WORKOUT MANAGER

Iteration 2 – Server Side Documentation

Team 11

Mahek Parmar 202052379

Tejus Revi 202061206

Brief update from Iteration 1

- The server side of the application has been implemented using the MVC framework.
- Our application contains 3 models;
    1. User
    2. Exercise
    3. WorkoutProgram
- The Exercise model was implemented according to the description from Iteration 1.
- The object called Routine was renamed to WorkoutProgram as it made more sense and made the application clearer and more concise. The WorkoutProgram object was also implemented accordingly.
- A User model has also been implemented, modeling several real-life user attributes.
- Going hand in hand with the models, we also have controllers that allow us to manipulate each model. The 3 controllers in our application include:
    1. UserController
    2. ExerciseController
    3. WorkoutProgramController
- As planned in the first iteration, we have used PassportJS as our authentication middleware to help verify users. Users can create their account either by inputting the required fields, or by using their Google Account.
- All data of our application is stored using MongoDb .

Functionalities implemented

- The user can create their accounts, either using their Google authentication, or by manually entering the username, email and password.
- The user can update/change their usernames and passwords, whenever they wish to.
- The user can delete their account if they want to
- Every user account has user-related information such as their age, height, weight, gender and target weight. The user may update these accordingly whenever they wish to.
- The user can view exercises – they can either search for single specific exercises, or they may search for a range of different exercises providing some exercise details such as equipment, target muscle and body part.
- The user can create their own custom made workout program
- They can set the workout program to either public (can be viewed by others) or private (can only be viewed by user themselves)
- After creating a workout program, the user may delete it if they want.
- The user can update the workout program details if they want to.
- The user can add exercise to the workout program, specifying how many repetitions and sets per exercise added to their workout program.
- The user can also remove/delete exercises from their workout programs.
- The user can view all the workout programs created by them
- The user can view workout programs created by others which are public

<u>**Functionalities not implemented.**</u>

- We opted out on the functionality of selecting a language preference due to the time constraint giving its time-consuming implementation.
- Also, the color accessibility/pallet settings are not implemented as of now. Since that includes the user's view/front end, they will be implemented in the next iteration as we design the front end.
- Finally, the functionality of exporting workout plans was temporarily paused due to the time constraint. We also feel that is not a crucial functionality because the user can simply open the application and view their workouts. Nonetheless, it will be thought of for the next iteration.

<u>**Additional functionalities.**</u>

- The remodeling of the previously known Routine to WorkoutProgram allows us to model WorkoutProgram closer to reality. As such we were able to implement more due to this (as will be explained in later sections)
- In addition to performing CRUD operations on them, users can now also search and retrieve other public workout programs (as will be explained in later sections).
- The workoutProgram model now has its access modifier – whether it is private or public. Hence, we are adding authorization in addition to authentication.

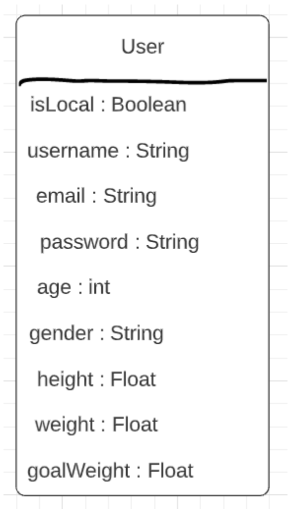
Models

As mentioned above we have 3 models:

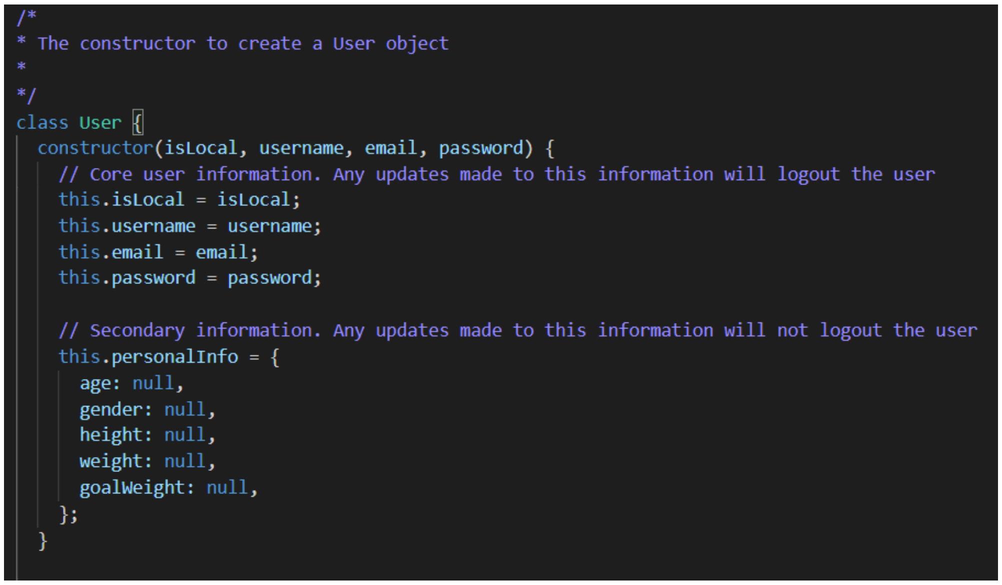
1. User

The User model models a real-life user. The user model is what interacts with the Exercise and WorkoutProgram model. This is where most of the functionality happens.

The User model works hand in hand with the *user* collection in the mongoDb database. We store all the User objects in the *user* collection. Our application requires user *email* to be unique, hence the email serves as the Primary key for this collection.



