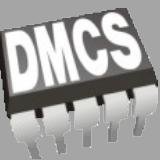




Interactive Web Applications



1. REST API – introduction and constraints
2. HTTP – methods and status codes
3. RESTful API example
4. Spring Boot – REST example, Postman

1

REST API – introduction and constraints

Definition

Resource-based

Representations

Constraints

REST - „REpresentational State Transfer”

Web services provide interoperability between computer systems on the Internet. REST-compliant web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations.

REST - „REpresentational State Transfer”

Roy Fielding defined REST in his 2000 PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures" at UC Irvine. He developed the REST architectural style in parallel with HTTP 1.1 of 1996–1999, based on the existing design of HTTP 1.0 of 1996.

Representations

- The resources states transferred between client and server
- Example:
 - Resource: person
 - Service: GET contact information
 - Representation: name, address, phone number – JSON or XML format

JSON Example

JavaScript Object Notation

```
{ "employees": [
  { "firstName": "John", "lastName": "Doe" },
  { "firstName": "Anna", "lastName": "Smith" },
  { "firstName": "Peter", "lastName": "Jones" }
]}
```

XML Example

Extensible Markup Language

```
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```

Client-Server
Layered system
Uniform Interface
Stateless
Cacheable
Code on demand

Client-Server

- A disconnected system – no direct connections to DB, assets or resources
- Separation of concerns
- Uniform interface – the link between the client-server

Layered system

- Client can not assume direct connection to server
- Software and/or hardware intermediaries between client and server
- Improves scalability

Uniform Interface

- Defines the interface between client and server
- Simplifies the architecture
- Fundamental to RESTful design
- Main assumptions:
 - HTTP methods (GET, POST, PUT, PATCH, DELETE)
 - URIs (resource name)
 - HTTP response (status, body)

<https://restapi.example.com/resources>

Stateless

- Server contains no client state
- Each request contains enough information to process the message
- Any session state is held on the client side

Cacheable

- Server responses (representations) are cacheable
 - implicitly
 - explicitly
 - negotiated

Code on demand

- Server can transfer logic to client
- Client executes logic
- Examples: Java applets, JavaScript technologies
- The only optional constraint

Summary

- Violating any constraint other than „Code on demand” means service is not strictly RESTful; it is only designed in RESTful like fashion
- Compliance with REST constraints allows:
Scalability, Simplicity, Modifiability, Visibility,
Portability, Reliability

2

HTTP – methods and status codes

The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, and hypermedia information systems.

HTTP is the foundation of data communication for the World Wide Web.

```
GET /doc/test.html HTTP/1.1
```

```
Host: www.test101.com
```

```
Accept: image/gif, image/jpeg, */*
```

```
Accept-Language: en-us
```

```
Accept-Encoding: gzip, deflate
```

```
User-Agent: Mozilla/4.0
```

```
Content-Length: 35
```

```
bookId=12345&author=Tan+Ah+Teck
```

Request Line

Request Headers

Request
Message
Header

A blank line separates header & body

Request Message Body

