Advanced Application Development (CSE-214) week-9 :Rest / CRUD / MVC / JPA / Hibernate

Dr. Alper ÖZCAN
Akdeniz University
alper.ozcan@gmail.com

1. BLOB (Binary Large Object)

How to handle **BLOB (Binary Large Object)** types like **PDFs or images**

MySQL Table with BLOB Column

```
CREATE TABLE document (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  file_name VARCHAR(255),
  file_type VARCHAR(100),
  data LONGBLOB
);
```

LONGBLOB can store large binary data (up to 4GB).

Spring Boot – Entity Class

```java
import jakarta.persistence.*;

@Entity
@Table(name = "document")
public class Document {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String fileName;

    private String fileType;

    @Lob
    @Column(name = "data", columnDefinition = "LONGBLOB")
    private byte[] data;

    // Getters and Setters
}
```

@Lob tells JPA this is a large binary field.

Spring Boot – DAO and Service

DAO (with EntityManager or Spring Data JPA)

```
public interface DocumentRepository extends JpaRepository<Document, Long> {}
```

you **only define an interface**.
You **don't** write the implementation manually.
Spring Data JPA automatically **creates the implementation at runtime!**

- JpaRepository<Document, Long> already provides all basic CRUD methods (save, findById, findAll, delete, etc.).
- Document is the entity class representing the **document** (which holds the PDF data).
- Spring Boot, when your application starts, **auto-generates** a real Java class behind the scenes that implements your DocumentRepository.
- You never have to write the .save(), .findAll(), .findById() methods manually — you just call them.

Service Layer

```
@Service
public class DocumentService {

    private final DocumentRepository repository;

    public DocumentService(DocumentRepository repository) {
        this.repository = repository;
    }

    public Document store(MultipartFile file) throws IOException {
        Document doc = new Document();
        doc.setFileName(file.getOriginalFilename());
        doc.setFileType(file.getContentType());
        doc.setData(file.getBytes());
        return repository.save(doc);
    }

    public Document getFile(Long id) {
        return repository.findById(id).orElse(null);
    }
}
```

Spring Boot – REST Controller

```java
@RestController
@RequestMapping("/api/files")
public class DocumentController {

    private final DocumentService service;

    public DocumentController(DocumentService service) {
        this.service = service;
    }

    @PostMapping("/upload")
    public ResponseEntity<String> upload(@RequestParam("file") MultipartFile file) throws IOException {
        service.store(file);
        return ResponseEntity.ok("Uploaded successfully.");
    }

    @GetMapping("/download/{id}")
    public ResponseEntity<byte[]> download(@PathVariable Long id) {
        Document doc = service.getFile(id);

        return ResponseEntity.ok()
            .header("Content-Disposition", "attachment; filename=" + doc.getFileName())
            .contentType(MediaType.parseMediaType(doc.getFileType()))
            .body(doc.getData());
    }
}
```

Angular UI – File Upload Component

HTML

```
<input type="file" (change)="onFileSelected($event)">
<button (click)="upload()">Upload</button>
```

This is an **input** of type **file**.
When the user selects a file, the **(change)** event is triggered.
The **$event** contains everything about the change — including the file the user picked.
event.target refers to the input HTML element.
event.target.files is a FileList — it contains the files the user selected.
event.target.files[0] is the first file

TypeScript

```
selectedFile: File;

onFileSelected(event: any) {
  this.selectedFile = event.target.files[0];
}

upload() {
  const formData = new FormData();
  formData.append('file', this.selectedFile);

  this.http.post('http://localhost:8080/api/files/upload', formData)
    .subscribe(response => console.log(response));
}
```

Angular UI – File Download Example

TypeScript

```
download(id: number) {
  this.http.get(`http://localhost:8080/api/files/download/${id}`, {
    responseType: 'blob'
  }).subscribe(blob => {
    const url = window.URL.createObjectURL(blob);
    const a = document.createElement('a');
    a.href = url;
    a.download = 'file.pdf';
    a.click();
  });
}
```

2. PDF inside an iframe using Angular and Spring Boot

Spring Boot – Controller for PDF Download

@GetMapping("/pdf/{id}"): This endpoint fetches the PDF binary data from the database.
ResponseEntity.ok(): Returns the PDF as a response with appropriate headers for rendering in an iframe.
Content-Type: application/pdf: This tells the browser the content type is a PDF.
Content-Disposition: inline: This tells the browser to display the PDF inline, instead of prompting the user to download it.

```
@RestController
@RequestMapping("/api/files")
public class DocumentController {

    private final DocumentService documentService;

    public DocumentController(DocumentService documentService) {
        this.documentService = documentService;
    }

    @GetMapping("/pdf/{id}")
    public ResponseEntity<byte[]> getPdf(@PathVariable Long id) {
        Document document = documentService.getFile(id);

        if (document == null) {
            return ResponseEntity.notFound().build();
        }

        return ResponseEntity.ok()
            .header(HttpHeaders.CONTENT_DISPOSITION, "inline; filename=" + document.getFileName())
            .contentType(MediaType.APPLICATION_PDF)
            .body(document.getData());
    }
}
```

Display the PDF in Angular Using an iframe

In your **Angular component**, use an iframe to display the PDF file served by the backend. We use **safeUrl** pipe to bypass Angular's security restrictions for dynamically loading URLs

Angular Component HTML

```
<iframe *ngIf="pdfUrl" [src]="pdfUrl | safeUrl" width="100%" height="600px"></iframe>
```

Angular Component TypeScript

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-pdf-viewer',
  templateUrl: './pdf-viewer.component.html',
  styleUrls: ['./pdf-viewer.component.css']
})
export class PdfViewerComponent implements OnInit {

  pdfUrl: string;

  constructor(private http: HttpClient) { }

  ngOnInit(): void {
    this.loadPdf(1);  // Assuming you're displaying the PDF with ID 1
  }

  loadPdf(id: number): void {
    this.pdfUrl = `http://localhost:8080/api/files/pdf/${id}`;
  }
}
```

CORS: If your Spring Boot backend is running on a different port or domain than your Angular frontend, you might need to enable CORS in Spring Boot.

SafeUrl Pipe

Angular has strict security rules when it comes to binding URLs in iframe tags. To bypass this, we need to create a **SafeUrl Pipe** to sanitize the URL. This pipe makes sure the URL is safe to use inside an iframe by bypassing Angular's security.

```
import { Pipe, PipeTransform } from '@angular/core';
import { DomSanitizer, SafeResourceUrl } from '@angular/platform-browser';

@Pipe({
  name: 'safeUrl'
})
export class SafeUrlPipe implements PipeTransform {

  constructor(private sanitizer: DomSanitizer) { }

  transform(url: string): SafeResourceUrl {
    return this.sanitizer.bypassSecurityTrustResourceUrl(url);
  }
}
```

3. CDN for BLOBs

Why Use a CDN for BLOBs?

- Directly storing and serving BLOBs from a database **is not efficient** for large files.
- Databases are **not optimized** for high-volume file delivery.
- **CDNs (Content Delivery Networks)** are **optimized** for fast, scalable, global file distribution.

How to Use a CDN for BLOBs?

Step	Action
1	Upload the file to **Object Storage** (like AWS S3, Azure Blob Storage, Google Cloud Storage).
2	The object storage **integrates with a CDN** (like CloudFront, Azure CDN, Cloudflare).
3	Save only the **URL or file path** into MySQL, not the file itself.
4	When client requests, your app/API returns the **CDN URL** for the file.
5	Angular UI loads the file directly from the **CDN** (very fast).

CDN for BLOBs Example Architecture

- User uploads a **PDF** through Angular → Spring Boot API.
- Spring Boot **uploads the file** to AWS S3.
- AWS S3 has **CloudFront CDN** attached.
- Spring Boot **stores the S3/CloudFront URL** in MySQL.
- Later, when a user requests the PDF, Angular fetches the URL from backend → displays in iframe or link.

Amazon CloudFront is a **Content Delivery Network (CDN)** service offered by **AWS (Amazon Web Services)**. It **delivers** your content (images, videos, PDFs, static files, APIs, websites) **quickly and securely** to users all over the world. It works by **caching** your files in **edge locations** (servers) that are close to your users

CDN for BLOBs Example Code

S3 stands for **Simple Storage Service**.
It is a service provided by **Amazon Web Services (AWS)** that allows you to **store** and **retrieve any amount of data** (images, PDFs, videos, backups, etc.) over the internet.
You can think of S3 like a **huge, safe, and scalable hard drive** on the cloud.

Spring Boot: Upload to S3

```
public String uploadFile(MultipartFile file) {
    String fileName = file.getOriginalFilename();
    ObjectMetadata metadata = new ObjectMetadata();
    metadata.setContentLength(file.getSize());
    amazonS3.putObject(new PutObjectRequest("your-bucket-name", fileName, file.getInputStream(), metadata));
    return amazonS3.getUrl("your-bucket-name", fileName).toString(); // This is the CDN URL!
}
```

Save in DB (only URL)

```
Student student = new Student();
student.setDocumentUrl("https://yourcdn.com/path-to-file.pdf");
studentRepository.save(student);
```

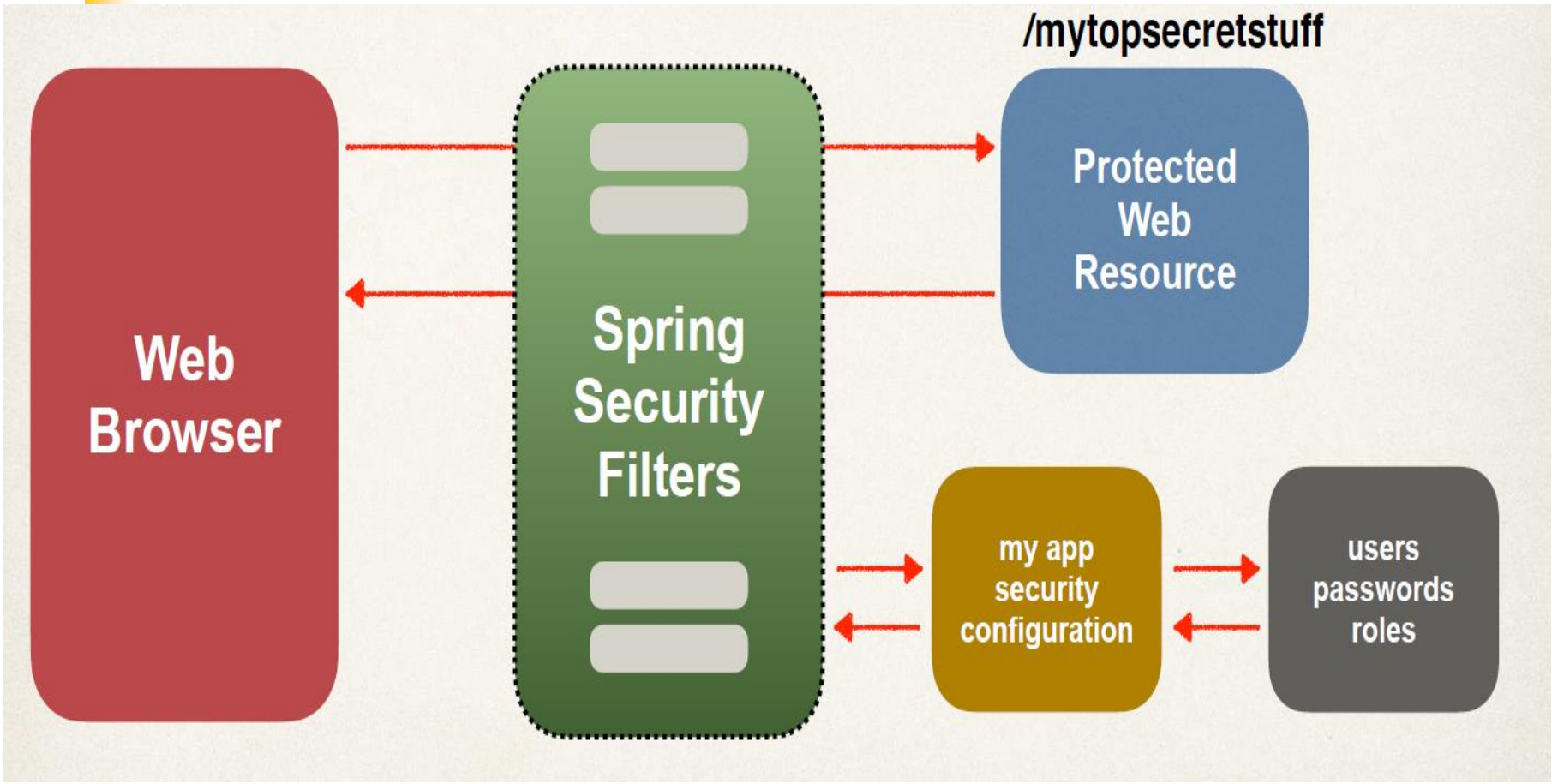
Angular: Display in iframe

```
<iframe [src]="student.documentUrl | safeUrl" width="600" height="800"></iframe>
```

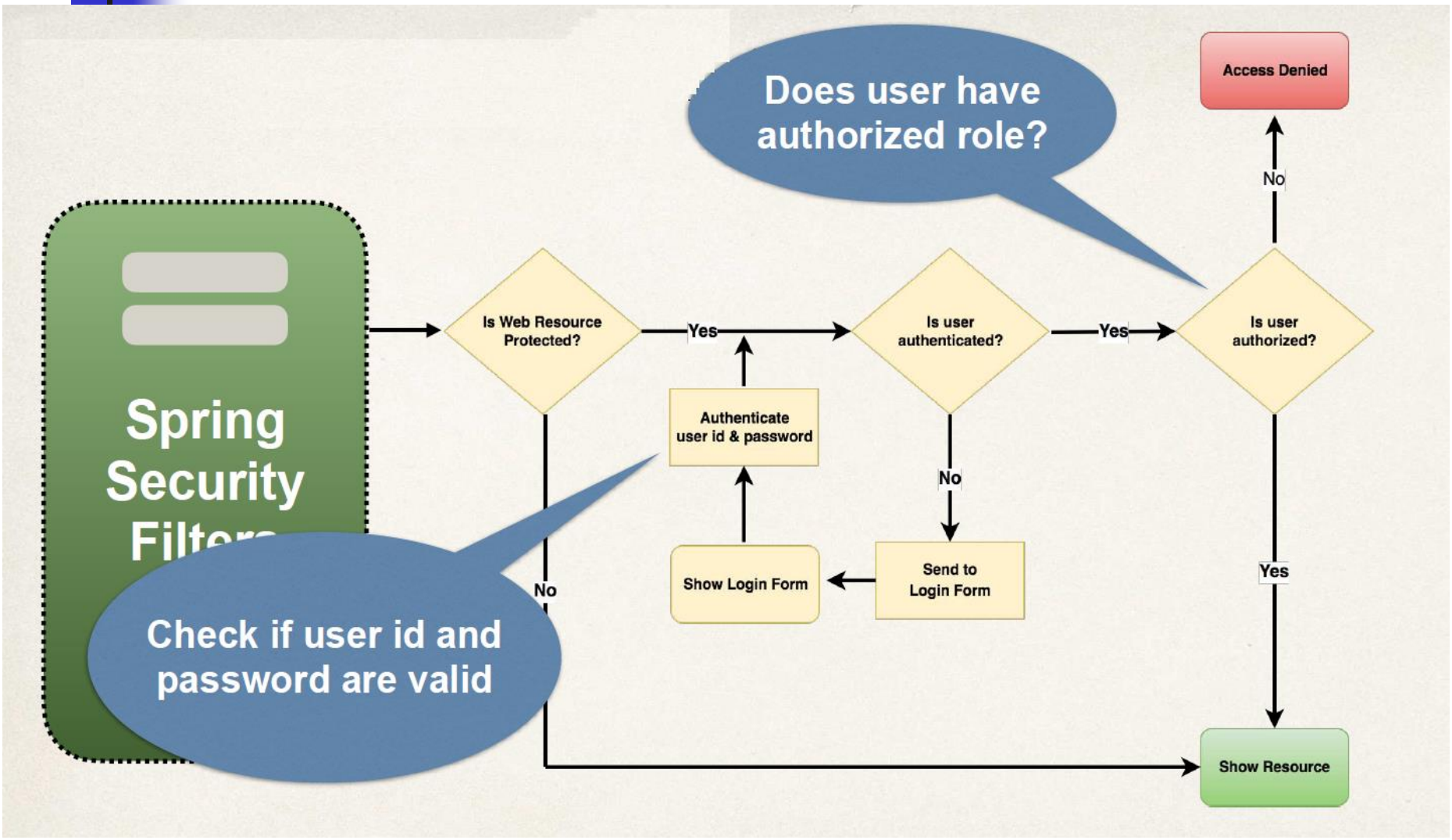
Spring Boot REST API Security

- Secure Spring Boot REST APIs
- Define users and roles
- Protect URLs based on role
- Store users, passwords and roles in DB (plain-text -> encrypted)

Spring Security Overview



Spring Security in Action



Security Concepts

- Authentication
  - Check user id and password with credentials stored in app / db
- Authorization
  - Check to see if user has an authorized role

Enabling Spring Security

1. Edit **pom.xml** and add **spring-boot-starter-security**

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

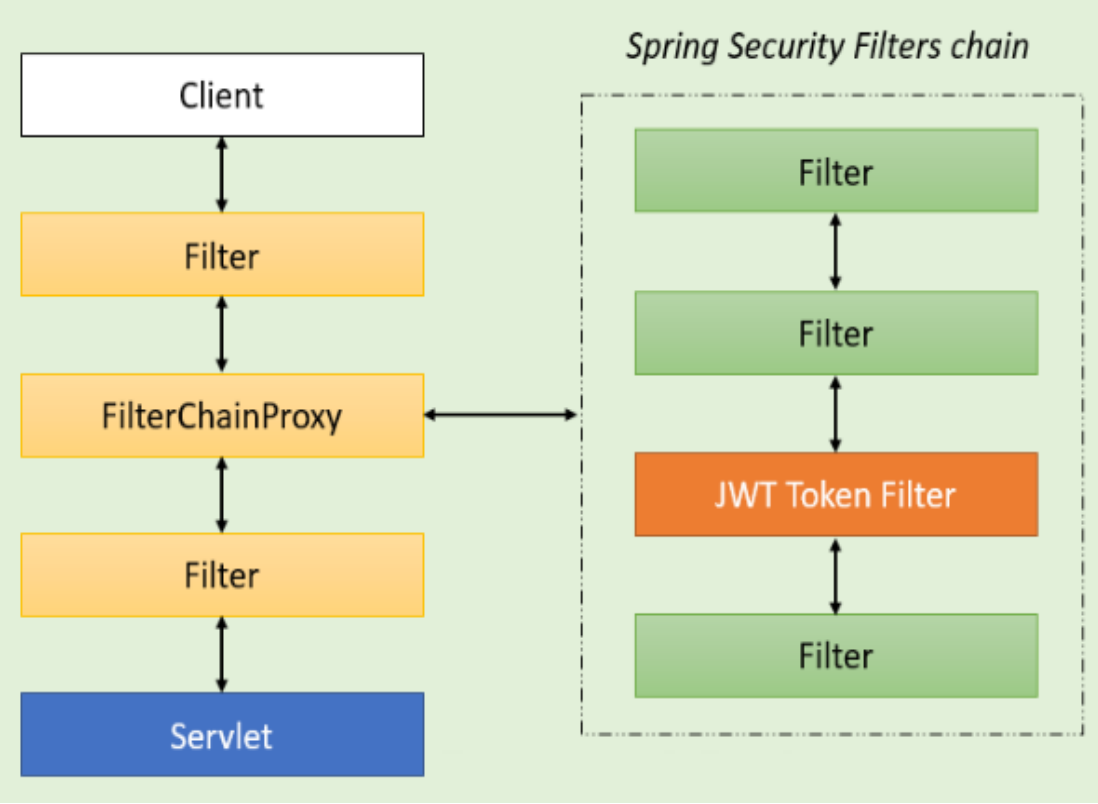
2. This will *automagically* secure all endpoints for application

Spring Security and JWT Token Authentication in a Spring Boot REST API

When a client sends a request to the server, **the request will go through a sequence of filters** before reaching the destination servlet/controller which is actually responsible for processing the request.

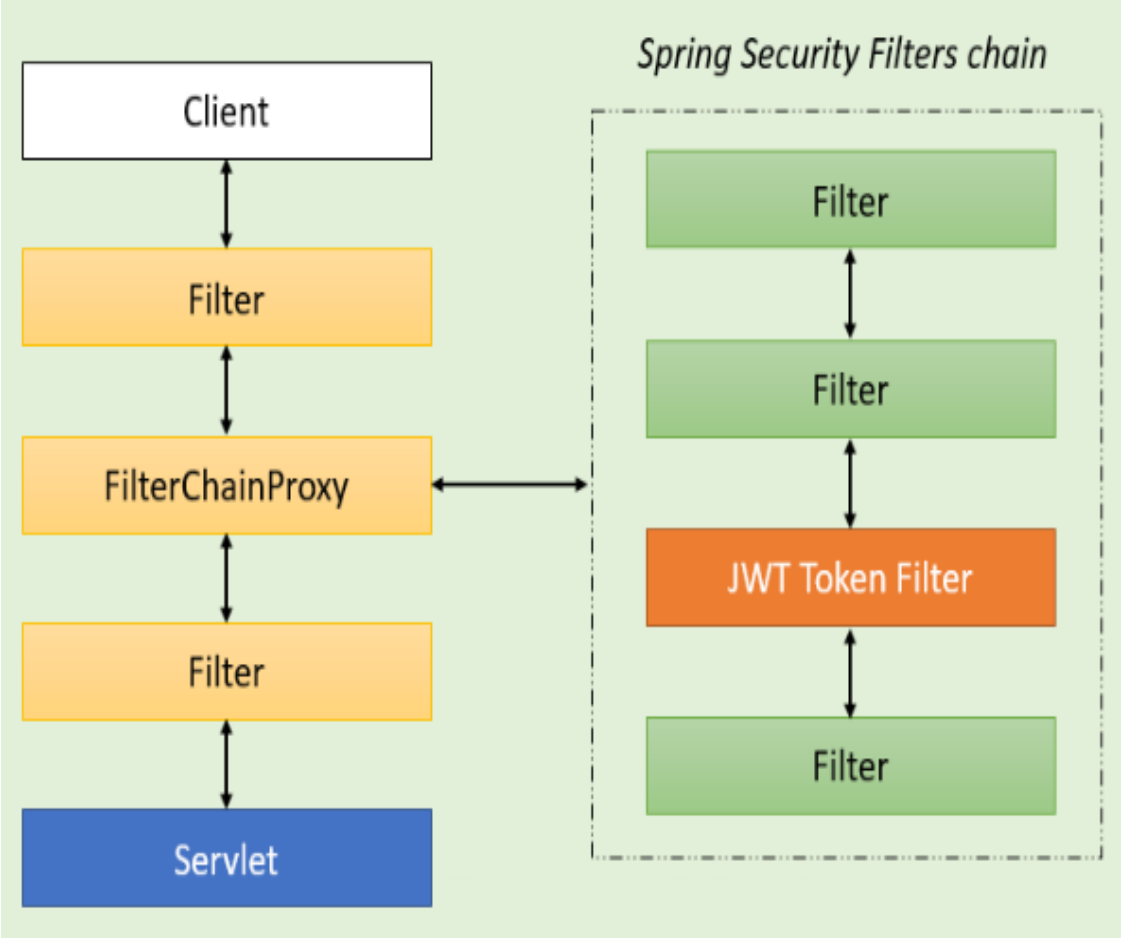
The Spring Web framework plugs in a special filter called ***FilterChainProxy*** that picks a chain of internal filters used by Spring Security, depending on the application's security configuration.

Each filter in the Spring Security filters chain is responsible for applying a specific security concern to the current request.



Spring Security and JWT Token Authentication in a Spring Boot REST API

We need to insert our own custom filter (e.g. **JWT Token Filter**) in the middle of Spring Security filters chain. This filter will **check availability** and verify integrity of the **access token**. **If** the **token is verified**, the request is passed through the downstream filters and finally reaching the **destination handler**. Otherwise, an **Unauthorized error should be raised**.



Spring Security and JWT Token Authentication

- Set up Spring Security with JWT authentication.
- Implement the **JWT Token Utility** class to generate and validate tokens.
- Create a **JWT Filter** to intercept requests and validate JWT tokens.
- Configure Spring Security to secure the REST API.
- Create a **RestController** to handle user login and access secured endpoints.

Necessary dependencies to your pom.xml file

```xml
<dependencies>
    <!-- Spring Boot Starter Security -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId
    </dependency>

    <!-- JWT dependencies -->
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt</artifactId>
        <version>0.11.5</version>
    </dependency>
```

```xml
    <!-- Spring Boot Starter Web for creating REST APIs -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Boot Starter Validation for validation support -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>

    <!-- Spring Boot Starter Data JPA (optional, for DB access) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- H2 Database (optional, for demonstration purposes) -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Create JwtUtil.java

This class will be used to generate and validate the JWT token.

```java
import io.jsonwebtoken.*;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import java.util.Date;

@Component
public class JwtUtil {

    @Value("${jwt.secret}")
    private String secretKey;

    @Value("${jwt.expiration}")
    private long expirationTime;

    // Generate JWT token
    public String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + expirationTime))
                .signWith(SignatureAlgorithm.HS512, secretKey)
                .compact();
    }

    // Validate JWT token
    public boolean validateToken(String token) {
        try {
            Jwts.parser()
                    .setSigningKey(secretKey)
                    .parseClaimsJws(token);
            return true;
        } catch (JwtException | IllegalArgumentException e) {
            return false;
        }
    }

    // Extract username from JWT token
    public String extractUsername(String token) {
        return Jwts.parser()
                .setSigningKey(secretKey)
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
    }
}
```

Create JWT Filter

- This filter will intercept every request and check if the request contains a valid JWT token.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;

public class JwtFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtil jwtUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request, javax.servlet.http.HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        String token = extractToken(request);

        if (token != null && jwtUtil.validateToken(token)) {
            String username = jwtUtil.extractUsername(token);
            UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(username, null, null);
            SecurityContextHolder.getContext().setAuthentication(authentication);
        }

        filterChain.doFilter(request, response);
    }

    private String extractToken(HttpServletRequest request) {
        String bearerToken = request.getHeader("Authorization");
        if (bearerToken != null && bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring(7); // Extract token after "Bearer "
        }
        return null;
    }
}
```

What is JWT Filter ?

This is a custom **filter** that runs for **every HTTP request**.

- Check if the request has a **valid JWT token** in the Authorization header.
- If the token is valid:
  - It sets the **authentication info** into the Spring Security context.
- If not valid:
  - The request continues **unauthenticated** and will be **blocked later** if the endpoint is protected.

1.User logs in via /login.

2.If login is successful, backend returns a **JWT token**.

3.User sends future requests with the JWT in the header:

```
Authorization: Bearer <your-jwt-token>
```

4.When the request comes in:

- **JwtFilter** catches it **before** Spring Security checks authentication.
- It validates the token using JwtUtil.
- If valid, it sets authentication into Spring's security context.

5.Spring Security then allows access to protected endpoints based on this context. JwtFilter checks every request and ensures it has a valid JWT before it hits the controller. Without a valid token, protected endpoints will return **401 Unauthorized**.

Create Security Configuration

This configuration will set up Spring Security to use JWT authentication and define security rules for different API endpoints.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private JwtFilter jwtFilter;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
                .authorizeRequests()
                .antMatchers("/login", "/register").permitAll()  // Public endpoints
                .anyRequest().authenticated() // All other requests require authentication
                .and()
                .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);
    }

    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import org.springframework.beans.factory.annotation.Autowired;
```

What is SecurityConfig.java ?