The constructor for creating a User is as bellows:



```
/*
* The constructor to create a User object
*
*/
class User {
  constructor(isLocal, username, email, password) {
    // Core user information. Any updates made to this information will logout the user
    this.isLocal = isLocal;
    this.username = username;
    this.email = email;
    this.password = password;

    // Secondary information. Any updates made to this information will not logout the user
    this.personalInfo = {
      age: null,
      gender: null,
      height: null,
      weight: null,
      goalWeight: null,
    };
  }
```

The User attributes are split into 2 groups

1. Core/Primary information – any updates to this would log the user out. These include:
    a. isLocal: a Boolean, false if account created using Google, else true
    b. username : the username the user wishes to use for their account
    c. email : the user's email address
    d. password : the user's password
2. Secondary information – any updates to these will not log the user out. These are non-mandatory and include:
    a. age : an integer representing the User's age
    b. gender : a String representing the User's gender
    c. height : a float representing the User's height
    d. weight : a float representing the User's weight
    e. goalWeight : a float representing the User's goal/target weight

One thing to note is that the User passwords are not stored as provided by the user, instead, in the *user* collection, the passwords are hashed before we store them. This ensures better privacy, and at the same time more integrity. At any time when the password needs to be retrieved (for example when changing the password), the password would be un-hashed before being provided to the user.

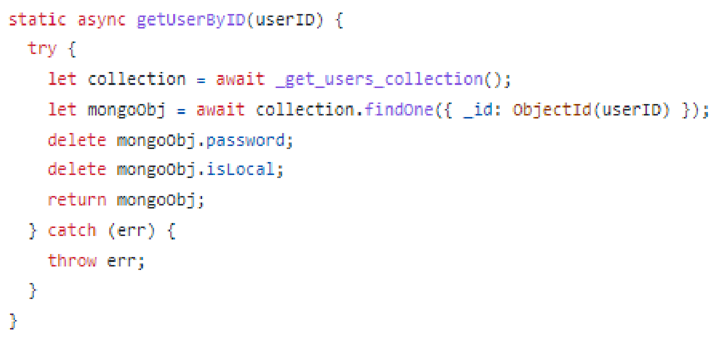
Methods involving the User model.

a. save()

- Saves the current instance of User class into *user* collection
- If the user was correctly inserted, then the function returns true. Else, it returns false.
- This function is invoked when a new user registers for the application.
- When a user who uses Google authentication signs up for the first time, this function is invoked. The parameter *isLocal* is set to false to remember that this particular user has used Google Authentication to sign up.

b. getUserByID()

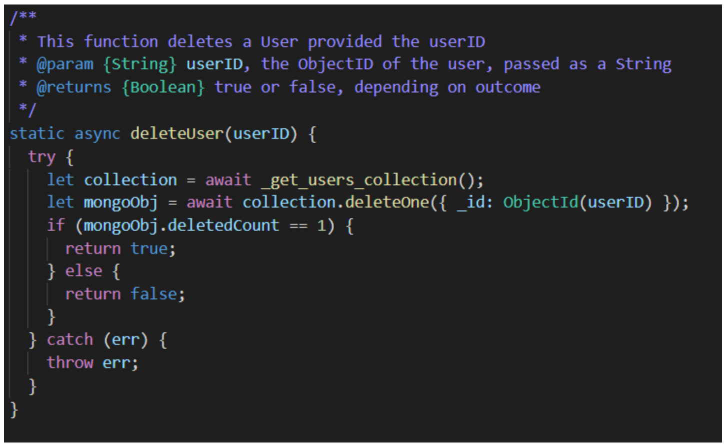
- Finds and returns a single User object from the *user* collection based on the user's id.
- Deletes the *password* and *isLocal* fields before returning the user object.



```
static async getUserByID(userID) {
  try {
    let collection = await _get_users_collection();
    let mongoObj = await collection.findOne({ _id: ObjectId(userID) });
    delete mongoObj.password;
    delete mongoObj.isLocal;
    return mongoObj;
  } catch (err) {
    throw err;
  }
}
```

c. <u>deleteUser(userID)</u>

- A simple method that deletes a User from the *user* collection given its userID
- We would query the *user* collection using the userID as the query parameter, and then delete the User from the collection.
- This is a desired functionality as users may no longer want to use the application, in that case we would have to delete all the details pertaining to them from our database.



```
/**
 * This function deletes a User provided the userID
 * @param {String} userID, the ObjectID of the user, passed as a String
 * @returns {Boolean} true or false, depending on outcome
 */
static async deleteUser(userID) {
  try {
    let collection = await _get_users_collection();
    let mongoObj = await collection.deleteOne({ _id: ObjectId(userID) });
    if (mongoObj.deletedCount == 1) {
      return true;
    } else {
      return false;
    }
  } catch (err) {
    throw err;
  }
}
```

d. <u>updateUser(userID, newUsername, newPassword)</u>

- A method that allows to update the users username and password given the userID
- This allows us to update the core user details. Hence, as mentioned above, doing so will log the user out.
- We would query the *user* collection using the userID as the query parameter, and then update the said fields.
- This is key functionality as users would want to change their usernames and passwords due to various reasons. As users, we usually change our passwords due to security reasons. This method provides us with this key functionality.

e. <u>updatePersonalInfo(userID, age, gender, height, weight, goalWeight)</u>

- This method allows us to update the secondary user information such as the user's age, gender, height, weight and goal weight, given the userID
- It will query the *user* collection using the userID as the query parameter and it will then update every attribute which was provided.
- This method also provides an important functionality as secondary details usually change, and users would want to update them to keep them upto mark.
- As mentioned above, since this is not the user's primary information, updating this will not log the user out.

f. getPasswordFor(email)

- This method finds and returns the hashed password for the user represented by the *email* parameter, provided that the email address exists in the collection.
- Since the application requires email addresses to be unique, we are guaranteed that a single password will be returned in a function call.

g. getUserDetails(email)

- This function provides passport authentication middleware with information about the currently logged in user.
- The function will find and return the user object represented by the *email* parameter.

h. addNonLocal(username, email)

- Adds a non local User object into the database. A non local user is any user who has used Google Authentication to sign up.
- For nonLocal users, their passwords are not saved in the *user* collection. However, the email address is stored to avoid the user from creating another account.

i. authenticateUser (email, Password)

- This function verifies if the password entered by the user matches the hashed password store in the *user* collection.
- Uses *verify* method of password-hash to check equality between unhashed password entered by the user and the hashed password stored in the database.

j. emailDoesNotExist(email)

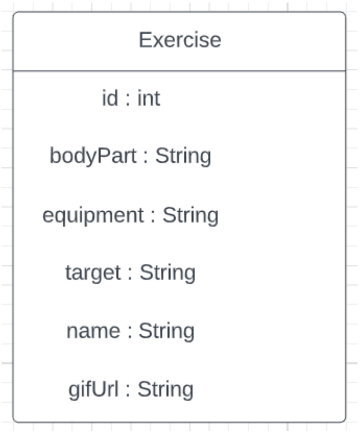
- This is a helper function that helps us to check whether a User with the provided email address exists in the *user* collection or not.
- It is more of an internal function that assists in other functions.
- Since the User email is the PK for this collection, we need this function.