```
HTTP/1.1 200 OK
```

```
Date: Sun, 08 Feb xxxx 01:11:12 GMT
```

```
Server: Apache/1.3.29 (Win32)
```

```
Last-Modified: Sat, 07 Feb xxxx
```

```
ETag: "0-23-4024c3a5"
```

```
Accept-Ranges: bytes
```

```
Content-Length: 35
```

```
Connection: close
```

```
Content-Type: text/html
```

```
<h1>My Home page</h1>
```

Status Line

Response Headers

Response
Message
Header

A blank line separates header & body

Response Message Body

GET – The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect. **(SAFE AND IDEMPOTENT)**

POST – The POST method requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI. The data POSTed might be, for example, an annotation for existing resources; a message for a bulletin board, newsgroup, mailing list, or comment thread; a block of data that is the result of submitting a web form to a data-handling process; or an item to add to a database. **(NOT SAFE AND NOT IDEMPOTENT)**

PUT – The PUT method requests that the enclosed entity be stored under the supplied URI. If the URI refers to an already existing resource, it is modified; if the URI does not point to an existing resource, then the server can create the resource with that URI. **(NOT SAFE BUT IDEMPOTENT)**

DELETE – The DELETE method deletes the specified resource. **(NOT SAFE BUT IDEMPOTENT)**

PATCH – The PATCH method applies partial modifications to a resource. **(NOT SAFE AND NOT IDEMPOTENT)**