This is a **configuration class** where you define how **Spring Security** behaves in your application. You use it to:

- Disable CSRF (if needed, usually for APIs).
- Define **which endpoints are public** (like /login) and which are protected.
- Register your **JWT filter** to be executed **before** any request reaches the controller.

Example from SecurityConfig.java

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable()
        .authorizeRequests()
        .antMatchers("/login", "/register").permitAll()
        .anyRequest().authenticated()
        .and()
        .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);
}
```

- The line .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class) tells Spring to:
- "Run my JwtFilter **before** the default username-password authentication filter.«
- SecurityConfig tells Spring Security how to behave and when to call your custom filter.

UsernamePasswordAuthenticationFilter class

It is a **built-in class** in **Spring Security** library.

```
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter
```

It comes automatically when you add this dependency in your project

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

It's **inside the spring-security-web.jar** file (part of the Maven dependency).

UsernamePasswordAuthenticationFilter is the **default filter** in Spring Security **that handles login**.

- It **listens** for HTTP POST requests to /login by default.
- It expects **username** and **password** parameters.
- It tries to **authenticate** the user.
- If successful, it calls the AuthenticationManager.
- If failed, it returns 401 Unauthorized.

Create the UserController for Authentication

create a **Login** API to authenticate users and generate JWT tokens.

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

@RestController
@RequestMapping("/api/auth")
public class UserController {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private JwtUtil jwtUtil;

    @PostMapping("/login")
    public ResponseEntity<String> login(@RequestBody UserLoginDto userLoginDto) {
        try {
            authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(userLoginDto.getUsername(), userLoginDto.getPassword())
            );
            String token = jwtUtil.generateToken(userLoginDto.getUsername());
            return ResponseEntity.ok(token);
        } catch (Exception e) {
            return ResponseEntity.status(401).body("Invalid credentials");
        }
    }
}
```

when users send a **POST request** to /api/auth/login with their credentials (username and password), the system validates their credentials and, if valid, generates a JWT token and returns it.

Frontend (Angular)

On the Angular side, you can send the JWT token along with the request in the **Authorization header**.

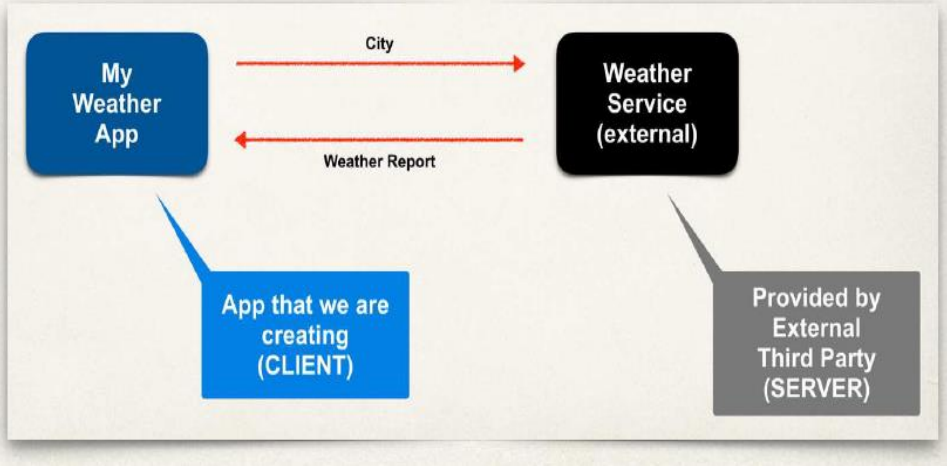
```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthService {

  private apiUrl = 'http://localhost:8080/api/auth/login';

  constructor(private http: HttpClient) {}

  login(username: string, password: string): Observable<any> {
    return this.http.post(this.apiUrl, { username, password });
  }

  getProtectedData(): Observable<any> {
    const token = localStorage.getItem('jwt_token');
    const headers = new HttpHeaders().set('Authorization', `Bearer ${token}`);
    return this.http.get('http://localhost:8080/protected-api', { headers });
  }
}
```

REST APIs - REST Web Services

- Create REST APIs / Web Services with Spring
- Discuss REST concepts, JSON and HTTP messaging
- Install REST client tool: Postman
- Develop REST APIs / Web Services with **@RestController**
- Build a CRUD interface to the database with Spring REST

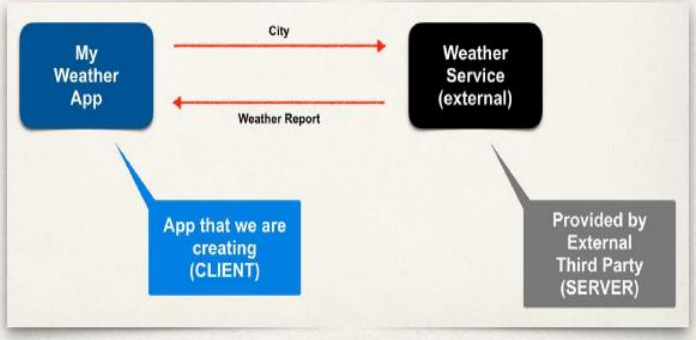
Questions

- How will we connect to the Weather Service?
- What programming language do we use?
- What is the data format?



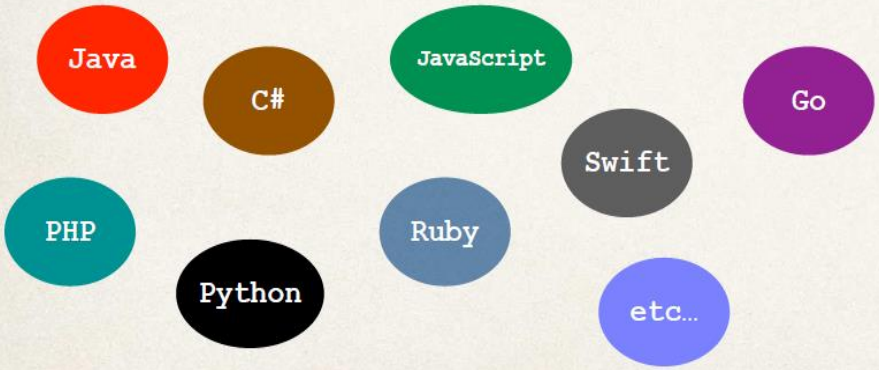
Answers

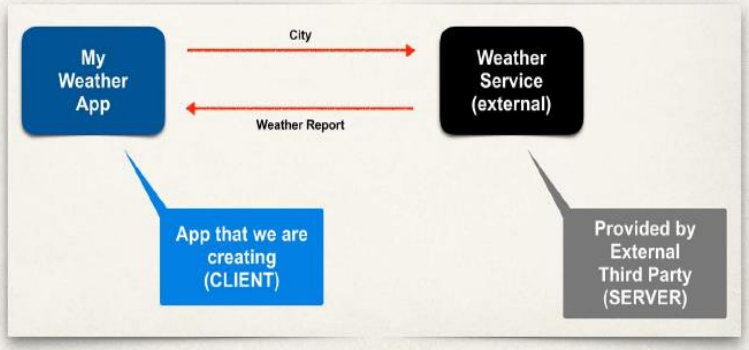
- **How will we connect to the Weather Service?**
  - We can make REST API calls over HTTP
  - REST: REpresentational State Transfer
  - Lightweight approach for communicating between applications



Answers

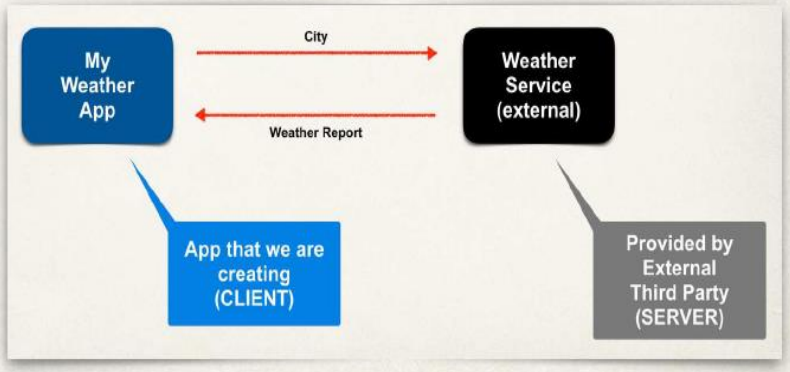
- **What programming language do we use?**
  - REST is language independent
  - The **client** application can use **ANY** programming language
  - The **server** application can use **ANY** programming language





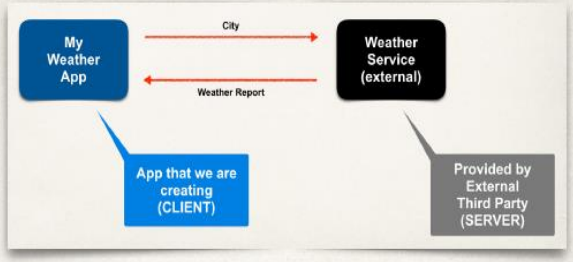
Answers

- **What is the data format?**
  - REST applications can use any data format
  - Commonly see XML and JSON
  - JSON is most popular and modern
    - JavaScript Object Notation



Response - Weather Report

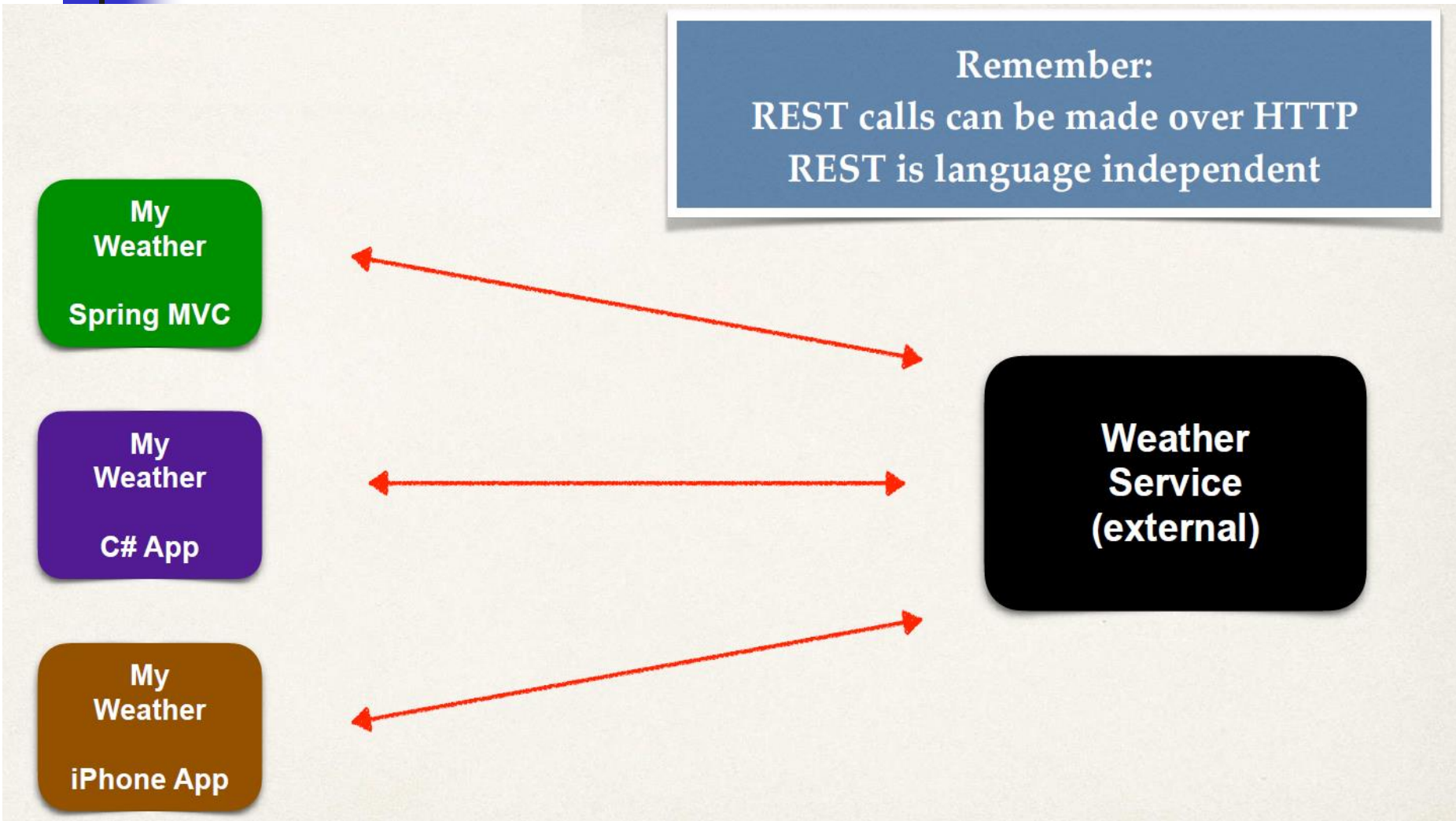
- The Weather Service responds with JSON




```
{
   …
   "temp": xxx,
   "feels_like": yyy,
   "humidity": zzz,
   …
}
```

Condensed version

Multiple Client Apps



What do we call it?

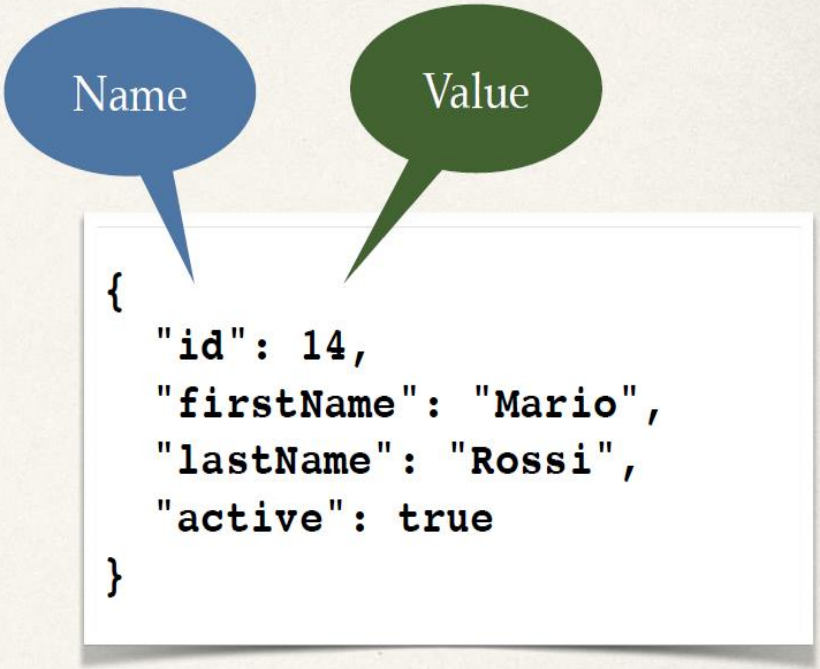
What is JSON?

- **J**ava**S**cript **O**bject **N**otation
- Lightweight data format for storing and exchanging data … plain text
- Language independent … not just for **JavaScript**
- Can use with any programming language: **Java, C#, Python** etc …

JSON is just plain text data

Simple JSON Example

- Curley braces define objects in JSON
- Object members are name / value pairs
  - Delimited by colons
- Name is **always** in double-quotes



Nested JSON Objects

```
{
   "id": 14,
   "firstName": "Mario",
   "lastName": "Rossi",
   "active": true,
   "address" : {
                  "street" : "100 Main St",
                  "city" : "Philadelphia",
                  "state" : "Pennsylvania",
                  "zip" : "19103",
                  "country" : "USA"
               }
}
```

JSON Arrays

```
{
   "id": 14,
   "firstName": "Mario",
   "lastName": "Rossi",
   "active": true,
   "languages" : ["Java", "C#", "Python", "Javascript"]
}
```

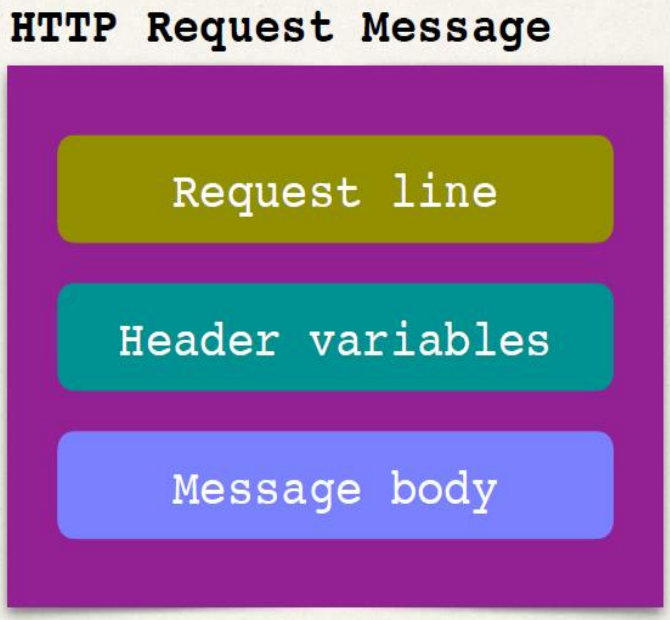
REST HTTP Basics - REST over HTTP

- Most common use of REST is over HTTP
- Leverage HTTP methods for CRUD operations

HTTP Method	CRUD Operation
POST	Create a new entity
GET	Read a list of entities or single entity
PUT	Update an existing entity
DELETE	Delete an existing entity

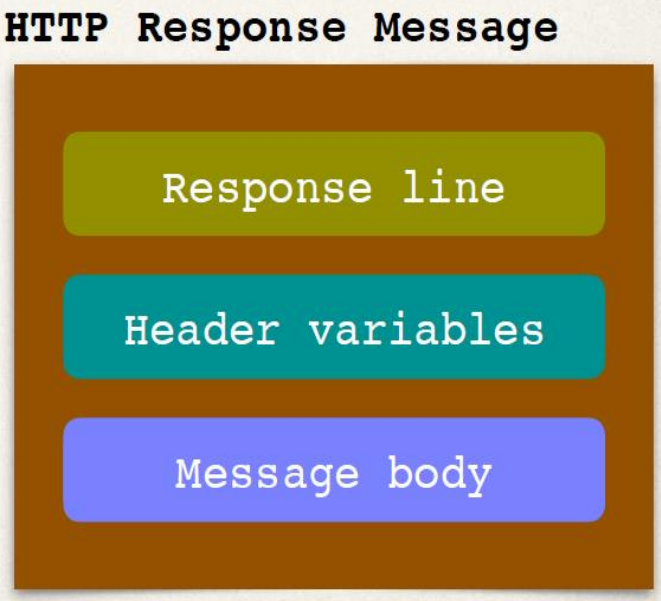
HTTP Request Message

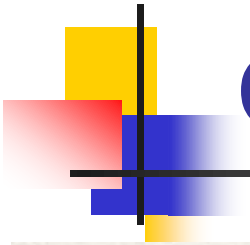
- Request line: the HTTP command
- Header variables: request metadata
- Message body: contents of message



HTTP Response Message

- Response line: server protocol and status code
- Header variables: response metadata
- Message body: contents of message

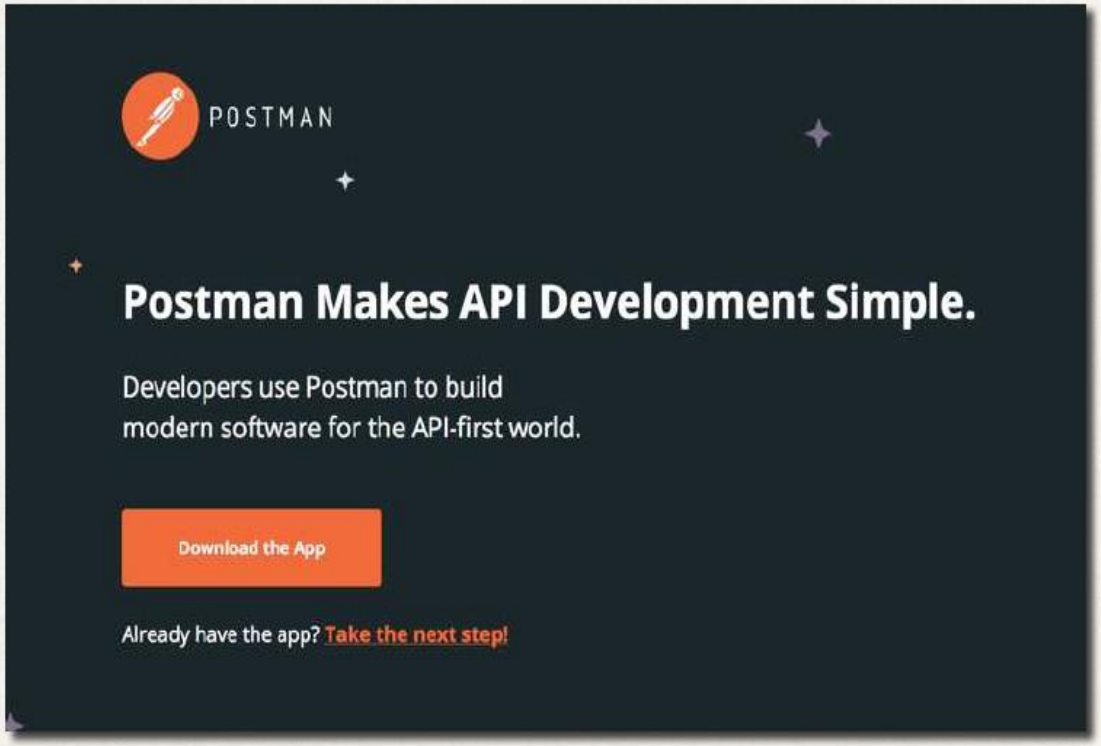




Client Tool

- We need a **client tool**
- Send HTTP requests to the REST Web Service / API
- Plenty of tools available: curl, Postman, etc …