2. Exercise

The Exercise model represents an Exercise Object and is crucial for our application. This Exercise model works hand in hand with the mongoDb *exercise* collection.

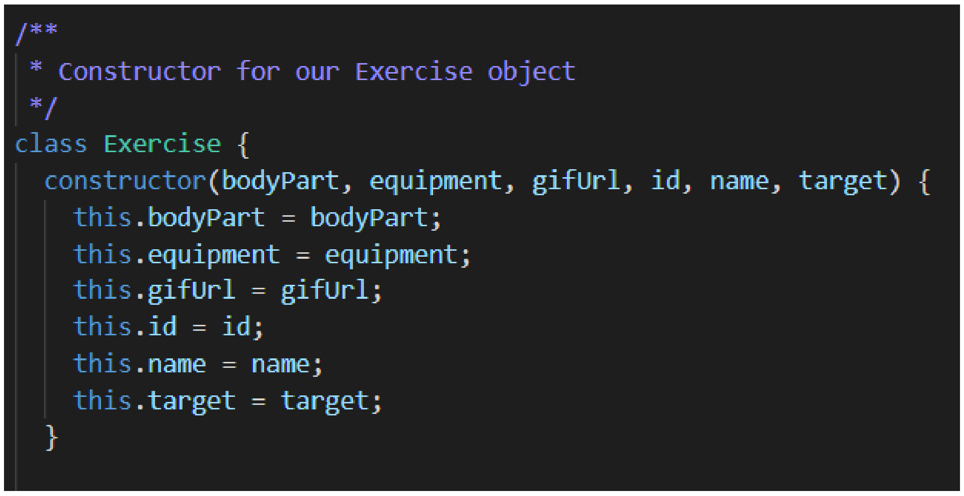
The mongoDb collection *exercise* serves as our database for the various Exercises. It contains 1327 entries of various different Exercises which were imported from our database API - ExerciseDB. Since the underlying basis of the application involves exercises, this model is crucial for our application as it connects us to the database collection which contains our exercises.

The Exercise model is modified using its controller exerciseController (discussed later)



Users can retrieve exercises of their choice by applying specific query parameters and the *exercise* collection provides the respective Exercises accordingly.

The constructor for creating an Exercise Object is as below.


```
/**
 * Constructor for our Exercise object
 */
class Exercise {
  constructor(bodyPart, equipment, gifUrl, id, name, target) {
    this.bodyPart = bodyPart;
    this.equipment = equipment;
    this.gifUrl = gifUrl;
    this.id = id;
    this.name = name;
    this.target = target;
  }
```

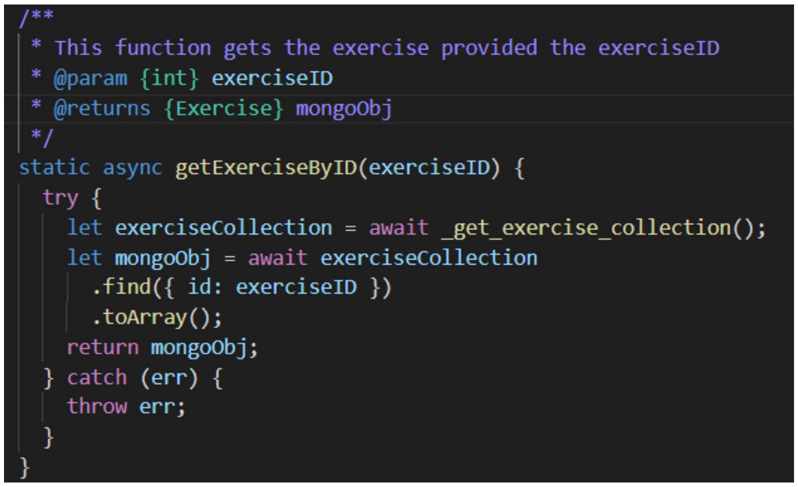
A brief description of the attributes;

- id: a unique int id, which serves as the Primary Key to identify Exercise Objects.
- bodyPart: the body part which the exercise works
- gifUrl: a URL that gets us a GIF which describes how to perform the exercise
- equipment: the type of equipment needed to perform the exercise
- name: the name of the exercise
- target: the specific muscle which the exercise targets.

Methods involving the Exercise model.

a. getExerciseByID(exerciseID)

- As was mentioned, the int attribute exerciseID is unique to every Exercise object and hence it serves as the Primary Key for the Exercises Collection.
- We use this method to retrieve an Exercise from the *exercise* collection by querying the *exercise* collection using exerciseID as the query parameter. The Exercise corresponding to the exerciseID is then returned.
- This method is useful as it provides the functionality for "Search a workout exercise". This variant of searching an exercise is specific and since it uses the exerciseID, it will only retrieve the single specific Exercise. The user can then view it or even add it to their workout program.

