

Spring Boot with REST API, MVC and Microservices



Spring Boot

(REST API, MVC and Microservices)

Introduction to Java Spring Boot

- Overview of Java Spring Boot
- Environment Set up
- Setting up a Spring Boot project
- Project structure and configuration
- Introduction to Maven
- Spring Actuator

Spring Boot

(REST API, MVC and Microservices)

Building Restful APIs with Spring Boot

- Introduction to RESTful web services
- Creating a simple RESTful API
- RESTful principles and best practices
- Handling HTTP methods (GET, POST, PUT, DELETE)
- Request and Response formats (JSON, XML)

Spring Boot

(REST API, MVC and Microservices)

Spring Boot and Spring MVC

- Understanding Model-View-Controller (MVC) pattern
- Creating controllers and handling requests
- Request mapping and parameter handling
- Exception handling in Spring MVC
- Thymeleaf Template engine
- Using Lombok

Spring Boot

(REST API, MVC and Microservices)

Spring Boot Data Access with JPA

- Overview of Java Persistence API (JPA)
- Integrating JPA with Spring Boot
- Entity classes and relationships
- Repository pattern and CRUD operations

Spring Boot

(REST API, MVC and Microservices)

Spring Boot and Security

- Authentication and Authorization
- Introduction to security in Spring Boot
- Implementing authentication with Spring Security
- Authorization and role-based access control

Microservices with Spring Boot

- Introduction to microservices
- Building microservices with Spring Boot

Thank You!



Spring Boot – Environment Set Up



Spring Boot Development Environment

- Assume –you already have some experience with Java
 - OOP, classes, interfaces, inheritance, exception handling, collections
- Ensure the software already installed
 - Java Development Kit (JDK) - *Spring Boot 3 requires JDK 17 or higher*
 - Java IDE (any Java IDE will work)

Eclipse, IntelliJ IDEA, NetBeans, VS Code, etc

Spring Boot Development Environment

Download and Install

- Java 17 or later
- IDE – Eclipse
- STS Plugin in Eclipse
- MySQL
- Lombok - optional

Make sure you can run a basic HelloWorld Java app in your Java IDE

Spring Boot (REST API, MVC and Microservices)

Introduction to Spring

Spring Website - Official

The screenshot shows the official Spring website at spring.io. The page features a large green and yellow graphic on the left. In the center, the text "Spring makes Java productive." is displayed above two buttons: "WHY SPRING" and "QUICKSTART". At the bottom left, there's a "NEWS" button and a link to the "Spring Health Assessment Report". The top navigation bar includes links for "Why Spring", "Learn", "Projects", "Academy", "Solutions", "Community", and a gear icon. A dropdown menu for "Projects" is open, listing various Spring projects: Overview, Spring Boot, Spring Framework, Spring Cloud, Spring Cloud Data Flow, Spring Data, Spring Integration, Spring Batch, Spring Security, a link to "View all projects", a section for "DEVELOPMENT TOOLS" (Spring Tools 4), and links for "Spring Initializr" and "Spring Initializr" (with a small icon).



Why Spring?

Simplify Java Enterprise Development

Very popular framework for building Java applications

Provides a large number of helper classes and annotations

Goals of Spring

- Lightweight development with Java POJOs (Plaint-Old-Java-Objects)
- Dependency injection to promote loose coupling
- Minimize boilerplate Java code