HEAD – The HEAD method asks for a response identical to that of a GET request, but without the response body. This is useful for retrieving meta-information written in response headers, without having to transport the entire content. **(SAFE AND IDEMPOTENT)**

OPTIONS – The OPTIONS method returns the HTTP methods that the server supports for the specified URL. This can be used to check the functionality of a web server by requesting '*' instead of a specific resource. **(SAFE AND IDEMPOTENT)**

CONNECT – The CONNECT method converts the request connection to a transparent TCP/IP tunnel, usually to facilitate SSL-encrypted communication (HTTPS) through an unencrypted HTTP proxy. **(SAFE AND IDEMPOTENT)**

TRACE – The TRACE method (a debugging tool) echoes the received request so that a client can see what (if any) changes or additions have been made by intermediate servers. **(SAFE AND IDEMPOTENT)**

Informational 1XX

100 - Continue

101 - Switching Protocols

110 - Connection Timed Out

Successful 2XX

200 - OK

201 - Created

202 - Accepted

204 - No content

Redirection 3XX

301 - Moved Permanently

302 - Found

304 - Not Modified

Client Error 4XX

400 - Bad Request

401 - Unauthorized

403 - Forbidden

404 - Not Found

405 - Method Not Allowed

Server Error 5XX

500 - Internal Server Error

503 - Service Unavailable

3

RESTful API example

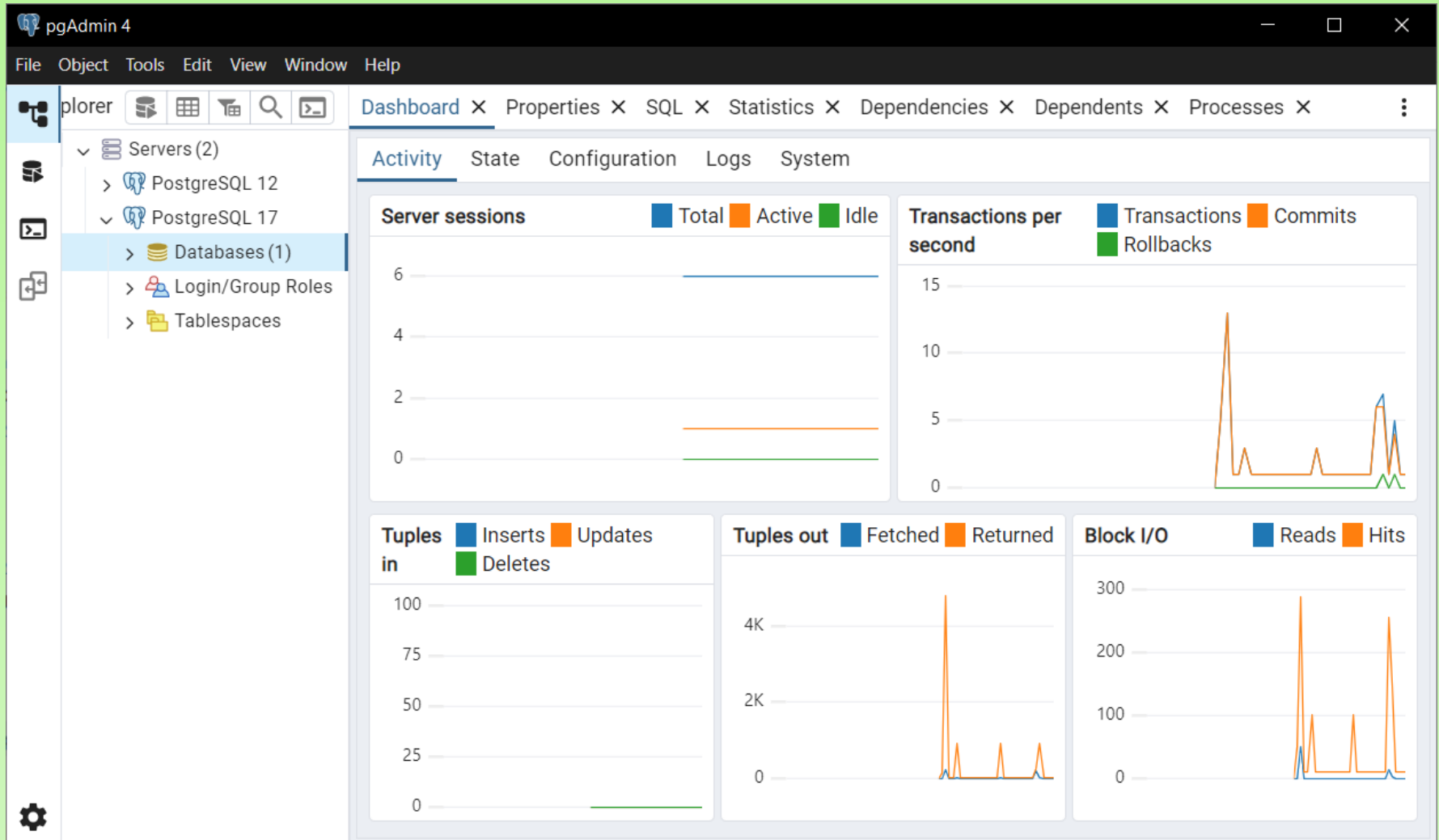
URL	GET	PUT