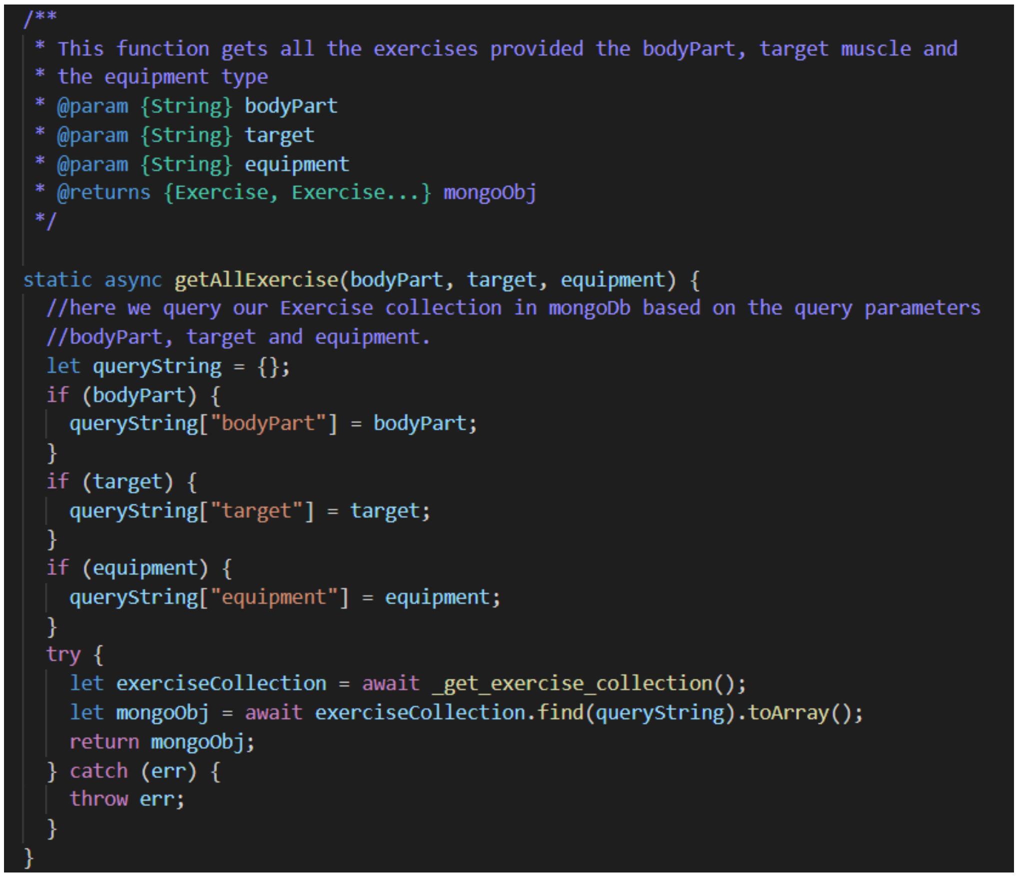


```
/**
 * This function gets the exercise provided the exerciseID
 * @param {int} exerciseID
 * @returns {Exercise} mongoObj
 */
static async getExerciseByID(exerciseID) {
  try {
    let exerciseCollection = await _get_exercise_collection();
    let mongoObj = await exerciseCollection
      .find({ id: exerciseID })
      .toArray();
    return mongoObj;
  } catch (err) {
    throw err;
  }
}
```

b. getAllExercise(bodyPart, target, equipment)

- This method provides the second variant of "Search a workout exercise". This search is much broader.
- Here, the user can search Exercises based on the body part which the exercise works on, the type of equipment the exercise requires and also the muscle that the exercise targets.
- We use these as our query parameters to query the *exercise* collection and all exercises matching the query parameters will be returned in an array.

- This way the user can retrieve several exercise options based on their choice of body part, target muscle and equipment.

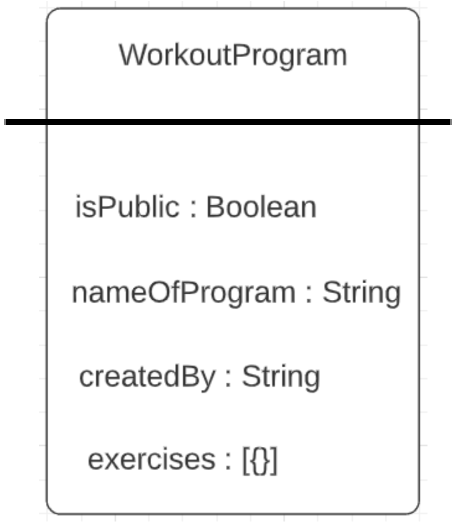


```
/**
 * This function gets all the exercises provided the bodyPart, target muscle and
 * the equipment type
 * @param {String} bodyPart
 * @param {String} target
 * @param {String} equipment
 * @returns {Exercise, Exercise...} mongoObj
 */

static async getAllExercise(bodyPart, target, equipment) {
  //here we query our Exercise collection in mongoDb based on the query parameters
  //bodyPart, target and equipment.
  let queryString = {};
  if (bodyPart) {
    queryString["bodyPart"] = bodyPart;
  }
  if (target) {
    queryString["target"] = target;
  }
  if (equipment) {
    queryString["equipment"] = equipment;
  }
  try {
    let exerciseCollection = await _get_exercise_collection();
    let mongoObj = await exerciseCollection.find(queryString).toArray();
    return mongoObj;
  } catch (err) {
    throw err;
  }
}
```

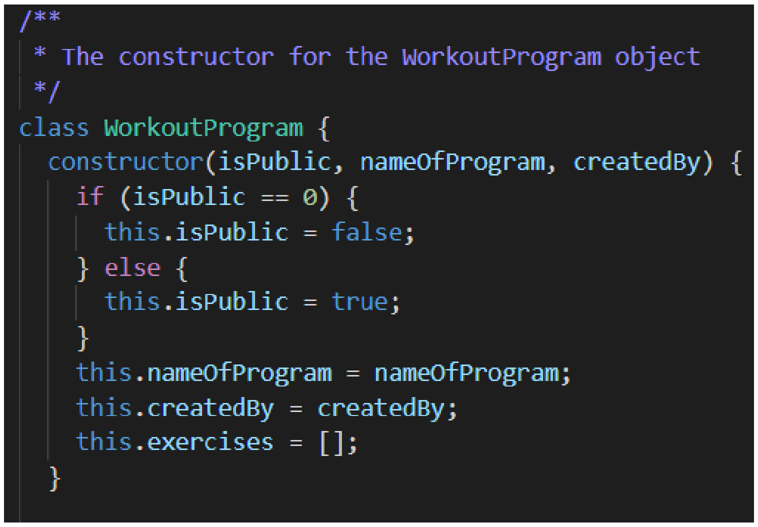
3. **WorkoutProgram**

This models a real life WorkoutProgram. The concept of a workout program is essential for our application because rather than searching for exercises every now and then, users can alternatively enter their selected exercises into a workout program and then simply refer to the workout program whenever they wish. This is a desired functionality that offers convenience for the user.