Spring Framework Runtime

Data Access/Integration

JDBC

ORM

OXM

JMS

Transactions

AOP

Aspects

Beans

Core

Web

WebSocket

Servlet

Web

Portlet

Instrumentation

Messaging

Core Container

SpEL

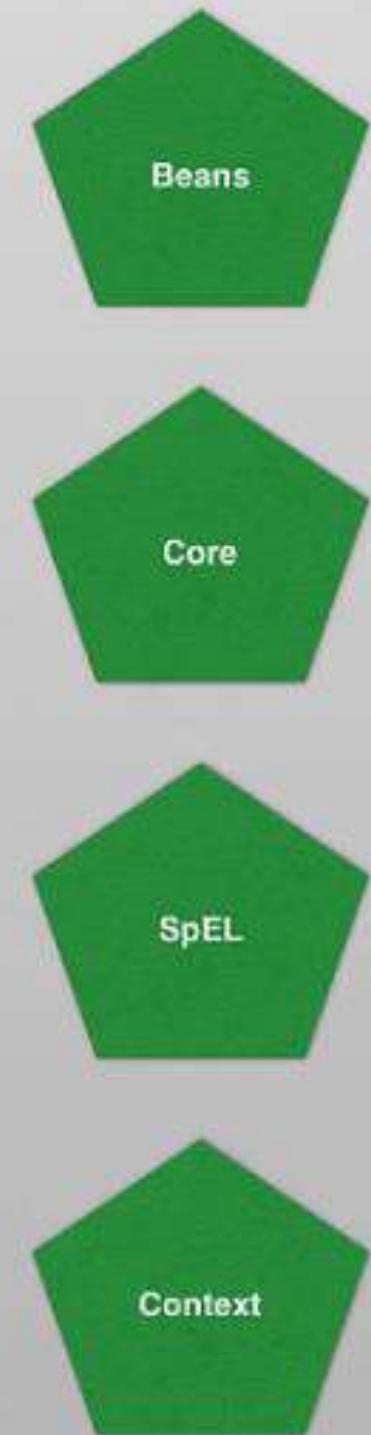
Context

Test

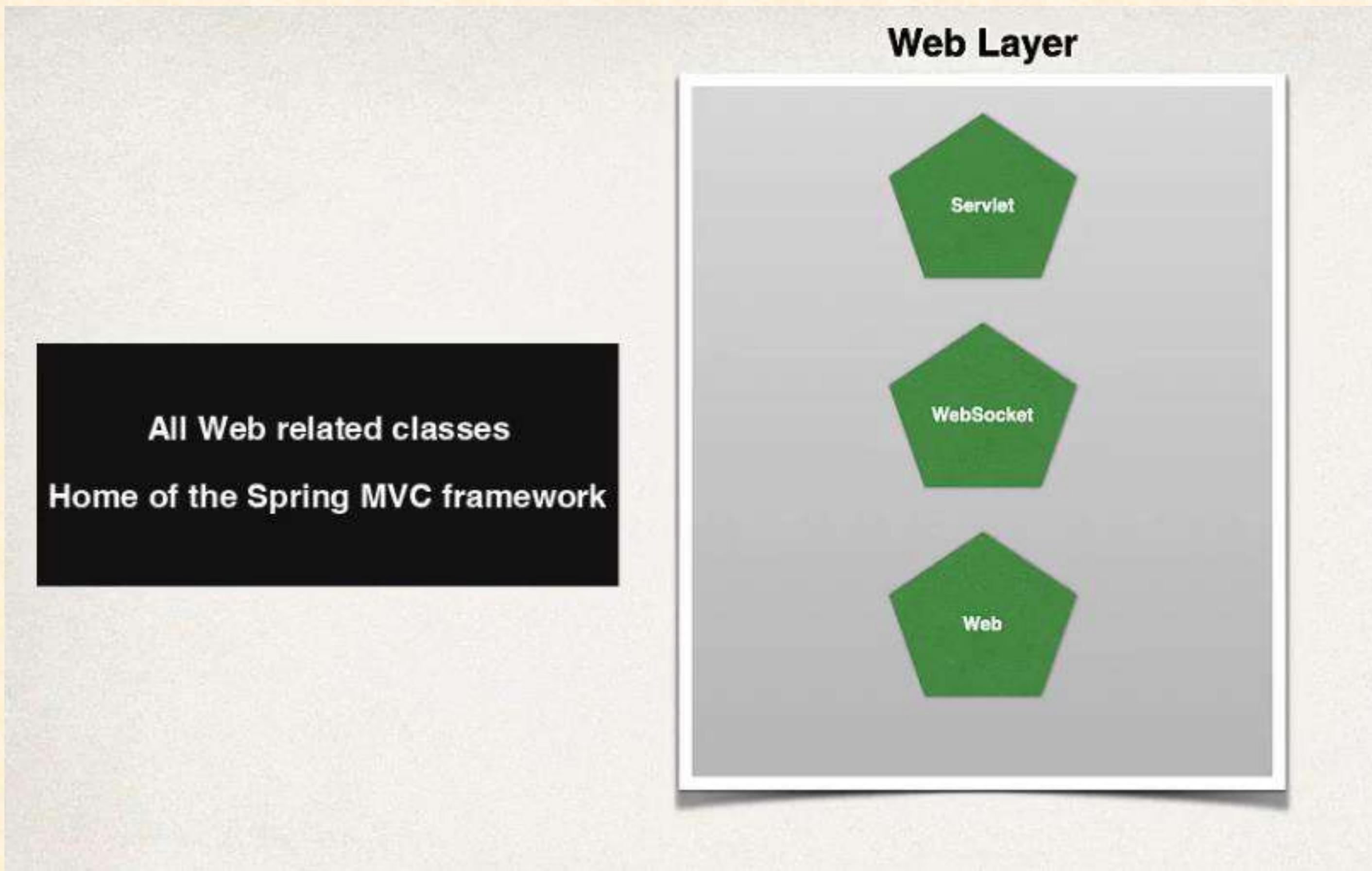
Core Container

Factory for creating beans
Manage bean dependencies

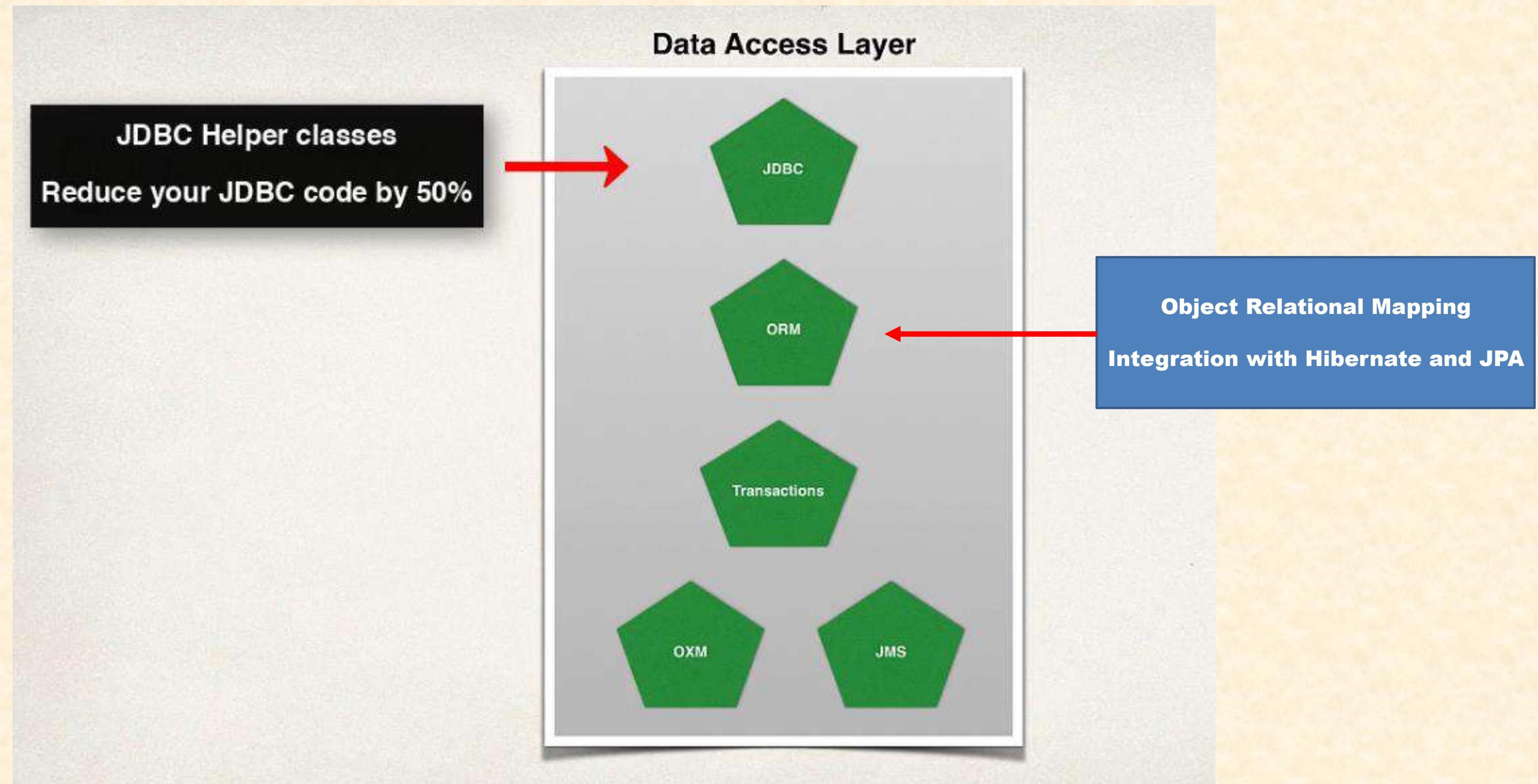
Core Container



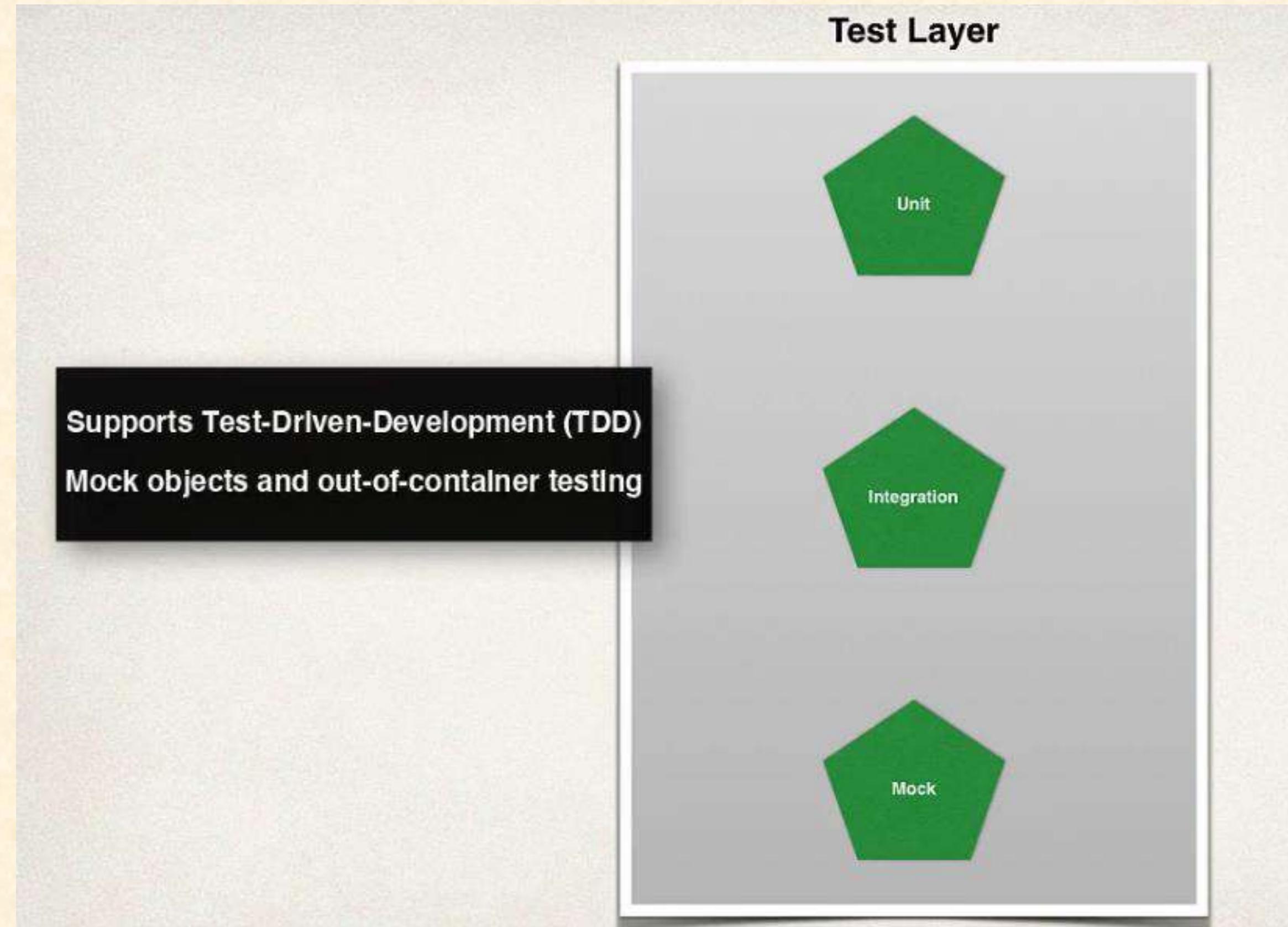
Web Layer



Data Access Layer



Test Layer



Spring “Projects”

- Additional Spring *modules* built-on top of the core Spring Framework
- Only use what you need ...
 - Spring Cloud, Spring Data
 - Spring Batch, Spring Security
 - Spring Web Services
 - *others* ...

The Problem

- Building a traditional Spring application is really HARD!!!

Q: Which JAR dependencies do I need?

And that's
JUST the basics
for getting started

Q: How do I set up configuration (xml or Java)?

Q: How do I install the server? (Tomcat, JBoss etc...)

Spring Boot Solution

- Makes it easier to get started with Spring app development
- Minimize the amount of manual configuration
 - Perform auto-configuration based on props files and JAR classpath
- Help to resolve dependency conflicts (Maven or Gradle)
- Provide an embedded HTTP server so you can get started quickly
 - Tomcat, Jetty, Undertow, ...

[OVERVIEW](#)[LEARN](#)[SUPPORT](#)[SAMPLES](#)

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need minimal Spring configuration.

If you're looking for information about a specific version, or instructions about how to upgrade from an earlier release, check out [the project release notes section](#) on our wiki.

Features

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' dependencies to simplify your build configuration
- Automatically configure Spring and 3rd party libraries whenever possible
- Provide production-ready features such as metrics, health checks, and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

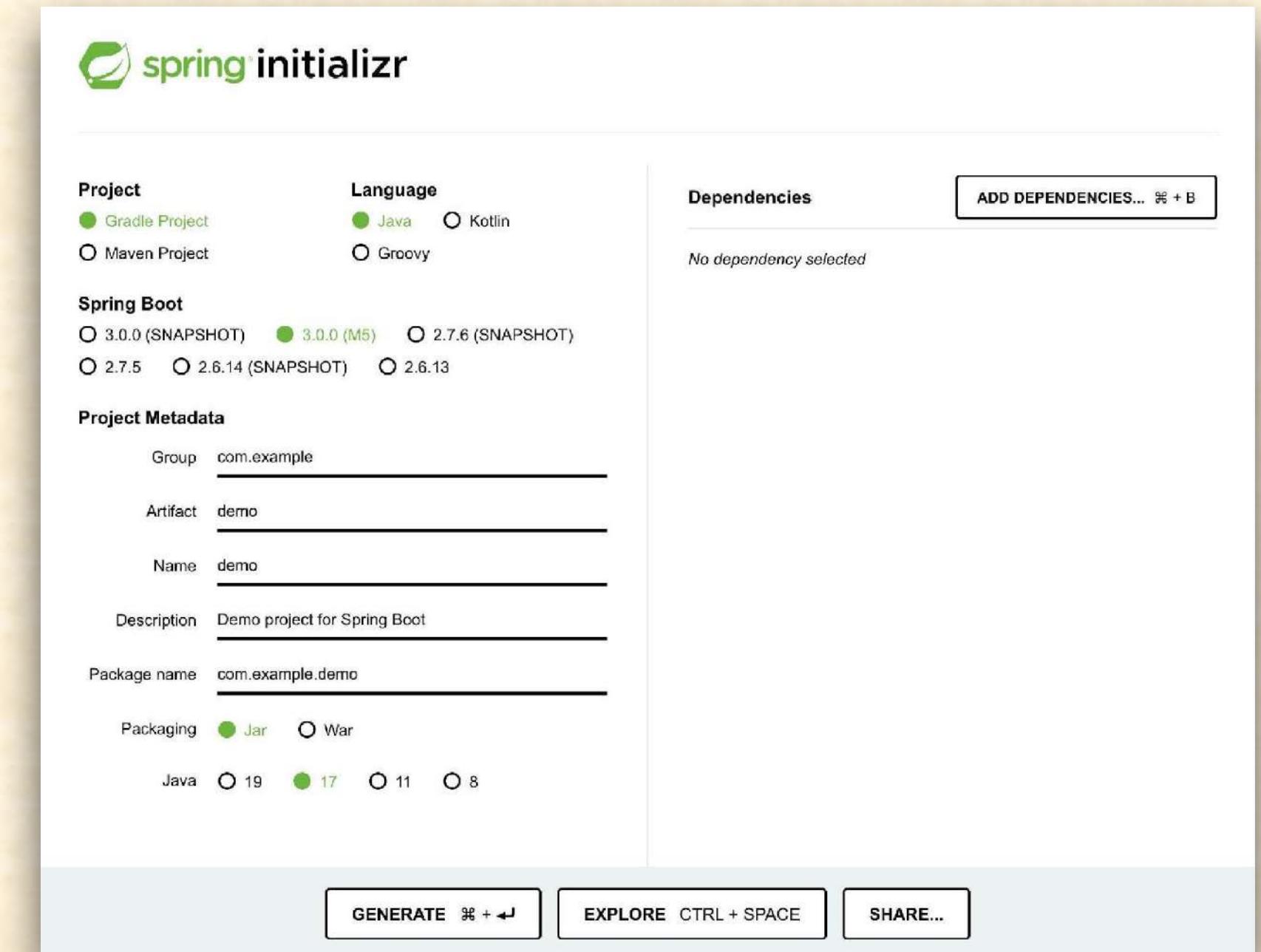
Spring Boot and Spring

- Spring Boot uses Spring behind the scenes
- Spring Boot simply makes it easier to use Spring

Spring Initializr

- Quickly create a starter Spring Boot project
- Select your dependencies
- Creates a Maven/Gradle project
- Import the project into your IDE
 - Eclipse, IntelliJ, NetBeans etc ...

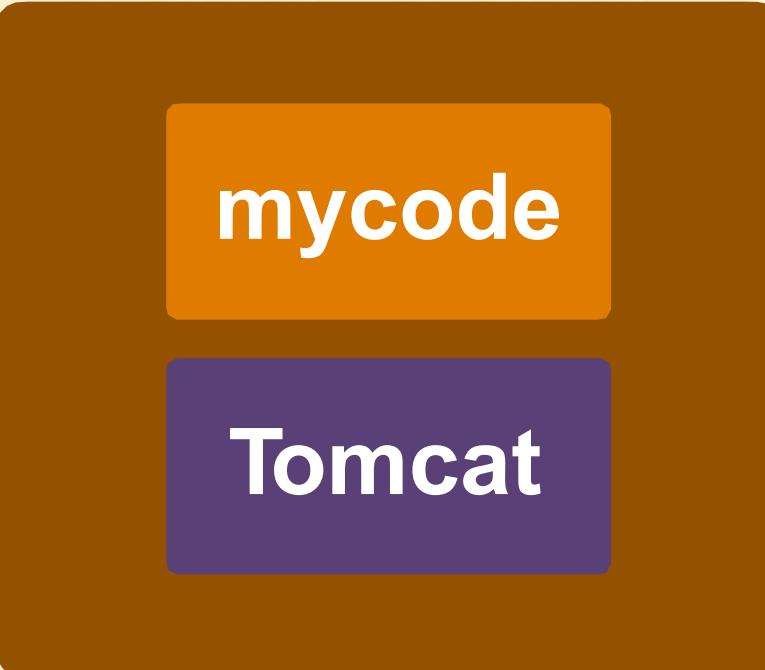
<http://start.spring.io>



Spring Boot Embedded Server

- Provide an embedded HTTP server so you can get started quickly
 - Tomcat, Jetty, Undertow, ...
 - No need to install a server separately

myapp.jar



JAR file
includes your application code
AND
includes the server

Self-contained unit
Nothing else to install

Running Spring Boot Apps

- Spring Boot apps can be run standalone (includes embedded tomcat server)
- Run the Spring Boot app from the IDE or command-line

myapp.jar

mycode

Tomcat

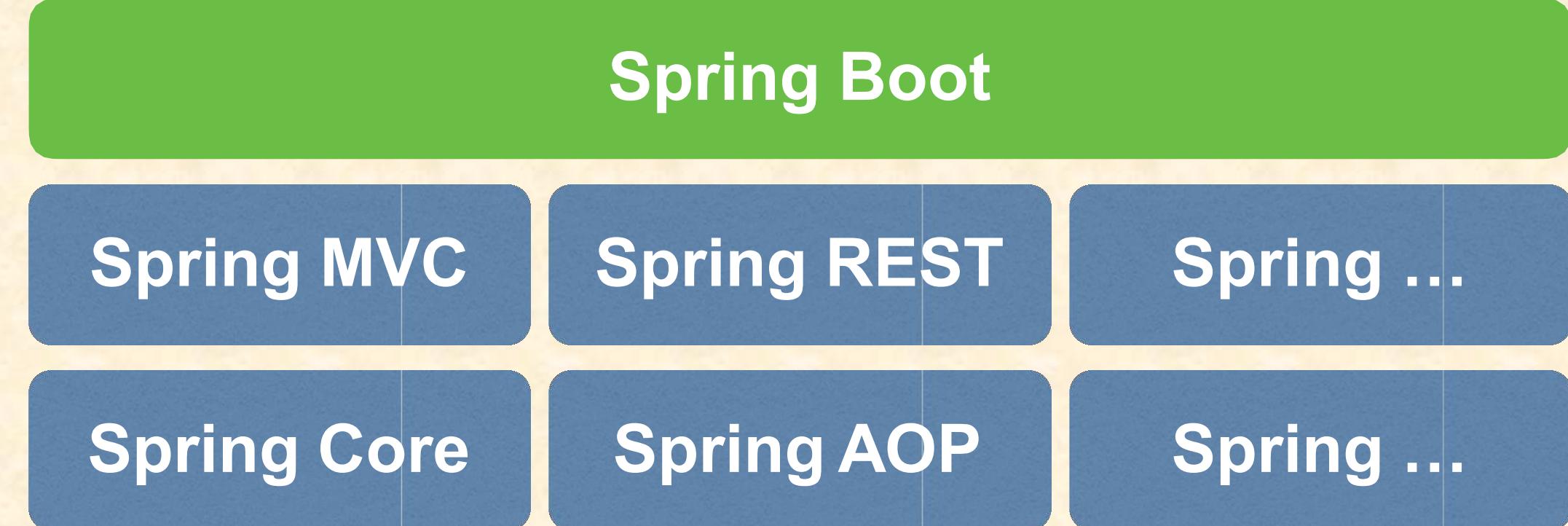
```
> java -jar myapp.jar
```

Name of our JAR file

Spring Boot Misconceptions

Q: Does Spring Boot replace Spring MVC, Spring REST etc ...?

