

Project Documentation

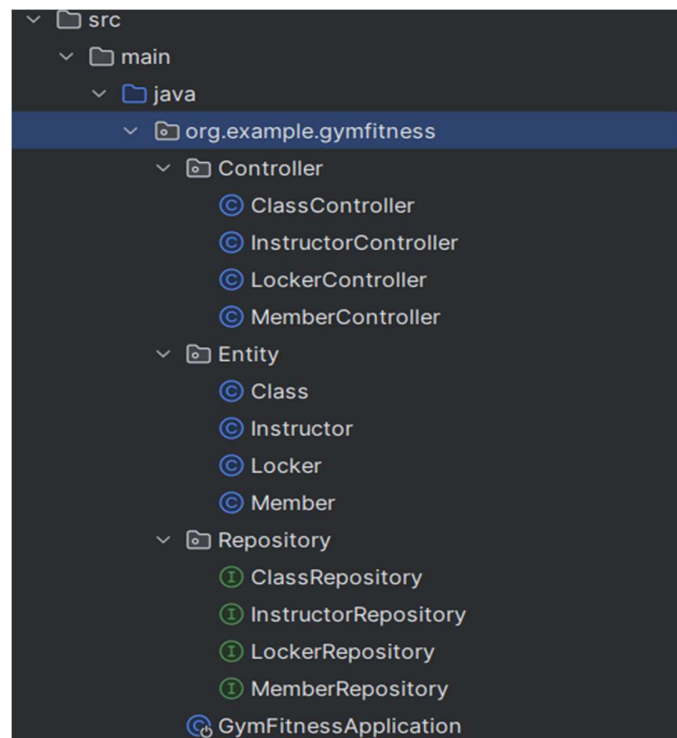
RASHAD ABD ALGHANI RASHAD KARAKI
B2205.010018

GymFitness

Overview: GymFitness project serves as a tool for efficiently managing and organizing various elements within a gym, contributing to the streamlined administration of classes, instructors, lockers, and members. It offers the potential for further development and customization to meet the evolving needs of a gym facility.

Project Structure

- Main Packages and Classes:



- **Entity Relationships**

- **Class Entity:**

- Represents a gym class and as a Many-to-One relationship with **Instructor** and a One-to-Many relationship with **Member**, each Class has one Instructor and many members.

```
• @Entity
public class Class {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Long classId;

    String className;

    int maxCapacity;
    @JoinColumn(name = "instructor")
    @ManyToOne
    Instructor instructor;

    @OneToMany(mappedBy = "aClass")
    @JsonManagedReference
    List<Member> members;
```

- **Instructor Entity:**

- Represents a gym instructor and has a One-to-One relationship with **Locker** and a One-to-Many relationship with **Class**, each Instructor has one locker and many classes.

```
• @Entity
public class Instructor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Long instructorId;

    String instructorName;
    String specialization;

    @OneToOne(cascade = CascadeType.ALL)
    Locker locker;

    @OneToMany(mappedBy = "instructor", cascade = CascadeType.ALL)
    @JsonBackReference
    List<Class> classes;
```

- **Locker Entity:**
 - Represents a locker and has a One-to-One relationship with **Instructor**.

```

@Entity
public class Locker {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Long lockerId;

    @Column(name = "LockerNumber")
    String lockerNumber;

    @OneToOne(mappedBy = "locker")
    @JsonBackReference
    Instructor instructor;
}

```

- **Member Entity:**
 - Represents a gym member and has a Many-to-One relationship with **Class**.

```

@Entity
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Long memberId;

    String memberName;
    String memberPhone;

    @ManyToOne
    @JsonBackReference
    @JoinColumn(name = "classId")
    Class aClass;
}

```

Repository:

The repository layer in a Spring Boot application typically serves as an interface between your application's business logic and the underlying database. It is responsible for performing database operations such as querying, saving, updating, and deleting entities. In the context of the GymFitness project you provided, the repository layer is crucial for managing the interaction with the database for the various entities (e.g., **Class**, **Instructor**, **Locker**, **Member**).

```

public interface ClassRepository extends JpaRepository<Class, Long> {
    List<Class> findByInstructor_InstructorId(Long instructorId);
}