The WorkoutProgram works hand in hand with the *workout-program* collection in the mongoDb database. We store all the WorkoutProgram objects in the *workout-program* collection.

The constructor for creating a WorkoutProgram object is as below.



```
/**
 * The constructor for the WorkoutProgram object
 */
class WorkoutProgram {
  constructor(isPublic, nameOfProgram, createdBy) {
    if (isPublic == 0) {
      this.isPublic = false;
    } else {
      this.isPublic = true;
    }
    this.nameOfProgram = nameOfProgram;
    this.createdBy = createdBy;
    this.exercises = [];
  }
```

A brief description of the WorkoutProgram attributes;

- isPublic: a Boolean attribute which specifies the access for the WorkoutProgram, whether it is public (can be viewed by others) or private (can only be viewed by user who created the WorkoutProgram)
- nameOfProgram: describes the name of the WorkoutProgram.
- createdBy: the ObjectId of the user who created the WorkoutProgram, passed as a string
- exercises: an array that stores the JSON objects pertaining to {Exercise, numReps, numSets} in a WorkoutProgram. This array is initially empty when a WorkoutProgram is created.

Methods involving the WorkoutProgram model.

a) save()
  - A simple method that saves the WorkoutProgram object to the *workout-program* collection in our mongoDb database.
  - When users create their WorkoutPrograms, we would store them for future reference in the *workout-program* collection

b) getWorkoutProgramByID(workoutProgramID)
  - User's would want to see their workout programs, they would also want to see other public workout programs. This method provides this functionality.
  - It is a simple method which returns a WorkoutProgram Object from the *workout-program* collection given its MongoDb ObjectId.

c) getAllPublicWorkoutPrograms()
  - This method allows the user to see all the workout programs which are public.
  - This is a desired functionality as many times we don't create our own workout programs due to lack of knowledge, experience or time.
  - This functionality allows the user to use workout programs which are created by others and made public.

d) deleteWorkoutProgram(workoutProgramID)
  - This method allows the user to delete a WorkoutProgram which they created, given its MongoDb ObjectId.
  - It is also desired functionality as users would indeed want to delete their workout programs due to various reasons.

e) addExerciseToWorkoutProgram(workoutProgramID, exerciseID, numSets, numReps)

  - On creating a workout program, users would want to be able to add exercises along with their frequency (sets and repetitions/reps) to their workout program.
  - This method provides this functionality.
  - We simply provide the ObjectId of the WorkoutProgram document where we want to add an exercise, and together with that we provide the id of the exercise to add, along with the desired sets and reps.

f) removeExerciseToWorkoutProgram(workoutProgramID, exercise)

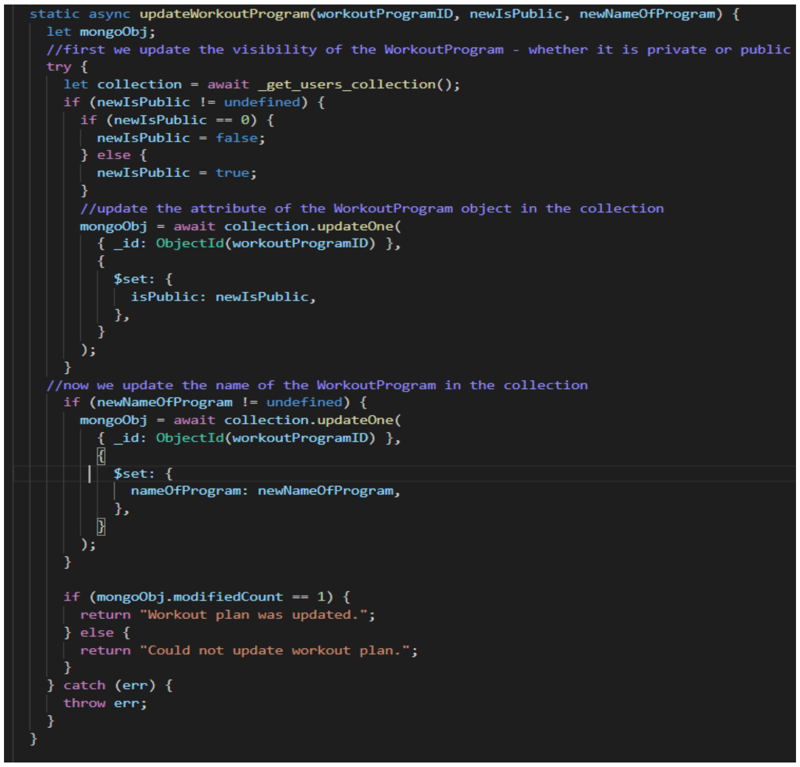
- Users would also want to remove exercises from their workout programs due to various reasons.
- This method provides the user with this functionality.
- We need to provide the ObjectId of the WorkoutProgram document from which we want to remove an Exercise.
- In addition to that, we need to provide the id of the exercise which we want to remove.

g) getWorkoutProgramsByUser(userID)

- Users would want to be able to see all the workout programs they created.
- This method provides all the workout programs created by a particular user.
- We query the *workout-program* collection based on the createdBy attribute and return all the resulting WorkoutProgram objects that match.

h) updateWorkoutProgramDetails(workoutProgramID, newIsPublic, newNameOfProgram)

- This is a key method involving the WorkoutProgram model
- It allows us to update the attributes of a WorkoutProgram object
- It provides us with a key practical functionality as users may want to change the name of their workout program, they may want to change the workout program to either public or private.



```
static async updateWorkoutProgram(workoutProgramID, newIsPublic, newNameOfProgram) {
  let mongoObj;
  //first we update the visibility of the WorkoutProgram - whether it is private or public
  try {
    let collection = await _get_users_collection();
    if (newIsPublic != undefined) {
      if (newIsPublic == 0) {
        newIsPublic = false;
      } else {
        newIsPublic = true;
      }
      //update the attribute of the WorkoutProgram object in the collection
      mongoObj = await collection.updateOne(
        { _id: ObjectId(workoutProgramID) },
        {
          $set: {
            isPublic: newIsPublic,
          },
        }
      );
    }
  //now we update the name of the WorkoutProgram in the collection
    if (newNameOfProgram != undefined) {
      mongoObj = await collection.updateOne(
        { _id: ObjectId(workoutProgramID) },
        {
          $set: {
            nameOfProgram: newNameOfProgram,
          },
        }
      );
    }

    if (mongoObj.modifiedCount == 1) {
      return "Workout plan was updated.";
    } else {
      return "Could not update workout plan.";
    }
  } catch (err) {
    throw err;
  }
}
```

Routes and Controllers

We have controllers for each model. The controllers allow the user to interact with the models, via the functionalities and achieve their user-level goals.