- No. Instead, Spring Boot actually uses those technologies



Spring Boot Misconceptions

Q: Does Spring Boot run code faster than regular Spring code?

- No.
- Behind the scenes, Spring Boot uses same code of Spring Framework
- Spring Boot is about making it easier to get started
- Minimizing configuration etc ...

Spring Boot Misconceptions

Q: Do I need a special IDE for Spring Boot?

- No.
- You can use any IDE for Spring Boot apps ... even use plain text editor
- The Spring team provides free *Spring Tool Suite (STS)* [IDE plugins]
- Not a requirement. Use the IDE that works best for you

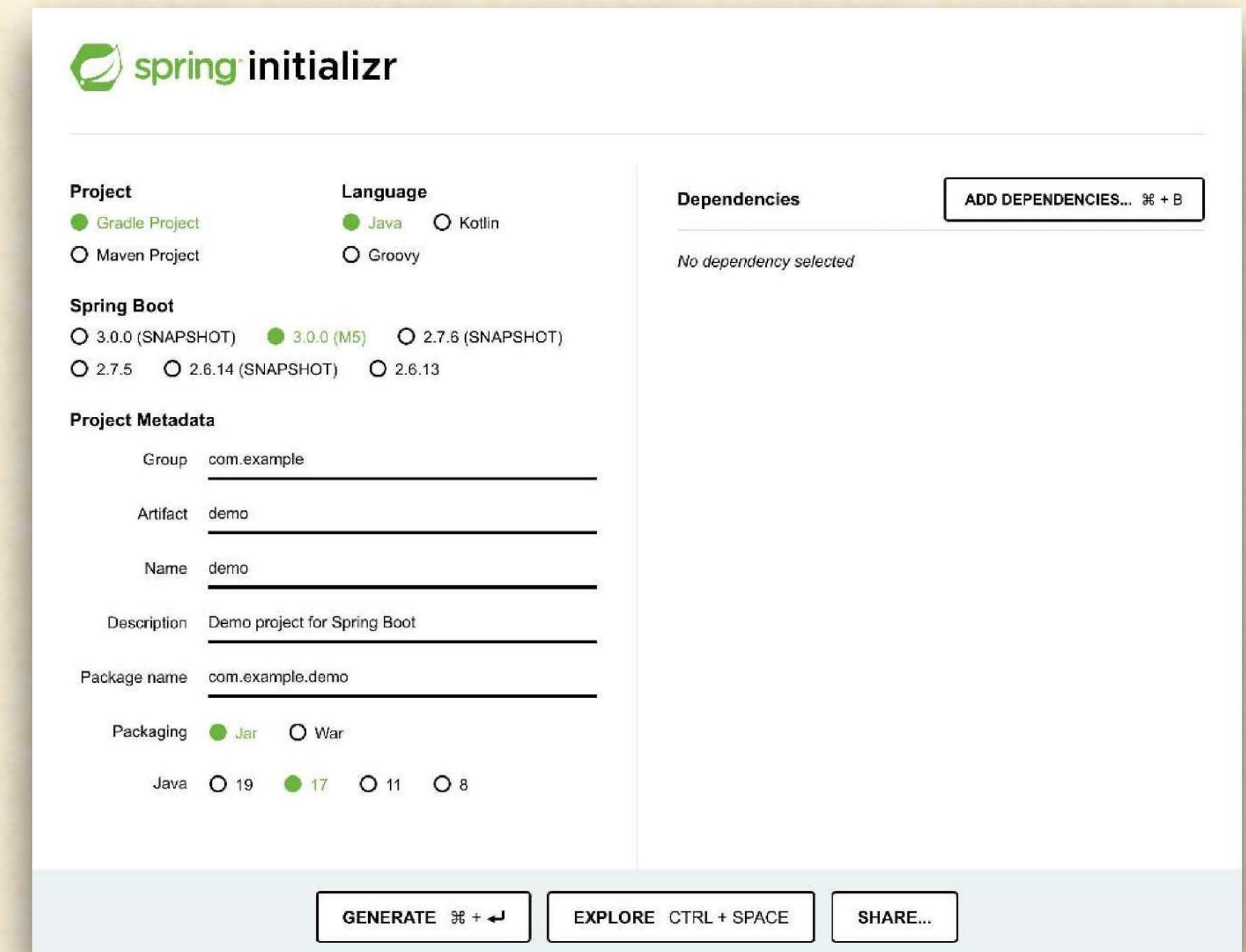
Spring Boot (REST API, MVC and Microservices)

Spring Initializr

Spring Initializr

- Quickly create a starter Spring project
- Select your dependencies
- Creates a Maven/Gradle project
- Import the project into your IDE
 - Eclipse, IntelliJ, NetBeans etc ...

<http://start.spring.io>



Maven – Brief Intro

- When building Java project, we may need additional JAR files
 - For example: Spring, Hibernate, Commons Logging, JSON etc...
- One approach is to download the JAR files from each project web site
- Manually add the JAR files to your build path / classpath

Maven Solution

- Tell Maven the projects we are working with (dependencies)
 - Spring, Hibernate etc
- Maven will go out and download the JAR files for those projects for us
- And Maven will make those JAR files available during compile/run
- Think of Maven as our friendly helper

Development Process -1

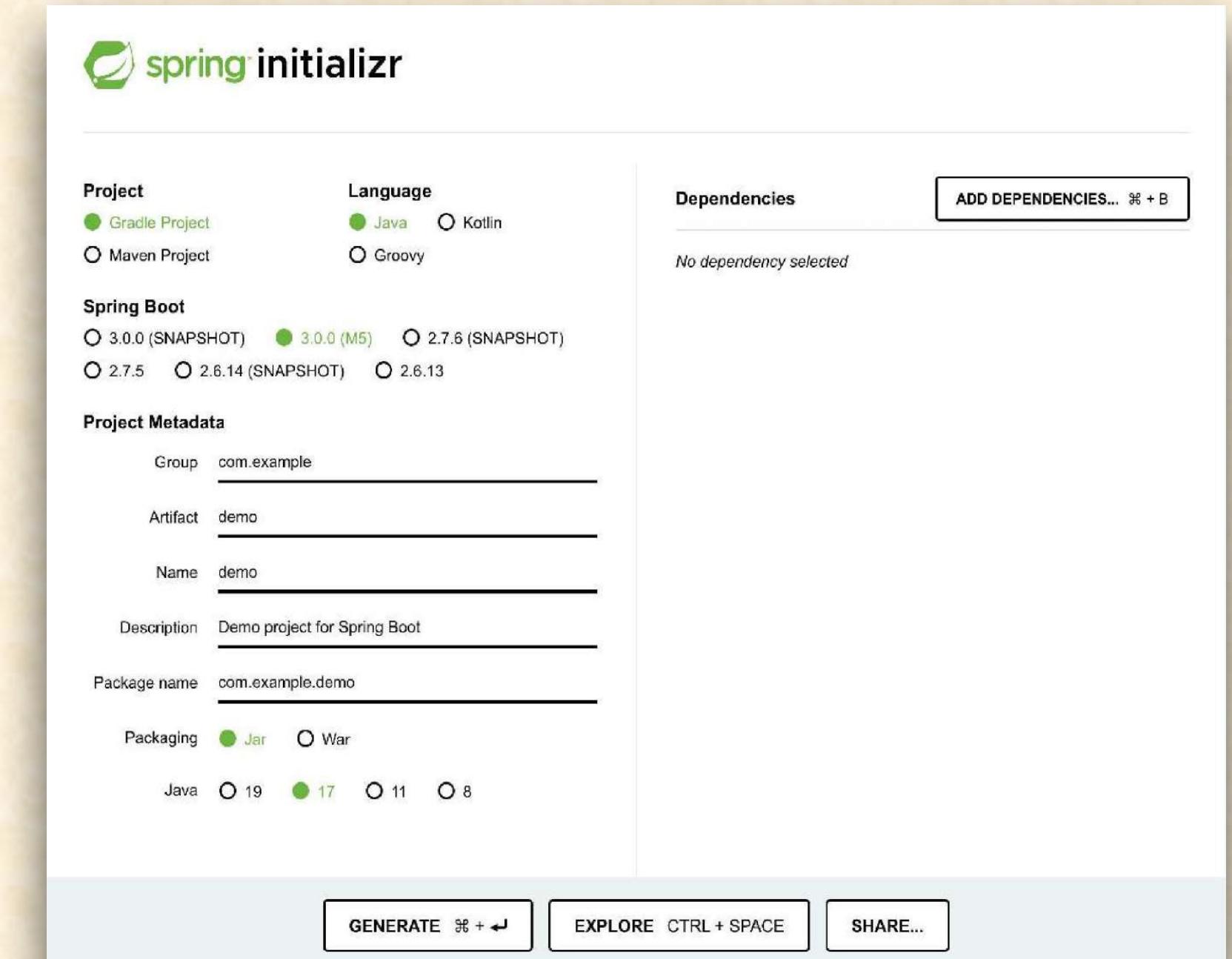
1. Configure our project at Spring Initializr website

