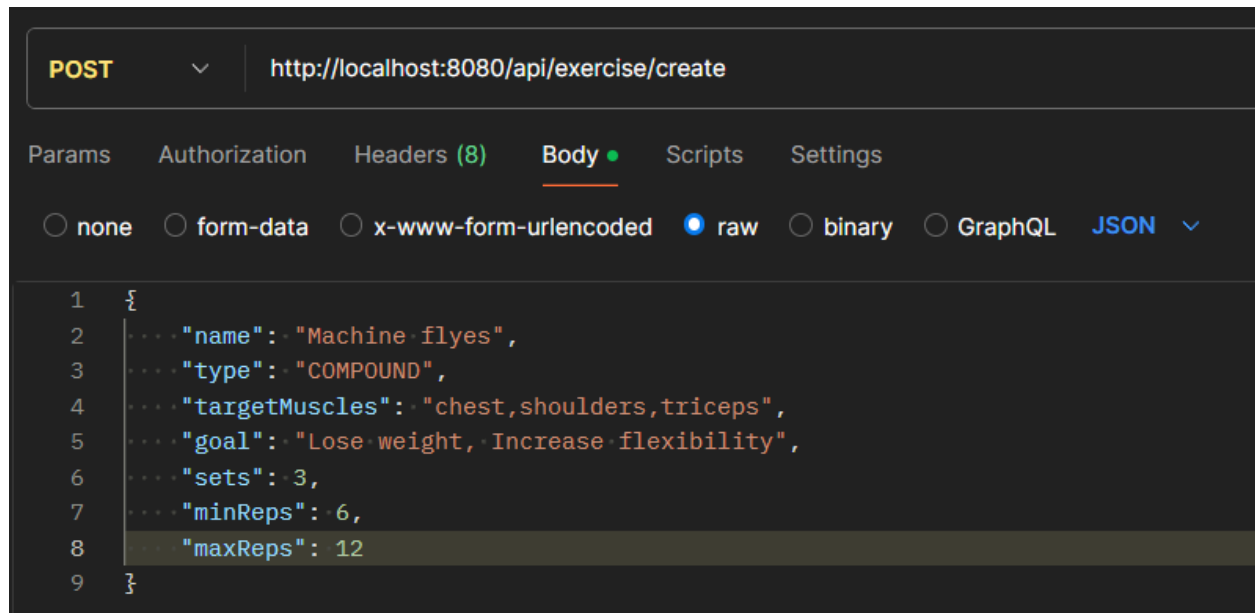
The workouts generated using this app consist of 2 lists of elements: workout exercises and cardio exercises. The workout exercises are entities called "Exercise", which contain id, name, type (compound/isolation), target muscles,

```

@Override
public Exercise create(String name, String type, String targetMuscles, String goal, int sets, int minReps, int maxReps) {
    Exercise exercise = new Exercise(name, type, targetMuscles, goal, sets, minReps, maxReps);
    return this.exerciseRepository.save(exercise);
}

```

Picture 4. Service function



Picture 5. Postman example

Same thing goes for the cardio exercises, called “Cardio”. They contain an id, name, goal variable and a time variable (to determine if the cardio exercise is used before or after the training). They can also be added to the database via post method from the cardio controller, which goes through the cardio service, where it’s created and saved in the database through the cardio repository. I also added the cardio exercises through Postman as well.

All of the table values I used to work with this program can be found in the 2 csv files located in the specified folder called “tables”.

Now there are 2 additional entities called ExerciseDto and CardioDto, these entities are the ones that the user receives when he finishes the workout survey. ExerciseDto contains the name, type, targetMuscles and the sets values coming from the Exercise entity with additional reps value (determined in the determineReps function in the ExerciseMapper class, picture 6) and weight category (light/moderate/heavy, also determined in the determineWeight function in the ExerciseMapper class, picture 7).

```
1 usage
public int determineReps(Exercise exercise, String currentBody, String targetBody){
    int reps = 0;
    if(
        (currentBody.equals("regular") || currentBody.equals("flabby")) && (targetBody.equals("regular"))
    ){
        reps = exercise.getMinReps();
    }
    if(
        (currentBody.equals("regular")) && (targetBody.equals("fit") || targetBody.equals("curvy"))
    ){
        reps = exercise.getMinReps();
    }
    if(
        (currentBody.equals("regular")) && (targetBody.equals("athletic") || targetBody.equals("flat stomach"))
    ){
        reps = exercise.getMinReps();
        reps++;
    }
    if(
        (currentBody.equals("flabby")) && (targetBody.equals("fit") || targetBody.equals("curvy"))
    ){
        reps = exercise.getMinReps();
        reps++;
    }
    if(
        (currentBody.equals("flabby")) && (targetBody.equals("athletic") || targetBody.equals("flat stomach"))
    ){
        reps = exercise.getMinReps();
        reps+=2;
    }
}
```