We will be describing the controller functionalities along with their respective routes, where applicable.

1. UserController.

The workoutProgramController allows the user of our application to interact with the User model. It deals with user-related tasks involving, validating, deleting, updating User. It does so via the following functions;

a. getUserByID

What it does?	Helps retrieve users based on their user id.
Interacts with?	User model via ● getUserByID(userID) method
Achieved via?	GET HTTP call `app.get("/user", auth.checkAuthenticated, userController.getUserByID); // Receives userID from the session`

b. addLocal

What it does?	Adds a local user. It validates the provided credentials first before adding a user.
Interacts with?	User model via ● save() method
Achieved via?	POST HTTP call `app.post("/user", userController.addLocal);`

c. deleteUser

What it does?	Allows the deletion of a User given the userID
Interacts with?	User model via ● deleteUser(userID) method
Achieved via?	DELETE HTTP call `app.delete("/user", auth.checkAuthenticated, userController.deleteUser);`

d. updateUser

What it does?	Allows the user to update the core User information given the userID. It validates the information before updating it.
Interacts with?	User model via ● updateUser(userID, newUsername, hashedNewPassword) method

Achieved via?	PUT HTTP call
	`app.put("/user", auth.checkAuthenticated,`
	`userController.updateUser);`

e. updatePersonalInformation

What it does?	Allows the user to update the secondary User information given the userID. It validates the information before updating it.
Interacts with?	User model via ● updatePersonalInfo(userID, age, gender, height, weight, goalWeight) method

Achieved via?	PUT HTTP call
	`app.put(`
	`  "/user/personalInformation",`
	`  auth.checkAuthenticated,`
	`  userController.updatePersonalInformation`
	`);`

f. logout

What it does?	Logs out the currently logged in user from the session
Interacts with?	No interaction with the model

Achieved via?	GET HTTP call
	`app.get("/logout", auth.checkAuthenticated,`
	`userController.logout); // Receives userID from the session`

Notes	The endpoint /logout requires authentication. Hence, only authenticated users will be able to logout.

2. exerciseController.

The exerciseController allows the user to interact with the Exercise model. Hence, users can view exercises of their choice via the following functions:

a) getExerciseByID

What it does?	This function helps provide the user with the Exercise given an exerciseID
Interacts with?	Exercise model via ● getExerciseByID(exerciseID) method
Parameters	Passed as endpoint variable
Achieved via?	GET HTTP call `app.get("/exercise/:exerciseID", exerciseController.getExerciseByID);`

b) getAllExercise

What it does?	This function provides the user with all Exercises according to the user's selection of body part, target muscle and equipment.
Interacts with?	Exercise model via ● getAllExercise(bodyPart, target, equipment) method
Parameters	Passed as query parameters
Achieved via?	GET HTTP call `app.get("/exercise", exerciseController.getAllExercise);`

3. WorkoutProgramController.

The workoutProgramController allows the user of our application to interact with both the WorkoutProgram model and the Exercise model. Hence, the user can create, view, add exercises, delete exercises, edit their workout programs with help of workoutProgramController. It does so via the following functions;

a) getAllPublicWorkoutPrograms

What it does?	Provides the user with all the workout programs which are public.
Interacts with?	WorkoutProgram model via ● getAllPublicWorkoutPrograms() method
Achieved via?	GET HTTP call `app.get("/workoutProgram", workoutProgramController.getAllPublicWorkoutPrograms);`
Note	User authentication is optional to achieve this functionality.

b) getWorkoutProgramByID

What it does?	Provides the user with a specific workout program given its workoutProgramID.
Interacts with?	WorkoutProgram model via ● getWorkoutPorgramByID(workoutProgramID) method
Parameters?	Passed through the endpoint variable
Achieved via?	GET HTTP call `app.get("/workoutProgram/:workoutProgramID", workoutProgramController.getWorkoutProgramByID);`
Notes	If a user is trying to access a private WorkoutProgram, the authentication middleware provides the currently logged in user's information. If the WorkoutProgram is public, this does not matter.

c) addWorkoutProgram

What it does?	Allows the user to add/create a new workout program. It performs validation before creating a new WorkoutProgram
Interacts with?	WorkoutProgram model by • the save() method
Parameters?	Passed through the request body

Achieved via?	POST HTTP call
	`app.post(`
	`"/workoutProgram",`
	`auth.checkAuthenticated,`
	`      workoutProgramController.addWorkoutProgram`
	`);`

Note	User needs to be authenticated to access this functionality.

d) deleteWorkoutProgram

What it does?	Allows the user to delete/remove a workout program. Only the user who created the WorkoutProgram can delete it. It checks if the workout program exists before deleting it
Interacts with?	WorkoutProgram model via • deleteWorkoutProgram(workoutProgramID)method
Parameters?	Passed through the endpoint variable

Achieved via?	DELETE HTTP call
	`    app.delete(`
	`      "/workoutProgram/:workoutProgramID",`
	`      auth.checkAuthenticated,`
	`      workoutProgramController.deleteWorkoutProgram`
	`    );`

Notes	User needs to be authenticated to access this functionality.

e) updateWorkoutProgramDetails

What it does?	Allows the user to update/change details of their workout programs. Again, only the user who created the workout program can edit its name and visibility (public or private). It checks if WorkoutProgram exists before updating it
Interacts with?	WorkoutProgram model via • updateWorkoutProgramDetails(workoutProgramID, isPublic, nameOfProgram) method
Parameters?	Parameters needed are • workoutProgramID (passed through endpoint variable) • isPublic and nameOfProgram (passed via request body)
Achieved via?	PUT HTTP call `app.put( "/workoutProgram/:workoutProgramID", auth.checkAuthenticated, workoutProgramController.updateWorkoutProgramDetails );`
Notes	Users need to be authenticated to access this functionality.