<http://start.spring.io>

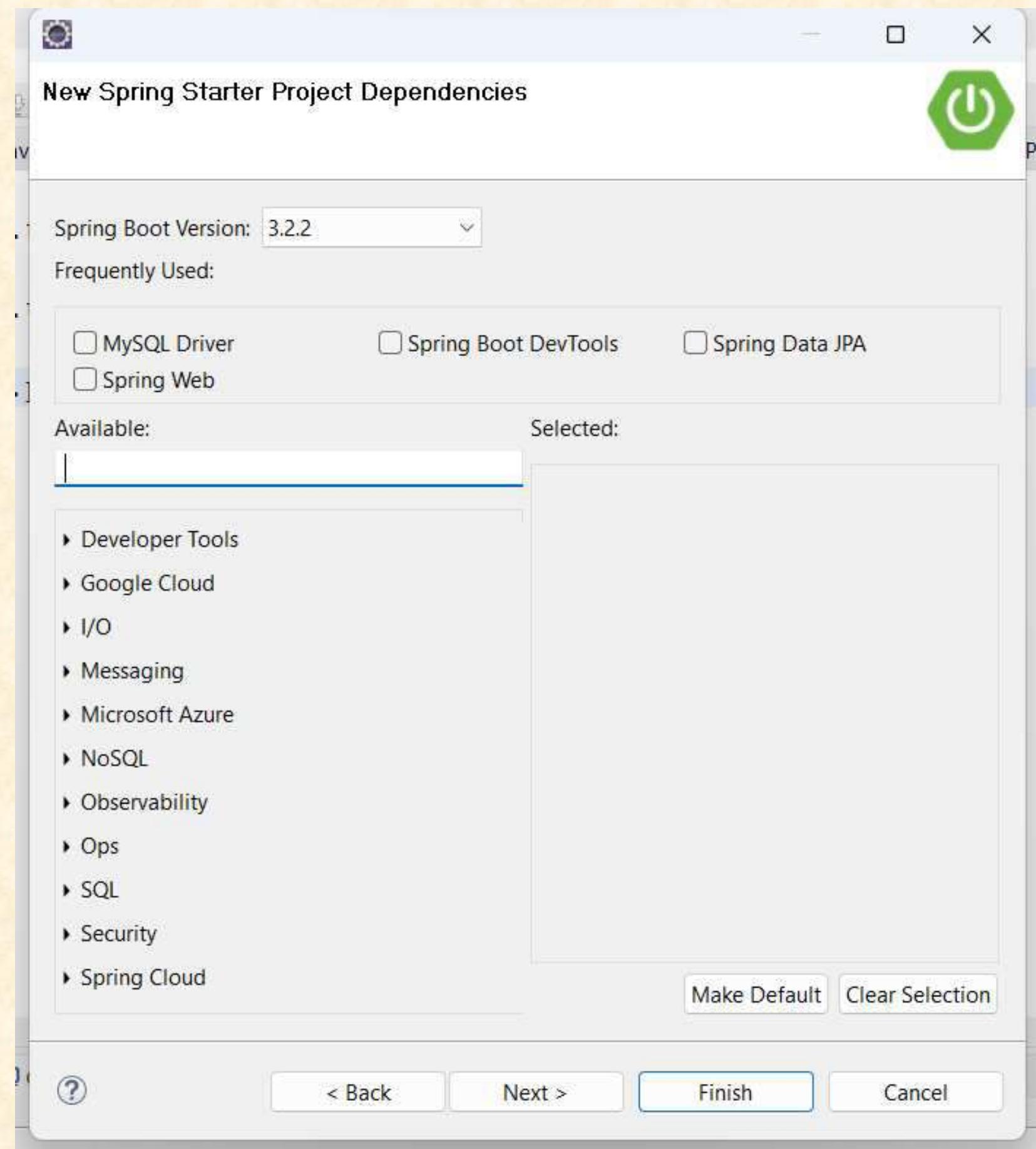
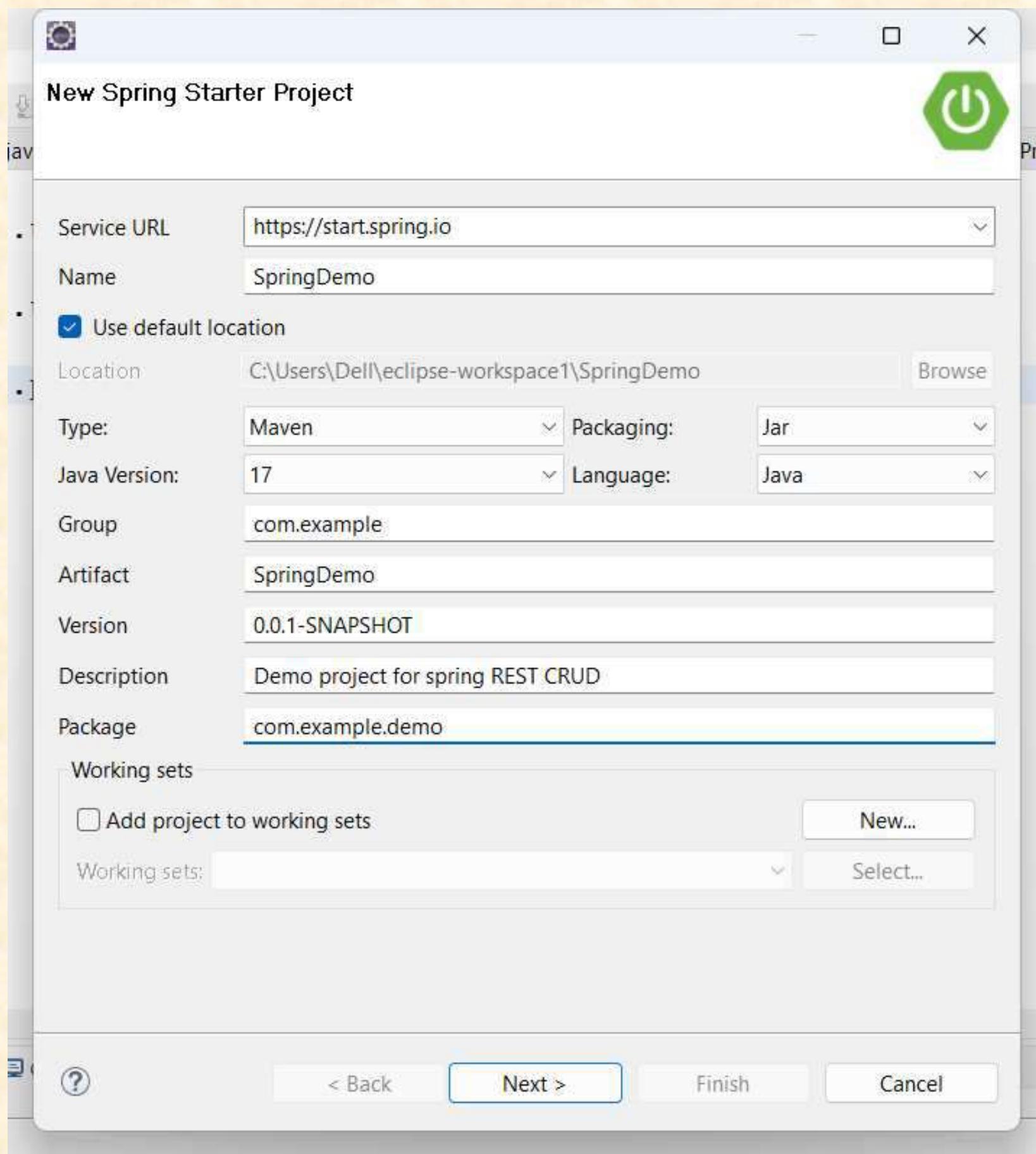
2. Download the zip file

3. Unzip the file

4. Import the project into our IDE



Development Process -2



SpringBootApplication

- *@SpringBootApplication* is composed of the following:

Annotation	Description
<code>@EnableAutoConfiguration</code>	Enables Spring Boot's auto-configuration support
<code>@ComponentScan</code>	Enables component scanning of current package Also recursively scans sub-packages
<code>@Configuration</code>	Able to register extra beans with @Bean or import other configuration classes

Spring Boot (REST API, MVC and Microservices)

Spring REST Controller

REST Controller

- Let's create a very simple REST Controller



Create REST Controller

Set up rest controller

```
@RestController  
public class MyRestController {  
  
    //expose "/" root that returns "Hello World"  
  
    @GetMapping("/")  
    public String sayHello() {  
        return "Hello World!";  
    }  
}
```

Handle HTTP GET requests

Spring Boot (REST API, MVC and Microservices)

Maven – Project Management Tool

Spring Boot and Maven

- When we generate projects using Spring Initializr: start.spring.io
 - It generates a Maven project for us
- In this section, we will learn the basics of Maven
 - Viewing dependencies in the Maven pom.xml file
 - Spring Boot Starters for Maven

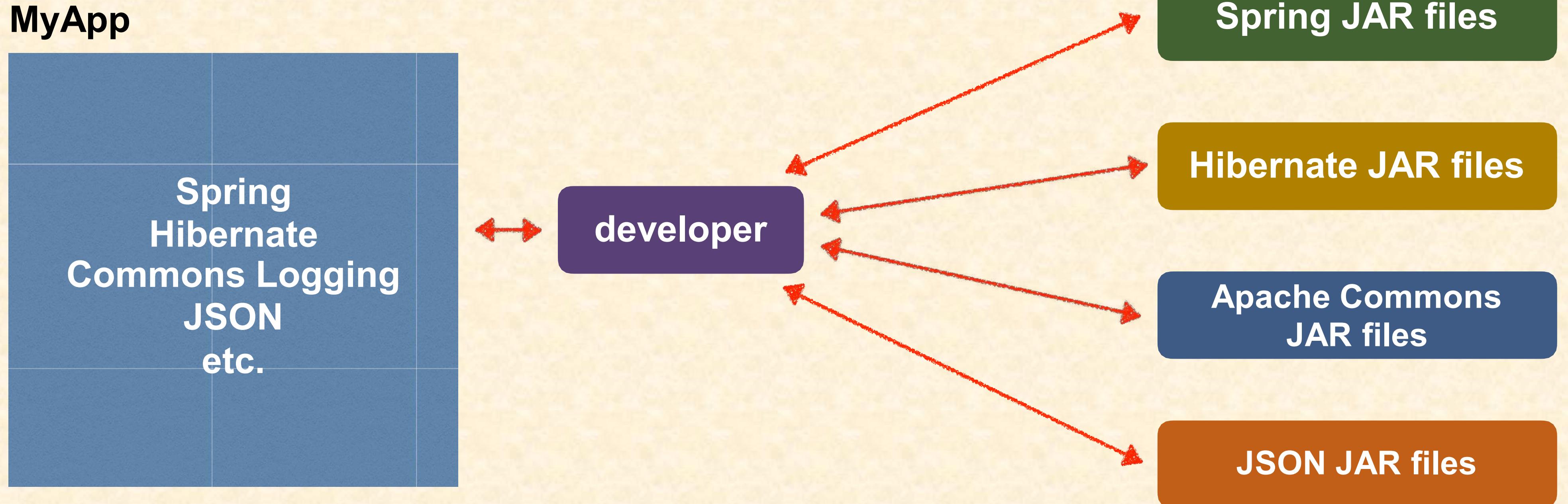
What is Maven?

- Maven is a Project Management tool
- Most popular use of Maven is for build management and dependencies

What Problems Does Maven Solve?

- When building our Java project, we may need additional JAR files
 - For example: Spring, Hibernate, Commons Logging, JSON etc...
- One approach is to download the JAR files from each project web site
- Manually add the JAR files to the build path / classpath

Our Project without Maven

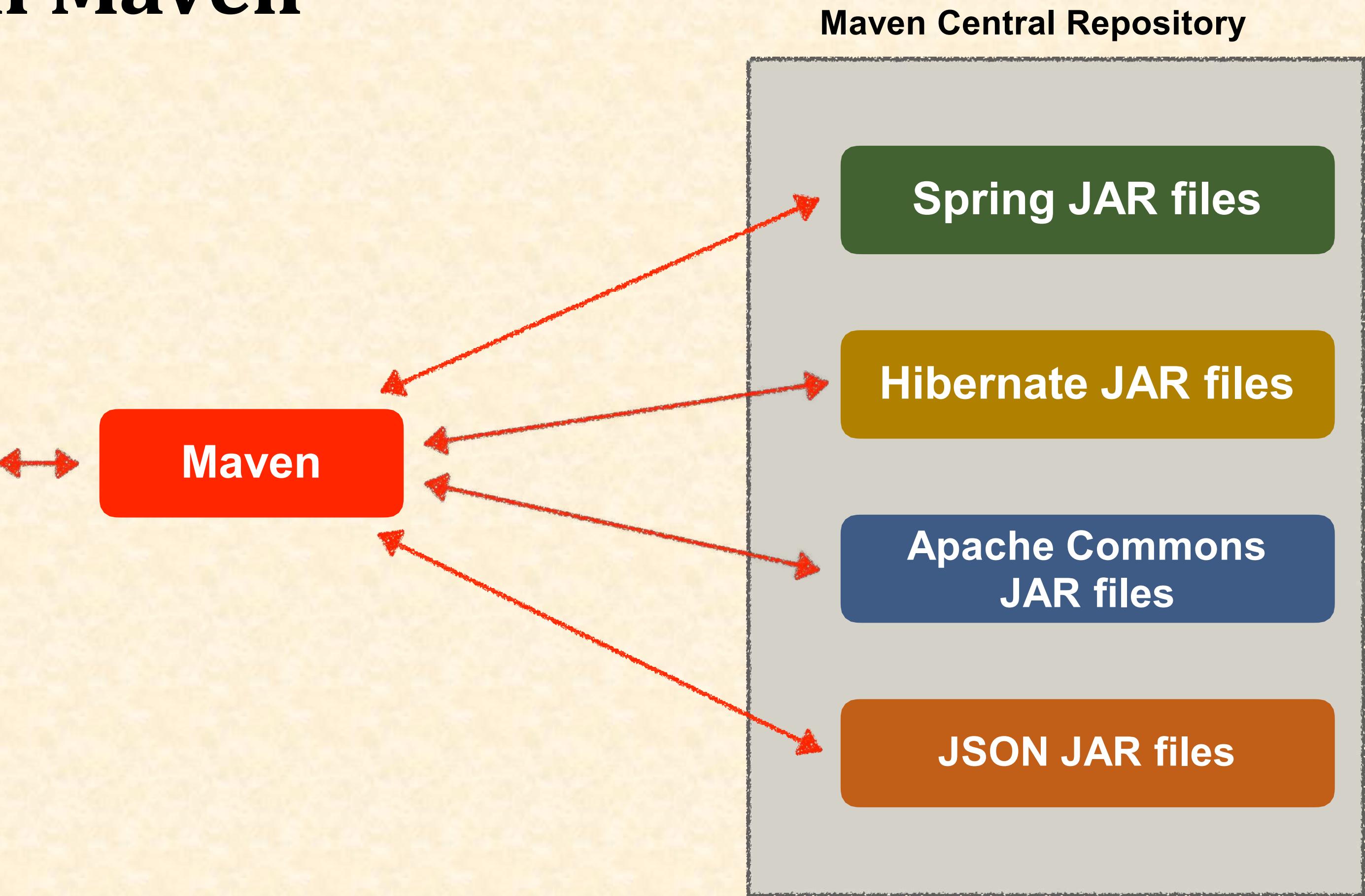
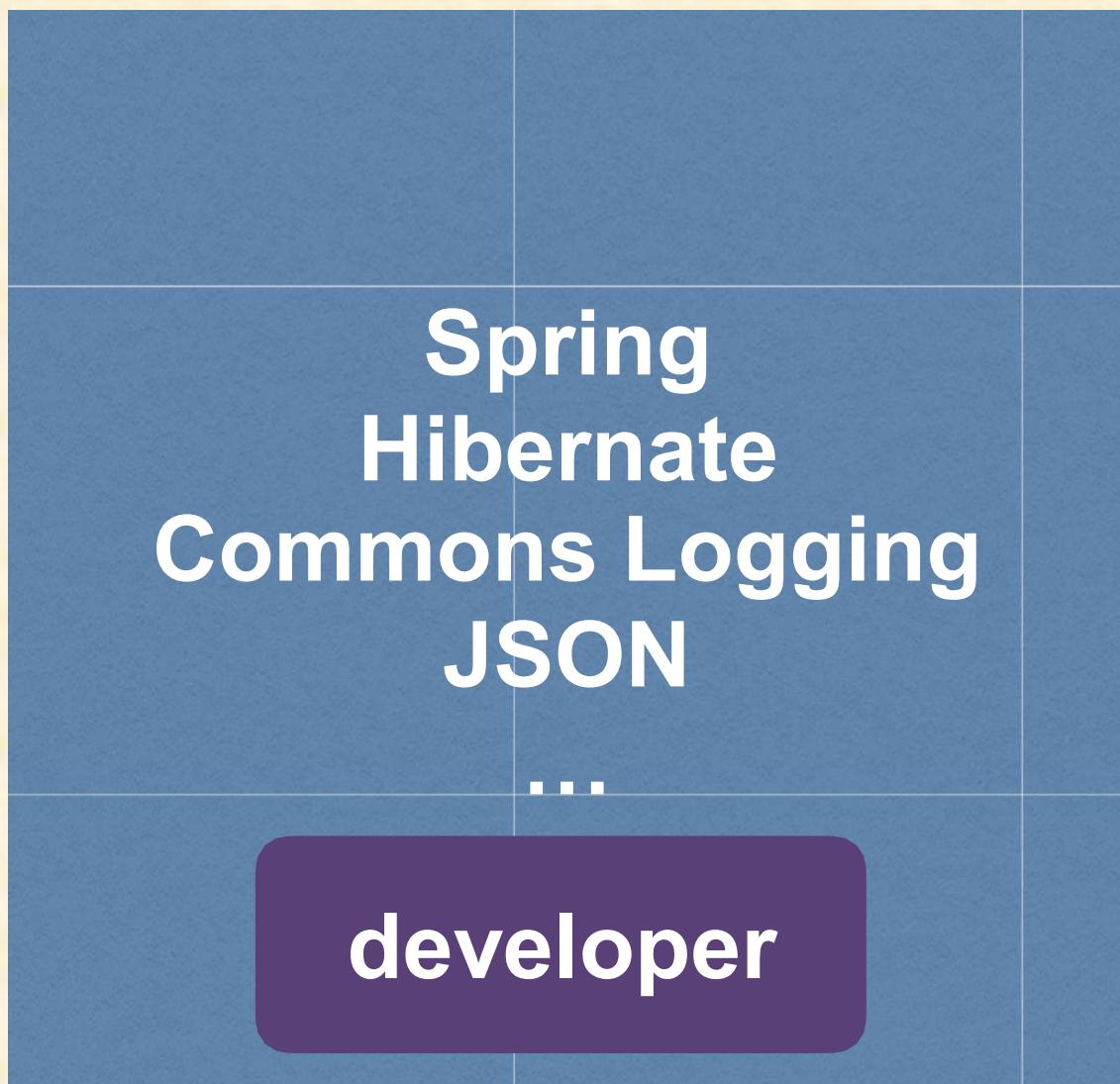