Picture 6. Partial view of determineReps function in ExerciseMapper

To determine the reps of the exercise, I created an algorithm that determines the repetitions based on what the user selected from the current body question and the target body question. For different selections, the algorithm takes out the minReps or the maxReps value from the Exercise entity and it saves those values, or it modifies them by addition or subtraction and saves them after. The algorithm looks like this:

(currentBody + targetBody = reps)

regular + regular = minReps

regular + fit/curvy = minReps

regular + athletic/flat stomach = +1 of minReps

flabby + regular = minReps

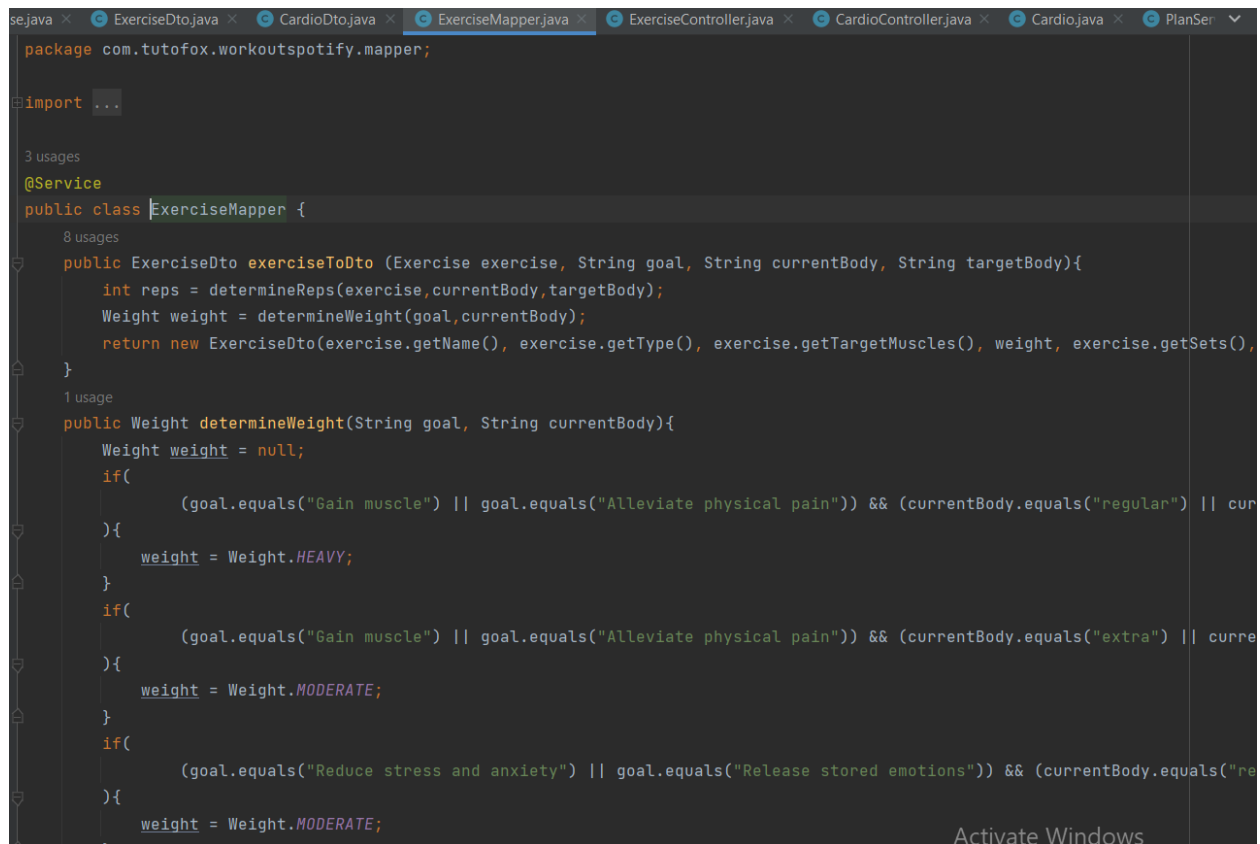
flabby + fit/curvy = +1 of minReps

flabby + athletic/flat stomach = +2 of minReps

extra + regular = +1 of minReps

extra + fit/curvy = +2 of minReps

extra + athletic/flat stomach = -2 of maxReps
 overweight + regular = +2 of minReps
 overweight + fit/curvy = -2 of maxReps
 overweight + athletic/flat stomach = -1 of maxReps
 obese + regular = -1 of maxReps
 obese + fit/curvy = maxReps
 obese + athletic/flat stomach = maxReps



```

package com.tutofox.workoutspotify.mapper;

import ...

@Service
public class ExerciseMapper {

    public ExerciseDto exerciseToDto (Exercise exercise, String goal, String currentBody, String targetBody){
        int reps = determineReps(exercise,currentBody,targetBody);
        Weight weight = determineWeight(goal,currentBody);
        return new ExerciseDto(exercise.getName(), exercise.getType(), exercise.getTargetMuscles(), weight, exercise.getSets(),
    }

    public Weight determineWeight(String goal, String currentBody){
        Weight weight = null;
        if(
            (goal.equals("Gain muscle") || goal.equals("Alleviate physical pain")) && (currentBody.equals("regular") || cur
        ){
            weight = Weight.HEAVY;
        }
        if(
            (goal.equals("Gain muscle") || goal.equals("Alleviate physical pain")) && (currentBody.equals("extra") || curre
        ){
            weight = Weight.MODERATE;
        }
        if(
            (goal.equals("Reduce stress and anxiety") || goal.equals("Release stored emotions")) && (currentBody.equals("re
        ){
            weight = Weight.MODERATE;
        }
    }
}
  
```

Picture 7. Partial view of determineWeight function in ExerciseMapper

To determine the weight that the user uses for that specific exercise, I implemented another algorithm, similar to the previous one. The parameters passed are the goal of the user and the current body type that he has. The outcomes are light weight, moderate weight or heavy weight (based on what the user's capabilities are). The algorithm looks like this:

(goal + currentBody = weight)

Gain muscle/Alleviate physical pain + regular/flabby = heavy weight

Gain muscle/Alleviate physical pain + extra/overweight/obese = moderate weight

Reduce stress and anxiety/Release stored emotions + regular/flabby/extra = moderate weight