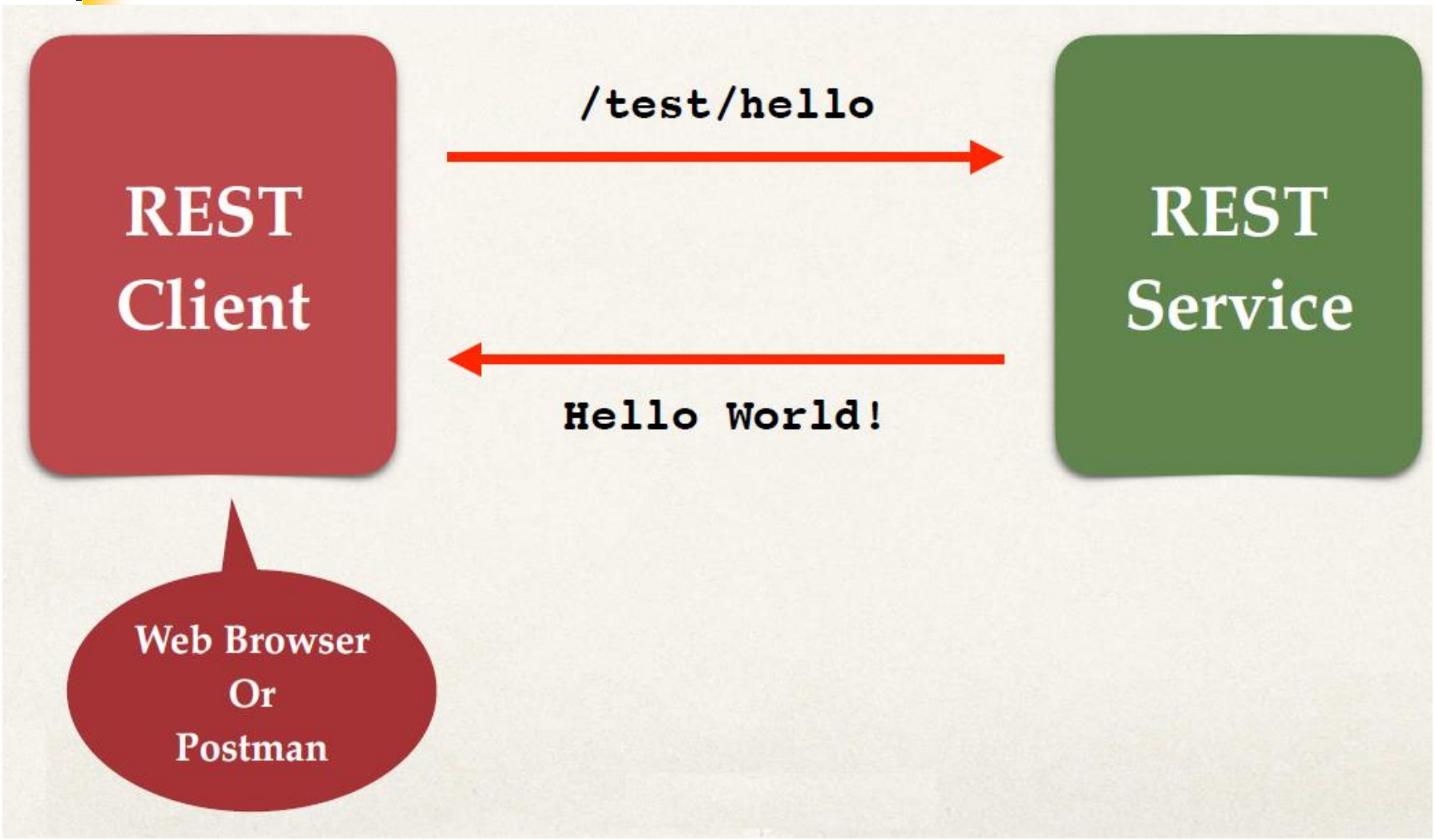
Postman



Free developer plan

www.getpostman.com

Spring REST Controller - Hello World



Testing with REST Client - Postman

Spring REST Controller - Dev Process

1. Add Maven dependency for Spring Boot Starter Web

2. Create Spring REST Service using @RestController

Step 1: Add Maven Dependency

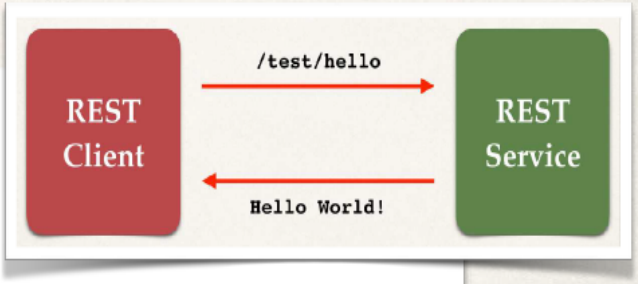
File: pom.xml

```
<!-- Add Spring Boot Starter Web -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

At Spring Initializr website,
can also select the "Web" dependency

Step 2: Create Spring REST Service

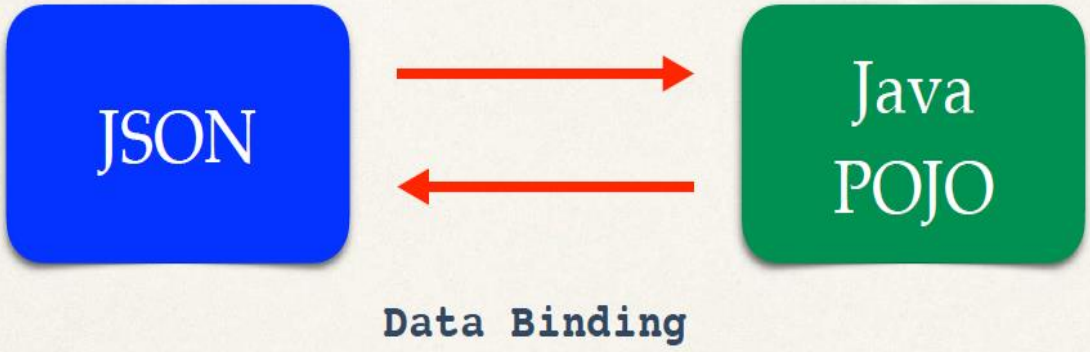
```
@RestController
@RequestMapping("/test")
public class DemoRestController {

  @GetMapping("/hello")
  public String sayHello() {
    return "Hello World!";
  }

}
```



Handles HTTP GET requests

Java JSON Data Binding

- Data binding is the process of converting JSON data to a Java POJO



Also known as

Mapping

Serialization / Deserialization

Marshalling / Unmarshalling

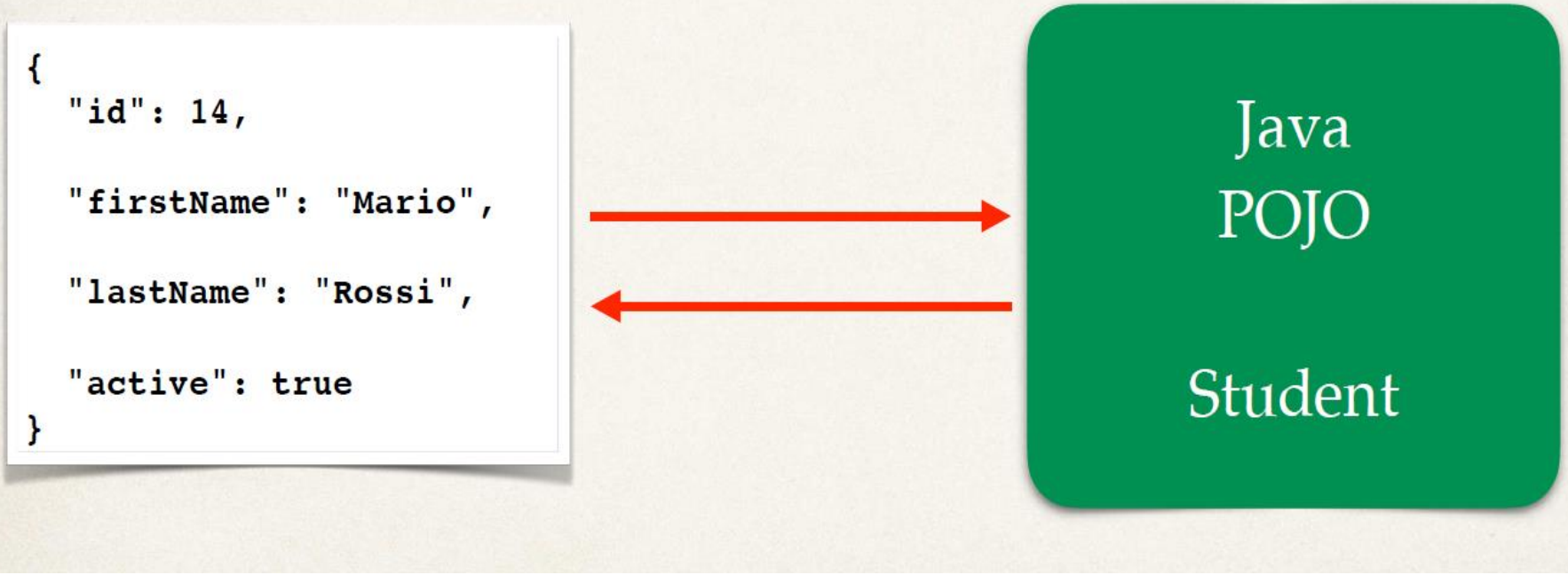
JSON Data Binding with Jackson

- Spring uses the **Jackson Project** behind the scenes
- Jackson handles data binding between JSON and Java POJO
- Details on Jackson Project:

https://github.com/FasterXML/jackson-databind

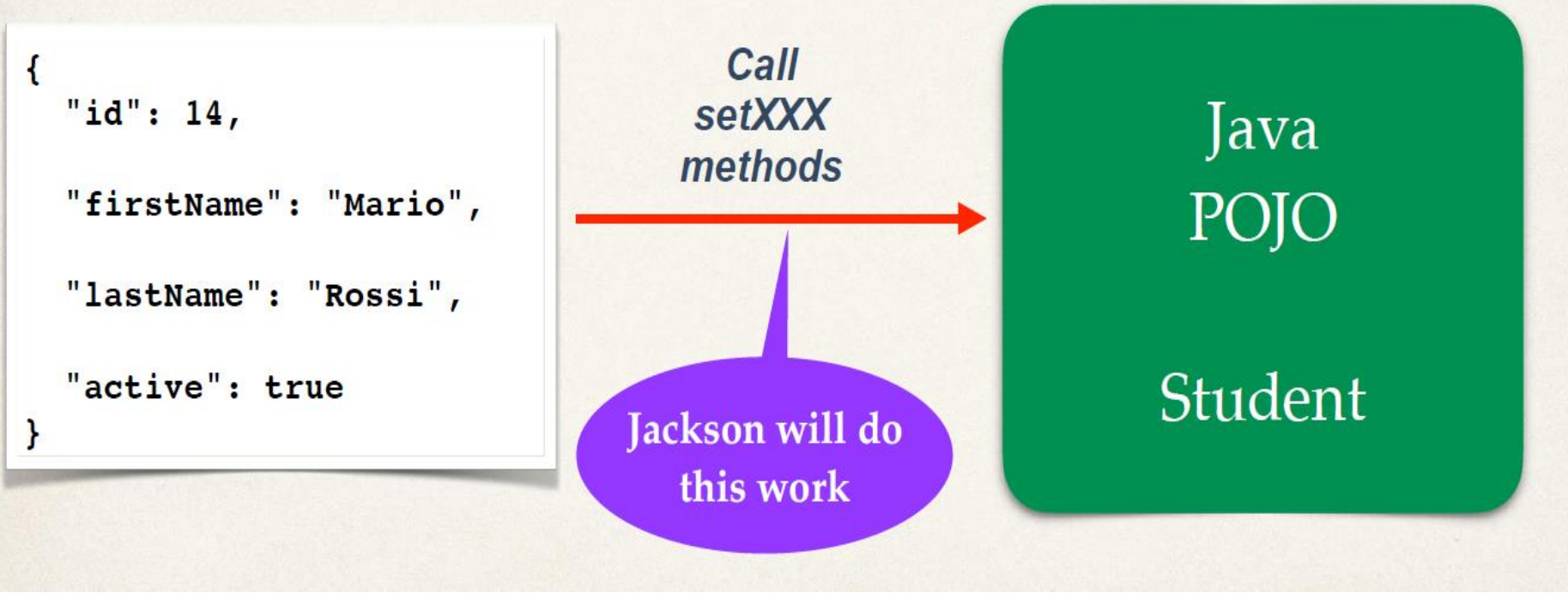
Jackson Data Binding

- By default, Jackson will call appropriate getter/setter method



JSON to Java POJO

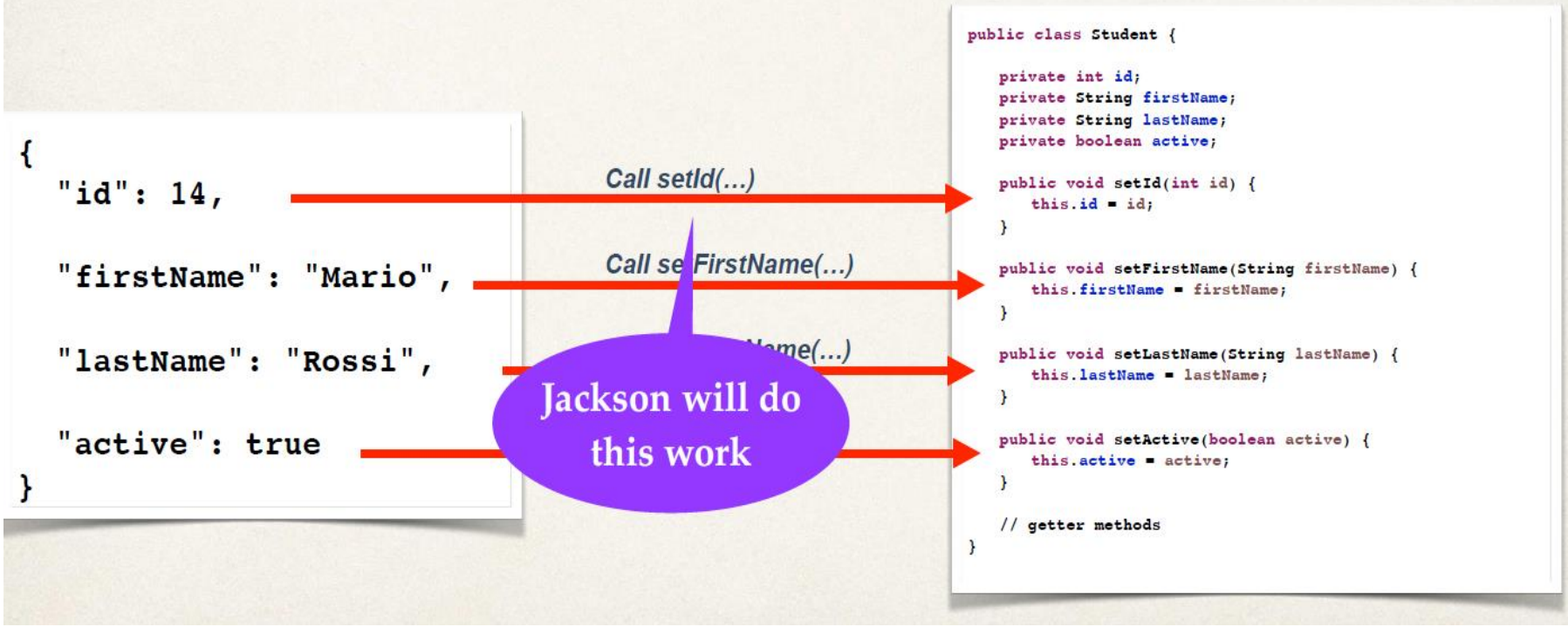
- Convert JSON to Java POJO … call setter methods on POJO



JSON to Java POJO

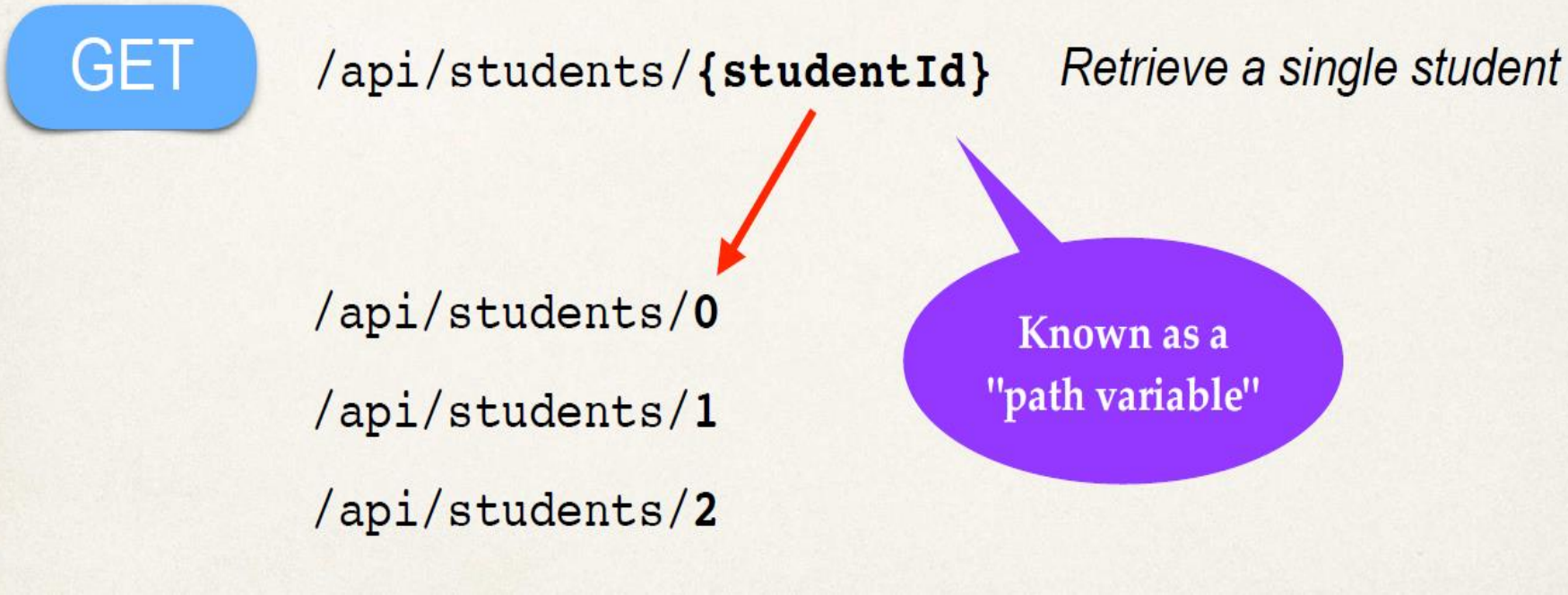
Note: Jackson calls the setXXX methods
It does NOT access internal private fields directly

- Convert JSON to Java POJO … call setter methods on POJO

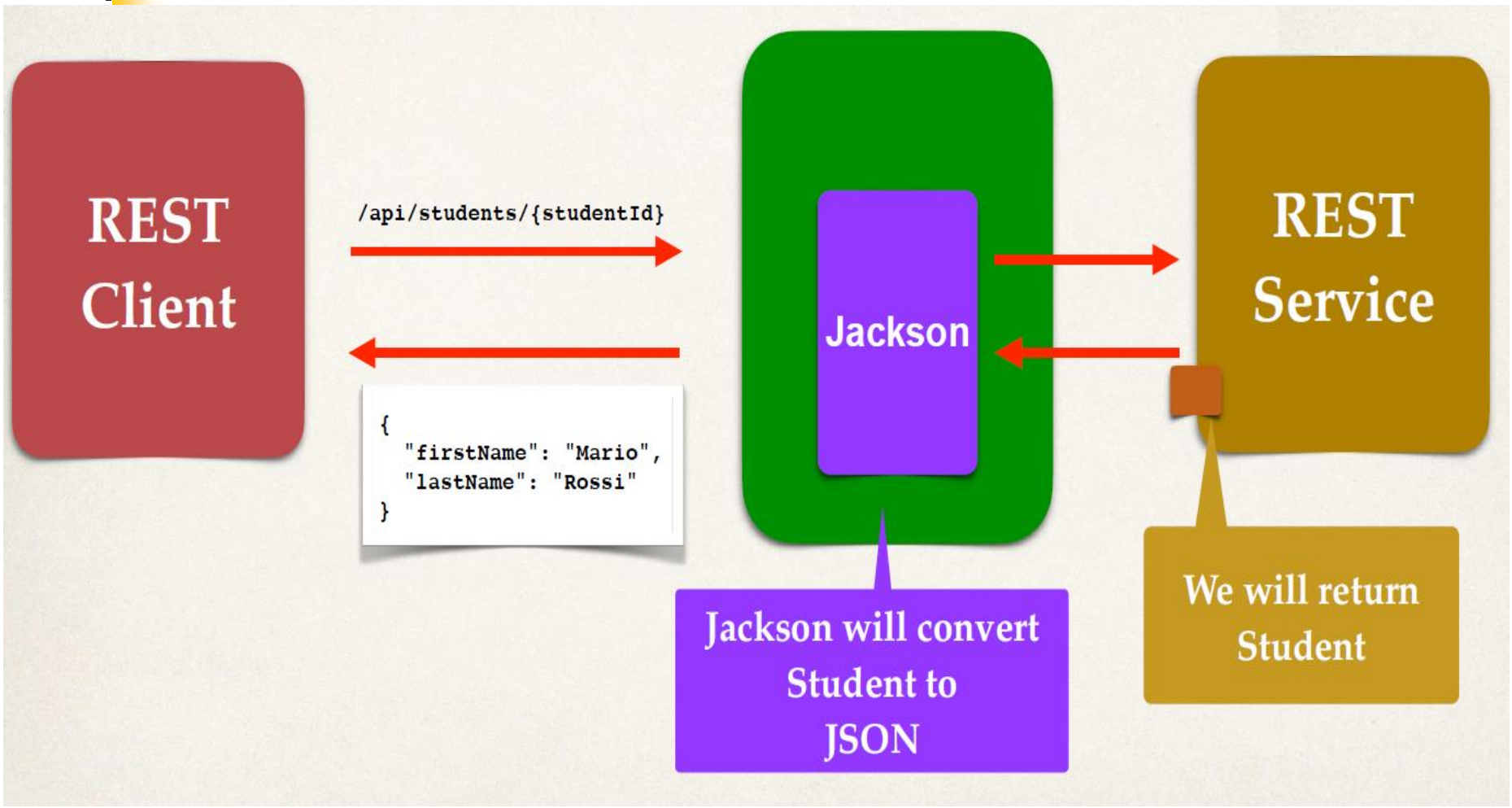


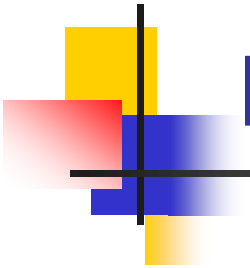
Spring REST - Path Variables

- Retrieve a single student by id



Spring REST Service - Jackson



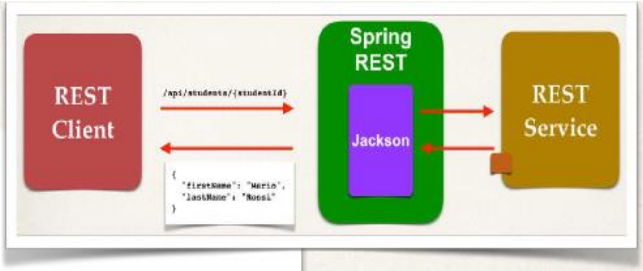


Development Process

1. Add request mapping to Spring REST Service

- Bind path variable to method parameter using `@PathVariable`

Step 1: Add Request Mapping



File: StudentRestController.java

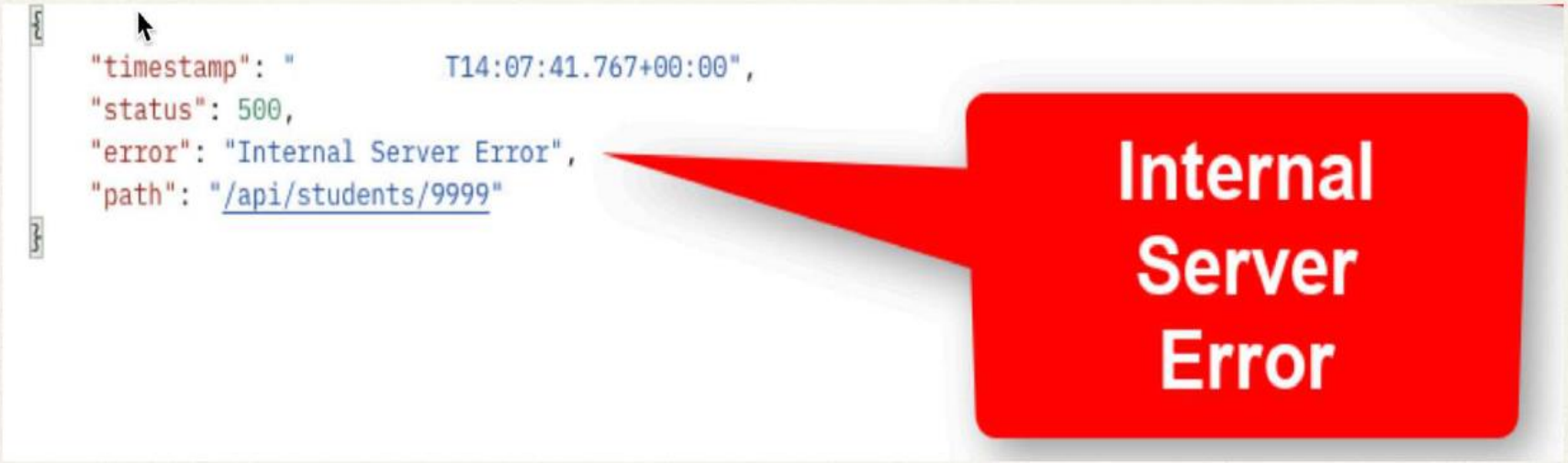
```
@RestController
@RequestMapping("/api")
public class StudentRestController {