f) addExerciseToWorkoutProgram

What it does?	Allows the user to add an exercise to their workout program. Only the user who created the workout program can add exercise to it.
Interacts with?	Exercise model via • getExerciseByID (exerciseID) method WorkoutProgram model via • getWorkoutProgramByID(workoutProgramID) • addExerciseToWorkoutProgram(workoutProgramID, exercise, numSets, numReps) methods
Parameters?	Parameters needed are • workoutProgramID (passed through endpoint variable) • exerciseID, numSets, numReps (passed through request body)
Achieved via?	PUT HTTP call `app.put( "/workoutProgram/addExercise/:workoutProgramID", auth.checkAuthenticated, workoutProgramController.addExerciseToWorkoutProgram );`
Note	Users need to be authenticated to access this functionality.

g) removeExerciseFromWorkoutProgram

What it does?	Allows the user to remove an delete/remove an exercise from their workout program. Only the user who created the workout program can delete any exercises from it.
Interacts with?	WorkoutProgram model via ● getWorkoutProgramByID(workoutProgramID) ● removeExerciseFromWorkoutProgram(workoutProgramID, exerciseID) methods
Parameters?	Parameters needed are: • exerciseID (passed as a query parameter) • workoutProgramID (passed as endpoint variable)

Achieved via?	PUT HTTP call
	`    app.put(`
	`      "/workoutProgram/removeExercise/:workoutProgramID", // Exercise id passed as query param`
	`      auth.checkAuthenticated,`
	`      workoutProgramController.removeExerciseFromWorkoutProgram`
	`    );`

Note	Users need to be authenticated to access this functionality.

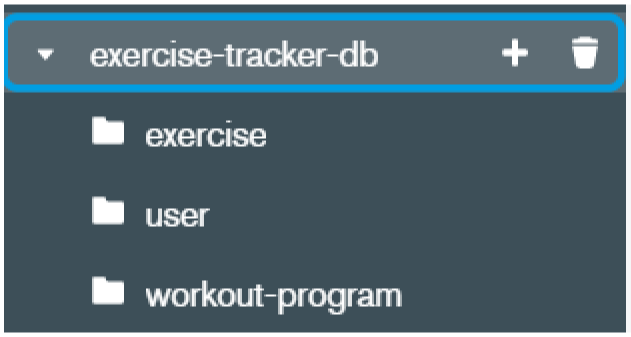
h) getWorkoutProgramsByUser

What it does?	Allows the user to retrieve all workout programs they created
Interacts with?	WorkoutProgram model via ● getWorkoutProgramByUser(userID) method

Achieved via?	GET HTTP call
	`    app.get(`
	`      "/user/workoutPrograms",`
	`      auth.checkAuthenticated,`
	`      workoutProgramController.getWorkoutProgramsByUser`
	`    )`

Note	Users need to be authenticated to access this functionality. Current user details will be received by our authentication middleware.

Description of the Database



Our MongoDd database is called *exercise-tracker-db,* it consists of 3 collections:

1. exercise

This collection consists of our data.

It is a read only collection

It contains a total of 1327 Exercise objects. Each Exercise object represents a single exercise. This data is imported from our database API - ExerciseDB.

2. user

This collection stores our User data.

3. workout-program

This collection stores the WorkoutProgram Objects. When a User creates a workout program, we store it in this collection. To retrieve and perform CRUD operations on them, we query this *workout-program* collection and then perform the necessary operations on the queried results.

When designing the database, we have used both the normalized and embedded data modeling approaches.

Users create their accounts, their details and credentials are stored in the *user* collection of our database. They can search exercises from the *exercise* collection. What brings the User and the Exercise together in the database is the *workout-program* collection. This collection stores WorkoutProgram objects.

Every WorkoutProgram object stores a reference(createdBy) to its User, hence using this reference we can search the *user* collection to retrieve the user (normalized).

- The normalized approach was chosen over the embedded because the *user* collection is dynamic, users are constantly added, edited and removed. Using an embedded approach here would not be appropriate as it would be inefficient when a User data changes. In this case, we would have to update the User both in the *user* collection and in the WorkoutProgram object

- Using a normalized approach avoids this problem, we use the user's id as a Primary Key and store this in the WorkoutProgram objects - so now, if a User does change, we only need to update the *user* collection and since we store the user id in the WorkoutProgram object, we do not need to update that - we simply use the user id (PK) and query the *user* collection to identify and retrieve the particular user.

At the same time, the WorkoutProgram object also stores an array that contains JSON entries of Exercise Objects, and their frequencies.

- Here, we use the embedded approach as we store the respective Exercise document inside the JSON entry.
- We know that the data in the *exercise* collection is static - it does not change. So rather than storing the exercise id's and then querying the *exercise* collection every time, we can just store the Exercise documents in WorkoutProgram objects. This would be more efficient as wouldn't need to query the *exercise* collection and would reduce the overall number of lookups.
- If the data in the *exercise* collection would change, then definitely we would have used the normalized approach using the exercise id as the PK, but the fact that data in *exercise* collection does not change, and that the data in the *workout-program* and *exercise* is used together, we feel it is better to use the embedded approach in this regard.

Tests

1-test-exercise.js

1. GET /exercise -
   1.1. [SUCCESS 1] Make a get request to /exercise. Should display all exercise objects as an array. The array should contain 1327 objects.
2. GET /exercise/:exerciseID -
   2.1. [FAIL 1] Makes a get request to /exercise/:exerciseID with invalid ID. Should return an error message.
   2.2. [SUCCESS 1] Makes a get request to /exercise/:exerciseID with valid ID. Should return the exercise object corresponding to the id.