Lose weight/Increase flexibility + regular = moderate weight

Reduce stress and anxiety/Release stored emotions + overweight/obese = light weight

Lose weight/Increase flexibility + flabby/extra/overweight/obese = light weight

CardioDto has only 3 variables, id, name and duration. The duration can be set based on 2 different algorithms located in the CardioMapper class. The first one is for the cardio exercise before the workout, and the key factor is the target body the user tries to achieve (picture 8).

```
public CardioDto cardioToDtoBefore(Cardio cardio, String targetBody){  
    int duration = 0;  
    if(targetBody.equals("regular")){  
        duration = 5;  
    }  
    if(targetBody.equals("fit") || targetBody.equals("curvy")){  
        duration = 10;  
    }  
    if(targetBody.equals("athletic") || targetBody.equals("flat stomach")){  
        duration = 15;  
    }  
    return new CardioDto(cardio.getName(), duration);  
}
```

Picture 8. Before workout cardio exercise generator in CardioMapper

The second algorithm (picture 9) is also similar to the previous one, but the only different thing is that this one is based on what the user selects from the activity level question. Also, to mention, there are 5 different answers the user can select: sedentary, somewhat active, moderately active, very active and highly active. If the user selects “sedentary”, then the user won’t receive any after workout cardio exercises in his workout plan.

```

1 usage
public CardioDto cardioToDtoAfter(Cardio cardio, String activity){
    int duration = 0;
    if(activity.equals("Somewhat active")){
        duration = 5;
    }
    if(activity.equals("Moderately active")){
        duration = 10;
    }
    if(activity.equals("Very active") || activity.equals("Highly active")){
        duration = 15;
    }
    return new CardioDto(cardio.getName(), duration);
}

```

Picture 9. After workout cardio exercise generator in CardioMapper

The workout exercises and the cardio exercises are generated through the survey from the frontend. Like mentioned before, the survey takes out key words from what the user selects (the key words are the titles of the selection cards) and those key words are going to be processed in the backend application. There are 3 main controller methods, which use those key words to generate the workout plan: findExercise, located in the ExerciseController class (which returns a list of ExerciseDto entities), findBeforeCardio, located in the CardioController class (which returns 1 CardioDto entity that has time variable "BEFORE") and findAfterCardio, also in the CardioController class (which returns 1 CardioDto entity that has time variable "AFTER"). Those methods are initiated from the frontend via post method (using the axios plugin installed in the React application), which sends out a FormData object including all those key words as arguments. The frontend post method calls can be seen on pictures 10, 11 and 12, while the backend controller methods can be seen on pictures 13, 14 and 15.