https://restapi.example.com/resources	List the URIs and perhaps other details of the collection's members.	Replace the entire collection with another collection.
https://restapi.example.com/resources/item_1	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	Replace the addressed member of the collection, or if it does not exist, create it.

URL	PATCH	POST	DELETE
https://restapi.example.com/resources	Not used	Create a new entry in the collection.	Delete the entire collection.
https://restapi.example.com/resources/item_1	Update the addressed member of the collection.	Not used	Delete the addressed member of the collection.

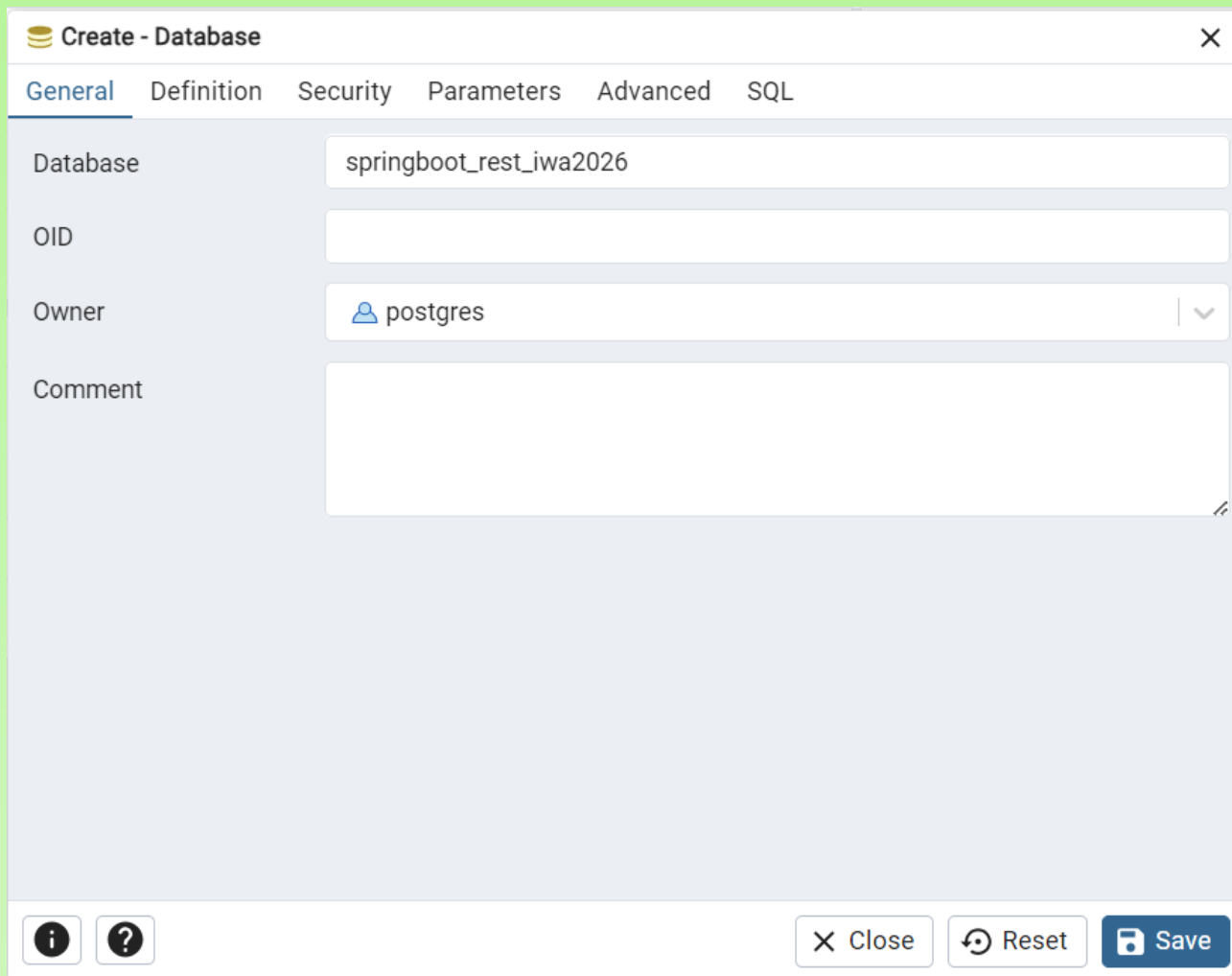
4

Spring Boot - REST example

To manage database you can run **pgAdmin4**.



Before running the application the database has to be created first.
To do that you can use pgAdmin tool.

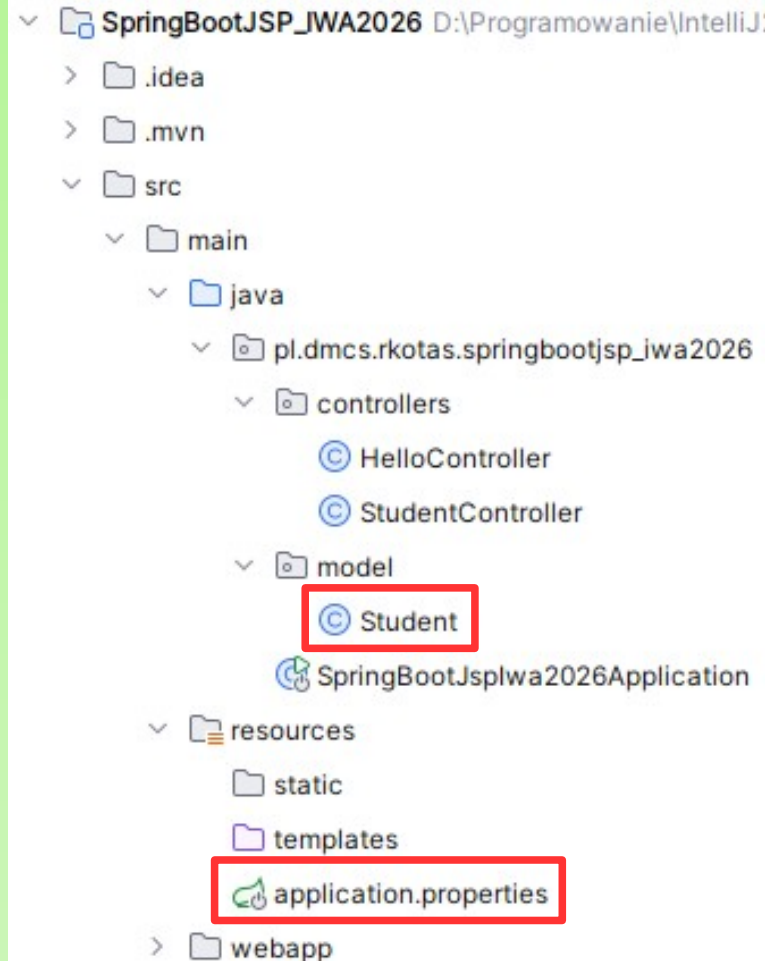


The image shows the 'Create - Database' dialog box in pgAdmin. The 'General' tab is selected, showing fields for Database name, OID, Owner, and Comment. The Database name is 'springboot_rest_iwa2026', the Owner is 'postgres', and the Comment field is empty. The dialog has tabs for General, Definition, Security, Parameters, Advanced, and SQL. At the bottom, there are buttons for Close, Reset, and Save.

Field	Value
Database	springboot_rest_iwa2026
OID	
Owner	postgres
Comment	

m pom.xml (SpringBootJSP_JWA2026) x