Maven Solution

- Tell Maven the projects we are working with (dependencies)
 - Spring, Hibernate etc
- Maven will go out and download the JAR files for those projects for us
- And Maven will make those JAR files available during compile/run

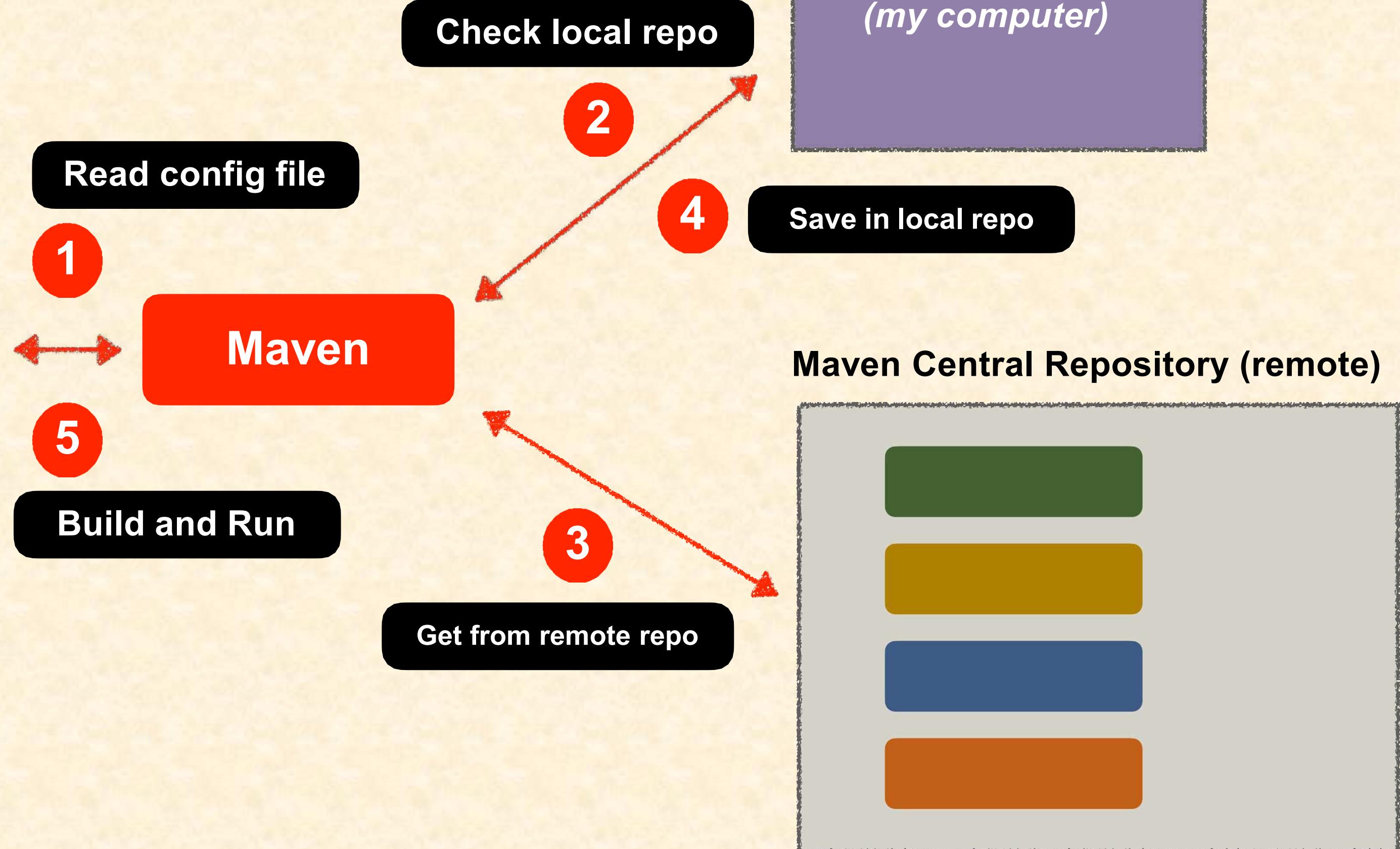
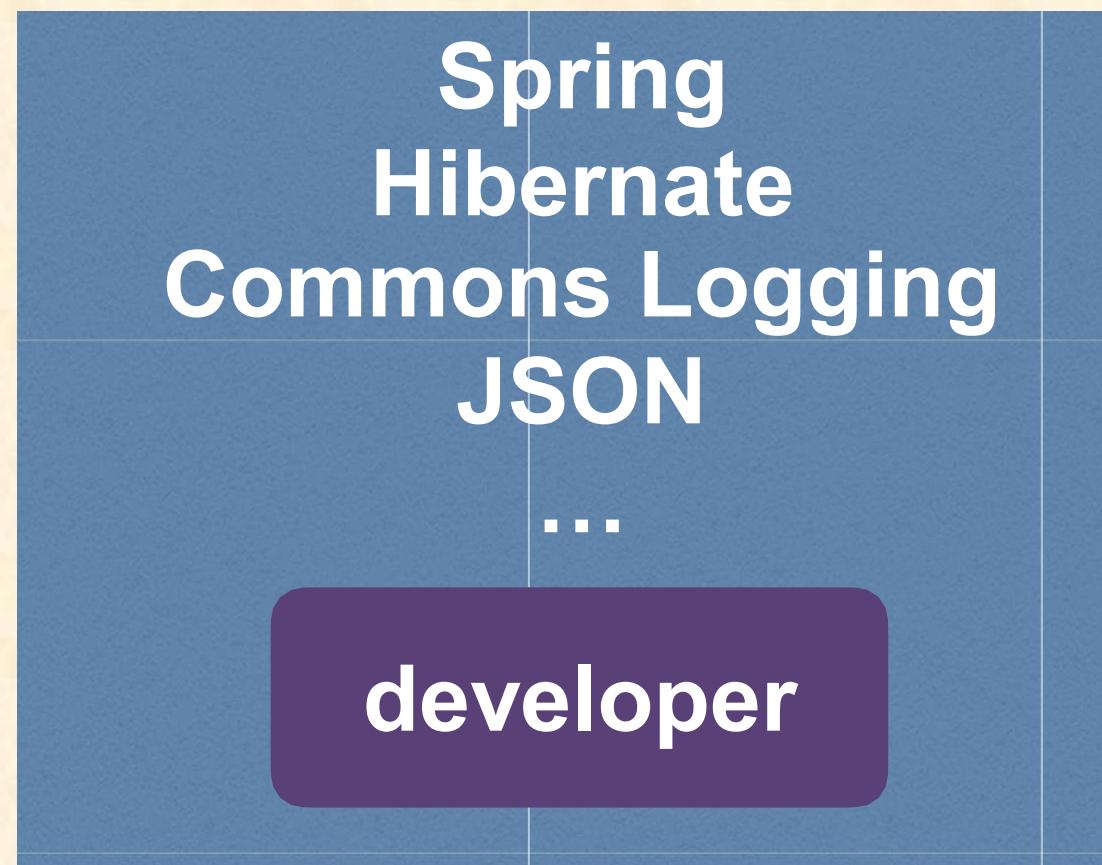
My Project with Maven

MyApp



Maven - How It Works?

Project Config file



Handling JAR Dependencies

- When Maven retrieves a project dependency
 - It will also download supporting dependencies
 - For example: Spring depends on commons-logging ...
- Maven will handle this for us automagically

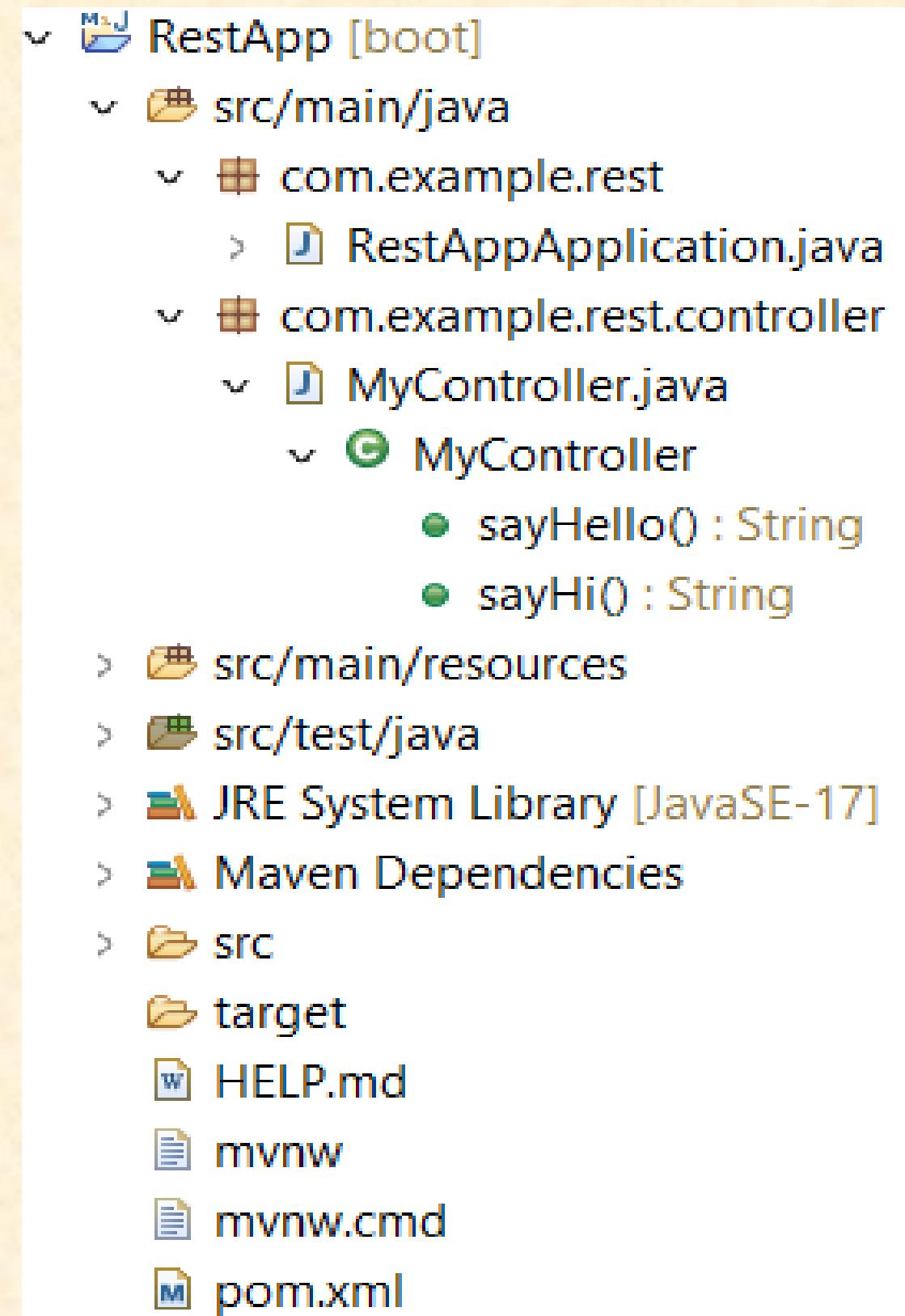
Building and Running

- When we build and run our app ...
- Maven will handle class / build path for us
- Based on config file, Maven will add JAR files accordingly

