



UNIVERSITETI I EVROPËS JUGLINDORE  
УНИВЕРЗИТЕТ НА ЈУГОИСТОЧНА ЕВРОПА  
SOUTH EAST EUROPEAN UNIVERSITY

**Faculty of Contemporary Sciences and Technologies**

**Course: Programming in Java**

**Project:**

# **DENTAL CLINIC MANAGEMENT SYSTEM**

**Students:**

**Hadisa Xheladin 128632**

**Kanita Jakupi 128633**

**Mentor:**

**Prof. Dr. Nuhi Besimi**

**Skopje, 2022**

## Contents

1.Abstract.....	3
1.1.Technologies used.....	3
1.2. The outcome of the project .....	4
2.Steps to run the application.....	4
2.1.Dependencies .....	6
2.2.DDL scripts .....	8
3.Usage.....	9
3.2.Queries.....	14
3.3.Examples in Postman .....	15
4.Unit Testing.....	19
Summary .....	20

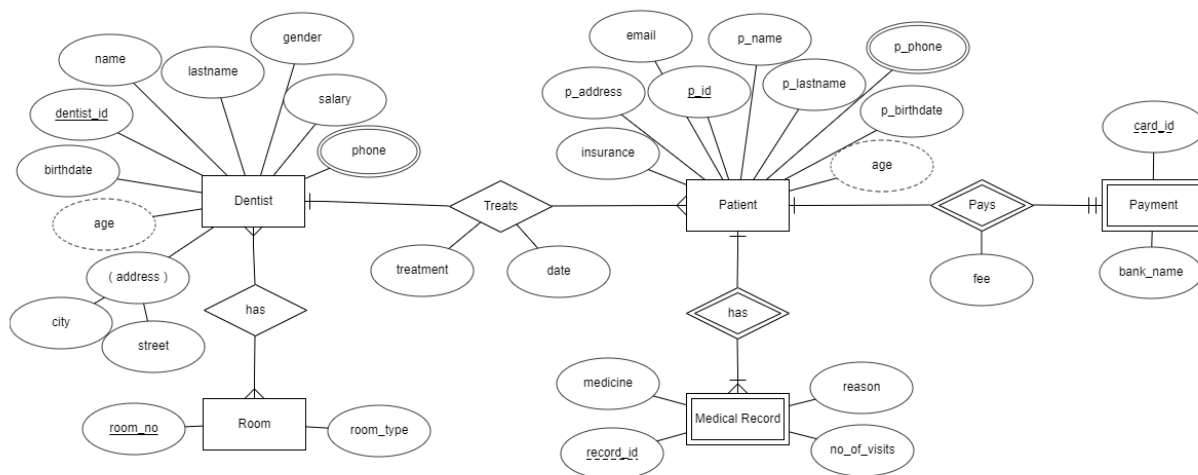
## 1.Abstract

In this semester we have decided to create a Rest API for patient controls management in the dental clinic.

We have carried out this project based on the given requirements as well as our personal preferences.

In this project we have tried to apply all the knowledge and material that we have learned during this semester.

With this project we think that we will achieve a proper goal, to facilitate the work of dentists in recording the examinations of their patients.



### 1.1.Technologies used

**IntelliJ IDEA Ultimate** - an Integrated Development Environment (IDE) for JVM languages designed to maximize developer productivity that we have used to implement the code of the project in this course.

**Postman** - an application that we have used for API testing. It is an HTTP client that tests HTTP requests, utilizing a graphical user interface, through which we have obtained different types of responses that need to be subsequently validated.

**PhpMyAdmin** - a free software tool written in PHP, intended to handle the administration of MySQL over the Web. We have used phpMyAdmin to perform most administration tasks, including creating a database and running queries required for the project.

**AnyDesk** - made it possible to interact with each other – no matter where we were. We shared our screens for presentations, brainstorm together by using the Whiteboard, and shared files through File Transfer. With AnyDesk, collaboration was possible with a few simple clicks.