```
46  @  
47  
48  
49  
50  
51  @  
52  
53  
54  
55  @  
56  
57  
58  
59  
60  
  
    <dependency>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-data-jpa</artifactId>  
        <version>4.0.3</version>  
    </dependency>  
    <dependency>  
        <groupId>org.postgresql</groupId>  
        <artifactId>postgresql</artifactId>  
    </dependency>  
    <dependency>  
        <groupId>tools.jackson.dataformat</groupId>  
        <artifactId>jackson-dataformat-xml</artifactId>  
        <version>3.1.0</version>  
    </dependency>  
</dependencies>
```

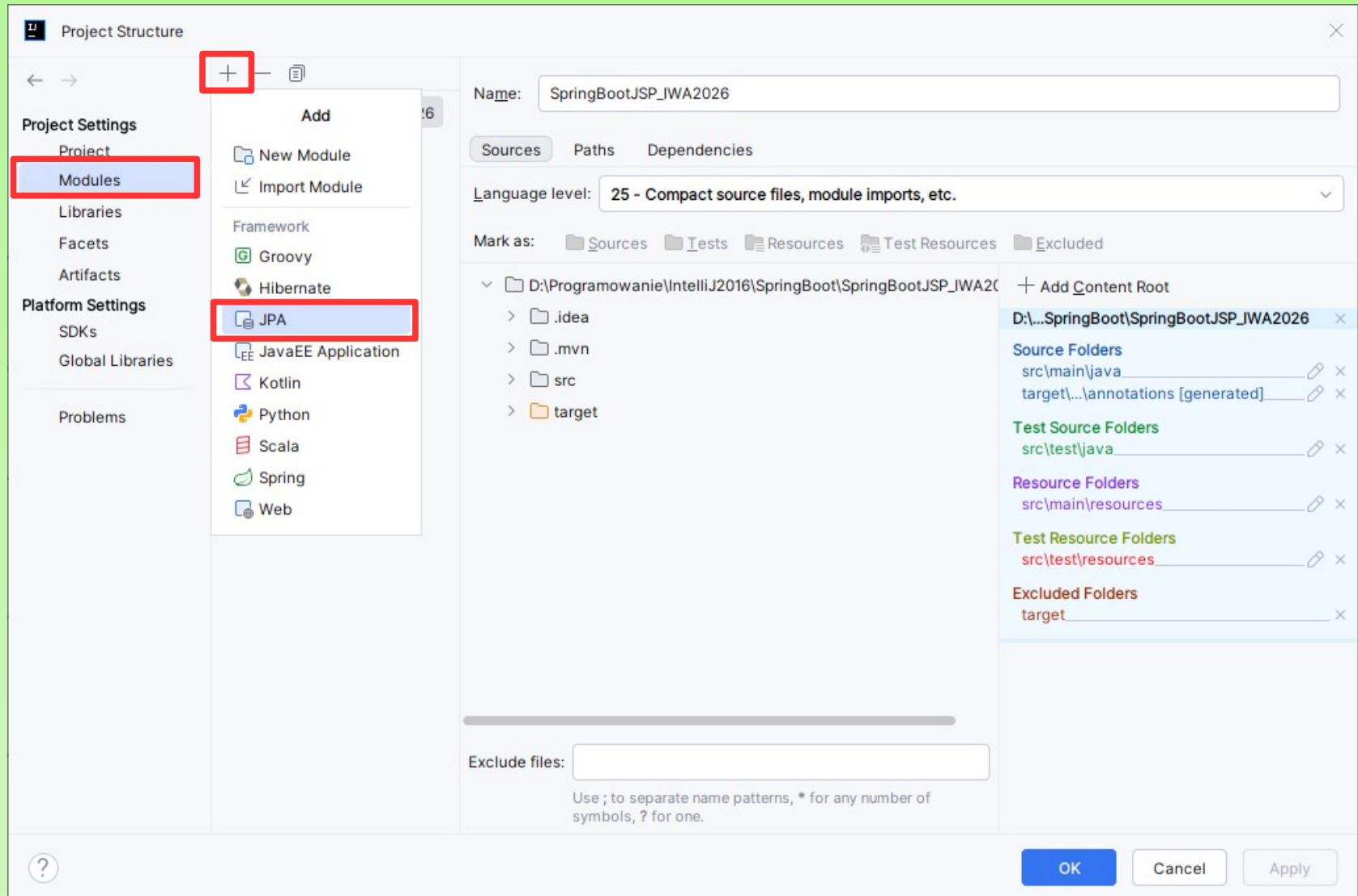



Spring Boot – REST example



34

dr inż. Rafał Kotas, rkotas@dmcs.pl

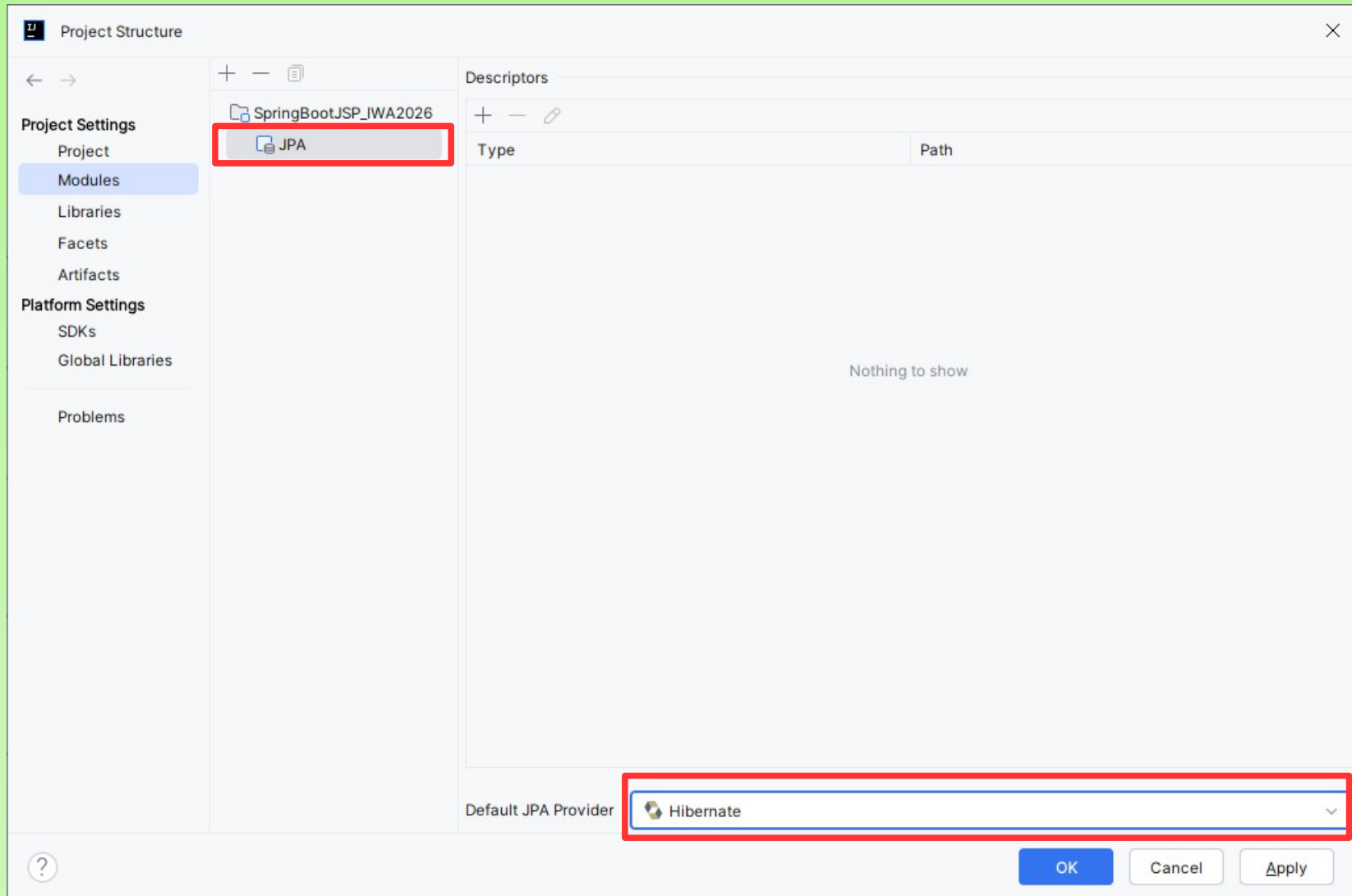


Spring Boot – REST example



35

dr inż. Rafał Kotas, rkotas@dmcs.pl



Spring Boot – REST example



36

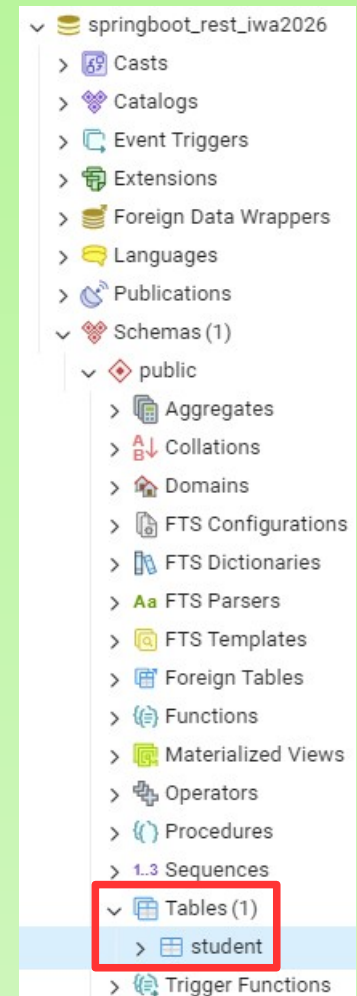
dr inż. Rafał Kotas, rkotas@dmcs.pl

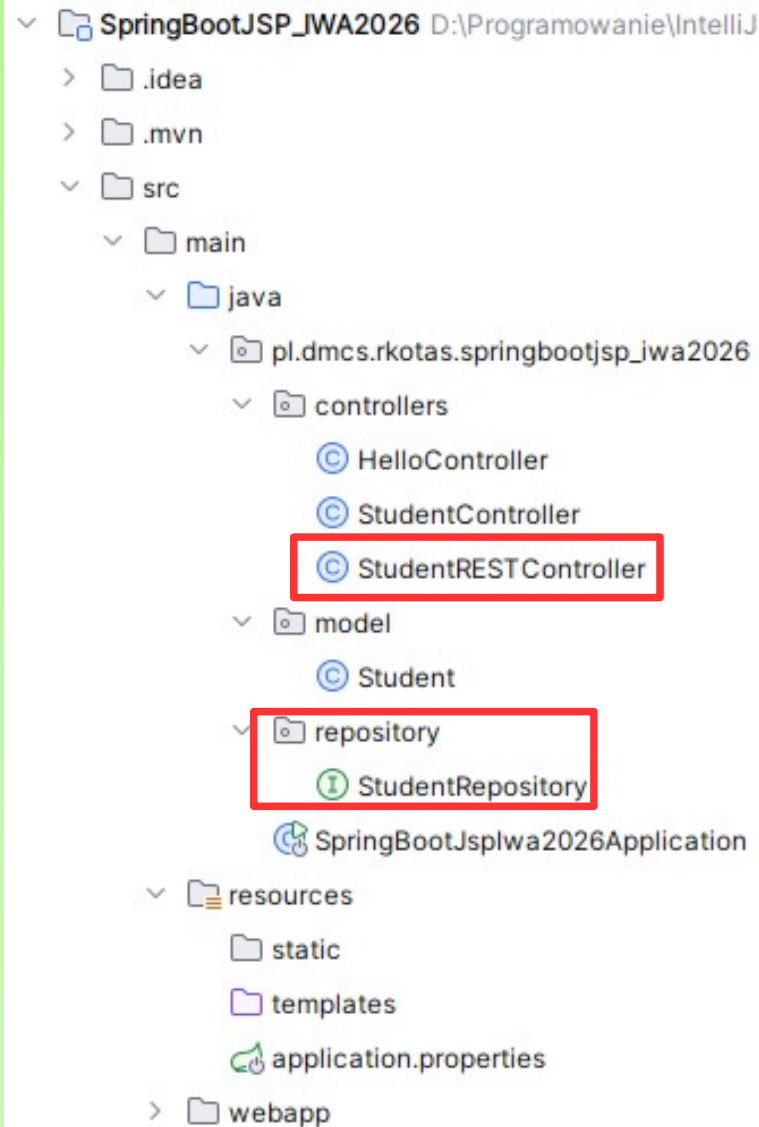
application.properties