Standard Directory Structure

- Normally when you join a new project
 - Each development team dreams up their own directory structure
 - Not ideal for new comers and not standardized
- Maven solves this problem by providing a standard directory structure

Standard Directory Structure



Place your Java source code here

Standard Directory Structure Benefits

- Most major IDEs have built-in support for Maven
 - Eclipse, IntelliJ, NetBeans etc
 - IDEs can easily read/import Maven projects
- Maven projects are portable
 - Developers can easily share projects between IDEs

Advantages of Maven

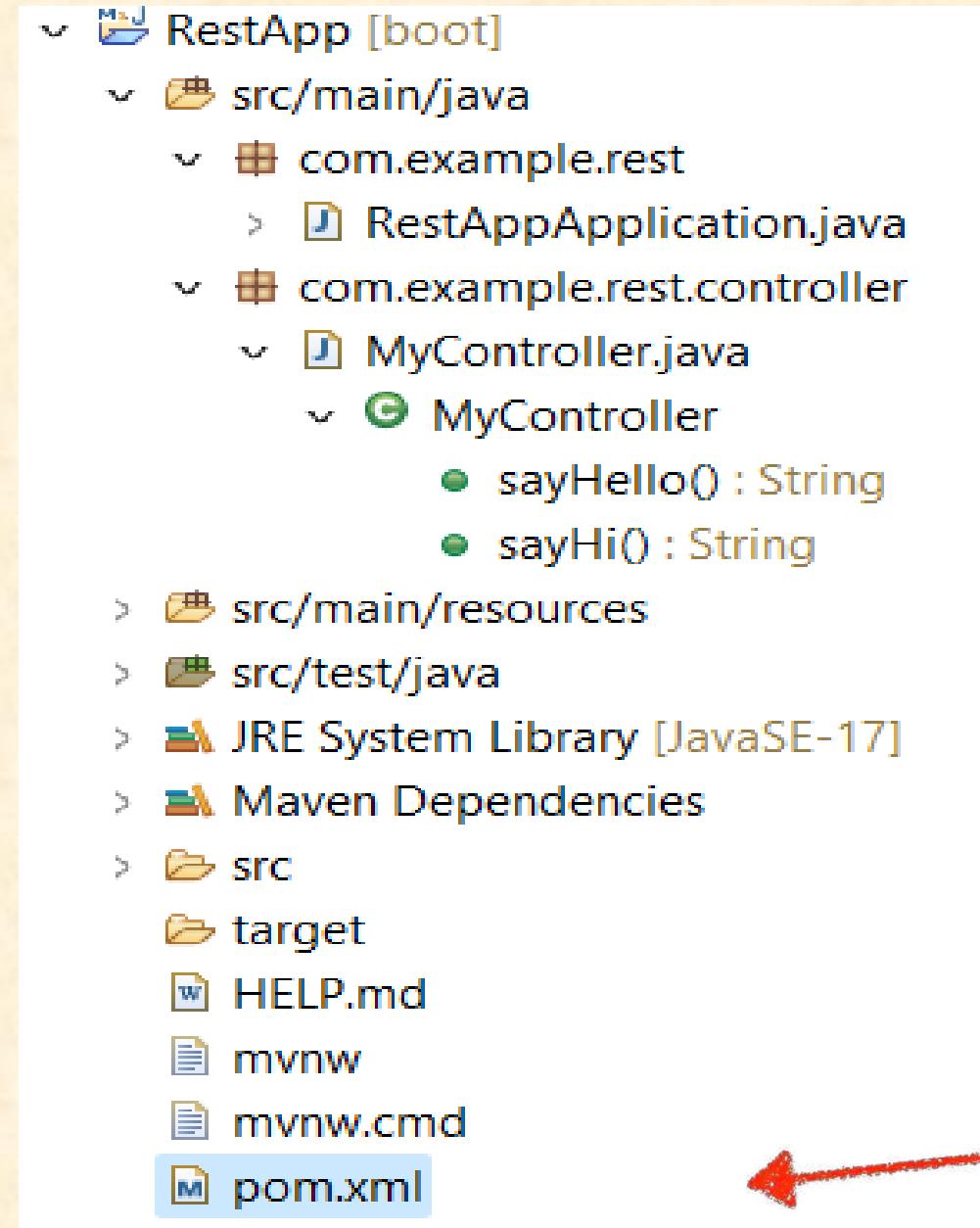
- Dependency Management
 - Maven will find JAR files for you
 - No more missing JARs
- Building and Running the Project
 - No more build path / classpath issues
- Standard directory structure

Maven Key Concepts

Maven Key Concepts

- POM File - pom.xml
- Project Coordinates

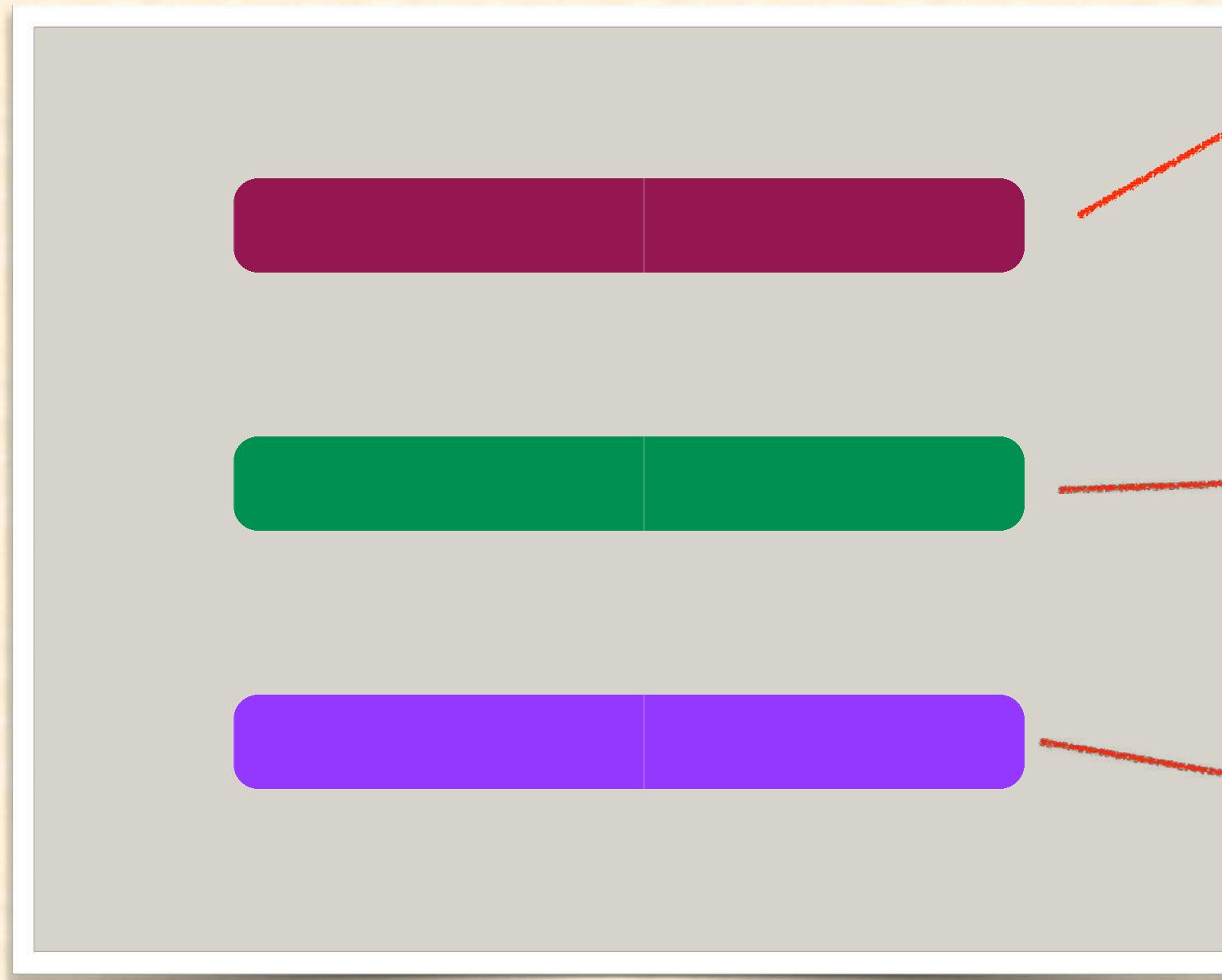
POM File - pom.xml



- Project Object Model file: POM file
- Configuration file for your project
- Basically your “shopping list” for Maven :-)
- Located in the root of your Maven project

POM File Structure

pom.xml



**Project name, version etc
Output file type: JAR, WAR, ...**

**List of projects we depend on
Spring, Hibernate, etc...**

**Additional custom tasks to run:
generate JUnit test reports etc...**

Simple POM File

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http  
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://  
<modelVersion>4.0.0</modelVersion>  
<parent>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-parent</artifactId>  
    <version>3.2.2</version>  
    <relativePath/> <!-- lookup parent from repository -->  
</parent>  
<groupId>com.example</groupId>  
<artifactId>RestApp</artifactId>  
<version>0.0.1-SNAPSHOT</version>  
<name>RestApp</name>  
<description>Demo project for Spring Boot</description>  
<properties>  
    <java.version>17</java.version>  
</properties>  
<dependencies>  
    <dependency>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-web</artifactId>  
    </dependency>  
  
    <dependency>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-test</artifactId>  
        <scope>test</scope>  
    </dependency>  
    </dependencies>  
<build>  
    <plugins>  
        <plugin>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-maven-plugin</artifactId>  
        </plugin>  
    </plugins>  
</build>  
</project>
```

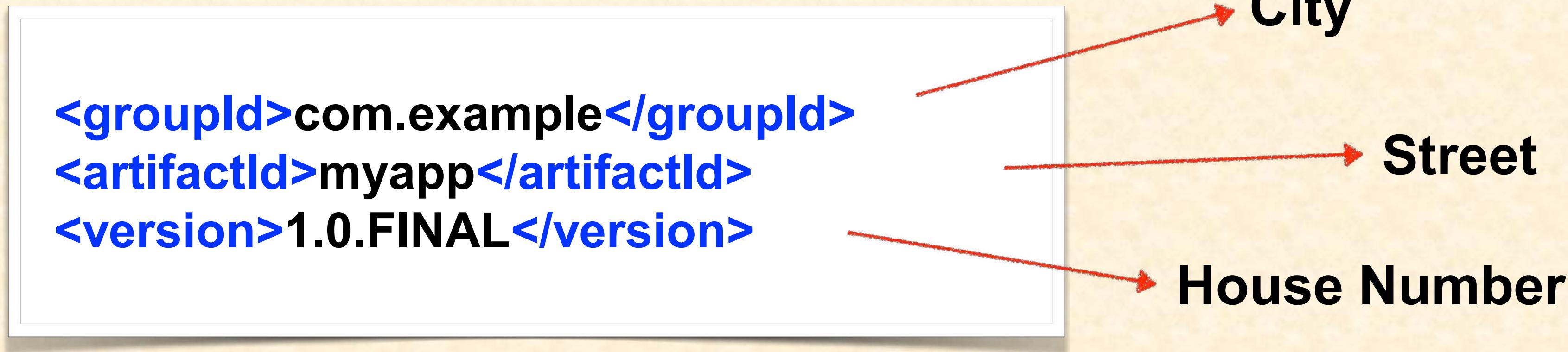
Project name, version etc
Output file type: JAR, WAR, ...

List of projects we depend on
Spring, Hibernate, etc...

Additional custom tasks to run:
generate JUnit test reports etc...

Project Coordinates

- Project Coordinates uniquely identify a project
 - Similar to GPS coordinates for your house: latitude / longitude
 - Precise information for finding your house (city, street, house #)



Example of Project Coordinates

```
<groupId>com. example</groupId>
<artifactId>myapp</artifactId>
<version>1.0.RELEASE</version>
```

```
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>6.0.0</version>
```

```
<groupId>org.hibernate.orm</groupId>
<artifactId>hibernate-core</artifactId>
<version>6.1.4.Final</version>
```

Adding Dependencies