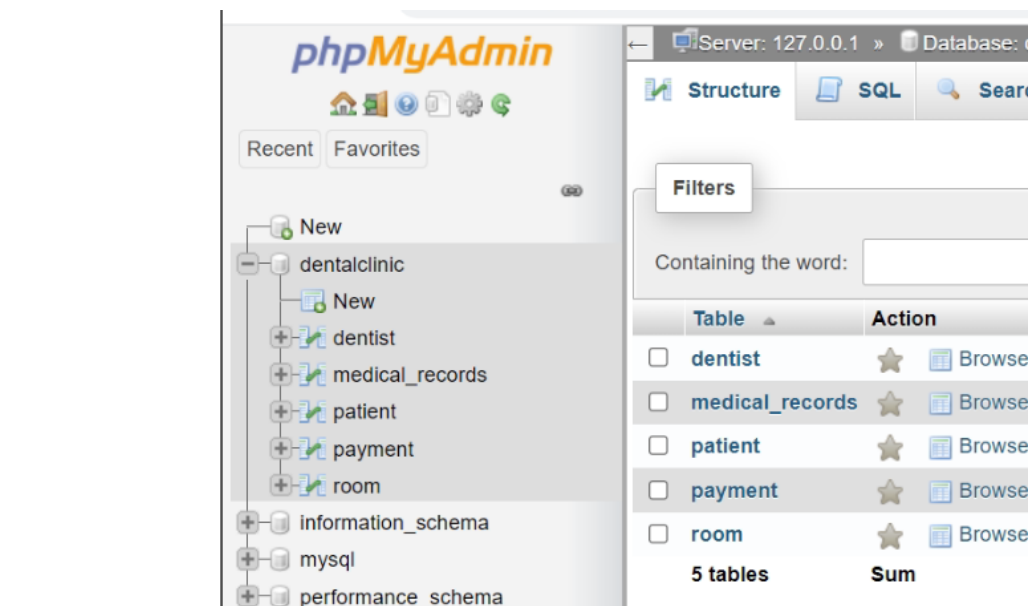
## 1.2. The outcome of the project

Our main goal with this project was to enable ease to a dentist working in a dental clinic and to a patient of that dentist. With this project there will be faster and easier access to the needed information (collection of data) during a dental clinic appointment.












## 2.Steps to run the application

First you should open the application from an IDE (IntelliJ Ultimate) and import the Dental Clinic project. You should open PhpMyAdmin and create a database with name dentalclinic. It should contain 5 tables: Dentist, Patient, Payment, Medical records and Room.












```
spring.datasource.url=jdbc:mysql://localhost:3306/dentalclinic
```











## Dentist Table Structure:

<div><div> Table structure</div><div> Relation view</div></div>									
#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
<input type="checkbox"/>	1 dentistId	int(30)			No	None		AUTO_INCREMENT	 Change  Drop  More
<input type="checkbox"/>	2 dentist_name	text	utf8mb4_general_ci		No	None			 Change  Drop  More
<input type="checkbox"/>	3 dentist_phone	int(30)			No	None			 Change  Drop  More











## Patient Table Structure:

<div><div> Table structure</div><div> Relation view</div></div>									
#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
<input type="checkbox"/>	1 patientId	int(30)			No	None		AUTO_INCREMENT	 Change  Drop  More
<input type="checkbox"/>	2 patient_name	text	utf8mb4_general_ci		No	None			 Change  Drop  More
<input type="checkbox"/>	3 patient_phone	int(30)			No	None			 Change  Drop  More

## Payment Table Structure:

<div><div> Table structure</div><div> Relation view</div></div>									
#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
<input type="checkbox"/>	1 paymentId	int(30)			No	None		AUTO_INCREMENT	 Change  Drop  More
<input type="checkbox"/>	2 total	int(30)			No	None			 Change  Drop  More

## Medical Records Table Structure:

<div><div> Table structure</div><div> Relation view</div></div>									
#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
<input type="checkbox"/>	1 recordId	int(30)			No	None		AUTO_INCREMENT	 Change  Drop  More
<input type="checkbox"/>	2 reason	text	utf8mb4_general_ci		No	None			 Change  Drop  More
<input type="checkbox"/>	3 medicine	text	utf8mb4_general_ci		No	None			 Change  Drop  More

## Room Table Structure:

+ Options

				roomId	room_type	room_no	dentist_id
<input type="checkbox"/>	 Edit	 Copy	 Delete	1	sterilization room	1010	1
<input type="checkbox"/>	 Edit	 Copy	 Delete	2	surgery room	1011	5
<input type="checkbox"/>	 Edit	 Copy	 Delete	3	dental work room	1012	4
<input type="checkbox"/>	 Edit	 Copy	 Delete	4	dental treatment room	1013	6

### 2.1. Dependencies

In the pom.xml file of our project we have added the following dependencies:

#### spring-boot-starter-web

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

It is used for building the web application, including RESTful applications using Spring MVC. It uses Tomcat as the default embedded container. spring-boot-starter-data-gemfire. It is used to GemFire distributed data store and Spring Data GemFire.