  // define endpoint for "/students/{studentId}" - return student at index

  @GetMapping("/students/{studentId}")
  public Student getStudent(@PathVariable int studentId) {

    List<Student> theStudents = new ArrayList<>();

    // populate theStudents
    …

    return theStudents.get(studentId);
  }

}
```

Bind the path variable (by default, must match)

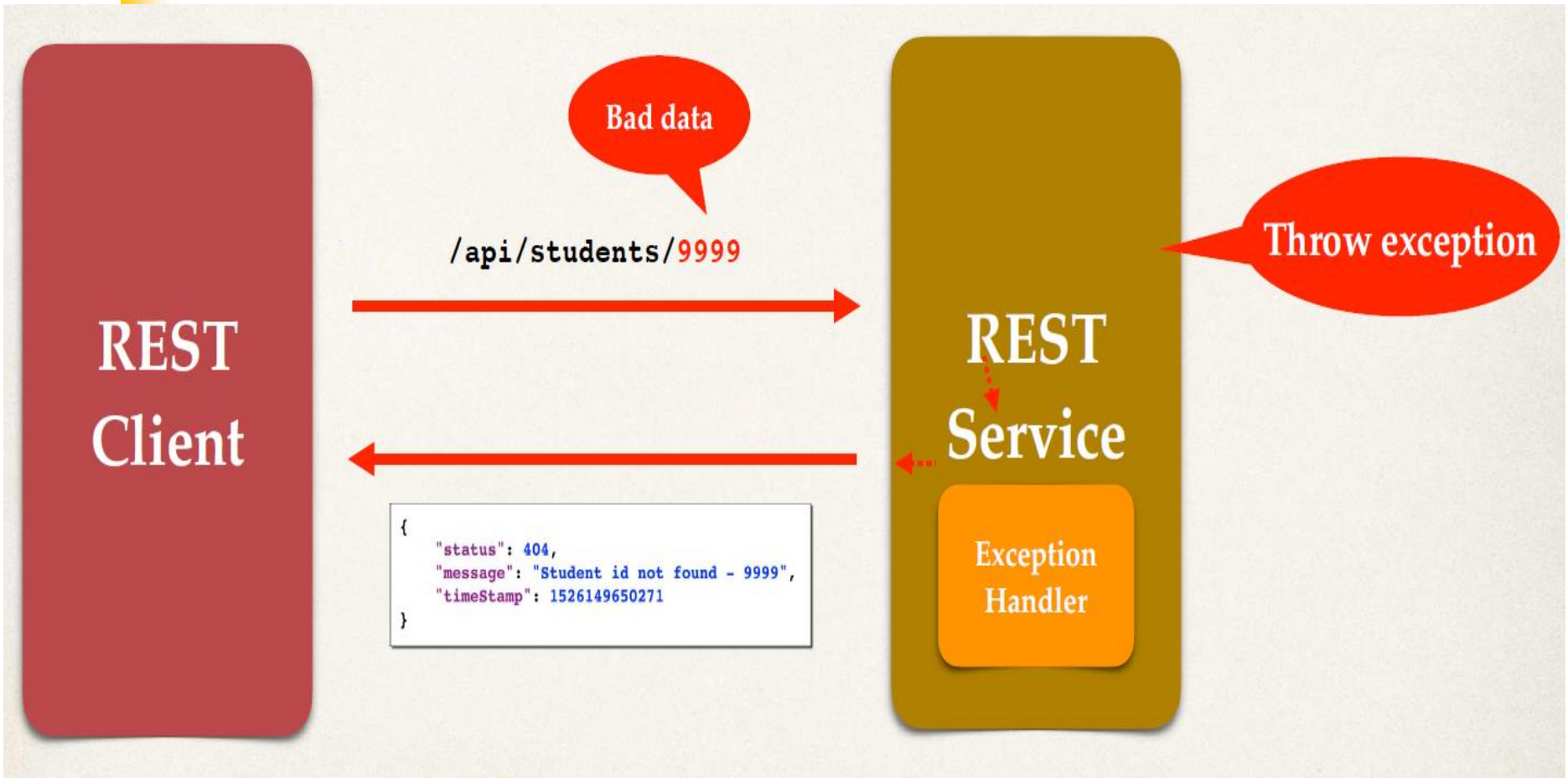
Jackson will convert Student to JSON

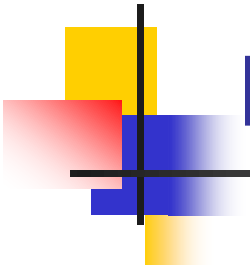
Spring REST - Exception Handling

- Bad student id of 9999 …



Spring REST Exception Handling





Development Process

1. Create a custom error response class

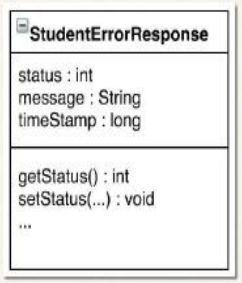
2. Create a custom exception class

3. Update REST service to throw exception if student not found

4. Add an exception handler method using `@ExceptionHandler`

Step 1: Create custom error response class

- The custom error response class will be sent back to client as JSON
- We will define as Java class (POJO)
- Jackson will handle converting it to JSON



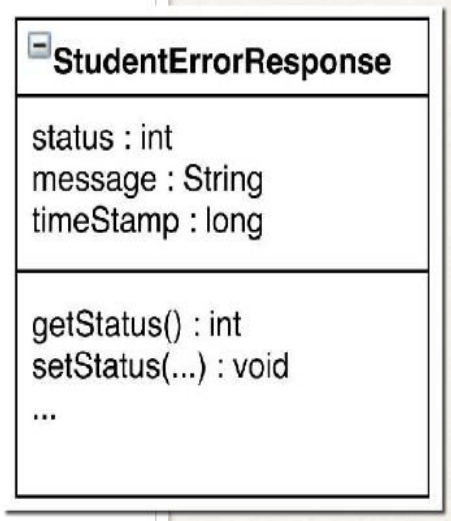
You can define any custom fields that you want to track

```
{
    "status": 404,
    "message": "Student id not found - 9999",
    "timeStamp": 1526149650271
}
```

Step 1: Create custom error response class

File: StudentErrorResponse.java

```
public class StudentErrorResponse {

  private int status;
  private String message;
  private long timeStamp;

  // constructors

  // getters / setters

}
```




```
{
    "status": 404,
    "message": "Student id not found - 9999",
    "timeStamp": 1526149650271
}
```

Step 2: Create custom student exception

- The custom student exception will used by our REST service
- In our code, if we can't find student, then we'll throw an exception
- Need to define a custom student exception class
  - `StudentNotFoundException`

Step 2: Create custom student exception

File: StudentNotFoundException.java

```
public class StudentNotFoundException extends RuntimeException {

  public StudentNotFoundException(String message) {
    super(message);
  }

}
```

Call super class constructor

Step 3: Update REST service to throw exception

File: StudentRestController.java

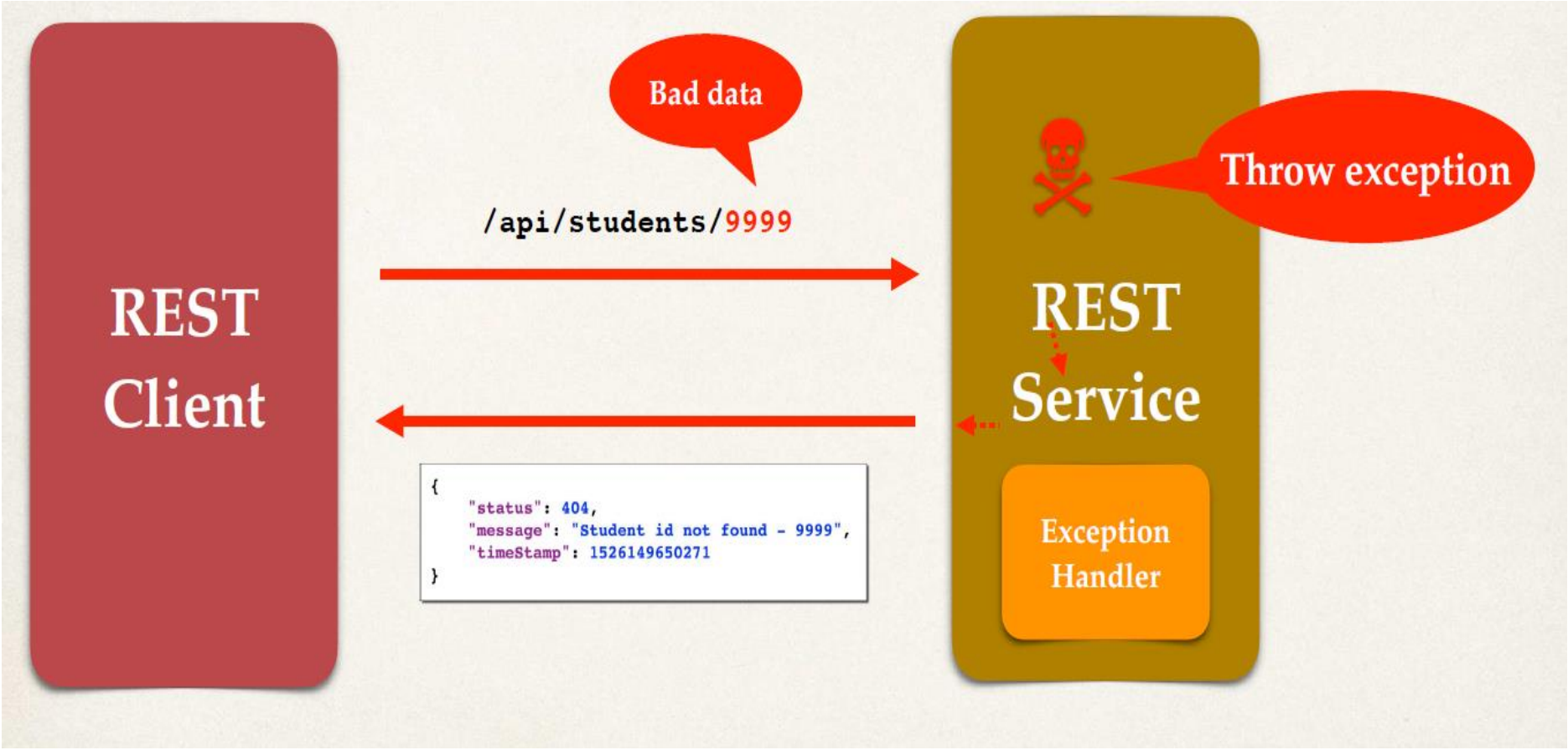
```
@RestController
@RequestMapping("/api")
public class StudentRestController {

  @GetMapping("/students/{studentId}")
  public Student getStudent(@PathVariable int studentId) {

    // check the studentId against list size

    if ( (studentId >= theStudents.size()) || (studentId < 0) ) {
      throw new StudentNotFoundException("Student id not found - " + studentId);
    }

    return theStudents.get(studentId);

  }
  ...
}
```

Could also check results from DB

Throw exception

Happy path

Spring REST Exception Handling



Step 4: Add exception handler method

- Define exception handler method(s) with `@ExceptionHandler` annotation
- Exception handler will return a `ResponseEntity`
- `ResponseEntity` is a wrapper for the HTTP response object
- `ResponseEntity` provides fine-grained control to specify:
  - HTTP status code, HTTP headers and Response body

Step 4: Add exception handler method

java

Exception handler method

Type of the response body

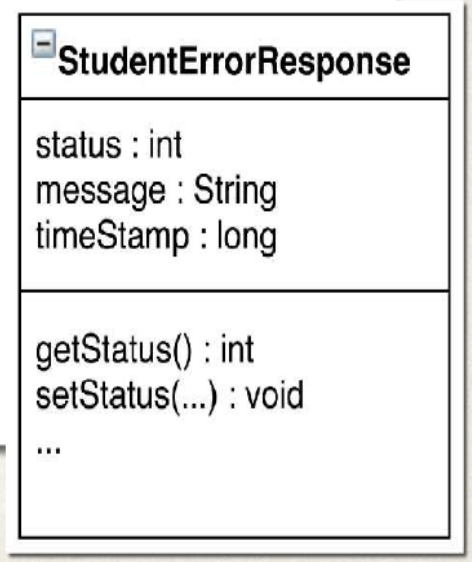
Exception type to handle / catch

```
pi")
public class StudentRestController {
  ...

  @ExceptionHandler
  public ResponseEntity<StudentErrorResponse> handleException(StudentNotFoundException exc) {

    StudentErrorResponse error = new StudentErrorResponse();

    error.setStatus(HttpStatus.NOT_FOUND.value());
    error.setMessage(exc.getMessage());
    error.setTimeStamp(System.currentTimeMillis());

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

}
```

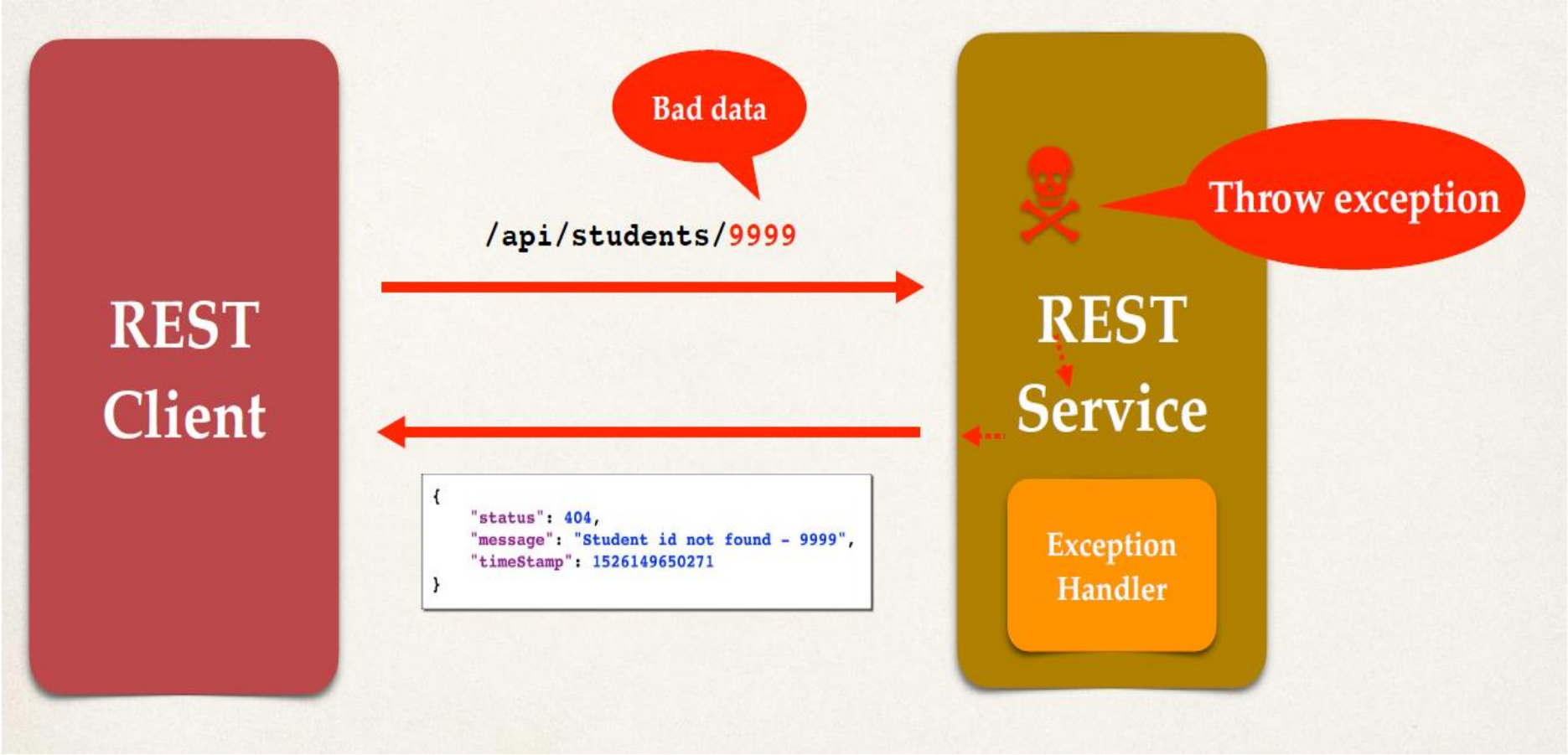
```
{
    "status": 404,
    "message": "Student id not found - 9999",
    "timeStamp": 1526149650271
}
```

Body

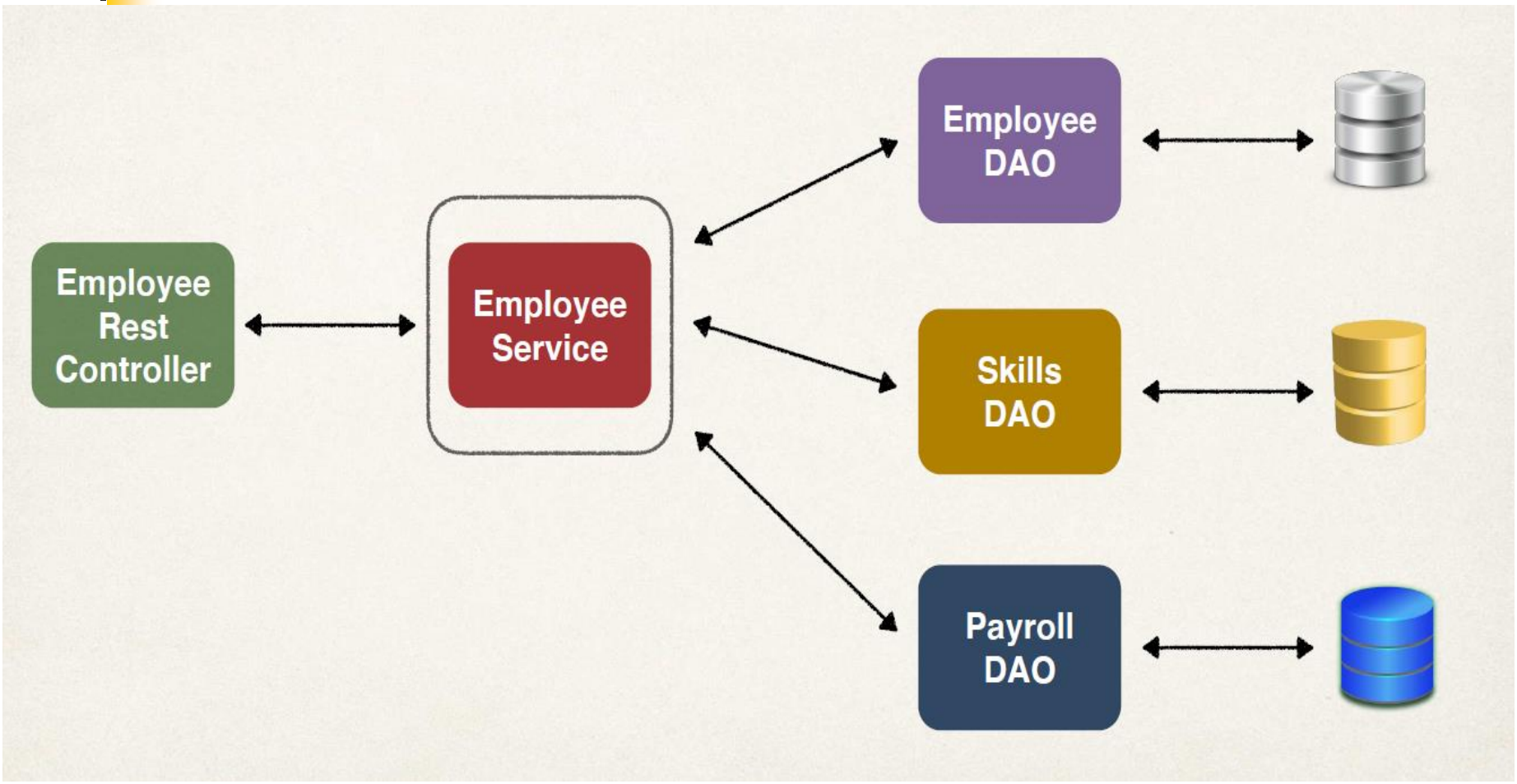
Status code



Spring REST Exception Handling

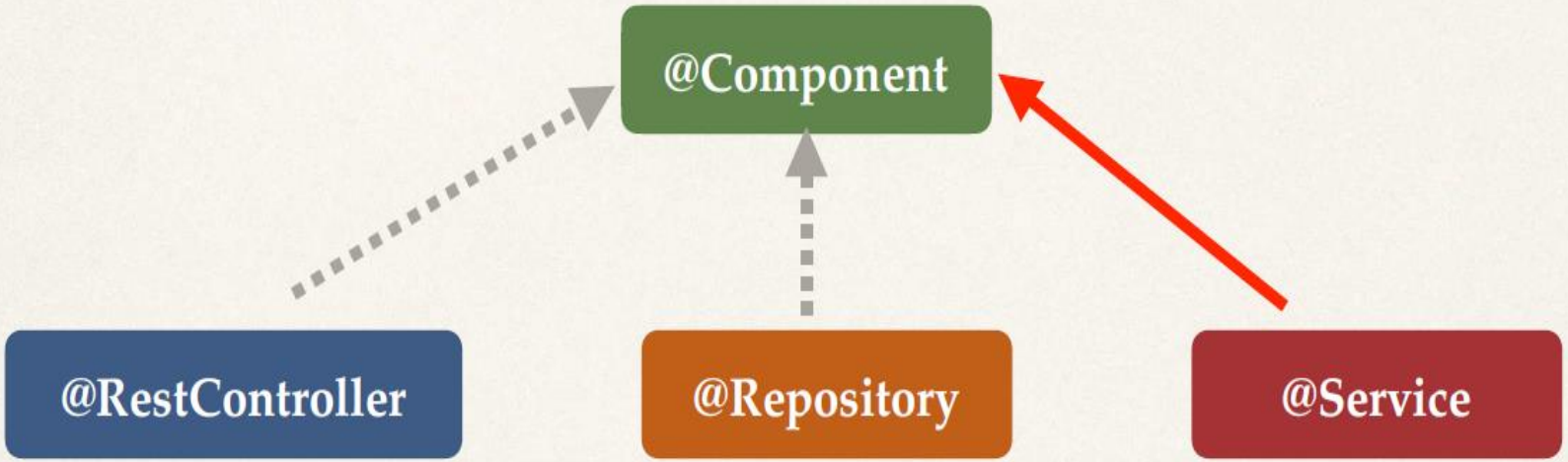


Integrate Multiple Data Sources



Specialized Annotation for Services

- Spring provides the **@Service** annotation



Specialized Annotation for Services

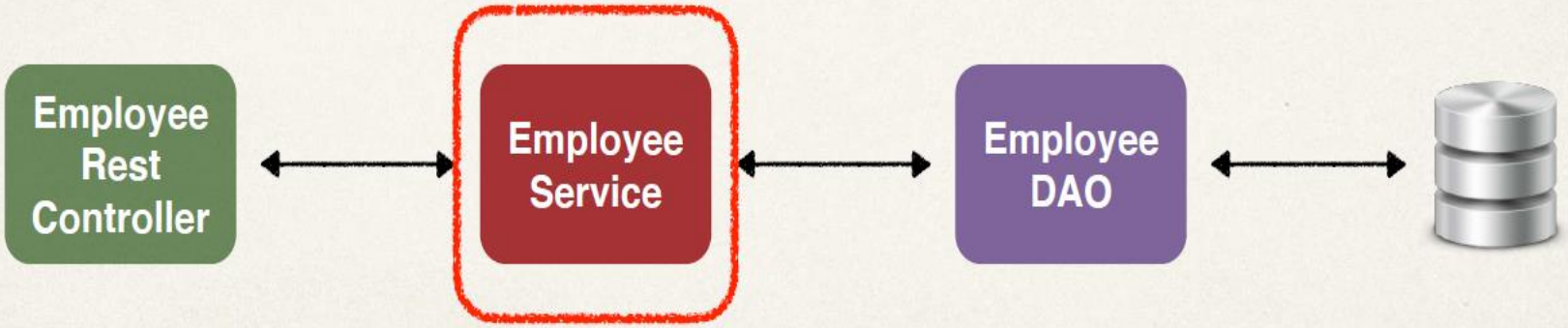
- **@Service** applied to Service implementations
- Spring will automatically register the Service implementation
  - thanks to component-scanning

Employee Service

1. Define Service interface

2. Define Service implementation

   - Inject the EmployeeDAO



Step 1: Define Service interface

```
public interface EmployeeService {

   List<Employee> findAll();

}
```

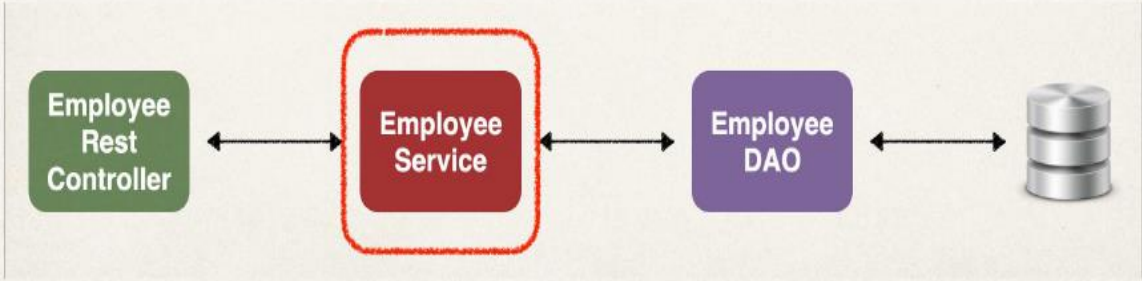
Step 2: Define Service implementation

@Service - enables component scanning

```
@Service
public class EmployeeServiceImpl implements EmployeeService {

  // inject EmployeeDAO ...