The findExercise method calls 8 different functions from the exercise service, and that's because the program is supposed to generate a workout that includes exercises which, all together, targets the entire body. There are 7 key muscles which the controller method tracks and calls functions: chest, back, shoulders, triceps, biceps, abs and legs (legs get 2 functions, one for finding compound exercise and one for isolation for the hamstrings, also known as the back muscle of the legs). All of these service functions use the built in repository function findExerciseByGoalContainingAndTargetMusclesStartsWithAndType, which automatically finds the exercise by giving the specified arguments in its name (picture 16).

Another thing to note is that the post method also saves a userId value in the FormData in all 3 post methods and a name value in the findExercise method, but that will be covered in detail in the User segment.

```

const handleSubmit = async (array) => {
  const formData1 = new FormData();
  const formData2 = new FormData();
  const formData3 = new FormData();
  let response;
  let response1;
  let response2;
  formData1.append("goal", array[0])
  formData1.append("currentBody", array[3])
  formData1.append("targetBody", array[4])
  if(sessionStorage.getItem( key: "user")!=null){
    const userInfo = JSON.parse(sessionStorage.getItem( key: "user"));
    formData1.append("name", array[0] + " workout");
    formData1.append("userId",userInfo.id);
  }
  else {
    formData1.append("name", "nothing");
    formData1.append("userId", "nothing");
  }
  try {
    response = await axios.post(
      url: 'http://localhost:8080/api/exercise/find',
      formData1, config: {
        headers: {
          'Content-Type': 'application/x-www-form-urlencoded',
        },
      },
    )
  }
}

```

Picture 10. Frontend post method for finding workout exercises


```

formData2.append("goal", array[0])
formData2.append("targetBody", array[4])
if(sessionStorage.getItem( key: "user")!=null){
    const userInfo = JSON.parse(sessionStorage.getItem( key: "user"));
    formData2.append("userId",userInfo.id);
}
else{
    formData2.append("userId", "nothing");
}
try {
    response1 = await axios.post(
        url: 'http://localhost:8080/api/cardio/find-before',
        formData2, config: {
            headers: {
                'Content-Type': 'application/x-www-form-urlencoded',
            },
        }
    )
}
catch(error){
    console.error('Error during POST request:', error);
}

```

Picture 11. Frontend post method for finding before workout cardio exercises

```
formData3.append("goal", array[0])
formData3.append("activity", array[5])
if(sessionStorage.getItem( key: "user")!=null){
    const userInfo = JSON.parse(sessionStorage.getItem( key: "user"));
    formData3.append("userId",userInfo.id);
}
else{
    formData3.append("userId", "nothing");
}
try {
    response2 = await axios.post(
        url: 'http://localhost:8080/api/cardio/find-after',
        formData3, config: {
            headers: {
                'Content-Type': 'application/x-www-form-urlencoded',
            },
        }
    )
}
catch(error){
    console.error('Error during POST request:', error);
}
```

Picture 12. Frontend post method for finding after workout cardio exercises

```

@PostMapping(value = "/find")
public ResponseEntity<List<ExerciseDto>> findExercise(
    @RequestParam String goal,
    @RequestParam String currentBody,
    @RequestParam String targetBody,
    @RequestParam String name,
    @RequestParam String userId
){
    List<ExerciseDto> exercise = new ArrayList<>();
    exercise.add(this.exerciseService.findExerciseForChestCompound(goal,currentBody.toLowerCase(),targetBody.toLowerCase()));
    exercise.add(this.exerciseService.findExerciseForBackCompound(goal,currentBody.toLowerCase(),targetBody.toLowerCase()));
    exercise.add(this.exerciseService.findExerciseForShouldersIsolation(goal,currentBody.toLowerCase(),targetBody.toLowerCase()));
    exercise.add(this.exerciseService.findExerciseForTricepsIsolation(goal,currentBody.toLowerCase(),targetBody.toLowerCase()));
    exercise.add(this.exerciseService.findExerciseForBicepsIsolation(goal,currentBody.toLowerCase(),targetBody.toLowerCase()));
    exercise.add(this.exerciseService.findExerciseForStomachIsolation(goal,currentBody.toLowerCase(),targetBody.toLowerCase()));
    exercise.add(this.exerciseService.findExerciseForLegsCompound(goal,currentBody.toLowerCase(),targetBody.toLowerCase()));
    exercise.add(this.exerciseService.findExerciseForLegsIsolation(goal,currentBody.toLowerCase(),targetBody.toLowerCase()));
    if(!userId.equals("nothing")){
        Plan plan = this.planService.createPlan(name,exercise,Integer.valueOf(userId));
        for(ExerciseDto ex : exercise){
            ex.setPlan(plan);
            this.exerciseDtoRepository.save(ex);
        }
    }
    return new ResponseEntity<>(exercise, HttpStatus.OK);
}