#### mysql-connector-java

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
```

MySQL provides connectivity for client applications developed in the Java programming language with MySQL Connector/J, a driver that implements the Java Database Connectivity (JDBC) API. MySQL Connector/J is a JDBC Type 4 driver. Different versions are available that are compatible with the JDBC 3.0 and JDBC 4.

### spring-boot-starter-test

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
```

Spring Boot provides a number of utilities and annotations to help when testing your application. Test support is provided by two modules: spring-boot-test contains core items, and spring-boot-test-autoconfigure supports auto-configuration for tests.

### spring-boot-starter-data-jpa

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Spring Boot provides spring-boot-starter-data-jpa dependency to connect Spring application with relational database efficiently. The spring-boot-starter-data-jpa internally uses the spring-boot-jpa dependency.

### hibernate-core

```
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
</dependency>
```

Hibernate is an open source Java persistence framework project. Perform powerful object relational mapping and query databases using HQL and SQL.

### junit

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
  <scope>compile</scope>
</dependency>
```

JUnit is the testing framework that is extensively used for java projects built in the maven project format for unit testing purposes. For our project to use JUnit features, we need to add JUnit as a dependency.

## Maven Plugin

```
<plugins>
  <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>2.5.8</version>
  </plugin>
</plugins>

</build>
```

"Maven" is really just a core framework for a collection of Maven Plugins. In other words, plugins are where much of the real action is performed, plugins are used to: create jar files, create war files, compile code, unit test code, create project documentation, and on and on.

## 2.2.DDL scripts

### Dentist Table

*INSERT INTO `dentist` (`dentistId`, `dentist name`, `dentist phone`) VALUES ('', '', '');*

```
INSERT INTO Dentist
VALUES
('1', 'Ana', 888701112),
('2', 'Maria', 389442255),
('3', 'Fisnik', 779966442),
('4', 'Mara', 888777444),
('5', 'Din', 999333222),
('6', 'Dua', 389701144)
```

### Patient Table

*INSERT INTO `patient` (`patientId`, `patient name`, `patient phone`, `dentistId`,) VALUES ('', '', '', '');*

```
INSERT INTO Patient
VALUES
('1', 'Hadisa', 789944551, '1'),
('2', 'Kanita', 885544332, '2'),
('3', 'Sara', 554422113, '3'),
('4', 'Aurora', 447723457, '1'),
('5', 'Dea', 389345876, '5'),
('6', 'Sulejma', 78339445, '6')
```



### Payment Table

*INSERT INTO `payment`(`paymentId`, `total`, `patientId`) VALUES ("","")*

```
INSERT INTO Payment
VALUES
```

```
('333', '600', '3'),
('444', '1500', '2'),
('111', '2000', '1'),
('222', '3000', '4'),
('448', '3600', '5'),
('449', '4300', '6')
```

### Medical records Table

*INSERT INTO `medical records`(`recordId`, `reason`, `medicine`, `patientId`) VALUES ("","","")*

```
INSERT INTO Medical_Record
VALUES
```

```
('123', 'malocclusions', 'braces', '1'),
('124', 'cavities', 'filling', '2'),
('125', 'weak tooth', 'crown', '3'),
('126', 'stains', 'teeth whitening', '4'),
('127', 'Teeth Grinding', 'Bonding Repair', '5'),
('128', 'Chipped Tooth', 'Filling', '6')
```

### Room Table

*INSERT INTO `room`(`roomId`, `room type`, `room no`, `dentist id`) VALUES ("","","")*

```
INSERT INTO Room
VALUES
```

```
('1', 'sterilization room', '1010', '1'),
('2', 'surgery room', '1011', '5'),
('3', 'dental work room', '1012', '4'),
('4', 'dental treatment room', '1013', '6')
```

## 3.Usage

The @GetMapping annotation is a specialized version of @RequestMapping annotation that acts as a shortcut for @RequestMapping(method = RequestMethod. GET) . The @GetMapping annotated methods in the @Controller annotated classes handle the HTTP GET requests matched with given URI expression.Annotation for mapping HTTP GET requests onto specific handler methods.

Annotation for mapping HTTP POST requests onto specific handler methods. Specifically, `@PostMapping` is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod. POST)`.

Annotation for mapping HTTP PUT requests onto specific handler methods. Specifically, `@PutMapping` is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod. PUT)`.

Annotation for mapping HTTP DELETE requests onto specific handler methods. Specifically, `@DeleteMapping` is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod. DELETE)`.

### **Dentist Controller**

For the Dentist Controller we have the following annotations:

```
@GetMapping("/dentist")
public List<Dentist> getAllDentist() { return dentistService.findAll(); }
```

```
@GetMapping("/{dentist_id}")
public Dentist getDentistById(@PathVariable Integer dentist_id) {
    return dentistService.findOneById(dentist_id);
}
```

```
@PostMapping("/dentist")
public Dentist insertNewDentist(@RequestBody DentistInput dentistInput) {
    return dentistService.save(dentistInput);
}
```

```

    @PutMapping("/{dentist/{dentist_id}")
    public Dentist updateDentist(@PathVariable Integer dentist_id,
                                @RequestBody DentistInput dentistInput) {

        return dentistService.update(dentist_id, dentistInput);
    }

```

```

    @DeleteMapping("/{dentist/{dentist_id}")
    public Boolean deleteDentist(@PathVariable Integer dentist_id) {
        dentistService.delete(dentist_id);
        return true;
    }

```

```

    @GetMapping("/{dentist/name/{dentistName}")
    public List<Dentist> findDentistByDentistName(@PathVariable String dentistName) {
        return dentistService.findDentistByDentistName(dentistName);
    }
}

```

## **Patient Controller**

For the Patient Controller we have the following annotations:

```

    @GetMapping("/{patient}")
    public List<Patient> getAllPatient() { return patientService.findAll(); }

```

```

    @GetMapping("/{patient/{patient_id}")
    public Patient getPatientById(@PathVariable Integer patient_id) {
        return patientService.findOneById(patient_id);
    }

```

```

@PostMapping("/{patient}")
public Patient insertNewPatient(@RequestBody PatientInput patientInput) {
    return patientService.save(patientInput);
}

```

```

@PutMapping("/{patient}/{patient_id}")
public Patient updatePatient(@PathVariable Integer patient_id,
    @RequestBody PatientInput patientInput) {
    return patientService.update(patient_id, patientInput);
}

```

```

@DeleteMapping("/{patient}/{patient_id}")
public Boolean deletePatient(@PathVariable Integer patient_id) {
    patientService.delete(patient_id);
    return true;
}

```

```

@GetMapping("/{patient/name}/{patientName}")
public List<Patient> findPatientByPatientName(@PathVariable String patientName) {
    return patientService.findPatientByPatientName(patientName);
}

```

## Payment Controller

For the Payment Controller we have the following annotations:

```

@GetMapping("/{payment}")
public List<Payment> findAll() { return paymentService.findAll(); }

```

```

@GetMapping("/{payment/{payment_id}")
public Payment findOne(@PathVariable Integer payment_id) { return paymentService.findOneById(payment_id); }

```

```

@GetMapping("/{payment/{patientId}")
public List<Payment> findPaymentByPatientId(@PathVariable Integer patientId) {
    return paymentService.findPaymentByPatientId(patientId);
}

```

## Medical records Controller

For the Medical\_records Controller we have the following annotations:

```

@GetMapping("/medical_records")
public List<Medical_records> findAll() { return medical_recordsRepository.findAll(); }

```

```

@GetMapping("/medical_records/{record_id}")
public Medical_records findOne(@PathVariable Integer record_id) { return medical_recordsRepository.findMedicalRecordByRecordId(record_id); }
}

```

## Room Controller

For the Room Controller we have the following annotations:

```

@GetMapping("/room")
public List<Room> findAll() { return roomService.findAll(); }

```

```

@GetMapping("/room/{room_id}")
public Room findOne(@PathVariable Integer room_id) { return roomService.findOneById(room_id); }
}

```

### 3.2.Queries

In order to define SQL to execute for a Spring Data repository method, we can annotate the method with the `@Query` annotation — its value attribute contains the JPQL or SQL to execute. Querying a database means searching through its data. You do so by sending SQL statements to the database. To do so, you first need an open database connection. Queries help you find and work with your data. A query can either be a request for data results from your database or for action on the data, or for both. A query can give you an answer to a simple question, perform calculations, combine data from different tables, add, change, or delete data from a database.

#### Dentist Repository

For the Dentist Repository we have created the following query:

```
@Query(
    "SELECT d FROM Dentist d WHERE d.dentistName = :dentistName"
)
List<Dentist> findDentistByDentistName(String dentistName);
}
```

This query finds the dentists based on the name given.

#### Patient Repository

For the Patient Repository we have created the following query:

```
@Query(
    "SELECT p FROM Patient p WHERE p.patientName = :patientName"
)
List<Patient> findPatientByPatientName(String patientName);
}
```

This query finds the patients based on the name given.