```

```
public interface InstructorRepository extends JpaRepository<Instructor,Long> {  
    Optional<Instructor> findByLocker_LockerNumber(String lockerNumber);  
}
```

```
public interface LockerRepository extends JpaRepository<Locker,Long> {  
}
```

```
public interface MemberRepository extends JpaRepository<Member,Long> {  
}
```

Controller:

The controller layer in a Spring Boot application is responsible for handling incoming HTTP requests, processing them, and returning appropriate responses. It serves as the entry point for external requests and acts as an intermediary between the client (e.g., a web browser or a mobile app) and the business logic of your application. In the context of the GymFitness project you provided, the controllers manage the different functionalities related to gym management for classes, instructors, lockers, and members.

For Controller CRUD operation methods will be implemented:

- 1- Create (save) as @POST
- 2- Read (get) / as @GET
- 3- Update / as @PUT
- 4- Delete / as @DELETE

For all entities in this project, in addition to a special method will be demonstrated in this report.

@GET is used for data retrieval, and it appends parameters to the URL.

@POST is used for data submission, and it sends parameters in the request body.

@PUT is used for updating or creating resources, and it typically sends the full representation of the resource.

@DELETE is used for requesting the removal of a resource identified by the URI.

Class Controller:

```
@RestController
@RequestMapping("/GymClass")
public class ClassController {

    @Autowired
    ClassRepository classRep;
    @Autowired
    MemberRepository memberRep;

    @GetMapping("/Classes")
    public List<Class> getClasses(){
        return classRep.findAll();
    }

    @PostMapping("/newClass")
    public String saveClass(@RequestBody Class cls){
        classRep.save(cls);
        return "Class Saved";
    }

    @GetMapping("/{classId}")
    public Optional<Class> getClass(@PathVariable Long classId){
        return classRep.findById(classId);
    }

    @DeleteMapping("remove/{classId}")
    public String deleteClass(@PathVariable Long classId){
        classRep.deleteByld(classId);
        return "class deleted";
    }

    @PutMapping("update/{classId}")
    public ResponseEntity<Class> updateClass(@PathVariable Long classId,
    @RequestBody Class updatedClass) {
        Optional<Class> existingClassOptional = classRep.findById(classId);

        if (existingClassOptional.isPresent()) {
            Class existingClass = existingClassOptional.get();
            existingClass.setClassName(updatedClass.getClassName());
            existingClass.setMaxCapacity(updatedClass.getMaxCapacity());
            existingClass.setInstructor(updatedClass.getInstructor());

            Class updatedEntity = classRep.save(existingClass);
            return new ResponseEntity<>(updatedEntity, HttpStatus.OK);
        } else {
```

```

        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}

@GetMapping("members/{instructorId}")
public List<Member> getMembersByInstructor(@PathVariable Long instructorId) {
    List<Class> classes = classRep.findByInstructor_InstructorId(instructorId);

    List<Member> members = new ArrayList<>();
    for (Class gymClass : classes) {
        List<Member> classMembers = gymClass.getMembers();
        if (classMembers != null) {
            members.addAll(classMembers);
        }
    }

    return members;
}
}

```

- **GET /classes** to returns a list of all gym classes.
- **POST /classes** to creates a new gym class.
- **GET /classes/{classId}** to retrieves information about a specific gym class.
- **PUT /classes/update/{classId}** to updates information for a specific gym class.
- **DELETE /classes/remove/{classId}** to deletes a gym class.
- **GET /classes/members/{instructorId}** to retrieves members associated with a specific instructor.

In addition to special method which is getting members by instructor id throw Class knowing that there is no relation between Member and Instructor So Logically Class entity is the Relation in this case.

- **GET /classes/members/{instructorId}** to retrieves members associated with a specific instructor.