```

Picture 13. Backend controller method for finding workout exercises

```

@PostMapping(value="/find-before")
public ResponseEntity<Object> findBeforeCardio(
    @RequestParam String goal,
    @RequestParam String targetBody,
    @RequestParam String userId
){
    if(!userId.equals("nothing")){
        CardioDto dto = this.cardioService.findCardioBefore(goal, targetBody);
        List<Plan> plans = this.planRepository.findAll();
        Plan plan = plans.get(plans.size()-1);
        dto.setPlan(plan);
        List<CardioDto> list = plan.getCardios();
        list.add(dto);
        plan.setCardios(list);
        this.planService.save(plan);
        return new ResponseEntity<>(this.cardioDtoRepository.save(dto),HttpStatus.OK);
    }
    else{
        return new ResponseEntity<>(this.cardioService.findCardioBefore(goal, targetBody), HttpStatus.OK);
    }
}

```

Picture 14. Backend controller method for finding before workout cardio exercises

```

@PostMapping(value="/find-after")
public ResponseEntity<Object> findAfterCardio(
    @RequestParam String goal,
    @RequestParam String activity,
    @RequestParam String userId
){
    if(!activity.equals("sedentary")){
        if(!userId.equals("nothing")){
            CardioDto dto = this.cardioService.findCardioAfter(goal, activity);
            List<Plan> plans = this.planRepository.findAll();
            Plan plan = plans.get(plans.size()-1);
            dto.setPlan(plan);
            List<CardioDto> list = plan.getCardios();
            list.add(dto);
            plan.setCardios(list);
            this.planService.save(plan);
            return new ResponseEntity<>(this.cardioDtoRepository.save(dto),HttpStatus.OK);
        }
        else{
            return new ResponseEntity<>(this.cardioService.findCardioAfter(goal, activity), HttpStatus.OK);
        }
    }
    else{
        return null;
    }
}

```

Picture 15. Backend controller method for finding after workout cardio exercises

```

1 usage
@Override
public ExerciseDto findExerciseForChestCompound(String goal, String currentBody, String targetBody) {
    return this.exerciseMapper.exerciseToDto(this.exerciseRepository.findExerciseByGoalContainingAndTargetMusclesStartsWithAndType(goal, targetMuscle: "chest", Type.COMPOUND).
}

1 usage
@Override
public ExerciseDto findExerciseForBackCompound(String goal, String currentBody, String targetBody) {
    return this.exerciseMapper.exerciseToDto(this.exerciseRepository.findExerciseByGoalContainingAndTargetMusclesStartsWithAndType(goal, targetMuscle: "back", Type.COMPOUND).
}

1 usage
@Override
public ExerciseDto findExerciseForShouldersIsolation(String goal, String currentBody, String targetBody) {
    return this.exerciseMapper.exerciseToDto(this.exerciseRepository.findExerciseByGoalContainingAndTargetMusclesStartsWithAndType(goal, targetMuscle: "shoulders", Type.ISOLATION).
}

1 usage
@Override
public ExerciseDto findExerciseForTricepsIsolation(String goal, String currentBody, String targetBody) {
    return this.exerciseMapper.exerciseToDto(this.exerciseRepository.findExerciseByGoalContainingAndTargetMusclesStartsWithAndType(goal, targetMuscle: "triceps", Type.ISOLATION).
}

```

Picture 16. Part of the service functions for finding workout exercises by goal, target muscles and type

2. Playlists

The playlists are components that are already taken as whole playlists from the Spotify app. After completing the workout survey and receiving the workout, the user can move onto the mood survey, which lets the user receive a Spotify playlist based on which emotion was submitted. The playlists are saved as a list of url links in the frontend and they're connected with the answers. When one answer is submitted, the corresponding url link will be taken out and the page will be redirected to that url (pictures 17, 18 and 19).

```
const questions = [
  {
    question: "How do you feel today?",
    answers: [
      { title: "Happy", image: "/images/emotions/happy.png" },
      { title: "Sad", image: "/images/emotions/sad.png" },
      { title: "Energetic", image: "/images/emotions/energetic.png" },
      { title: "Joyful", image: "/images/emotions/joyful.png" },
      { title: "Relaxed", image: "/images/emotions/relaxed.png" },
      { title: "Drowsy", image: "/images/emotions/drowsy.png" },
    ],
  },
];

const playlists = {
  happy: 'https://open.spotify.com/playlist/37i9dQZF1EIqG2NE0hqsD7',
  sad: 'https://open.spotify.com/playlist/37i9dQZF1DX76Wlfdnj7AP',
  energetic: 'https://open.spotify.com/playlist/0e6BLt9vFkefYCF543CxaY',
  joyful: 'https://open.spotify.com/playlist/37i9dQZF1EQp9BVPsNVof1',
  relaxed: 'https://open.spotify.com/playlist/37i9dQZF1EIhshGKK0SEkb',
  drowsy: 'https://open.spotify.com/playlist/37i9dQZF1EIqbUtlWmHt'
};
```