```
@Query(
    "SELECT t FROM Patient t "+
    "JOIN FETCH Payment p ON p.patient.patientId = t.patientId "+
    "WHERE p.paymentId = :paymentId"
)
List<Patient> findPatientByPaymentId(@Param("paymentId") Integer paymentId);
}
```

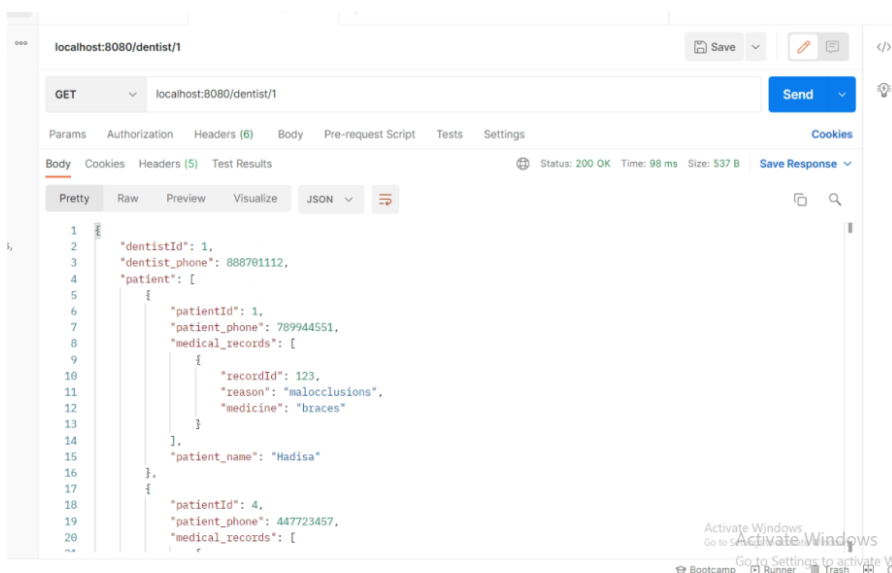
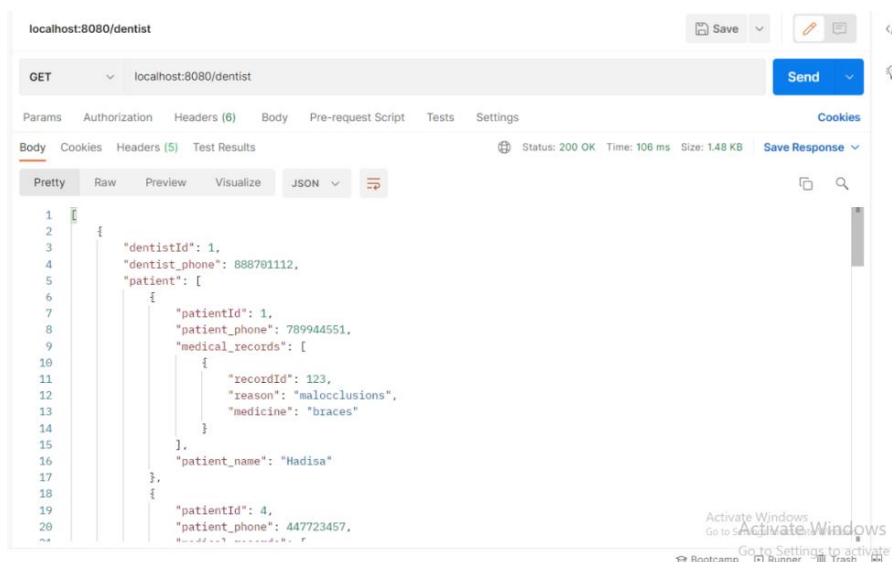
This query finds the patients based on the paymentId.

### 3.3.Examples in Postman

Postman makes it easy to create and send API requests. Send a request to test a endpoint, retrieve data from a data source, or try out an API's functionality.

#### GET

A GET request gets the information from the server. When you make the GET request on the server, then the server responds to the request. GET request will not affect any data on the server. Means, there is no creation, updation, addition, or deletion of data on the server when you are making a GET request.



localhost:8080/patient/name/Hadisa

GET localhost:8080/patient/name/Hadisa

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Body Cookies Headers (5) Test Results Status: 200 OK Time: 77 ms Size: 313 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "patientId": 1,
3   "patient_phone": 789944551,
4   "medical_records": [
5     {
6       "recordId": 123,
7       "reason": "malocclusions",
8       "medicine": "braces"
9     }
10  ],
11   "patient_name": "Hadisa"
12 }
```