2-test-user.js

1. Test Models
   1.1. [FAIL 1] Tests if validation.validUserInfo returns false when an incorrect email is provided
   1.2. [SUCCESS 1] Test if validation.validUserInfo returns true when valid user info is provided
2. POST /user
   2.1. [FAIL 1] Try to make a post request to /user with invalid email address in body params. The request should return body.success: false as the operation has failed.
   2.2. [FAIL 2] Try to make a post request to /user with no email address in body params. The request should return body.success: false as the operation has failed.
   2.3. [FAIL 3] Trying to make a post request to /user with no password in body params. The request should return body.success: false as the operation has failed.
   2.4. [FAIL 4] Trying to make a post request to /user with no username in body params. The request should return body.success: false as the operation has failed.
   2.5. [FAIL 5] Trying to make a post request to /user with empty email in body params. The request should return body.success: false as the operation has failed.
   2.6. [FAIL 6] Trying to make a post request to /user with empty password in body params. The request should return body.success: false as the operation has failed.
   2.7. [FAIL 7] Trying to make a post request to /user with empty username in body params. The request should return body.success: false as the operation has failed.

2.8. [SUCCESS 1] Trying to make a post request to /user with valid information in body params. The request should return body.success: true as the operation has succeeded.

2.9. [FAIL 8] Trying to make a post request to /user with valid information, but with the same email address as the previous request.

3. POST /auth/local

   3.1. [FAIL 1] Trying to make a post request to /auth/local with incorrect password.

   3.2. [SUCCESS 1] Trying to make a post request to /auth/local with the correct password.

4. GET /user

   4.1. [FAIL 1]Trying to make a get request to /user as an unauthenticated user. Since this endpoint requires authentication, the request returns an error message.

   4.2. [SUCCESS 1] Trying to make a get request to /user as an authenticated user. The user's info is returned, except their password.

5. PUT /user & PUT user/personalInformation

   5.1. [FAIL 1] Trying to make a put request to /user as an unauthenticated user. Since this endpoint requires authentication, the request returns an error message.

   5.2. [SUCCESS 1] Trying to make a put request to /user as an authenticated user. The request will succeed, and display the result of the operation.

   5.3. [SUCCESS 2] Trying to make a put request to /user as an authenticated user. The request will succeed, and display the result of the operation.

   5.4. [FAIL 2] Trying to make a put request to /user/personalInformation as an unauthenticated user. Since this endpoint requires authentication, the request returns an error message.

   5.5. [FAIL 3] Trying to make a put request to /user as an authenticated user. The body parameters passed are invalid, and therefore the request will return an error message.

   5.6. [SUCCESS 3] Trying to make a put request to /user as an authenticated user. The request will succeed, and display the result of the operation.

6. DELETE /user

   6.1. [FAIL 1] Trying to make a delete request to /user as an unauthenticated user. The request will fail.

6.2. [SUCCESS 1] Trying to make a delete request to /user as an authenticated user. The request will succeed.

3-test-workoutProgram.js

1. Test Models
   1.1. [FAIL 1] Tests if validation.validWorkoutProgramInfo returns false when nameOfProgram is empty
   1.2. [SUCCESS 1] Tests if validation.validWorkoutProgramInfo returns true when valid if is provided
2. POST /workoutProgram
   2.1. [FAIL 1] Trying to make a post request to /workoutProgram as an unauthenticated user. The request returns an error message as this endpoint requires authentication
   2.2. [FAIL 2] Trying to make a post request to /workoutProgram as an authenticated user, but with an empty nameOfProgram. The request returns an error message.
   2.3. [FAIL 3] Trying to make a post request to /workoutProgram as an authenticated user, but with invalid isPublic. isPublic has to be either 0 or 1. The request returns an error message.
   2.4. [FAIL 4] Trying to make a post request to /workoutProgram as an authenticated user, but with no nameOfProgram in body params. The request returns an error message.
   2.5. [FAIL 5] Trying to make a post request to /workoutProgram as an authenticated user, but with no isPublic in body params. The request returns an error message.
   2.6. [SUCCESS 1] Trying to make a post request to /workoutProgram as an authenticated user, with valid isPublic and nameOfProgram. The request succeeds. A new private program is created
   2.7. [SUCCESS 2] Trying to make a post request to /workoutProgram as an authenticated user, with valid isPublic and nameOfProgram. The request succeeds. A new public program is created.
3. GET /workoutProgram/:workoutProgramID

   3.1. [SUCCESS 1] Trying to make a get request to /workoutProgram/:workoutProgramID as an unauthenticated user. The workout program being accessed is public, therefore the request succeeds.
   3.2. [FAIL 1] Trying to make a get request to /workoutProgram/:workoutProgramID as an unauthenticated user. The workout program being accessed is private, therefore the request fails.
   3.3. [SUCCESS 1] Trying to make a get request to /workoutProgram/:workoutProgramID as an authenticated user. The workout program being accessed is private, but was created by the user trying to access it. Therefore the request succeeds.
4. PUT /workoutProgram/workoutProgramID
   4.1. [FAIL 1] Trying to make a put request to /workoutProgram/workoutProgramID as an unauthenticated user. The request fails as this endpoint requires authentication.
   4.2. [SUCCESS 1] Trying to make a put request to /workoutProgram/workoutProgramID as an authenticated user. The request succeeds.
   4.3. [SUCCESS 2] Trying to make a get request to /workoutProgram as an unauthenticated user. The request will return all public workout programs.
5. PUT /workoutProgram/addExercise/:workoutProgramID
   5.1. [SUCCESS 1] Trying to make a put request to /workoutProgram/addExercise/:workoutProgramID as an authenticated user. The request succeeds.
   5.2. [SUCCESS 2] Trying to make a put request to /workoutProgram/removeExercise/:workoutProgramID as an authenticated user. The request succeeds.
6. DELETE /workoutProgram/:workoutProgramID
   6.1. [FAIL 1] Trying to make a delete request to /workoutProgram/workoutProgramID as an unauthenticated user. The request fails as this endpoint requires authentication.
   6.2. [SUCCESS 1] Trying to make a delete request to /workoutProgram/removeExercise/:workoutProgramID as an authenticated user. The request succeeds.

6.3. [SUCCESS 2] Trying to make a delete request to /workoutProgram/removeExercise/:workoutProgramID as an authenticated user. The request succeeds.