Instructor Controller:

```

@RestController
@RequestMapping("/Instructor")

public class InstructorController {

    @Autowired
    InstructorRepository instructorRep;

    @GetMapping("/Instructors")
    public List<Instructor> getInstructors() {

```

```

        return instructorRep.findAll();
    }

    @GetMapping("/{InstructorId}")
    public Optional<Instructor> getInstructor(@PathVariable Long InstructorId) {
        return instructorRep.findById(InstructorId);
    }

    @PostMapping("/newInstructor")
    public String saveInstructor(@RequestBody Instructor inst) {
        instructorRep.save(inst);
        return "Class Saved";
    }

    @DeleteMapping("/remove/{InstructorId}")
    public String deleteInstructor(@PathVariable Long InstructorId) {
        instructorRep.deleteById(InstructorId);
        return "Instructor deleted";
    }

    @PutMapping("/update/{instructorId}")
    public ResponseEntity<Instructor> updateInstructor(@PathVariable Long instructorId,
        @RequestBody Instructor upInstructor) {
        Optional<Instructor> exInstructorOptional = instructorRep.findById(instructorId);

        if (exInstructorOptional.isPresent()) {
            Instructor exInstructor = exInstructorOptional.get();
            exInstructor.setInstructorName(upInstructor.getInstructorName());
            exInstructor.setSpecialization(upInstructor.getSpecialization());

            Instructor updatedEntity = instructorRep.save(exInstructor);
            return new ResponseEntity<>(updatedEntity, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }

    @GetMapping("/byLockerNumber/{lockerNumber}")
    public ResponseEntity<Instructor> getInstructorByLockerNumber(@PathVariable String
lockerNumber) {
        Optional<Instructor> instructorOptional =
instructorRep.findByLocker_LockerNumber(lockerNumber);

        return instructorOptional
            .map(instructor -> new ResponseEntity<>(instructor, HttpStatus.OK))
            .orElseGet(() -> new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }
}

```

- **GET /instructors** to returns a list of all gym instructors.
- **POST /instructors** to creates a new gym instructor.
- **GET /instructors/{instructorId}** to retrieves information about a specific gym instructor.
- **PUT /instructors/update/{instructorId}** to updates information for a specific gym instructor.
- **DELETE /instructors/remove/{instructorId}** to deletes a gym instructor.
- **GET /instructors/byLockerNumber/{lockerNumber}** to retrieves an instructor by locker number which has a one to one relation with Locker

Locker Controller:

```
@RestController
@RequestMapping("/Locker")

public class LockerController {
    @Autowired
    LockerRepository lockerRep;
    @GetMapping("Lockers")
    public List<Locker> getLockers(){
        return lockerRep.findAll();
    }

    @PostMapping("newLocker")
    public String saveMember(@RequestBody Locker locker) {
        lockerRep.save(locker);
        return "member saved";
    }

    @GetMapping("{lockerId}")
    public Optional<Locker> getLocker(@PathVariable Long lockerId){
        return lockerRep.findById(lockerId);
    }

    @PutMapping("/update/{lockerId}")
    public ResponseEntity<Locker> updateLocker(@PathVariable Long lockerId, @RequestBody
Locker upLocker) {
        Optional<Locker> exLockerOptional = lockerRep.findById(lockerId);

        if (exLockerOptional.isPresent()) {
            Locker exLocker = exLockerOptional.get();
            exLocker.setLockerNumber(upLocker.getLockerNumber());

            Locker updatedEntity = lockerRep.save(exLocker);
            return new ResponseEntity<>(updatedEntity, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
}
```



```

@DeleteMapping("remove/{LockerId}")
public String deleteLocker(@PathVariable Long LockerId) {
    lockerRep.deleteByld(LockerId);
    return "Locker deleted";
}
}

```

- GET /lockers to returns a list of all lockers.
- GET /lockers/{lockerId} to retrieves information about a specific locker.
- POST /newLocker to creates a new Locker.
- DELETE /Locker/remove/{LockerNum} to deletes a Locker.
- PUT /Locker/update/{lockerId} to updates information for a specific locker.