localhost:8080/patient/1

GET localhost:8080/patient/1

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Body Cookies Headers (5) Test Results Status: 200 OK Time: 31 ms Size: 311 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "patientId": 1,
3   "patient_phone": 789944551,
4   "medical_records": [
5     {
6       "recordId": 123,
7       "reason": "malocclusions",
8       "medicine": "braces"
9     }
10  ],
11   "patient_name": "Hadisa"
12 }
```

GET localhost:8080/dentist/name/Mara

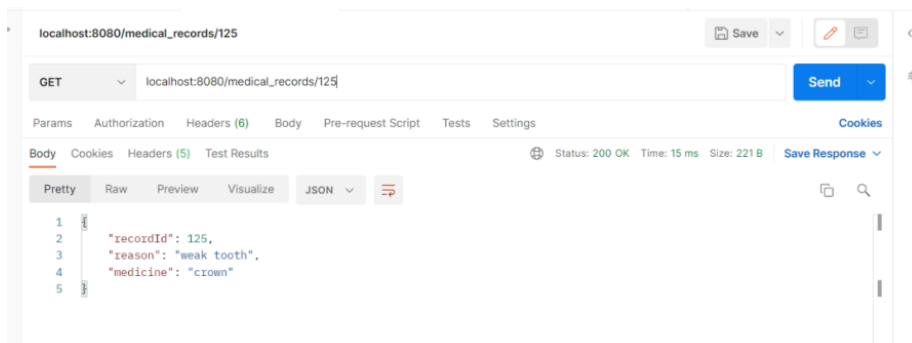
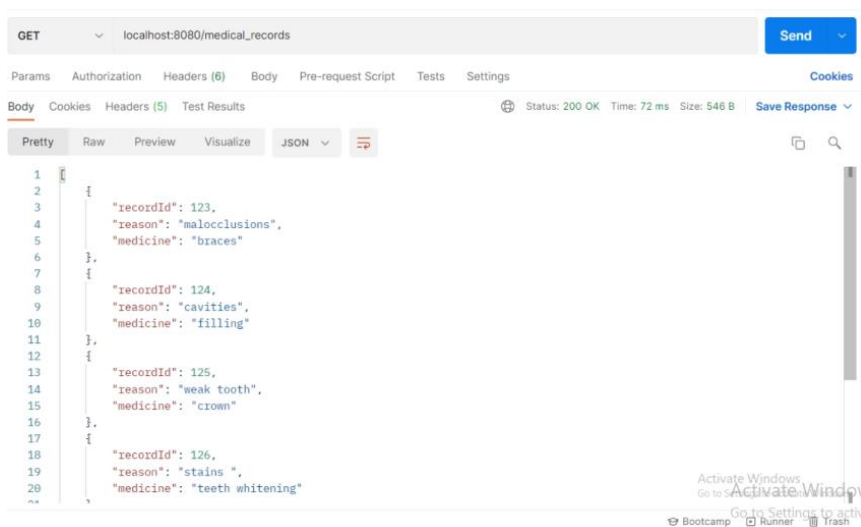
Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Body Cookies Headers (5) Test Results Status: 200 OK Time: 26 ms Size: 392 B Save Response

Pretty Raw Preview Visualize JSON

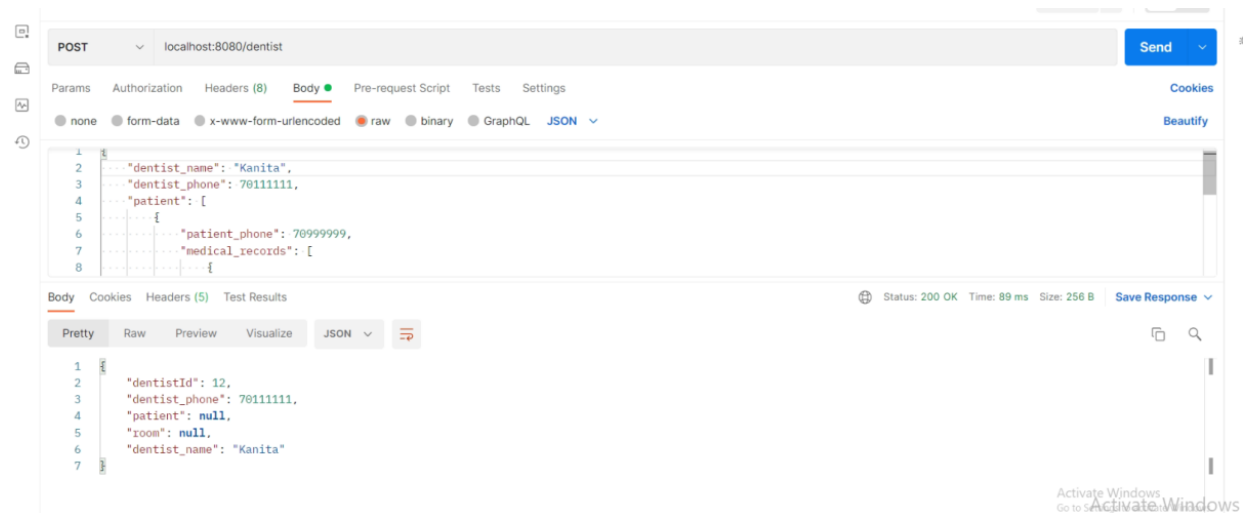
```
1 {
2   "dentistId": 4,
3   "dentist_phone": 888777444,
4   "patient": [
5     {
6       "patientId": 4,
7       "patient_phone": 447723457,
8       "medical_records": [
9         {
10          "recordId": 126,
11          "reason": "stains ",
12          "medicine": "teeth whitening"
13        }
14      ],
15      "patient_name": "Auroza"
16    }
17  ],
18   "dentist_name": "Mara"