Picture 17. Answers from mood survey and playlists list in frontend

```

const handleSubmit = async () => {
  setLoading( value: true);
  const link = determineLink();
  if(sessionStorage.getItem( key: 'user')!=null){
    const userInfo = JSON.parse(sessionStorage.getItem( key: "user"));
    const formData = new FormData();
    formData.append("name", selectedAnswer + " playlist")
    formData.append("url", link);
    formData.append("id", userInfo.id)
    try{
      const response = await axios.post(
        url: 'http://localhost:8080/api/playlist',
        formData, config: {
          headers: {
            'Content-Type': 'application/x-www-form-urlencoded',
          },
        }
      )
    }
    catch(error){
      alert(error.response.data);
    }
  }
  setTimeout( handler: () => {
    setLoading( value: false);
    window.location.href=link;
  }, timeout: 2000);
};

```

Picture 18. Main redirect page function in the mood survey page

```

const determineLink = () => {
  let emotionLink;
  if(selectedAnswer=="Happy"){
    emotionLink=playlists.happy
  }
  else if(selectedAnswer=="Sad"){
    emotionLink=playlists.sad
  }
  else if(selectedAnswer=="Energetic"){
    emotionLink=playlists.energetic
  }
  else if(selectedAnswer=="Joyful"){
    emotionLink=playlists.joyful
  }
  else if(selectedAnswer=="Relaxed"){
    emotionLink=playlists.relaxed
  }
  else if(selectedAnswer=="Drowsy"){
    emotionLink=playlists.drowsy
  }
  return emotionLink;
}

```

Picture 19. DetermineLink function from the main redirect page function

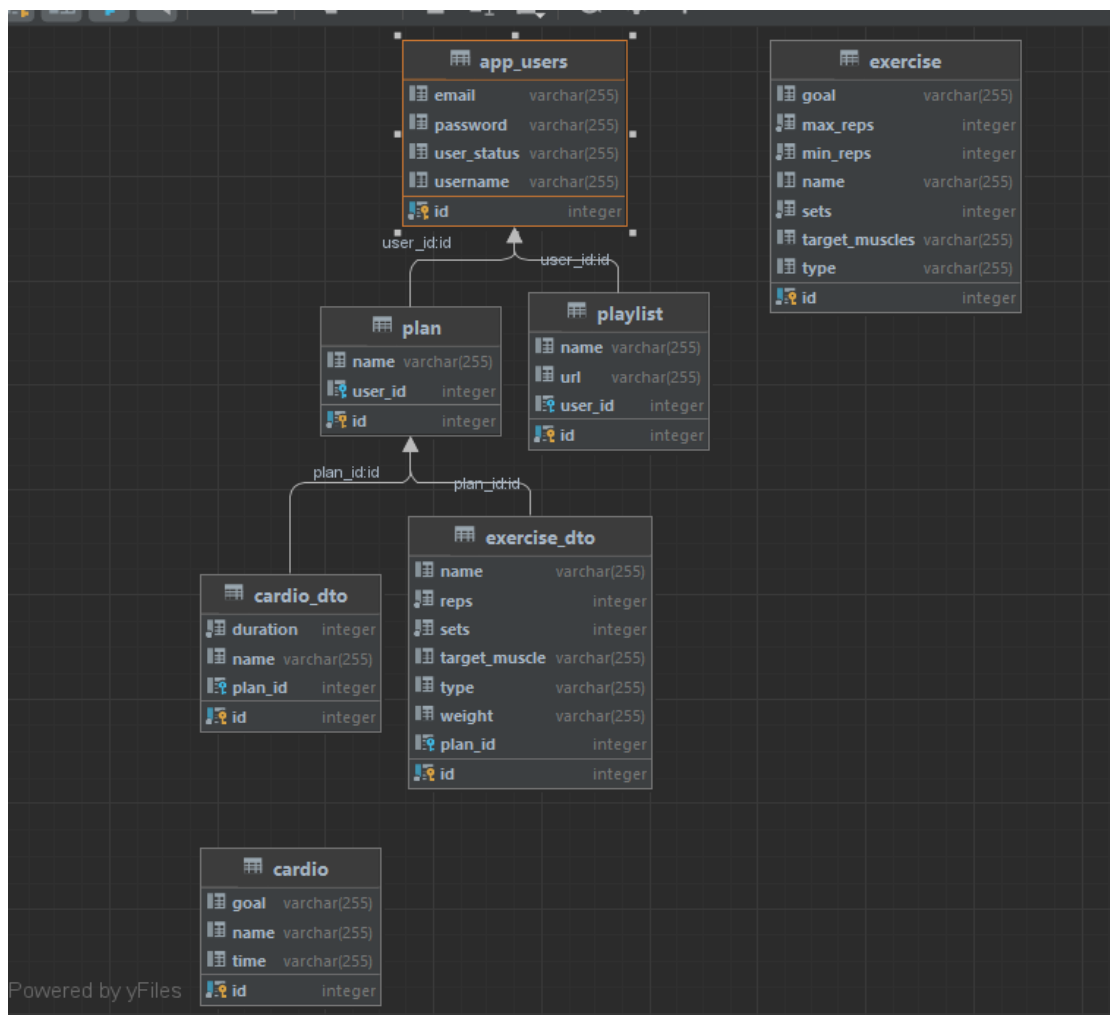
The selection of playlists that fit the emotion was pretty straight-forward: if it's a positive/high energy emotion, put a playlist to let the user keep the emotion or uplift it, otherwise, put a playlist to let the user feel more energized and motivated to workout. For example, if the user selects "energetic", it'll redirect the page to a playlist that's fast paced, if it's "relaxed", it'll transfer the user to a trap playlist so he/she can feel like working out.