Member Controller:

```

@RestController
@RequestMapping("/Member")

public class MemberController {

    @Autowired
    MemberRepository memberRep;

    @GetMapping("/Members")
    public List<Member> getMembers(){
        return memberRep.findAll();
    }

    @GetMapping("/{memberId}")
    public Optional<Member> getMember(@PathVariable Long memberId){
        return memberRep.findByld(memberId);
    }

    @PostMapping("newMember")
    public String saveMember(@RequestBody Member member){
        memberRep.save(member);
        return "member saved";
    }

    @PutMapping("/update/{memberId}")
    public ResponseEntity<Member> updateMember(@PathVariable Long memberId,
    @RequestBody Member upMember) {
        Optional<Member> exMemberOptional = memberRep.findByld(memberId);

        if (exMemberOptional.isPresent()) {
            Member exMember = exMemberOptional.get();

```

```

        exMember.setMemberName(upMember.getMemberName());

        Member updatedEntity = memberRep.save(exMember);
        return new ResponseEntity<>(updatedEntity, HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}

@DeleteMapping("remove/{memberId}")
public String deleteMember(Long memberId){
    memberRep.deleteById(memberId);
    return "member deleted";
}
}

```

- **GET /members** to returns a list of all gym members.
- **GET /members/{memberId}** to retrieves information about a specific gym member.
- **POST /members** to creates a new gym member.
- **DELETE /members/remove/{memberId}** to deletes a gym member.
- **PUT /members/update/{memberId}** to updates information for a specific gym member.

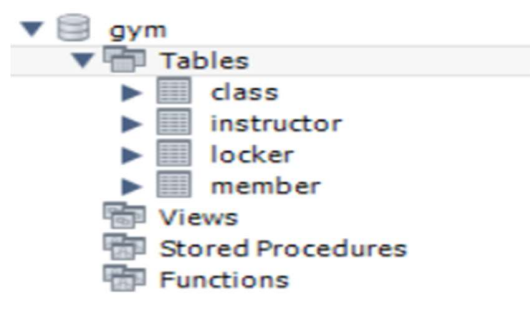
Running the Application:

- application properties must be set to create a connection to server and mysql .
- ```

spring.jpa.hibernate.ddl-auto = update
spring.main.banner-mode = off
spring.datasource.url = jdbc:mysql://localhost:3306/gym
spring.datasource.username = root
spring.datasource.password =
spring.jpa.show-sql = false
server.port=8088

```

When running the program, a new schema will be created and has the following structure:



Each table has the attributes of its entity where we can see and manipulate data.

## Code Output:

Member Entity output will be taken to show the procedure; other entities has same procedure.

**Reqbin** will be used as API Testing tool.

**getMember:**

GET ▾

EXT ▾

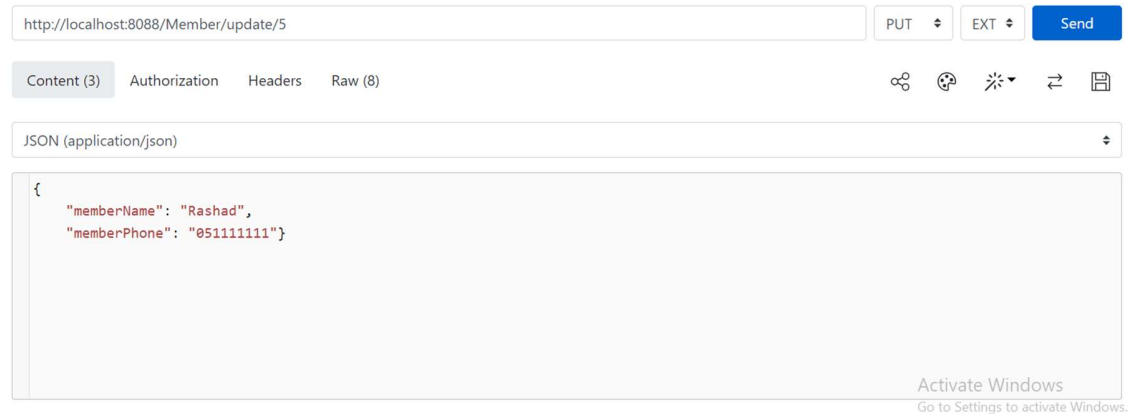
Send

**Output is:**

```
[{
 "memberId": 5,
 "memberName": "Ahmad",
 "memberPhone": "0599368410"
}, {
 "memberId": 6,
 "memberName": "Mohammed",
 "memberPhone": "0543440639"
}, {
 "memberId": 7,
 "memberName": "Yaseen",
 "memberPhone": "0547956959"
}]
```

## Update members:

To update member 5 name and number this method is used in riqbin.