19 }
20
```





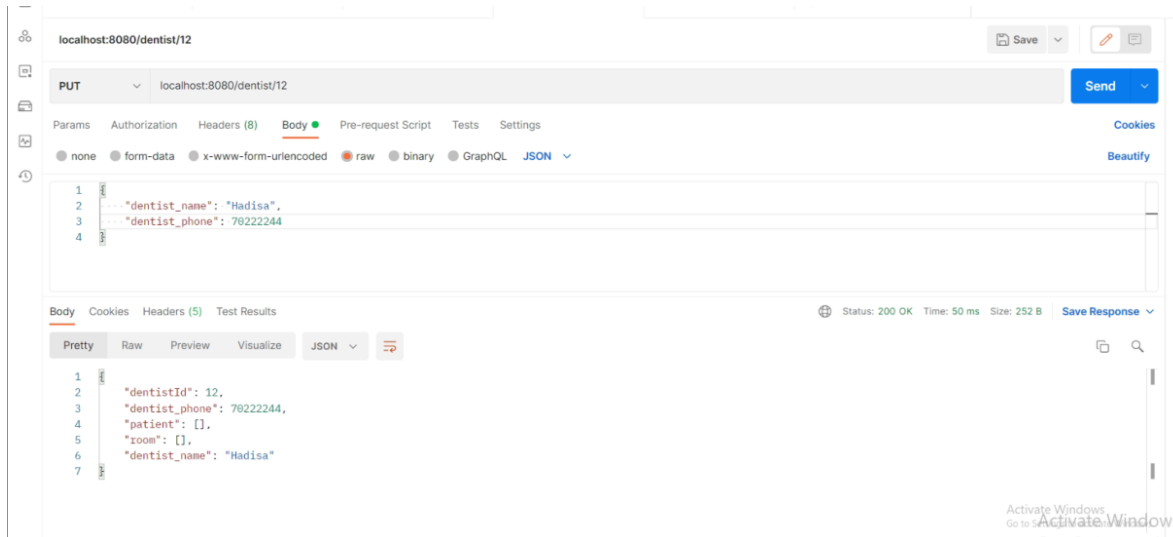
## POST

The post is an HTTP method like GET. We use this method when additional information needs to be sent to the server inside the body of the request. In general, when we submit a POST request, we expect to have some change on the server, such as updating, removing or inserting.



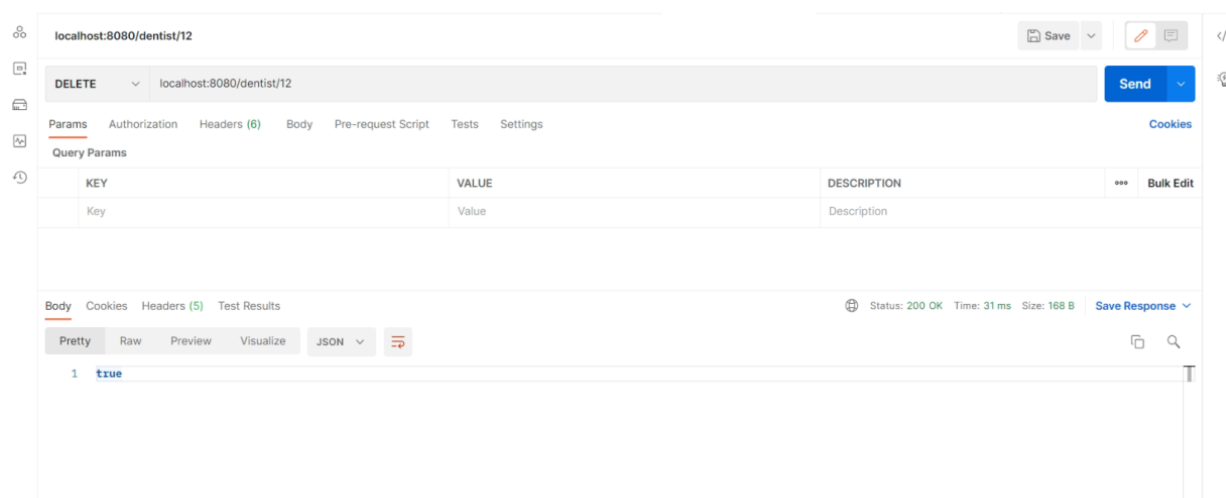
## PUT

A Postman PUT request is used to pass data to the server for the creation or modification of a resource. The difference between POST and PUT is that POST request is not idempotent. This means invoking the same PUT request numerous times will always yield the same output.



## DELETE

Postman DELETE request deletes a resource already present in the server. The DELETE method sends a request to the server for deleting the request mentioned in the endpoint. Thus it is capable of updating data on the server.



## 4. Unit Testing

Unit testing is a software testing method where “units”—the individual components of software—are tested. Developers write unit tests for their code to make sure that the code works correctly. This helps to detect and protect against bugs in the future.