```
<project ...>
...
<dependencies>

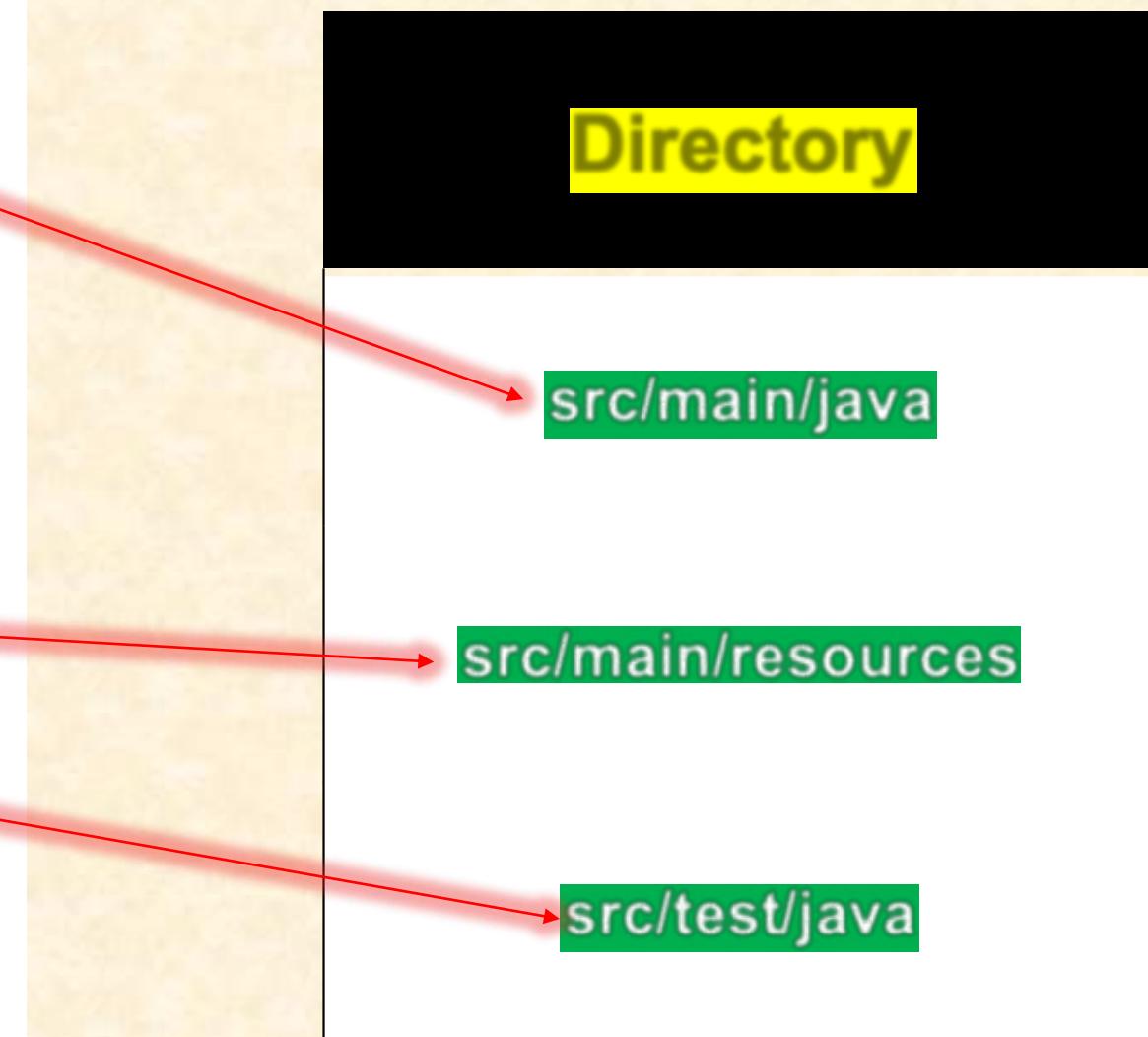
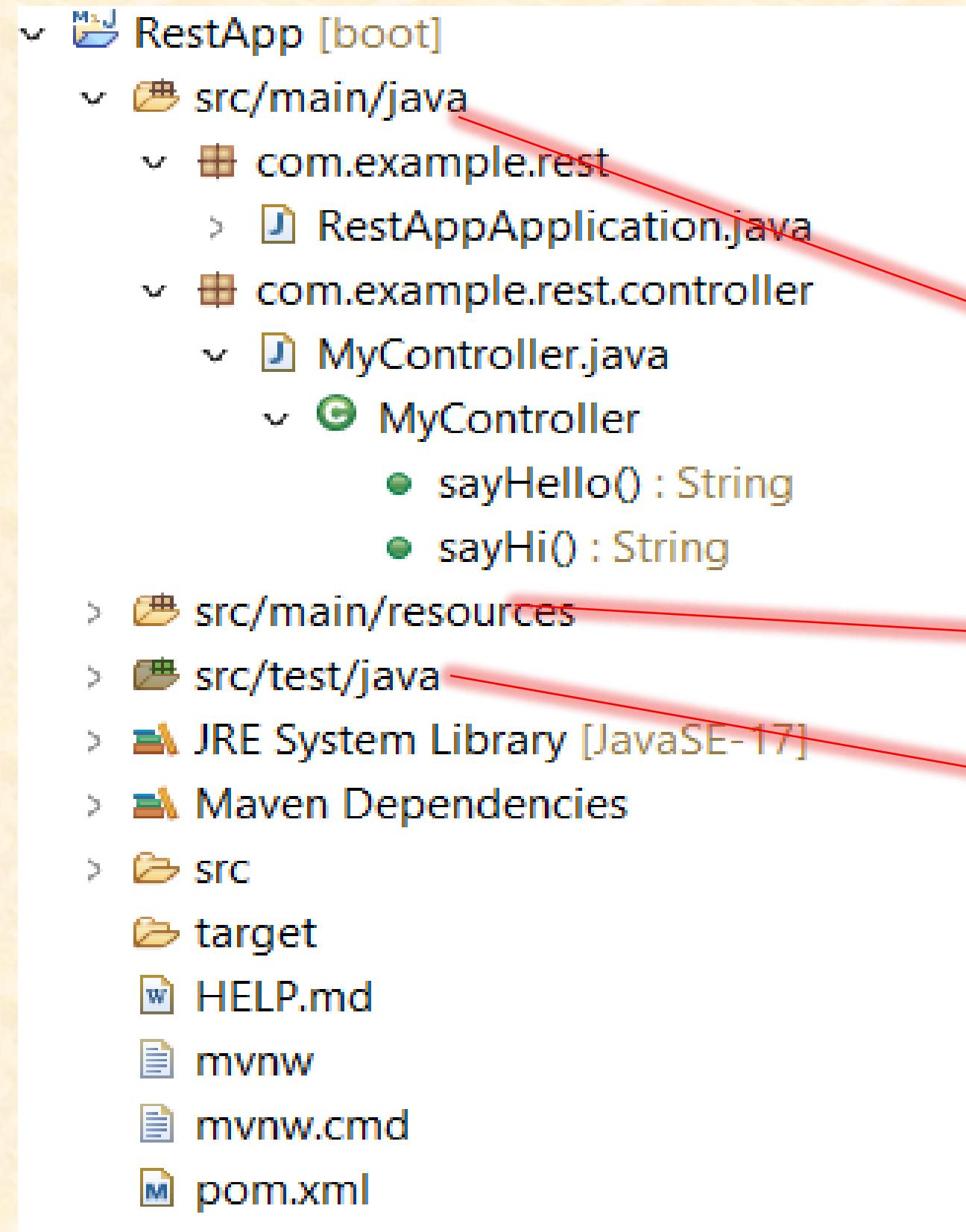
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.0.0</version>
    </dependency>

    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.1.4.Final</version>
    </dependency>
    ...
</dependencies>
</project>
```

Dependency Coordinates

- To add given dependency project, we need
 - **Group ID, Artifact ID**
 - **Version** is optional ...
 - Best practice is to include the version (repeatable builds)
- May see this referred to as: **GAV**
 - **Group ID, Artifact ID and Version**

Maven Standard Directory Structure



Description

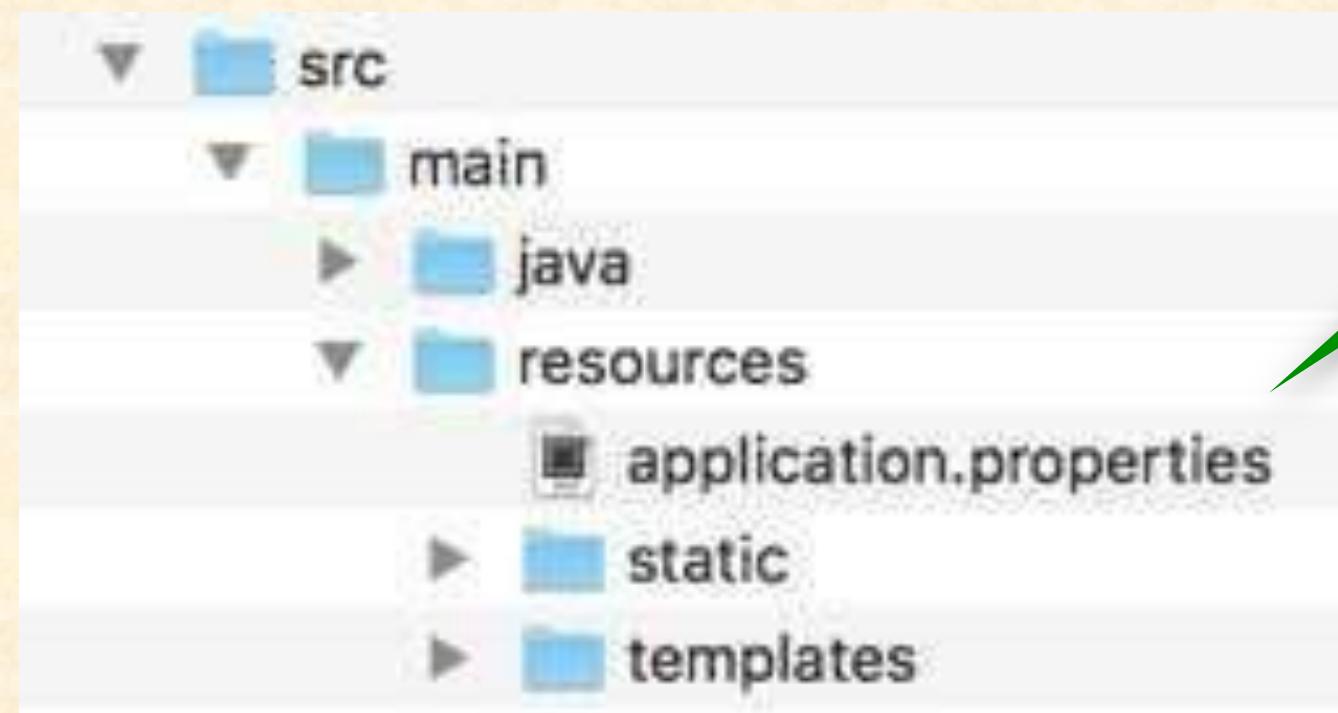
Your Java source code

Properties / config files used by your app

Unit testing source code

Application Properties

- By default, Spring Boot will load properties from: **application.properties**



Created by Spring Initializr

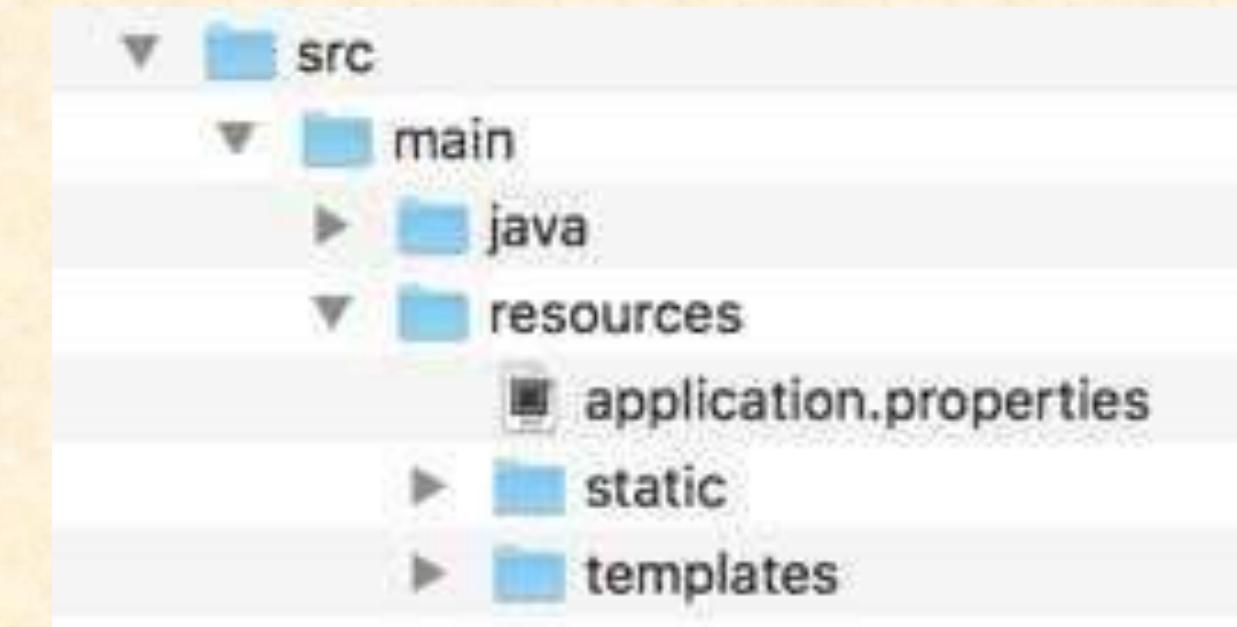
Empty at the beginning

Can add Spring Boot properties
`server.port=8585`

Also add your own custom properties
`coach.name=Mickey Mouse`

Application Properties

- Read data from: `application.properties`