  @Override
  public List<Employee> findAll() {
    return employeeDAO.findAll();
  }
}
```

DAO: Find, Add, Update and Delete
Service Layer - Best Practice

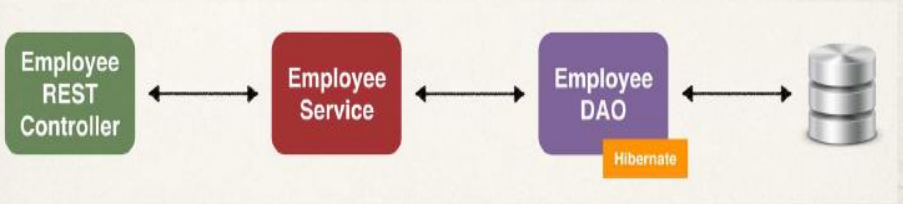
- Best practice is to apply transactional boundaries at the service layer
- It is the service layer's responsibility to manage transaction boundaries
- For implementation code
  - Apply @Transactional on service methods
  - Remove @Transactional on DAO methods if they already exist



Development Process

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

DAO methods



DAO: Get a single employee

```
@Override
public Employee findById(int theId) {

  // get employee
  Employee theEmployee = entityManager.find(Employee.class, theId);

  // return employee
  return theEmployee;
}
```

DAO: Add or Update employee

Note: We don't use @Transactional at DAO layer
It will be handled at Service layer

if id == 0
then save/insert
else update

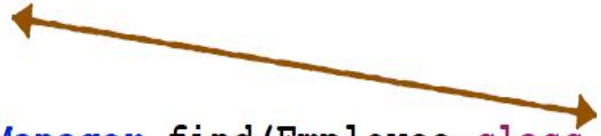
```
@Override
public Employee save(Employee theEmployee) {

  // save or update the employee
  Employee dbEmployee = entityManager.merge(theEmployee);

  // return dbEmployee
  return dbEmployee;
}
```

Return dbEmployee
It has updated id from the database
(in the case of insert)

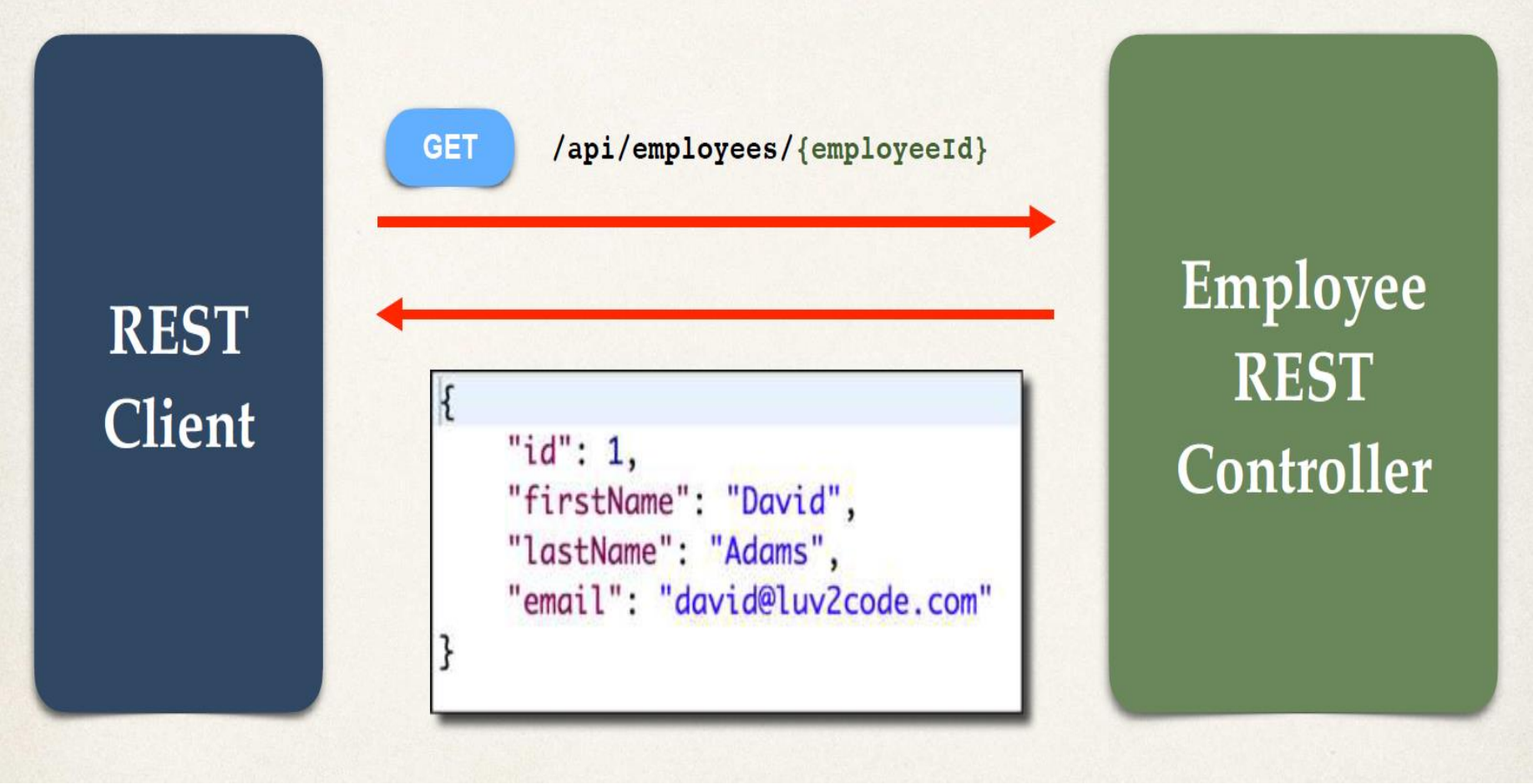
DAO: Delete an existing employee

Note: We don't use @Transactional at DAO layer
It will be handled at Service layer

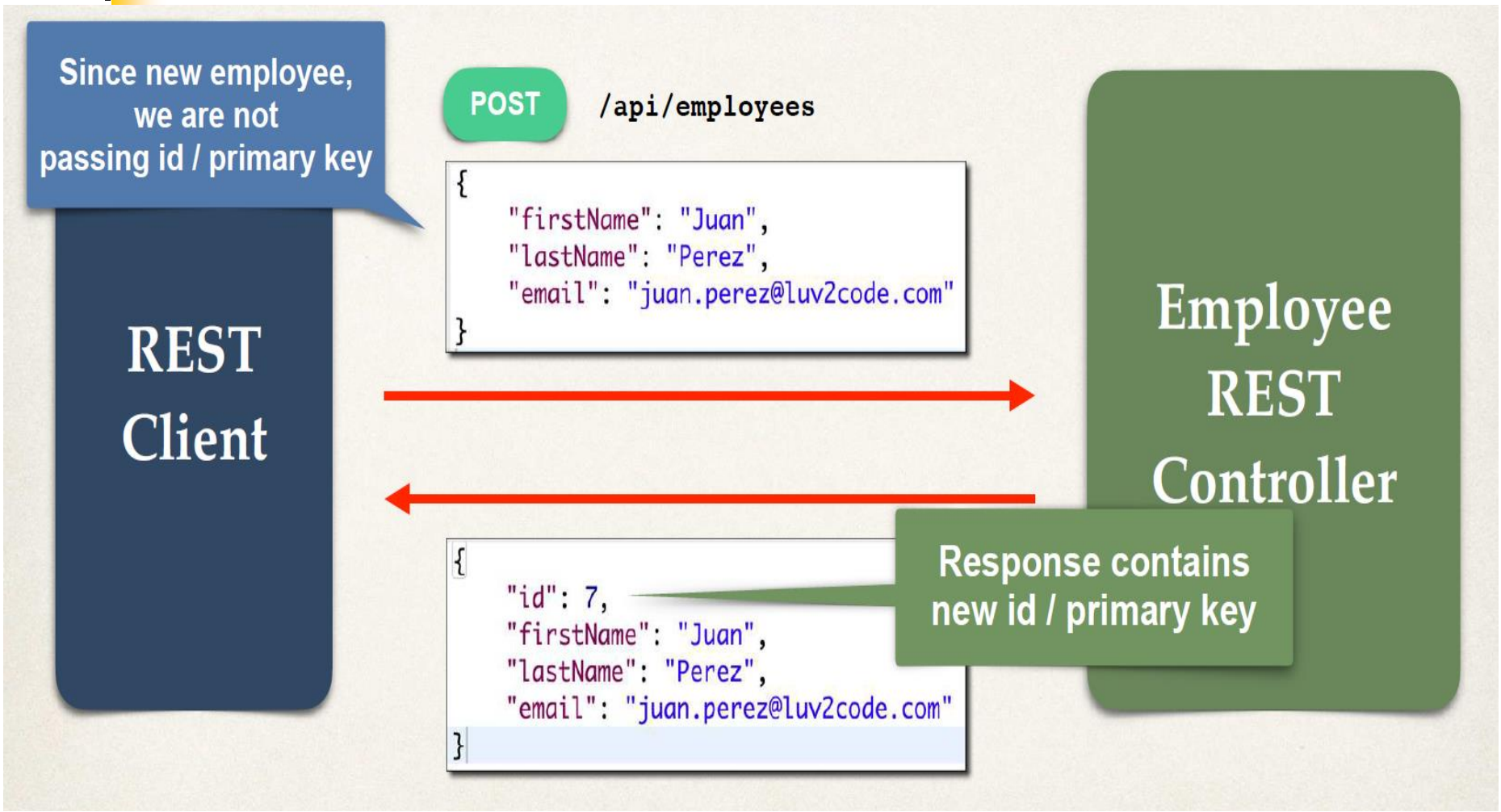
```
@Override
public void deleteById(int theId) {

  // find the employee by id
  Employee theEmployee = entityManager.find(Employee.class, theId);

  // delete the employee
  entityManager.remove(theEmployee);
}
```



Read a Single Employee



Create a New Employee

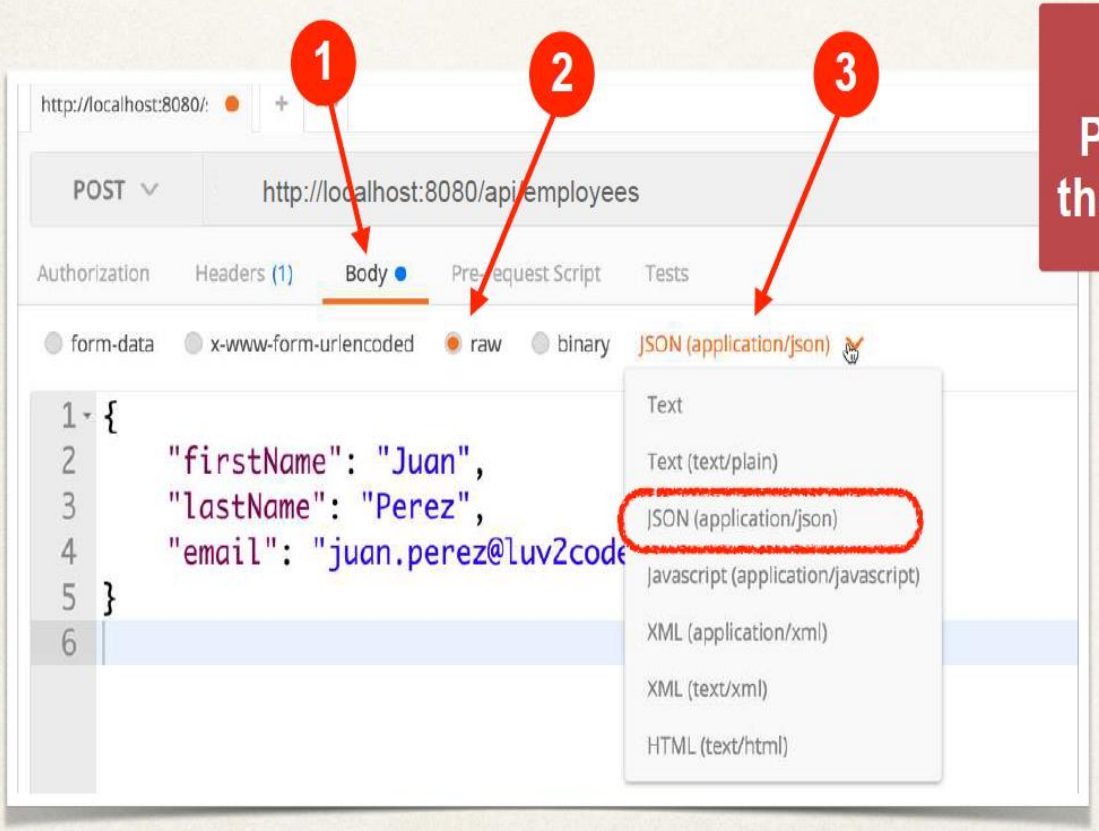


Sending JSON to Spring REST Controllers

- When sending JSON data to Spring REST Controllers
- For controller to process JSON data, need to set a HTTP request header
  - `Content-type: application/json`
- Need to configure REST client to send the correct HTTP request header

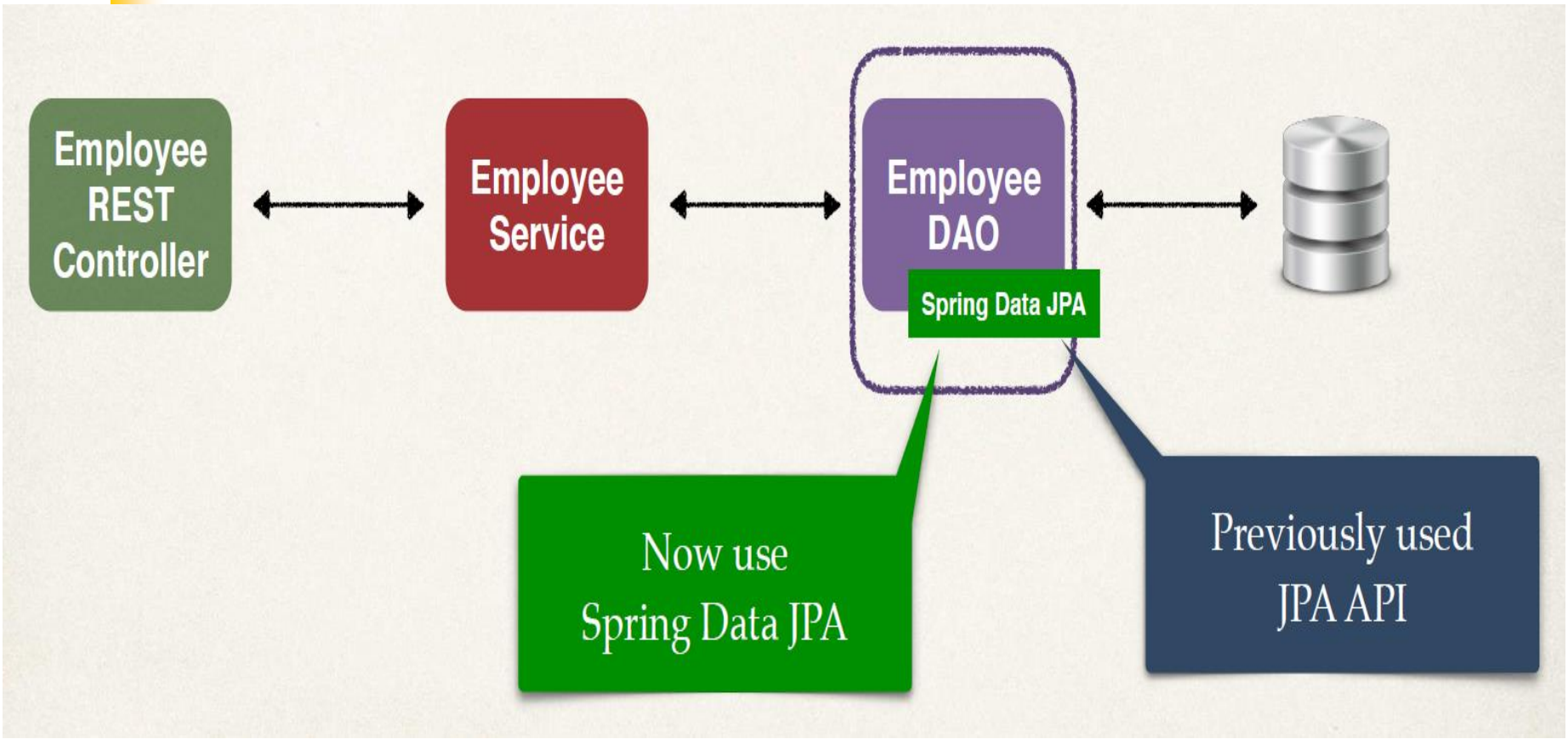
Postman - Sending JSON in Request Body

- Must set HTTP request header in Postman



Based on these configs, Postman will automatically set the correct HTTP request header

Spring Data JPA in Spring Boot
Application Architecture



Difference between JPA (Java Persistence API) and Spring Data JPA

Feature	JPA (Java Persistence API)	Spring Data JPA
Definition	A *specification* (standard) for object-relational mapping (ORM) in Java. Examples: Hibernate, EclipseLink are JPA implementations.	A *library* built by Spring to simplify JPA-based database operations even further.
What it Provides	Defines annotations like `@Entity`, `@Table`, `@Id`, `@GeneratedValue`, and basic `EntityManager` API for CRUD operations.	Provides `CrudRepository`, `JpaRepository`, `PagingAndSortingRepository` interfaces, automatically generating CRUD queries, pagination, and dynamic queries.
Level	Low-level, boilerplate-heavy (you manually write queries using `EntityManager`).	High-level, reduces boilerplate a lot (repositories automatically provide save, delete, findAll, etc.).
Boilerplate	More — you have to create queries manually (`entityManager.persist()`, `entityManager.find()`, etc.).	Less — you just define interfaces, Spring Data JPA generates the implementation automatically.
Usage Example	`entityManager.persist(student);` and `entityManager.find(Student.class, id);`	`studentRepository.save(student);` and `studentRepository.findById(id);`
Query Writing	You must manually create JPQL or native SQL queries if needed.	You can define methods by naming conventions like `findByName()`, or use `@Query` for custom queries.
Framework	Part of Java EE / Jakarta EE specification.	A part of the larger Spring ecosystem. Depends on JPA underneath.
Custom Queries	Must write queries manually with `EntityManager` or JPQL.	Supports method naming (`findByNameAndAge`), JPQL via `@Query`, or Native Queries easily.

Difference between JPA (Java Persistence API) and Spring Data JPA

JPA API Example

```java
@PersistenceContext
private EntityManager entityManager;

public Student findStudent(Long id) {
    return entityManager.find(Student.class, id);
}
```

Spring Data JPA Example

```java
public interface StudentRepository extends JpaRepository<Student, Long> {
    // No implementation needed, Spring generates it
}
```

The Problem

- We saw how to create a DAO for **`Employee`**
- What if we need to create a DAO for another entity?
  - **`Customer`**, **`Student`**, **`Product`**, **`Book`** …
- Do we have to repeat all of the same code again???