```
@SpringBootTest
public class DentalClinicApplicationTests {

    @Autowired
    private DentistRepository dentistRepository;
```

We have created the “repo” pointers with the @Autowired annotation tag. . Before every test method we have used the @Test annotation.

```
@Test
@Order(1)
public void testInsertDentist() {
    Dentist dentist = new Dentist( dentist_id: 7,  dentist_name: "Nuh",  dentist_phone: 776634521);
    dentistRepository.save(dentist);
    assertNotNull(dentistRepository.findDentistById(7));
}
```

**testInsertDentist()** method to insert the dentist in the database, we create a new dentist object, save the object through save(). We have used assertNotNull to determine the findDentistById method is not returning null.

```
@Test
@Order(2)
public void testDentistAttribute() {
    Dentist dentist = dentistRepository.findDentistById(7);
    assertEquals( expected: "Nuh", dentist.getDentist_name());
}
```

**testDentistAttribute()** method to see if an attribute (name) has the correct value.

```

@Test
@Order(3)
public void testDentistUpdate() {
    Dentist dentist = dentistRepository.findDentistById(7);
    dentist.setDentist_name("Anisa");
    dentistRepository.save(dentist);
    assertEquals( unexpected: "Nuh", dentistRepository.findDentistById(7).getDentist_name());
}

```

**testDentistUpdate()** to check if the data was updated correctly.

```

@Test
@Order(4)
public void testDeleteDentist() {
    Dentist dentist = dentistRepository.findDentistById(7);
    dentistRepository.delete(dentist);
    assertNull(dentistRepository.findDentistById(7));
}

```

**testDeleteDentist()** to test the delete method.

## Summary

We have built a simple web application for a Dental Clinic with Spring Boot and learned how it can ramp up our development pace. We also turned on some handy production services. This is only a small sampling of what Spring Boot can do.

<https://github.com/KJakupi/DentalClinicApp.git>