The playlist component in the backend application is designed for registered website guests, who registered using the authentication system. For regular website guests, the two surveys only allow to show the workout plan and the playlist on the page and isn't saved anywhere.

3. Users

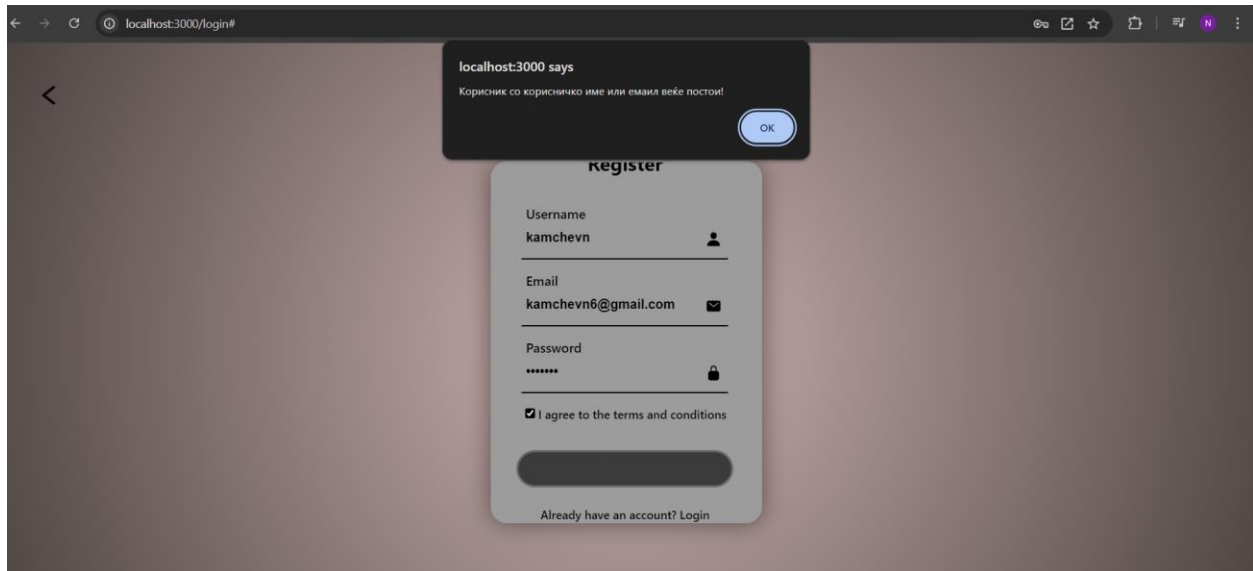
There is an option for the users to register on the web application, and that is through the authentication system. The main purpose for this entity to exist is to save all of the workout plans and playlists the user got from finishing both surveys when the user is logged in, to review them at any time during the logged-in session and to delete them if the user wanted to.

The User entity contains a few variables, like username, email, password and user status, to determine whether is logged in or logged out. It also contains a list of saved playlists (consisting of the name of the playlist and the url link) and saved plans, which they all contain their own list of cardio exercises and workout exercises separately. They are all connected through one-to-many relationships, which can be seen on picture 20.