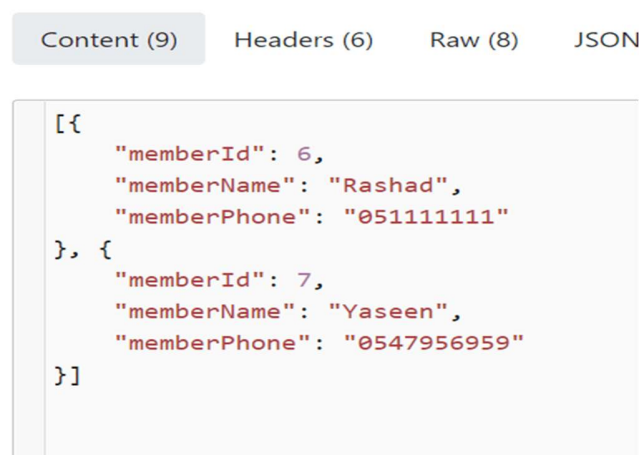
To obtain members from class using instructor id the method in Class controller is used:

`http://localhost:8088/GymClass/members/1`

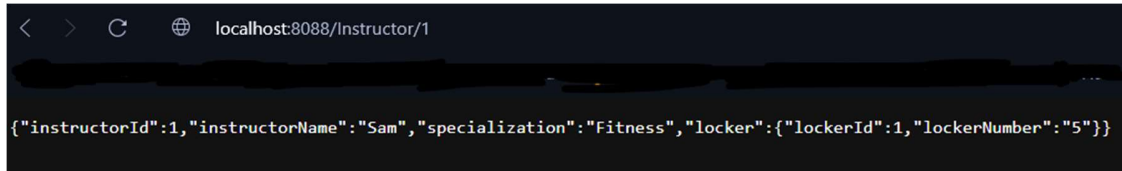


Only member from classes that instructor number 1 will be shown which is:

Status: 200 () Time: 19 ms Size: 0.13 |

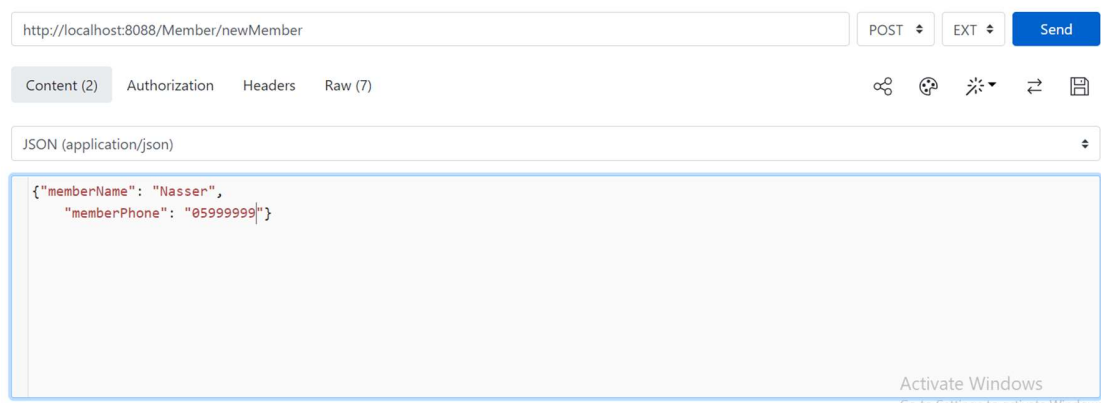


**Another request done by another way (browser search) which shows instructor by id and Locker to show the one to one relation.**



```
{"instructorId":1,"instructorName":"Sam","specialization":"Fitness","locker":{"lockerId":1,"lockerNumber":"5"}}
```

**Adding new member:**



And database will be like this:

| member_id | member_name | member_phone | class_id |
|-----------|-------------|--------------|----------|
| 6         | Rashad      | 051111111    | 1        |
| 7         | Yaseen      | 0547956959   | 1        |
| 8         | Sara        | 0544789593   | 2        |
| 9         | Nasser      | 05999999     | 2        |
| NULL      |             | NULL         | NULL     |

## Conclusion:

This document provides an overview of the GymFitness project, its structure, entity relationships, API endpoints, and instructions for running the application. For more detailed information, refer to the codebase and Javadoc comments in the source code.