```
5  # DataSource settings: set here your own configurations for the database
6  # connection.
7  spring.datasource.url = jdbc:postgresql://localhost:5433/springboot_rest_iwa2026
8  spring.datasource.username = postgres
9  spring.datasource.password = postgres
10
11 # Keep the connection alive if idle for a long time (needed in production)
12 spring.datasource.testWhileIdle = true
13 spring.datasource.validationQuery = SELECT 1
14
15 # Show or not log for each sql query
16 spring.jpa.show-sql = true
17
18 # Hibernate ddl auto (create, create-drop, update)
19 spring.jpa.hibernate.ddl-auto = update
20
21 # Naming strategy
22 spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy
23
24 # Use spring.jpa.properties.* for Hibernate native properties (the prefix is
25 # stripped before adding them to the entity manager)
26
27 # The SQL dialect makes Hibernate generate better SQL for the chosen database
28 spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
29
30 # Fix Postgres JPA Error (Method org.postgresql.jdbc.PgConnection.createClob() is not yet implemented).
31 spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
32
33 #To change logging levels in the app
34 # logging.level.org.springframework.web = trace
35 # logging.level.org.hibernate = trace
```

To check if the connection configuration is good, run the app and look for „student” table.

```
Student.java x
1 package pl.dmcs.rkotas.springbootjsp_iwa2026.model;
2
3 import jakarta.persistence.Entity;
4 import jakarta.persistence.GeneratedValue;
5 import jakarta.persistence.Id;
6
7 @Entity
8 public class Student {
9
10     @Id
11     @GeneratedValue
12     private long id;
13
14     private String firstname;
15     private String lastname;
16     private String email;
17     private String telephone;
18
19     public long getId() { return id; }
20
21     public void setId(long id) { this.id = id; }
```





StudentRepository.java ×



```
1 package pl.dmcs.rkotas.springbootjsp_iwa2026.repository;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4 import org.springframework.stereotype.Repository;
5 import pl.dmcs.rkotas.springbootjsp_iwa2026.model.Student;
6
7 @Repository
8 public interface StudentRepository extends JpaRepository<Student, Long>{
9     Student findById(long id);
10 }
```

© StudentRestController.java ×

```
1 package pl.dmcs.rkotas.springbootjsp_iwa2026.controllers;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.http.HttpStatus;
5 import org.springframework.http.ResponseEntity;
6 import org.springframework.web.bind.annotation.*;
7 import pl.dmcs.rkotas.springbootjsp_iwa2026.model.Student;
8 import pl.dmcs.rkotas.springbootjsp_iwa2026.repository.StudentRepository;
9 import java.util.List;
10 import java.util.Map;
11
12 @RestController
13 @RequestMapping("/students")
14 public class StudentRestController {
15
16     private StudentRepository studentRepository;
17
18     @Autowired
19     public StudentRestController(StudentRepository studentRepository) {
20         this.studentRepository = studentRepository;
21     }
```


© StudentRESTController.java ×

```
14      public class StudentRESTController {  
23          @RequestMapping(method = RequestMethod.GET/*, produces = "application/xml"*/)   
24              // @GetMapping  
25              public List<Student> findAllStudents() {  
26                  return studentRepository.findAll();  
27              }  
28  
29              @RequestMapping(method = RequestMethod.POST)   
30              // @PostMapping  
31              public ResponseEntity<Student> addStudent(@RequestBody Student student) {  
32                  studentRepository.save(student);  
33                  return new ResponseEntity<Student>(student, HttpStatus.CREATED);  
34              }
```

StudentRESTController.java ×

```
14 public class StudentRESTController {
```

```
36     @RequestMapping(value="/{id}", method = RequestMethod.DELETE)
```

```
37     // @DeleteMapping("/{id}")
```

```
38     public ResponseEntity<Student> deleteStudent (@PathVariable("id") long id) {
```

```
39         Student student = studentRepository.findById(id);
```

```
40         if (student == null) {
```

```
41             System.out.println("Student not found!");
```

```
42             return new ResponseEntity<Student>(HttpStatus.NOT_FOUND);
```

```
43         }
```

```
44         studentRepository.deleteById(id);
```

```
45         return new ResponseEntity<Student>(HttpStatus.NO_CONTENT);
```

```
46     }
```

```
48     @RequestMapping(value="/{id}", method = RequestMethod.PUT)
```

```
49     // @PutMapping("/{id}")
```

```
50     public ResponseEntity<Student> updateStudent(@RequestBody Student student, @PathVariable("id") long id) {
```

```
51         student.setId(id);
```

```
52         studentRepository.save(student);
```

```
53         return new ResponseEntity<Student>(student, HttpStatus.OK);
```

```
54     }
```

StudentRestController.java ×

```
14 public class StudentRestController {
56     @RequestMapping(value="/{id}", method = RequestMethod.PATCH)
57     // @PatchMapping("/{id}")
58     public ResponseEntity<Student> updatePartOfStudent(@RequestBody Map<String, Object> updates, @PathVariable("id") long id) {
59         Student student = studentRepository.findById(id);
60         if (student == null) {
61             System.out.println("Student not found!");
62             return new ResponseEntity<Student>(HttpStatus.NOT_FOUND);
63         }
64         partialUpdate(student, updates);
65         return new ResponseEntity<Student>(HttpStatus.NO_CONTENT);
66     }
67
68     private void partialUpdate(Student student, Map<String, Object> updates) {
69         if (updates.containsKey("firstname")) {
70             student.setFirstname((String) updates.get("firstname"));
71         }
72         if (updates.containsKey("lastname")) {
73             student.setLastname((String) updates.get("lastname"));
74         }
75         if (updates.containsKey("email")) {
76             student.setEmail((String) updates.get("email"));
77         }
78         if (updates.containsKey("telephone")) {
79             student.setTelephone((String) updates.get("telephone"));
80         }
81         studentRepository.save(student);
82     }
```

Spring Boot – REST example – Postman



44

dr inż. Rafał Kotas, rkotas@dmcs.pl

POST **Send**

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings Cookies Beautify

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON**