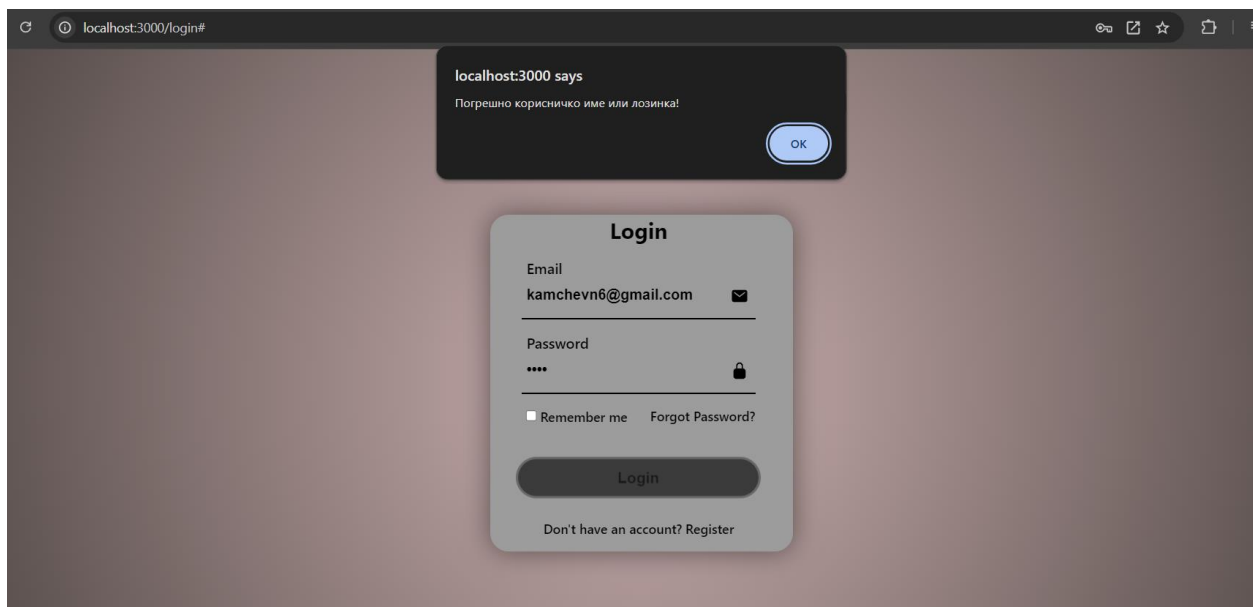


Picture 20. Visualization of all entities and connections through a graph map

The user registers through the authentication system, through a valid input, which contains a unique username, a unique email and a password no shorter than 6 characters. If the user tries to submit the form with a username or email that exists, or with a password less than or equal to 6 characters, it'll display specified error message (picture 21). Same thing goes when trying to log in using an incorrect email and/or password.



Picture 21. Example of a register attempt error message



Picture 22. Example of a login attempt error message

When the user logs in, his information will be added into a session storage and the home page navbar will be modified with a profile page for the user to access and a logout button to exit the session. The user can see his information on the profile page, like his username and password.

Also, 2 separate tables show below the user information box for all the workout plans and playlists the user has saved. The view button on every playlist redirects the user to the playlist page on Spotify, while the view button on every plan redirects the user to a page similar to the workout survey result page (named as “Workout.js”) and lists all the cardio exercises for before and after the workout, as well as all of the workout exercises the plan contains. Both of these tables, for each value, contain a delete button, which on click, not only do they delete the entire playlist/plan from the database, but it also removes the connection with the corresponding user, so the user has more storage to store other playlists/plans.

When the user presses on the “Logout” button on the home webpage, the user is logged out and all of the user information is removed from the session storage.

Lastly, how are the workout plans and playlists saved into those user tables? From what I mentioned before in the workout segment, I add inside the FormData component a “userId” variable for all of the post methods (in the cardio, exercise, plan and playlist controllers), and a “name” variable for the findExercises method in the Exercise controller, so it generates a plan with that specific name (picture 23).

```
if(sessionStorage.getItem( key: "user")!=null){
  const userInfo = JSON.parse(sessionStorage.getItem( key: "user"));
  formData3.append("userId",userInfo.id);
}
else{
  formData3.append("userId", "nothing");
}
```

Picture 23. Modification in the post methods from the frontend for active users in session

If there is no active user in session, then the default value of those 2 variables is “nothing” and the methods from the controllers only return the plans/playlists on the web page, without saving those in the user and plan/playlist tables inside the database. Otherwise, the methods create entities, saves them in their own specific tables, and establishes the necessary connections between each other and with the userId in session(as shown on picture 20).