```
public interface EmployeeDAO {

    public List<Employee> findAll();

    public Employee findById(int theId);

    public void save(Employee theEmployee);

    public void deleteById(int theId);

}
```

```
@Repository
public class EmployeeDAOJpaImpl implements EmployeeDAO {

    private EntityManager entityManager;

    @Autowired
    public EmployeeDAOJpaImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }

    @Override
    public List<Employee> findAll() {

        // create a query
        TypedQuery<Employee> theQuery =
                entityManager.createQuery("from Employee", Employee.class);

        // execute query and get result list
        List<Employee> employees = theQuery.getResultList();

        // return the results
        return employees;
    }

    @Override
    public Employee findById(int theId) {

        // get employee
        Employee theEmployee =
                entityManager.find(Employee.class, theId);

        // return employee
        return theEmployee;
    }
```

Creating DAO

- You may have noticed a pattern with creating DAOs

```
@Override
public Employee findById(int theId) {

  // get data
  Employee theData = entityManager.find(Employee.class, theId);

  // return data
  return theData;
}
```

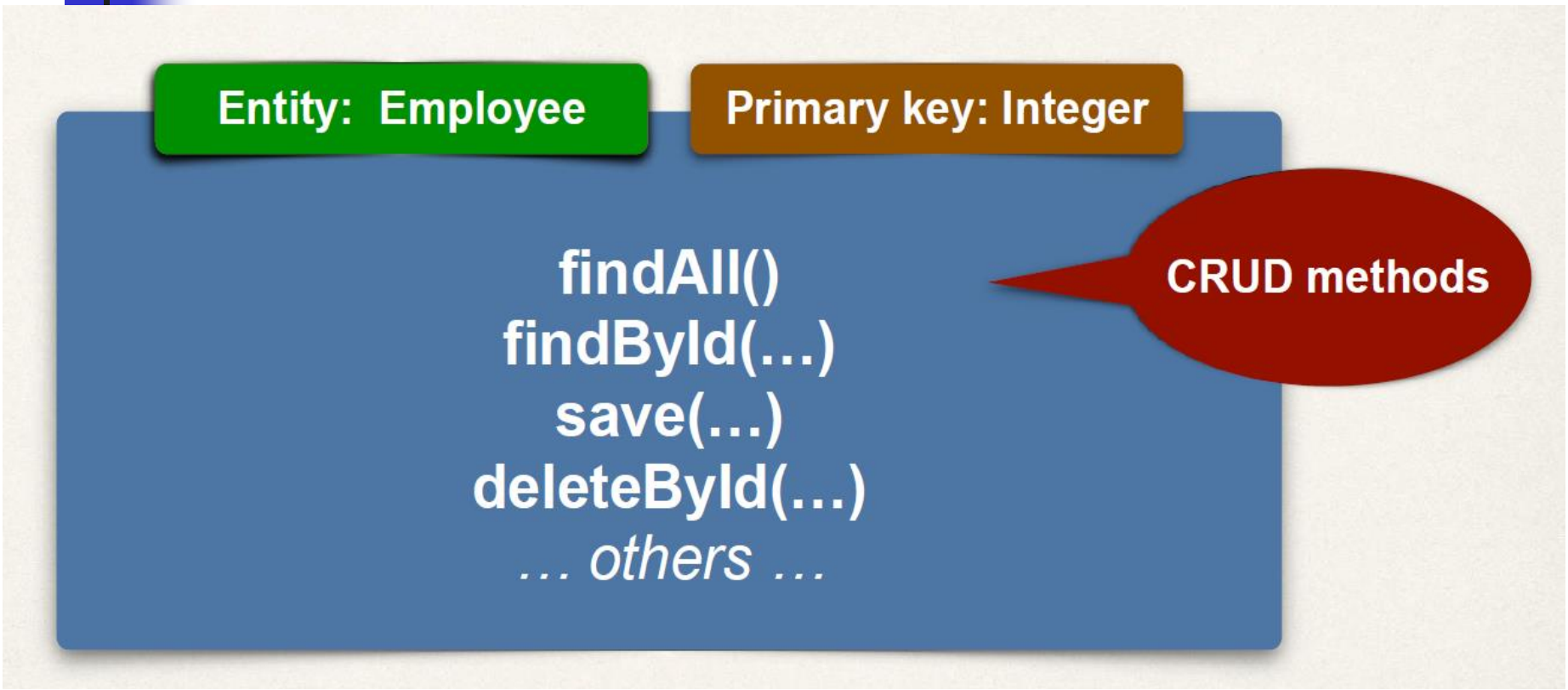
Most of the code is the same

Only difference is the entity type and primary key

Entity type

Primary key

Creating DAO for all entity type



Create a DAO, Plug in entity type and primary key and then give all of the basic CRUD features for free

Spring Data JPA - Solution

- **Spring Data JPA** is the solution!!!!

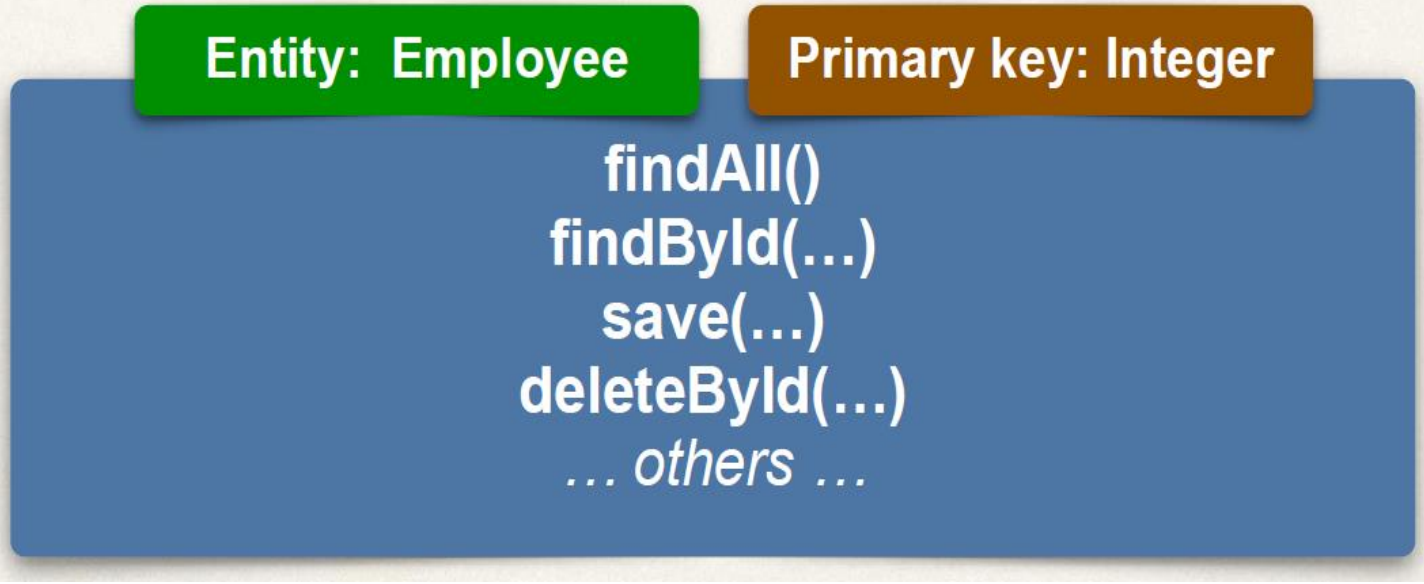
https://spring.io/projects/spring-data-jpa

- Create a DAO and just plug in your entity type and primary key
- Spring will give you a CRUD implementation for FREE
  - Helps to minimize boiler-plate DAO code

More than 70% reduction in code … depending on use case

JpaRepository

- Spring Data JPA provides the interface: **`JpaRepository`**
- Exposes methods (some by inheritance from parents)



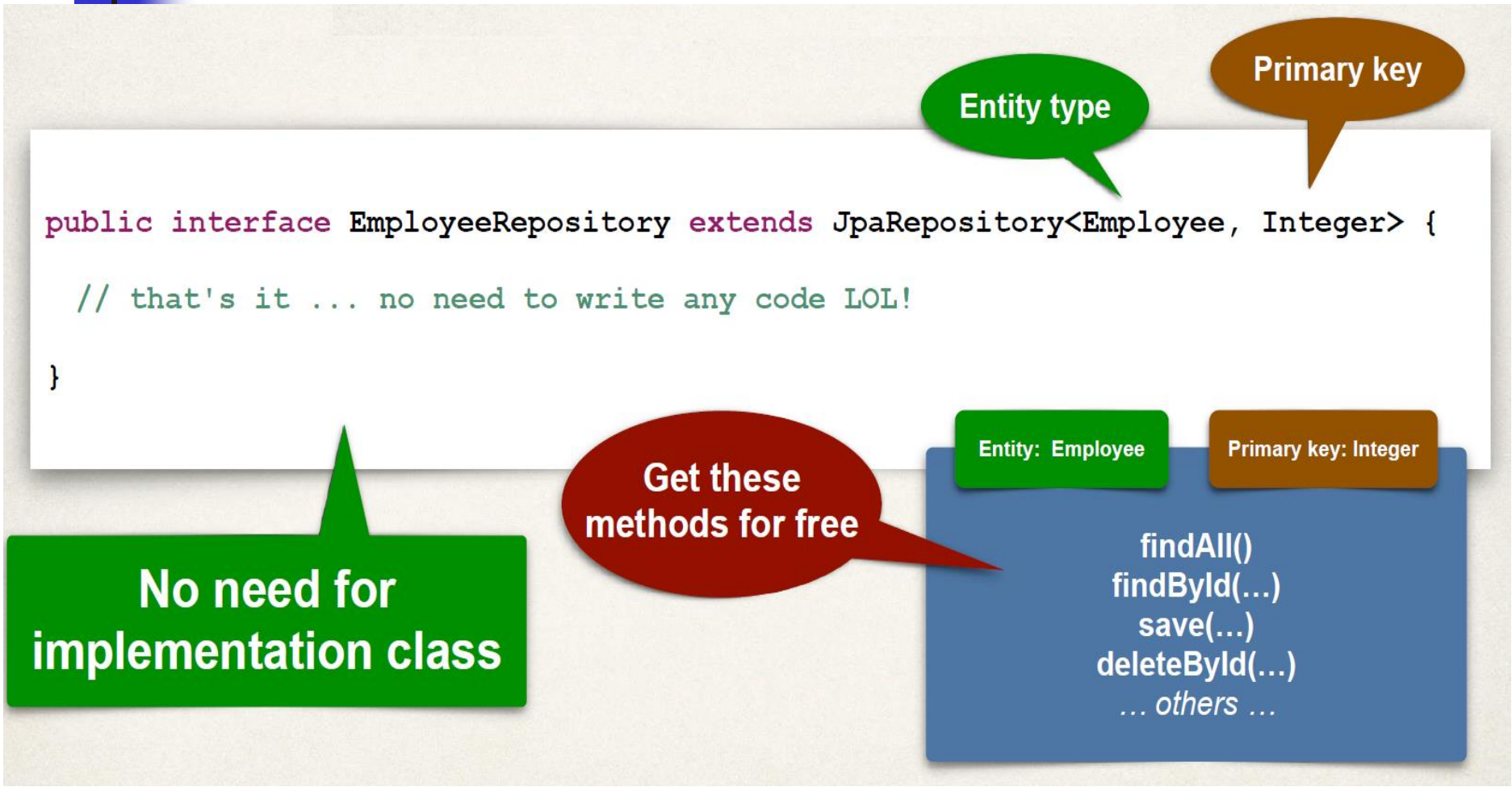
Development Process

1. Extend **`JpaRepository`** interface

2. Use your Repository in your app

No need for implementation class

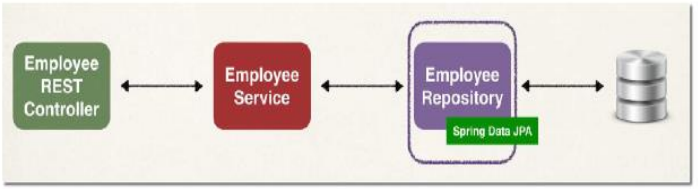
Step 1: Extend JpaRepository interface



Step 2: Use Repository in your app

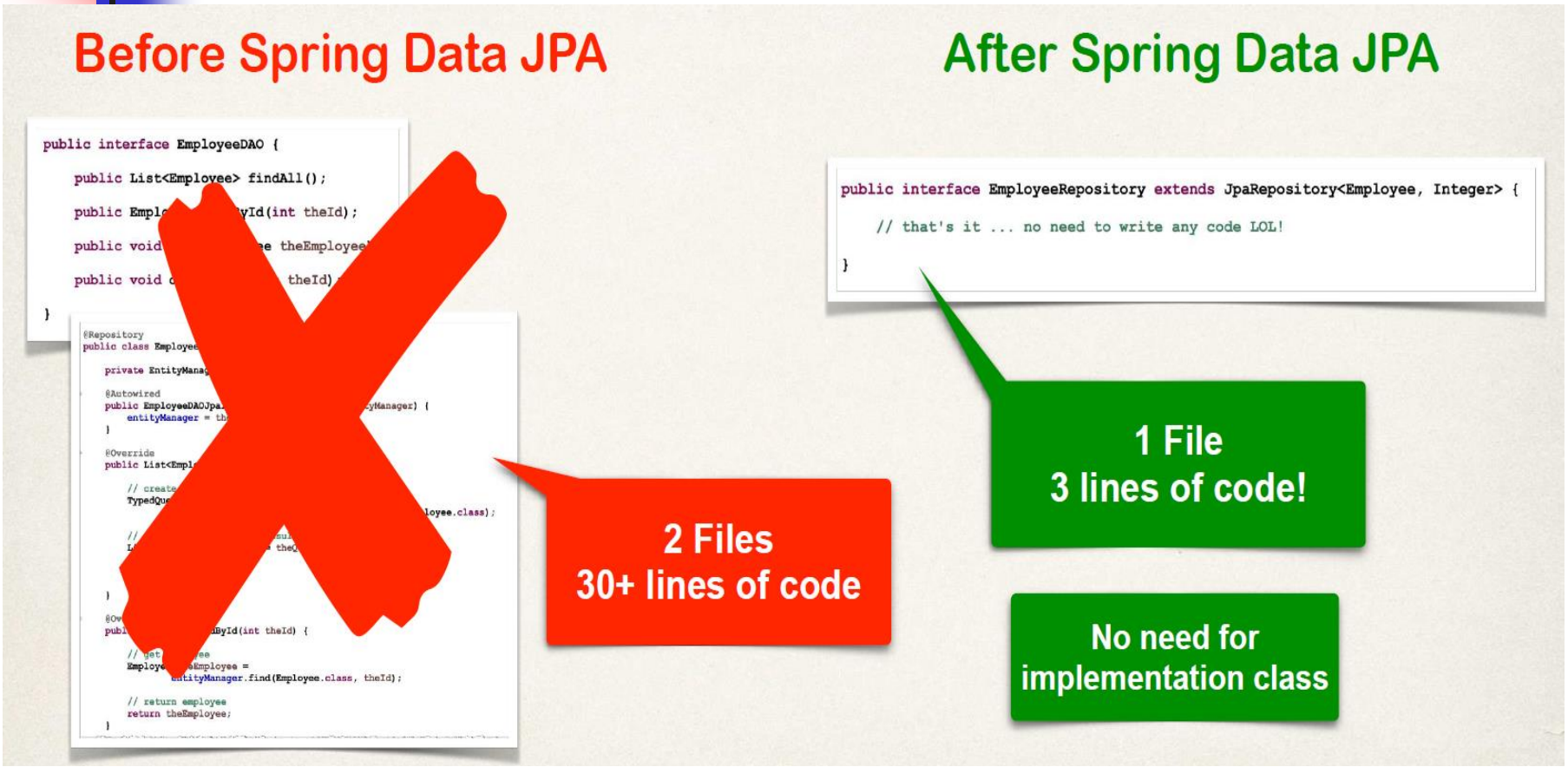
```
@Service
public class EmployeeServiceImpl implements EmployeeService {

  private EmployeeRepository employeeRepository;

  @Autowired
  public EmployeeServiceImpl(EmployeeRepository theEmployeeRepository) {
    employeeRepository = theEmployeeRepository;
  }

  @Override
  public List<Employee> findAll() {
    return employeeRepository.findAll();
  }
  ...
}
```

Our repository



Magic method that is available via repository

Minimized Boilerplate Code




```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}
```

Spring Data REST - Solution

- **Spring Data REST** is the solution!!!!
- Leverages your existing **`JpaRepository`**
- Spring will give you a REST CRUD implementation for FREE
  - Helps to minimize boiler-plate REST code!!!
  - No new coding required!!!

Create a REST API using existing JpaRepository (entity, primary key) and give all of the basic REST API CRUD features for free

Spring Data REST - How Does It Work?

- Spring Data REST will scan your project for **JpaRepository**
- Expose REST APIs for each entity type for your **JpaRepository**

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}
```

REST Endpoints

- By default, Spring Data REST will create endpoints based on entity type
- Simple pluralized form
  - First character of Entity type is lowercase
  - Then just adds an "s" to the entity

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}
```

Development Process

1. Add Spring Data REST to your Maven POM file

That's it!!!

Absolutely NO CODING required

Step 1: Add Spring Data REST to POM file

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

That's it!!!

Absolutely NO CODING required

Spring Data REST will scan for JpaRepository

HTTP Method		CRUD Action
POST	/employees	Create a new employee
GET	/employees	Read a list of employees
GET	/employees/{employeeId}	Read a single employee
PUT	/employees/{employeeId}	Update an existing employee
DELETE	/employees/{employeeId}	Delete an existing employee

Get these REST endpoints for free

Hibernate Advanced Mappings

- In the database, you most likely will have
  - Multiple Tables
  - Relationships between Tables
- Need to model this with Hibernate

Advanced Mappings

- One-to-One
- One-to-Many, Many-to-One
- Many-to-Many

One-to-One Mapping (Uni-Directional)

```java
@Entity
public class Instructor {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "instructor_detail_id") // Foreign key
    private InstructorDetail instructorDetail;

    // getters and setters
}
```

@OneToOne sets up the relationship.

@JoinColumn defines which column will hold the foreign key.

FetchType.LAZY → Only loads InstructorDetail **when accessed**.

```java
@Entity
public class InstructorDetail {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String hobby;

    // getters and setters
}
```

One-to-One Mapping (Uni-Directional)

```java
import org.springframework.data.jpa.repository.JpaRepository;

public interface InstructorRepository extends JpaRepository<Instructor, Long> {
    // Optional: You can add custom queries if needed later
}
```

Thanks to Spring Data JPA:

- You **don't need to manually write SQL** to find/save/update/delete.
- It automatically provides methods like .findById(), .save(), .findAll(), .delete(), etc.
- **No need to manually implement it** — Spring Data JPA generates the implementation at runtime

Spring Boot **automatically injects** (@Autowired) this repository wherever you need it.

```java
@Service
public class InstructorService {

    @Autowired
    private InstructorRepository instructorRepository;

    public Instructor findInstructor(Long id) {
        return instructorRepository.findById(id).orElse(null);
    }
}
```

One-to-One Mapping (Uni-Directional)

Now imagine this code in your service:

```
Instructor instructor = instructorRepository.findById(1L).orElse(null);
System.out.println("Instructor loaded!");
```

When findById(1L) is called:

- **Instructor** will be loaded from the database.
- **InstructorDetail** will NOT be loaded yet!
- **Only a proxy object** is created. No InstructorDetail loaded yet

```
String hobby = instructor.getInstructorDetail().getHobby();
System.out.println("Hobby: " + hobby);
```

At this point, Hibernate will send an extra SQL query:

- It will now fetch **InstructorDetail from the database**

Why is FetchType.LAZY Useful?

It **improves performance**.
It **avoids unnecessary SQL queries** if you don't actually need related entities.
It **saves memory and processing time**.

One-to-One Mapping (Bi-Directional)

In **Bi-Directional Mapping**, **both entities** (for example Instructor and InstructorDetail) have references to each other:

- Instructor knows about its InstructorDetail
- InstructorDetail knows about its Instructor

So you can **navigate the relationship from both sides**!

Cascade operations from either side

If you delete/update one, the relationship ensures the other side is also handled correctly (especially important for CascadeType.ALL).

Better Query Possibilities

You can write JPA queries (JPQL) that go from InstructorDetail to Instructor easily without manually joining tables.

One-to-One Mapping (Bi-Directional)

```java
@Entity
public class Instructor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "instructor_detail_id") // Foreign Key in Instru
    private InstructorDetail instructorDetail;

    // getters and setters
}
```

```java
@Entity
public class InstructorDetail {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String hobby;

    @OneToOne(mappedBy = "instructorDetail", cascade = CascadeType.ALL)
    private Instructor instructor;

    // getters and setters
}
```

@OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY):

- **Cascade.ALL** → When you **save/delete/update** Instructor, it also **affects** the InstructorDetail.
- **Fetch.LAZY** → InstructorDetail will **NOT** be fetched immediately — only when you call getInstructorDetail().

@JoinColumn(name = "instructor_detail_id") → This creates a **foreign key column** (instructor_detail_id) inside the **Instructor** table linking to InstructorDetail.id

@OneToOne(mappedBy = "instructorDetail", cascade = CascadeType.ALL): mappedBy = "instructorDetail" → Means InstructorDetail is the inverse side. JPA looks inside the Instructor class to manage the relationship.

One-to-Many Mapping (Bi-Directional)

```java
@Entity
public class Instructor {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "instructor", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private List<Course> courses;

    // getters and setters
}
```

```java
@Entity
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "instructor_id") // Foreign Key
    private Instructor instructor;

    // getters and setters
}
```

Cascade Types

Cascade refers to a mechanism where operations performed on a parent entity are automatically propagated to its associated child entities.

Cascade Type	Meaning
PERSIST	When saving parent, child is saved too.
REMOVE	When deleting parent, child is deleted too.
MERGE	When updating parent, child is updated too.
DETACH	When detaching parent, child is detached too.
REFRESH	Refresh child entity when parent is refreshed.
ALL	Applies all of the above actions.

Example: One-to-Many Relationship

Consider the example where an Instructor (parent) has multiple Courses (children).

•**Parent Entity**: Instructor (because it contains the reference to the collection of Courses).

•**Child Entity**: Course (because it's referenced by the Instructor and dependent on the Instructor).

CascadeType.PERSIST

When the parent (Instructor) is saved, the child (Course) is saved automatically as well.

```
Instructor instructor = new Instructor();
instructor.setName("John Doe");

Course course1 = new Course();
course1.setCourseName("Math 101");

Course course2 = new Course();
course2.setCourseName("Science 101");

// Associate the courses with the instructor
instructor.setCourses(Arrays.asList(course1, course2));

// Save instructor, and courses will be saved as well due to CascadeType.PERSIST
instructorRepository.save(instructor);
```

CascadeType.REMOVE

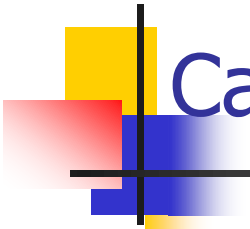
When the parent (Instructor) is deleted, the child (Course) is also deleted.

```
Instructor instructor = instructorRepository.findById(1L).orElse(null);
if (instructor != null) {
    instructorRepository.delete(instructor); // When instructor is deleted, courses will also be d
}
```

CascadeType.MERGE

When the parent (Instructor) is updated, the child (Course) is also updated.

```
Instructor instructor = instructorRepository.findById(1L).orElse(null);
if (instructor != null) {
    instructor.setName("John Smith");

    // Updating courses for the instructor
    for (Course course : instructor.getCourses()) {
        course.setCourseName(course.getCourseName() + " - Updated");
    }

    // Save the changes; courses will be merged (updated) as well due to CascadeType.MERGE
    instructorRepository.save(instructor);
}
```



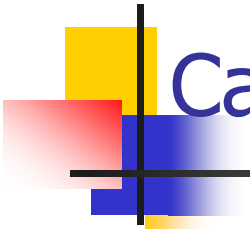
CascadeType.DETACH

- When the parent (Instructor) is detached from the persistence context, the child (Course) is also detached.
- **Detached** means the entity is no longer managed by the EntityManager.
- Changes to a detached entity will not be automatically persisted to the database unless you explicitly call merge() or refresh().
- You can manage detached entities outside the current transaction scope and reattach them when needed.

CascadeType.REFRESH

When the parent (Instructor) is refreshed, the child (Course) is also refreshed.