```
1 {
2   "firstname": "Rafal",
3   "lastname": "Kotas",
4   "email": "rkotas@dmcs.pl",
5   "telephone": "123123123"
6 }
```

Body Cookies Headers (5) Test Results Status: 201 Created Time: 463 ms Size: 265 B Save Response

Pretty Raw Preview Visualize **JSON**

```
1 {
2   "id": 1,
3   "firstname": "Rafal",
4   "lastname": "Kotas",
5   "email": "rkotas@dmcs.pl",
6   "telephone": "123123123"
7 }
```

Spring Boot – REST example, Postman



45

dr inż. Rafał Kotas, rkotas@dmcs.pl

GETlocalhost:8080/students

Send

ParamsAuthorizationHeaders (7)BodyPre-request ScriptTestsSettingsCookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

BodyCookiesHeaders (5)Test Results

Save Response

PrettyRawPreviewVisualizeJSON

```
1  [
2    {
3      "id": 1,
4      "firstname": "Rafal",
5      "lastname": "Kotas",
6      "email": "rkotas@dmcs.pl",
7      "telephone": "123123123"
8    },
9    {
10     "id": 2,
11     "firstname": "Jan",
12     "lastname": "Nowak",
13     "email": "jnowak@dmcs.pl",
14     "telephone": "456456456"
15   }
16 ]
```

Spring Boot – REST example – Postman



46

dr inż. Rafał Kotas, rkotas@dmcs.pl

The image shows the Postman application interface. At the top, a red box highlights the request configuration: the method is 'PUT' and the URL is 'localhost:8080/students/3'. Below this, the 'Body' tab is selected, and the 'raw' radio button is chosen. The JSON body is:

```
{  "firstname": "Jerzy",  "lastname": "Kowalski",  "email": "jkowalski@dmcs.pl",  "telephone": "789789789"}
```

. A red box highlights this body content. At the bottom, another red box highlights the response section. The status is '200 OK', time is '25 ms', and size is '266 B'. The response body is shown in 'Pretty' format:

```
{  "id": 3,  "firstname": "Jerzy",  "lastname": "Kowalski",  "email": "jkowalski@dmcs.pl",  "telephone": "789789789"}
```

PUT localhost:8080/students/3 Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary JSON Beautify

```
1 {
2   "firstname": "Jerzy",
3   "lastname": "Kowalski",
4   "email": "jkowalski@dmcs.pl",
5   "telephone": "789789789"
6 }
```

Body Cookies Headers (5) Test Results Status: 200 OK Time: 25 ms Size: 266 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 3,
3   "firstname": "Jerzy",
4   "lastname": "Kowalski",
5   "email": "jkowalski@dmcs.pl",
6   "telephone": "789789789"
7 }
```

Spring Boot – REST example – Postman



47

dr inż. Rafał Kotas, rkotas@dmcs.pl

The image shows the Postman application interface. At the top, the method is set to **PATCH** and the URL is `localhost:8080/students/3`. The **Body** tab is selected, and the format is set to **JSON**. The JSON body is:

```
{
  "firstname": "Jerzy",
  "telephone": "777888999"
}
```

The response status is **204 No Content**, with a time of 69 ms and a size of 112 B.

Spring Boot – REST example – Postman



48

dr inż. Rafał Kotas, rkotas@dmcs.pl

The image shows the Postman application interface. At the top, a red box highlights the request configuration bar, which includes the method 'DELETE', a dropdown arrow, the URL 'localhost:8080/students/3', and a blue 'Send' button with a dropdown arrow. Below this, the 'Body' tab is selected, showing radio buttons for 'none', 'form-data', 'x-www-form-urlencoded', 'raw', 'binary', and 'GraphQL'. The 'none' option is selected. The main area displays the message 'This request does not have a body'. At the bottom, another red box highlights the response section, which includes tabs for 'Body', 'Cookies', 'Headers (3)', and 'Test Results'. The 'Body' tab is active, showing the status 'Status: 204 No Content', the time 'Time: 43 ms', the size 'Size: 112 B', and a 'Save Response' button with a dropdown arrow.

1. Implement REST endpoints from the above example (GET all, POST one, DELETE one, PUT one, PATCH one) – „blue boxes”
2. Add REST endpoints for other missing HTTP methods (GET one, DELETE all, PUT all) – „red boxes”
3. Test all endpoints with the use of Postman



THE END

WIKAMP → IWA_3.zip