```
Instructor instructor = instructorRepository.findById(1L).orElse(null);
if (instructor != null) {
    entityManager.refresh(instructor);  // Refresh the instructor, courses will be refreshed too
}
```



CascadeType.ALL

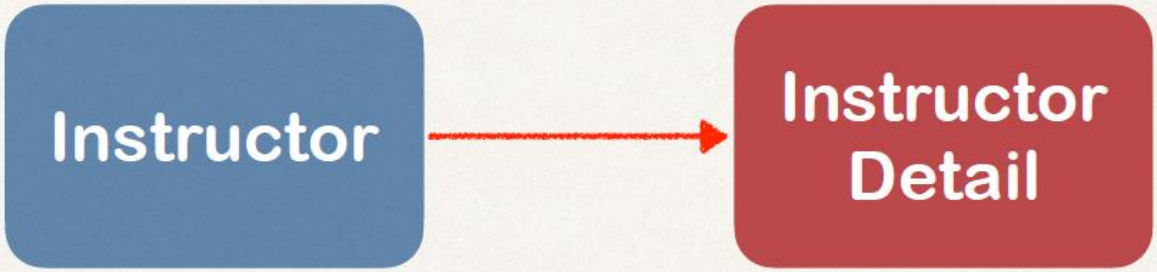
Applies all of the above actions (PERSIST, REMOVE, MERGE, DETACH, REFRESH).

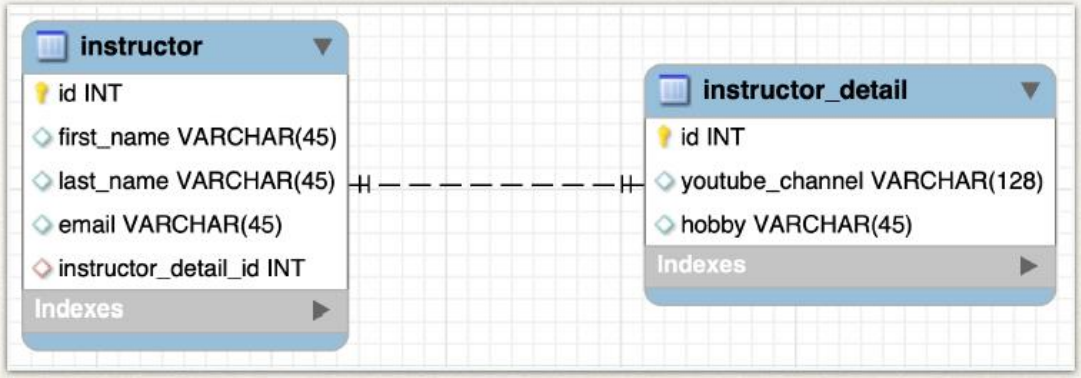
```
@OneToMany(cascade = CascadeType.ALL)
private List<Course> courses;
```

Effect: Every cascade action (save, remove, update, detach, refresh) will apply to both the Instructor and the Courses

One-to-One Mapping

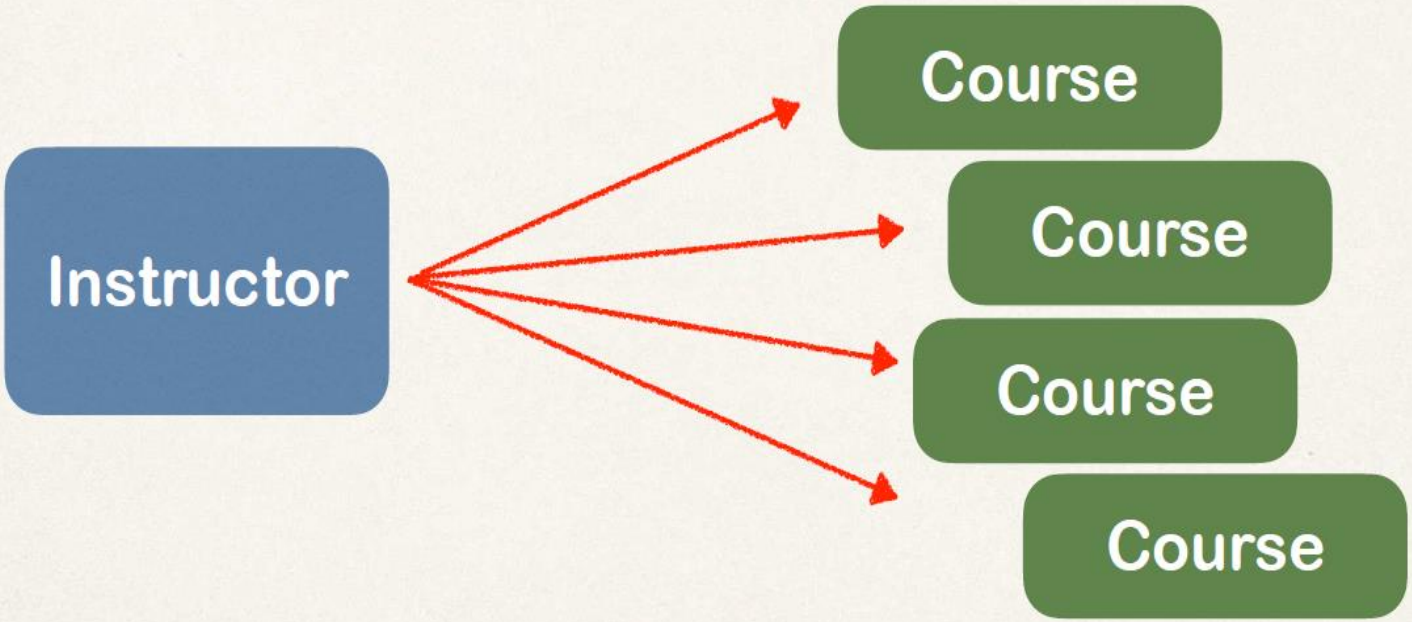
- An instructor can have an "instructor detail" entity
  - Similar to an "instructor profile"





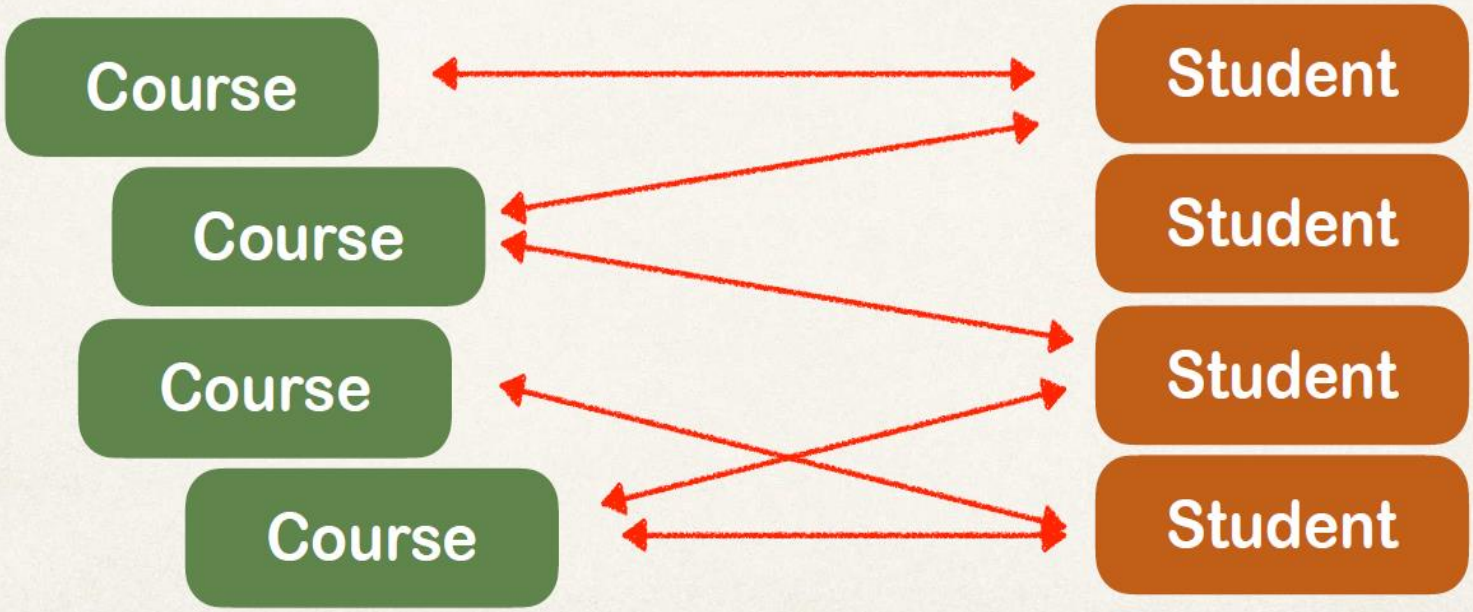
One-to-Many Mapping

- An instructor can have many courses



Many-to-Many Mapping

- A course can have many students
- A student can have many courses

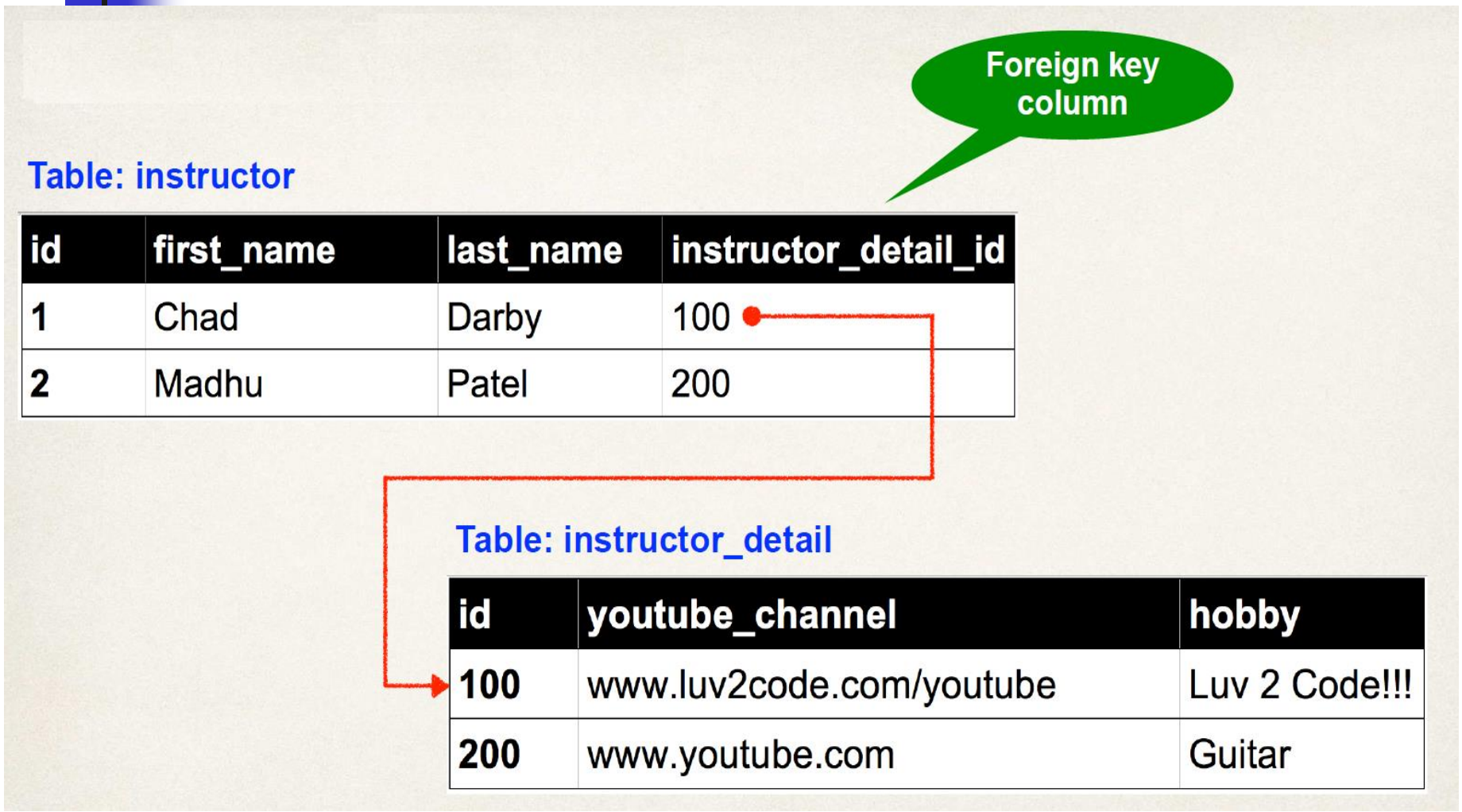


Important Database Concepts
Primary Key and Foreign Key

- Primary key and foreign key
- Cascade

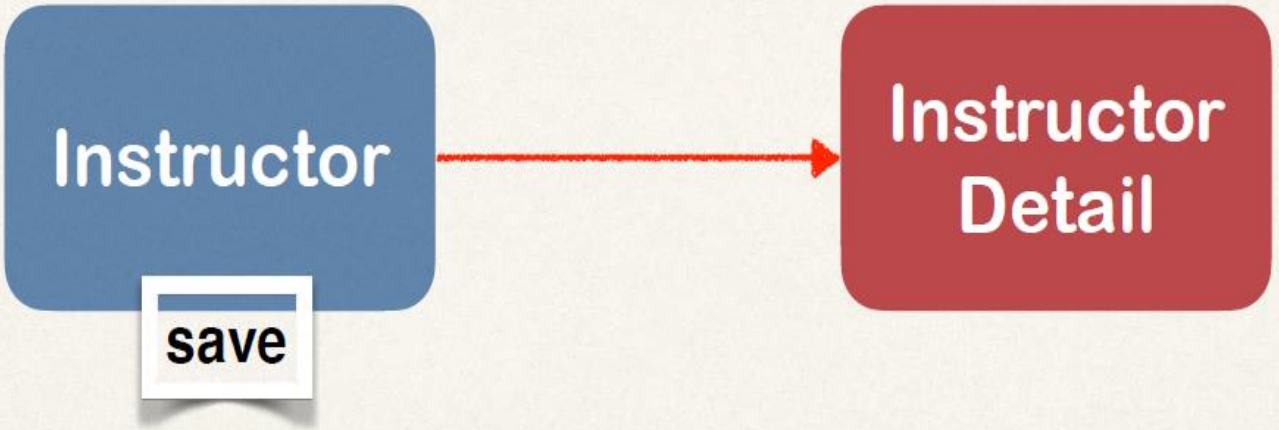
- Primary key: identify a unique row in a table
- Foreign key:
  - Link tables together
  - a field in one table that refers to primary key in another table

Foreign Key Example



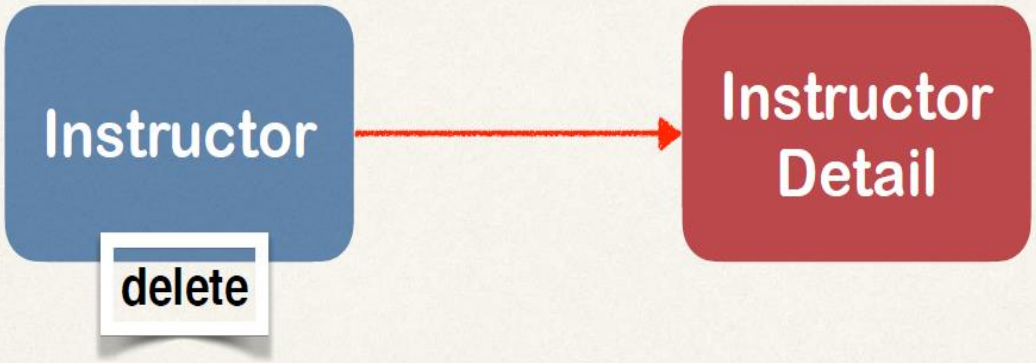
Cascade

- You can **cascade** operations
- Apply the same operation to related entities

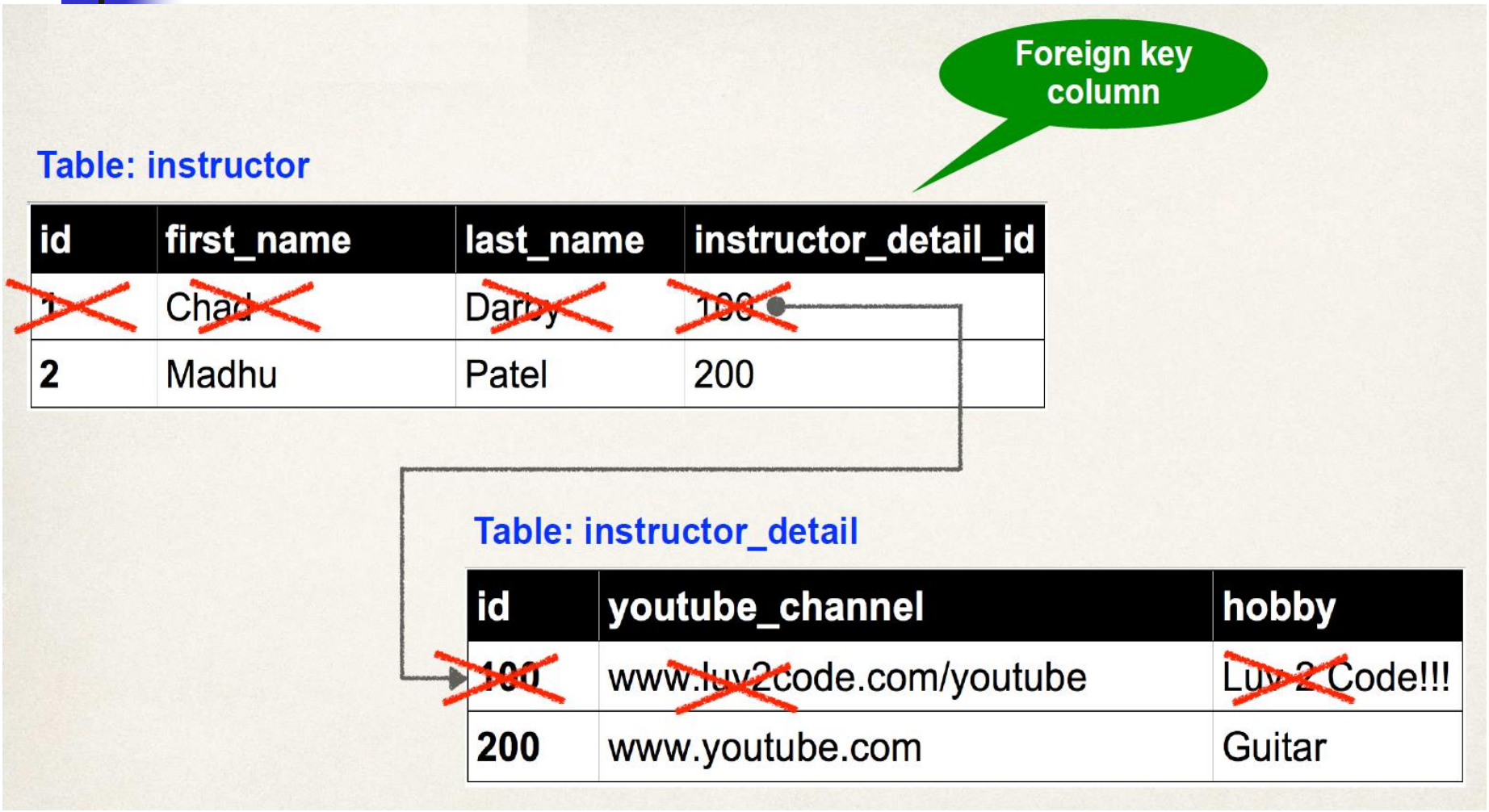


Cascade

- If we delete an **instructor**, we should also delete their **instructor_detail**
  - This is known as "CASCADE DELETE"

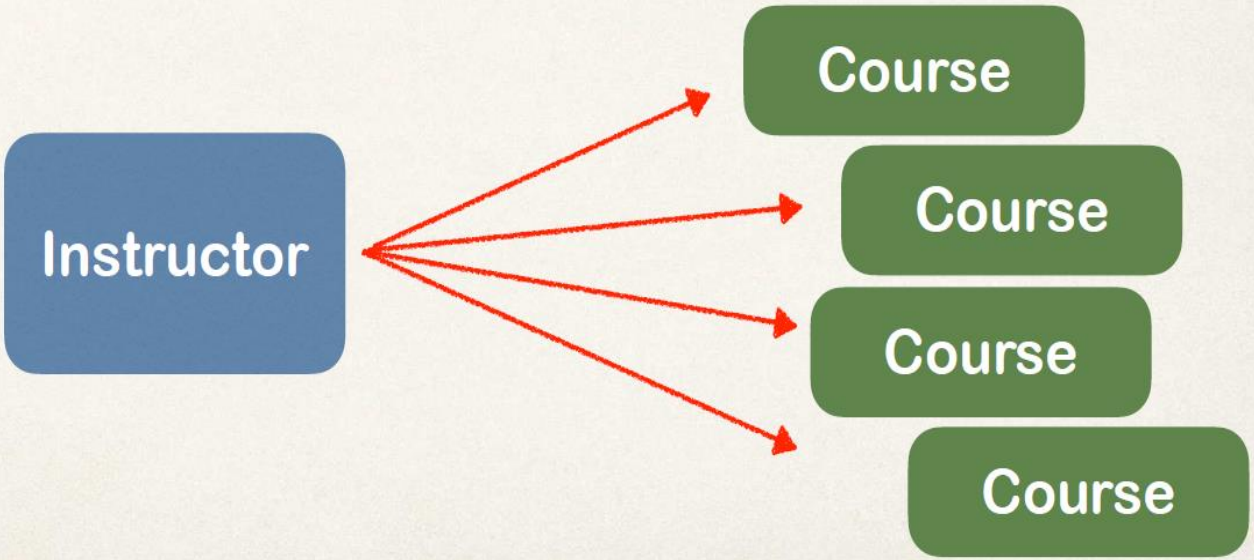


Cascade Delete

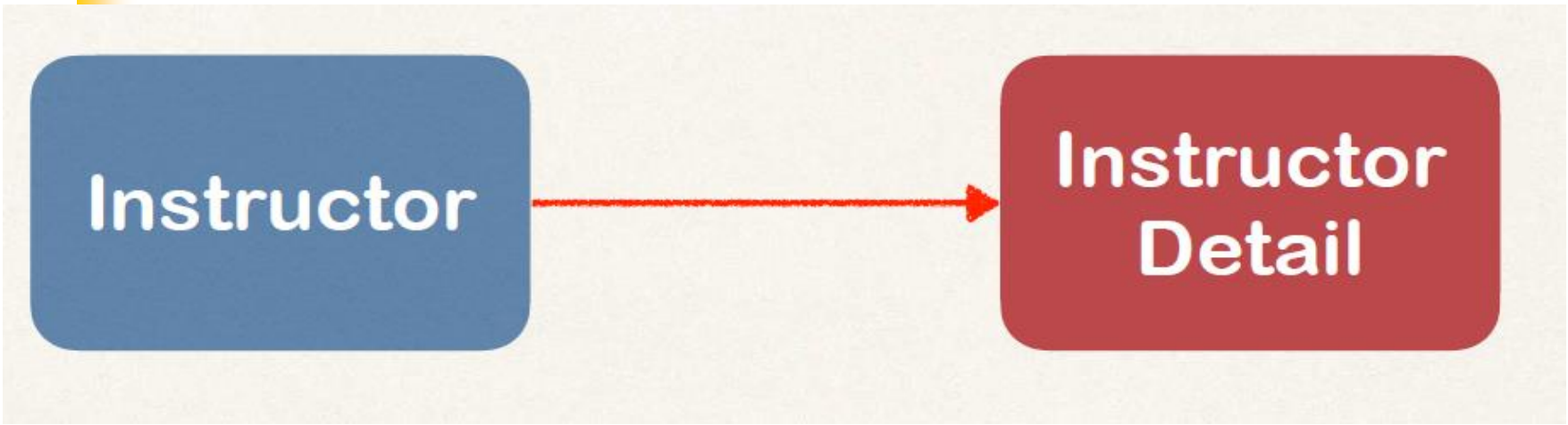


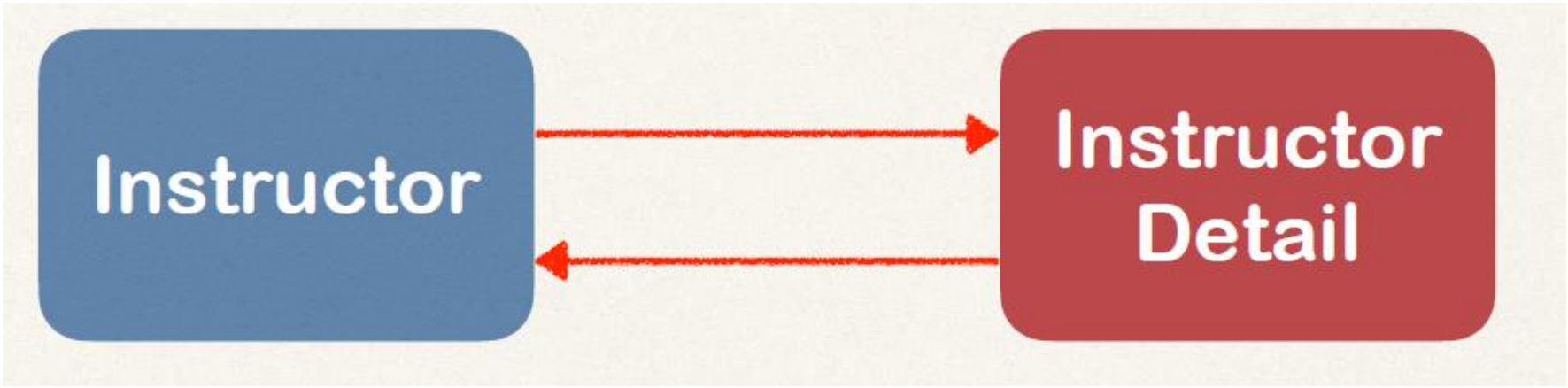
Fetch Types: Eager vs Lazy Loading

- When we fetch / retrieve data, should we retrieve EVERYTHING?
  - **Eager** will retrieve everything
  - **Lazy** will retrieve on request



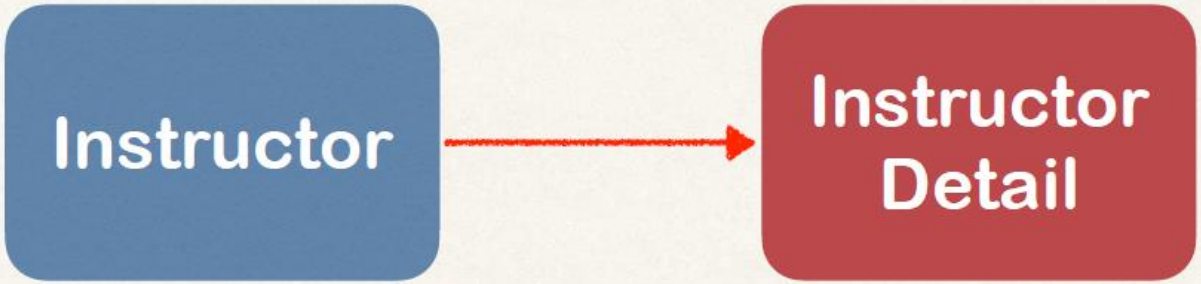
Uni-Directional - Bi-Directional

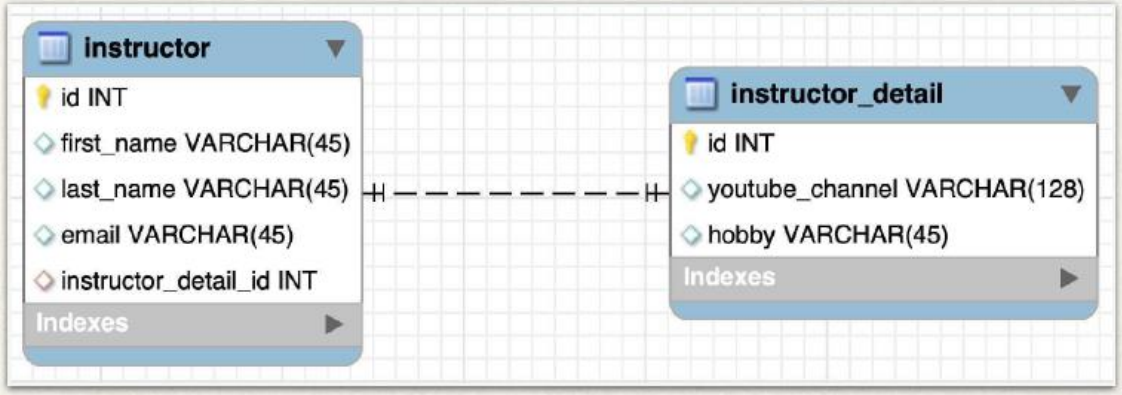




JPA / Hibernate
One-to-One Mapping

- An instructor can have an "instructor detail" entity
  - Similar to an "instructor profile"





Development Process: One-to-One

1. Prep Work - Define database tables
2. Create InstructorDetail class
3. Create Instructor class
4. Create Main App

table: instructor_detail

File: create-db.sql

```
CREATE TABLE `instructor_detail` (

  `id`              int(11)      NOT NULL AUTO_INCREMENT,
  `youtube_channel` varchar(128) DEFAULT NULL,
  `hobby`           varchar(45)  DEFAULT NULL,

  PRIMARY KEY (`id`)
);

...
```

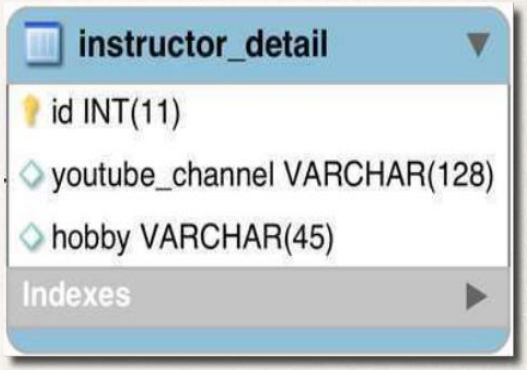


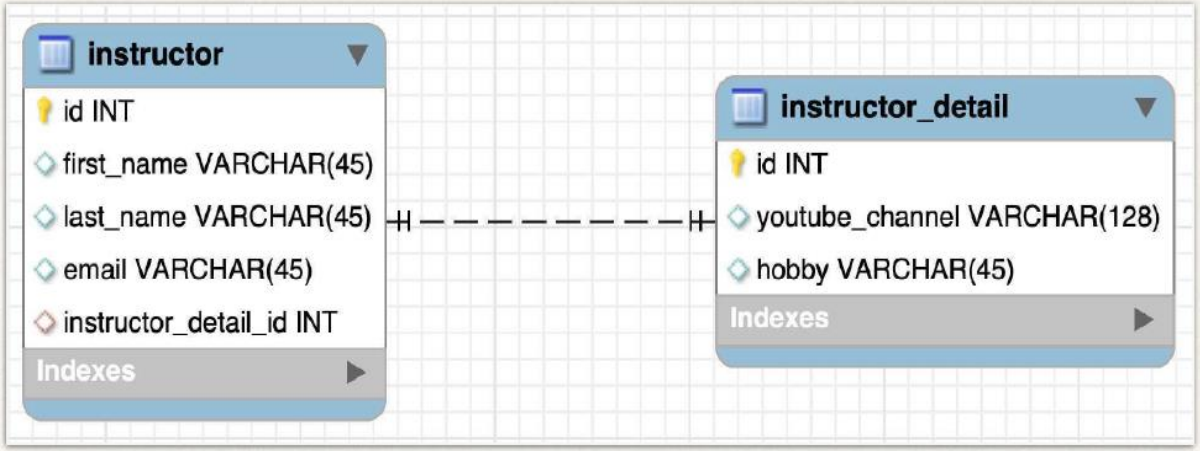
table: instructor

File: create-db.sql

```
...

CREATE TABLE `instructor` (

  `id`         int(11) NOT NULL AUTO_INCREMENT,
  `first_name` varchar(45) DEFAULT NULL,
  `last_name`  varchar(45) DEFAULT NULL,
  `email`      varchar(45) DEFAULT NULL,
  `instructor_detail_id` int(11) DEFAULT NULL,

  PRIMARY KEY (`id`)
    ...
);
```

Foreign Key

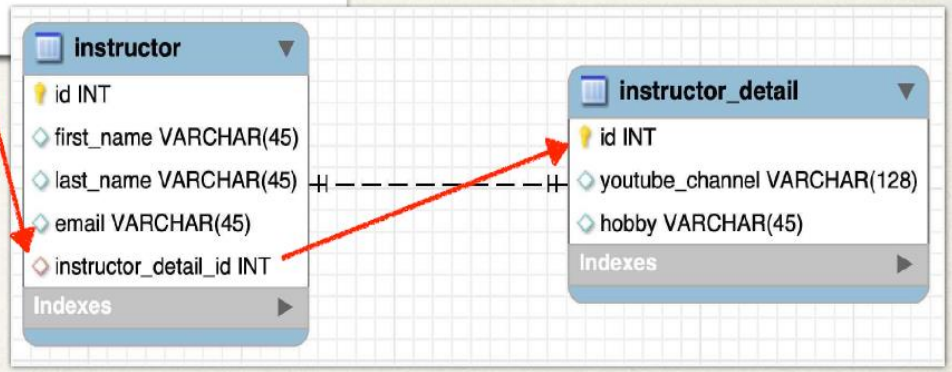
- Link tables together
- A field in one table that refers to primary key in another table



Defining Foreign Key

File: create-db.sql

```
...
CREATE TABLE `instructor` (
  ...

  CONSTRAINT `FK_DETAIL` FOREIGN KEY (`instructor_detail_id`)
  REFERENCES `instructor_detail` (`id`)

);
```

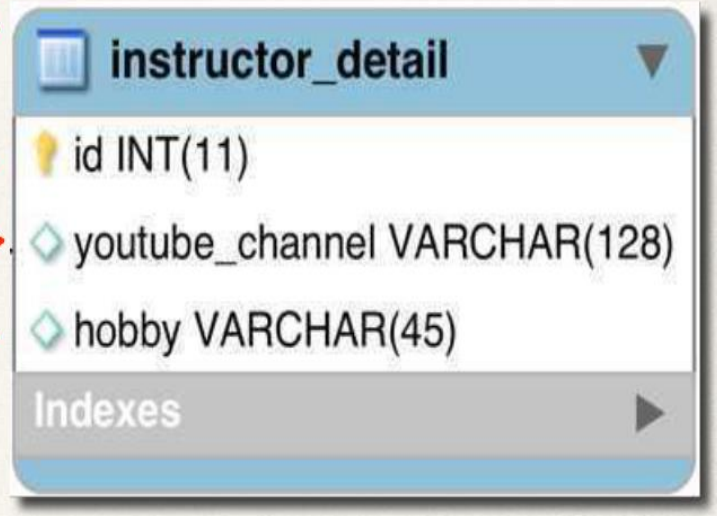


More on Foreign Key

- Main purpose is to preserve relationship between tables
  - Referential Integrity
- Prevents operations that would destroy relationship
- Ensures only valid data is inserted into the foreign key column
  - Can only contain valid reference to primary key in other table

Step 2: Create InstructorDetail class

```
@Entity
@Table(name="instructor_detail")
public class InstructorDetail {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private int id;

    @Column(name="youtube_channel")
    private String youtubeChannel;

    @Column(name="hobby")
    private String hobby;

    // constructors

    // getters / setters
}
```

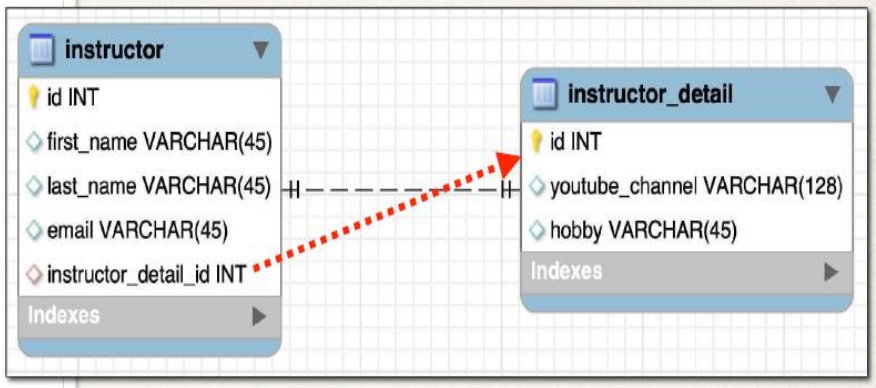


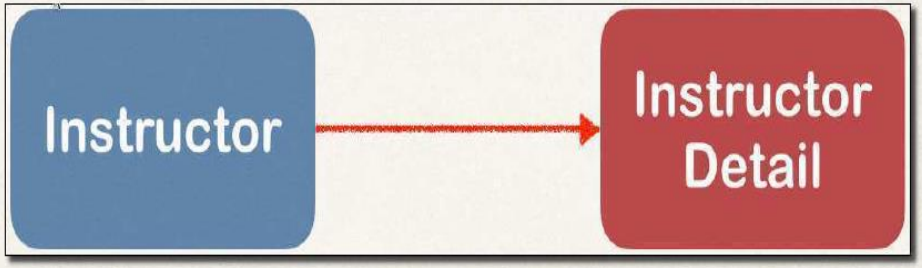
Step 3: Create Instructor class

```
@Entity
@Table(name="instructor")
public class Instructor {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private int id;

    @Column(name="first_name")
    private String firstName;

    @Column(name="last_name")
    private String lastName;

    @Column(name="email")
    private String email;

    ...
    // constructors, getters / setters
}
```

Step 3: Create Instructor class - @OneToOne

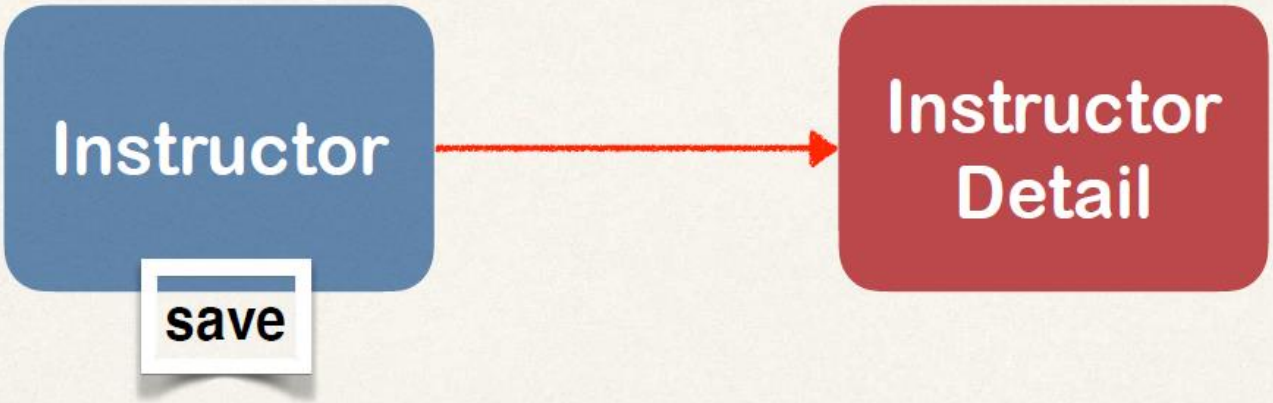
```
@Entity
@Table(name="instructor")
public class Instructor {
    ...

    @OneToOne
    @JoinColumn(name="instructor_detail_id")
    private InstructorDetail instructorDetail;

    ...
    // constructors, getters / setters
}
```



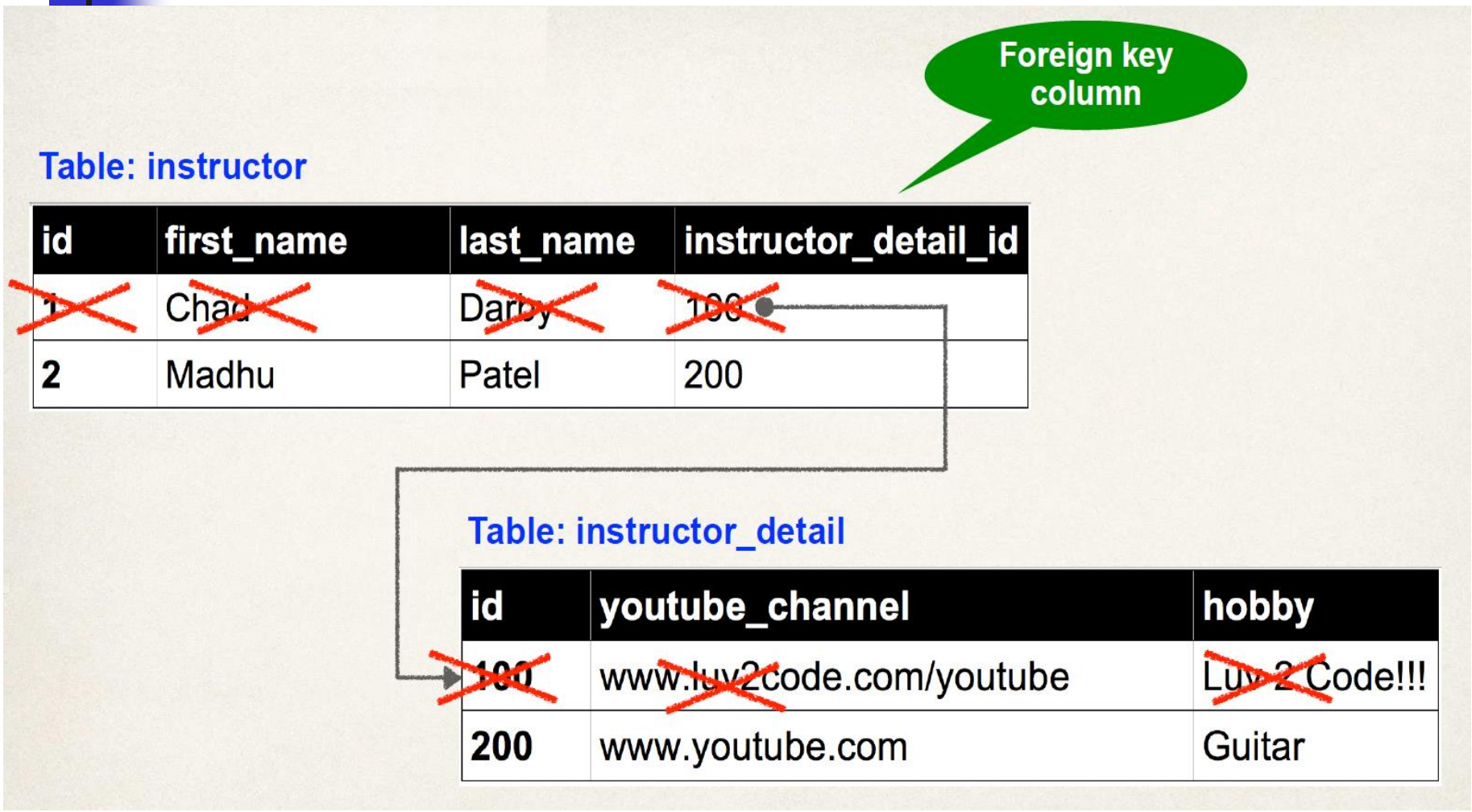


Cascade

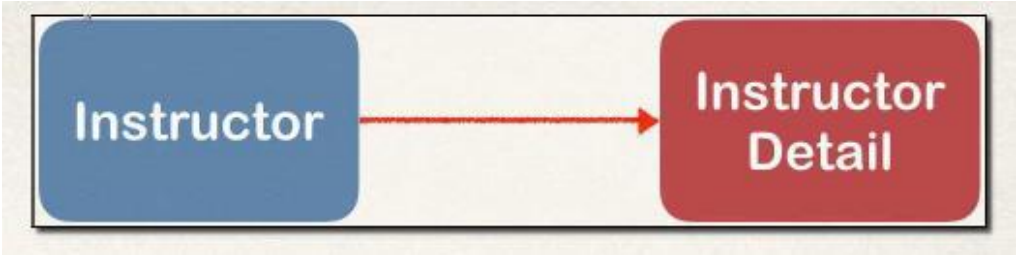
- Recall: You can **cascade** operations
- Apply the same operation to related entities



Cascade Delete



@OneToOne - Cascade Types



Cascade Type	Description
PERSIST	If entity is persisted / saved, related entity will also be persisted
REMOVE	If entity is removed / deleted, related entity will also be deleted
REFRESH	If entity is refreshed, related entity will also be refreshed
DETACH	If entity is detached (not associated w/ session), then related entity will also be detached
MERGE	If entity is merged, then related entity will also be merged
ALL	All of above cascade types

Configure Cascade Type

```
@Entity
@Table(name="instructor")
public class Instructor {
   ...

   @OneToOne(cascade=CascadeType.ALL)
   @JoinColumn(name="instructor_detail_id")
   private InstructorDetail instructorDetail;

   ...
   // constructors, getters / setters
}
```

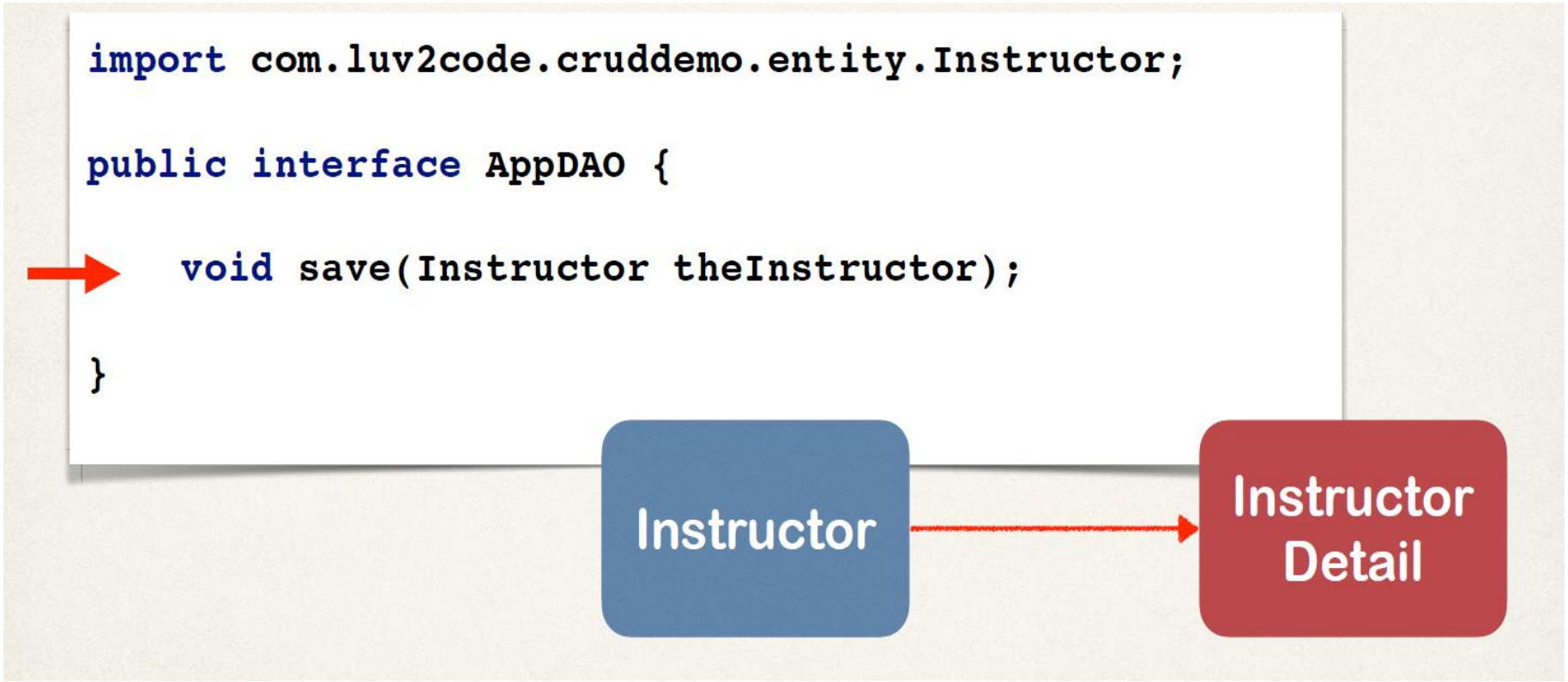
By default, no operations are cascaded.

Configure Multiple Cascade Types

```
@OneToOne(cascade={CascadeType.DETACH,
                   CascadeType.MERGE,
                   CascadeType.PERSIST,
                   CascadeType.REFRESH,
                   CascadeType.REMOVE})
```

Define DAO interface

```
import com.luv2code.cruddemo.entity.Instructor;

public interface AppDAO {

    void save(Instructor theInstructor);

}
```

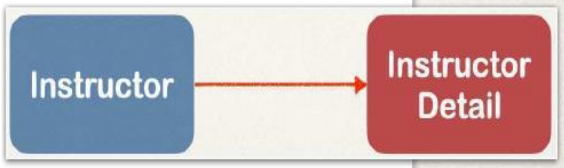


Define DAO implementation

```java
import com.luv2code.cruddemo.entity.Instructor;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@Repository
public class AppDAOImpl implements AppDAO {

    // define field for entity manager
    private EntityManager entityManager;

    // inject entity manager using constructor injection
    @Autowired
    public AppDAOImpl(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    @Override
    @Transactional
    public void save(Instructor theInstructor) {
        entityManager.persist(theInstructor);
    }

}
```

Inject the Entity Manager

Save the Java object

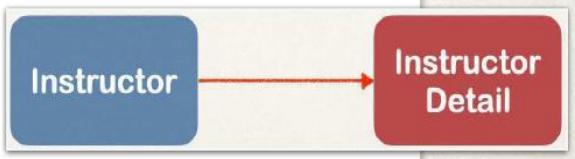


Define DAO implementation

```java
import com.luv2code.cruddemo.entity.Instructor;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@Repository
public class AppDAOImpl implements AppDAO {

    // define field for entity manager
    private EntityManager entityManager;

    // inject entity manager using constructor injection
    @Autowired
    public AppDAOImpl(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    @Override
    @Transactional
    public void save(Instructor theInstructor) {
        entityManager.persist(theInstructor);
    }

}
```

This will ALSO save the details object

Because of CascadeType.ALL



Update main app

```java
@SpringBootApplication
public class MainApplication {

    public static void main(String[] args) {
        SpringApplication.run(MainApplication.class, args);
    }

    @Bean
    public CommandLineRunner commandLineRunner(AppDAO appDAO) {
        return runner -> {

            createInstructor(appDAO);
        };
    }

    ...
```

Inject the AppDAO

```java
    private void createInstructor(AppDAO appDAO) {

        // create the instructor
        Instructor tempInstructor =
                new Instructor("Chad", "Darby", "darby@luv2code.com");

        // create the instructor detail
        InstructorDetail tempInstructorDetail =
                new InstructorDetail(
                        "http://www.luv2code.com/youtube",
                        "Luv 2 code!!!");

        // associate the objects
        tempInstructor.setInstructorDetail(tempInstructorDetail);

        // save the instructor
        System.out.println("Saving instructor: " + tempInstructor);
        appDAO.save(tempInstructor);

        System.out.println("Done!");
    }
}
```

Remember:

This will ALSO save the details object

Because of CascadeType.ALL

In AppDAO, delegated to entityManager.persist(...)

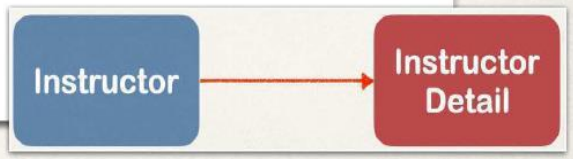
JPA / Hibernate

One-to-One Mapping : Find an entity - Define DAO implementation

```
@Repository
public class AppDAOImpl implements AppDAO {

    ...

    @Override
    public Instructor findInstructorById(int theId) {
        return entityManager.find(Instructor.class, theId);
    }

}
```

This will ALSO retrieve the instructor details object

Because of default behavior of @OneToOne fetch type is eager ... more on fetch types later



JPA / Hibernate

One-to-One Mapping : Delete an entity - Define DAO implementation

```
@Repository
public class AppDAOImpl implements AppDAO {
    ...

   @Override
   @Transactional
   public void deleteInstructorById(int theId) {

       // retrieve the instructor
       Instructor tempInstructor = entityManager.find(Instructor.class, theId);

       // delete the instructor
       entityManager.remove(tempInstructor);
   }

}
```

This will ALSO delete the instructor details object

Because of CascadeType.ALL