```
# configure server port
server.port=8484

# configure my props
coach.name=Mickey Mouse
team.name=The Mouse Crew
```

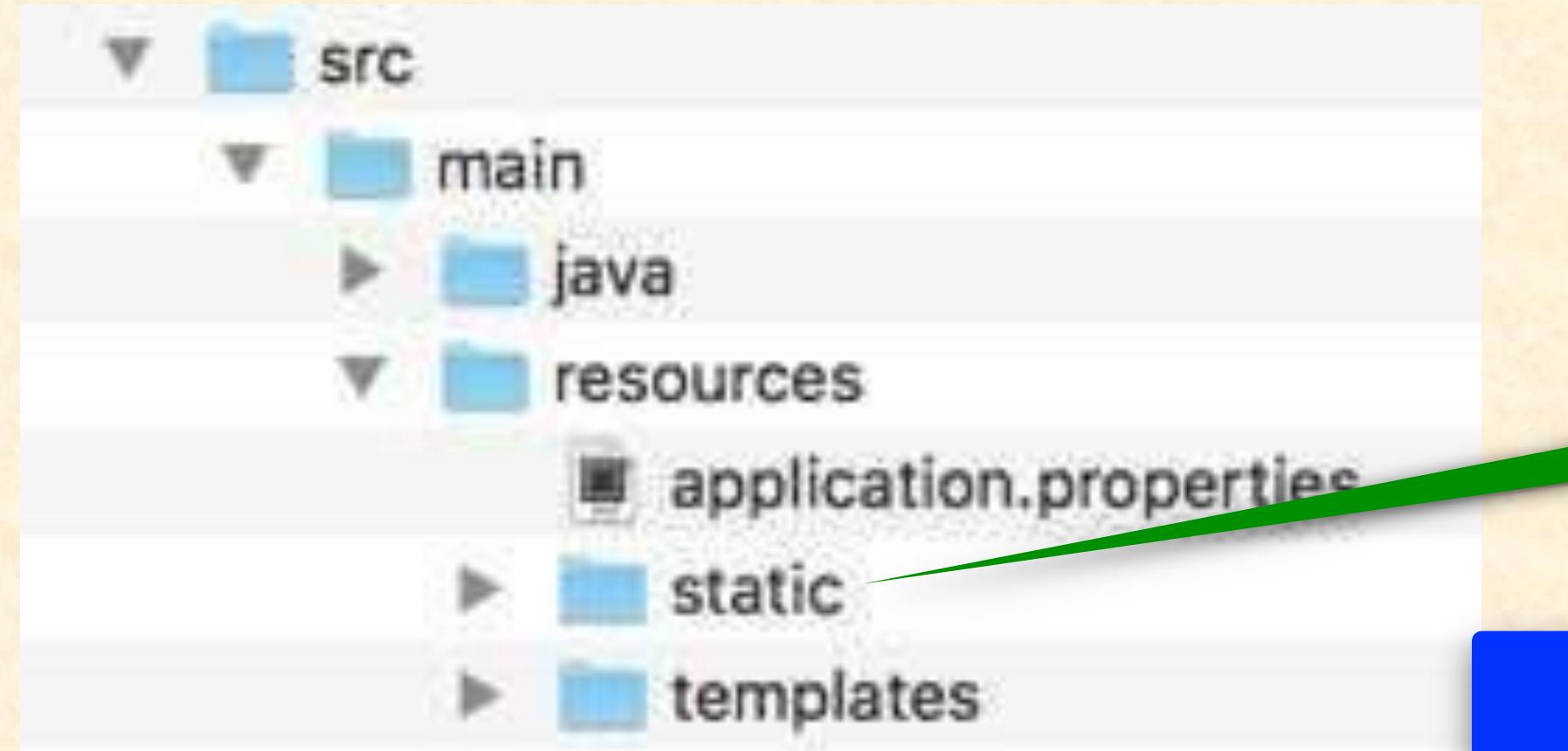
```
@RestController
public class FunRestController {

    @Value("${coach.name}")
    private String coachName;

    @Value("${team.name}")
    private String teamName;

    ...
}
```

Static Content

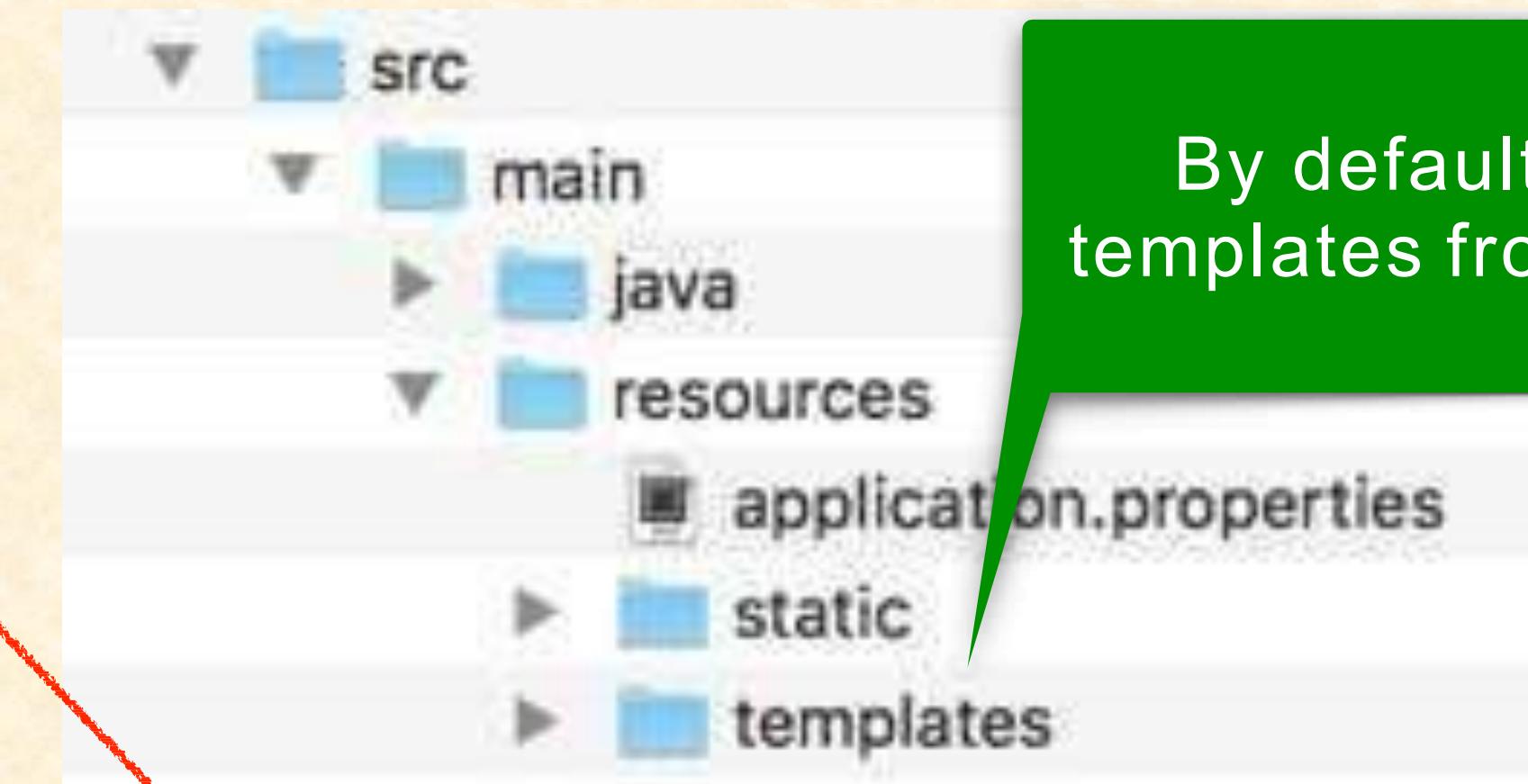


By default, Spring Boot will load static resources from "/static" directory

Examples of static resources
HTML files, CSS, JavaScript, images, etc ...

Templates

- Spring Boot includes auto-configuration for following template engines
 - FreeMarker
 - Thymeleaf
 - Mustache



By default, Spring Boot will load templates from "/templates" directory

Thymeleaf is a popular template engine
We will use it later in the course

Unit Tests

```
~ RestController [boot]
  ~ src/main/java
    ~ com.example.rest
      ~ RestAppApplication.java
        ~ RestAppApplication
          main(String[]) : void
    ~ com.example.rest.controller
      ~ MyController.java
  ~ src/main/resources
    static
    templates
    application.properties
~ src/test/java
  ~ com.example.rest
    ~ RestAppApplicationTests.java
      ~ RestAppApplicationTests
        contextLoads() : void
  JRE System Library [JavaSE-17]
  Maven Dependencies
```

Spring Boot unit test class

Created by Spring Initializr

You can add unit tests to the file

Thank You!



Spring Boot (REST API, MVC and Microservices)

Spring Boot Starters and Dev Tools



The Problem

- Building a Spring application is really HARD!!!

FAQ: Which Maven dependencies do I need?

The Problem

```
<!-- Spring support -->  
  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-webmvc</artifactId>  
    <version>6.0.0-RC1</version>  
</dependency>  
  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-tx</artifactId>  
    <version>6.0.0-RC1</version>  
</dependency>  
  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-orm</artifactId>  
    <version>6.0.0-RC1</version>  
</dependency>
```

Spring
version

Very error-prone

Easy to make
a simple mistake

```
<!-- Spring Security -->  
  
<dependency>  
    <groupId>org.springframework.security</groupId>  
    <artifactId>spring-security-web</artifactId>  
    <version>6.0.0-RC1</version>  
</dependency>  
  
<!-- Hibernate ORM -->  
  
<dependency>  
    <groupId>org.hibernate.orm</groupId>  
    <artifactId>hibernate-core</artifactId>  
    <version>5.4.1.Final</version>  
</dependency>  
  
<dependency>  
    <groupId>org.hibernate.validator</groupId>  
    <artifactId>hibernate-validator</artifactId>  
    <version>7.0.5.Final</version>  
</dependency>
```

Which versions
are compatible?

Why Is It So Hard?

- It would be great if there was a simple list of Maven dependencies
- Collected as a group of dependencies ... one-stop shop
- So We don't have to search for each dependency

There should be an
easier solution

The Solution - Spring Boot Starters

Spring Boot Starters

- A curated list of Maven dependencies
- A collection of dependencies grouped together
- Tested and verified by the Spring Development team
- Makes it much easier for the developer to get started with Spring
- Reduces the amount of Maven configuration

Spring MVC

- For example, when building a Spring MVC app, you normally need

```
<!-- Spring support -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>6.0.0-RC1</version>
</dependency>

<!-- Hibernate Validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>7.0.5.Final</version>
</dependency>

<!-- Web template: Thymeleaf -->
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf</artifactId>
    <version>3.0.15.RELEASE</version>
</dependency>
```

Solution: Spring Boot Starter - Web

- Spring Boot provides: **spring-boot-starter-web**

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

Saves the developer from having to list all of the individual dependencies

Also, makes sure you have compatible versions

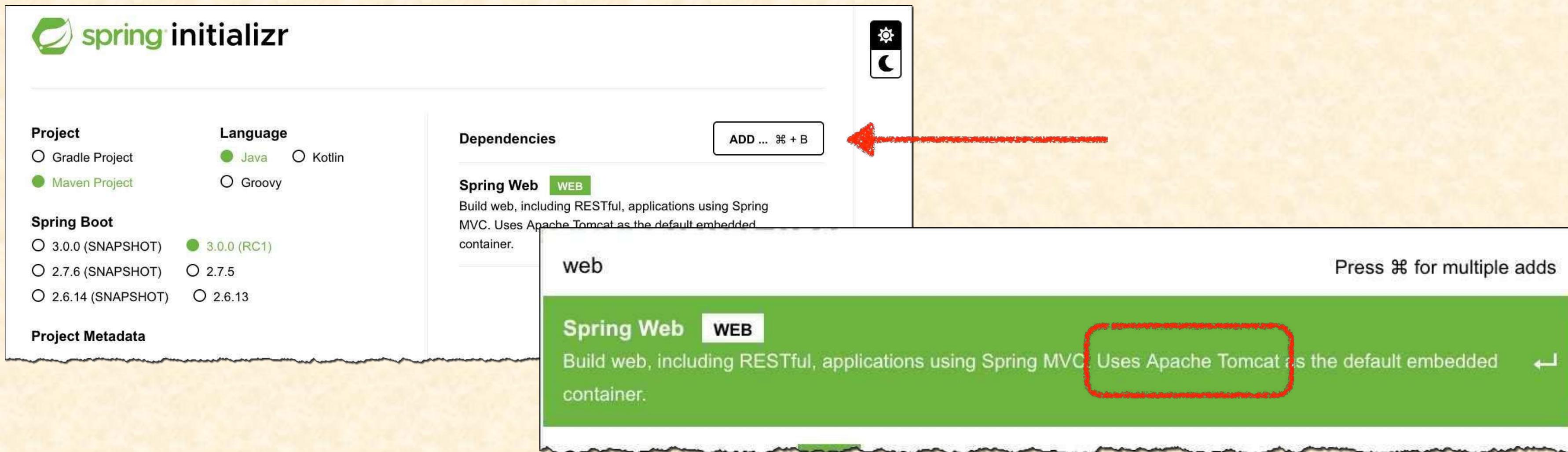
Spring Boot Starters

A collection of Maven dependencies
(Compatible versions)

CONTAINS
spring-web
spring-webmvc
hibernate-validator
json
tomcat
...

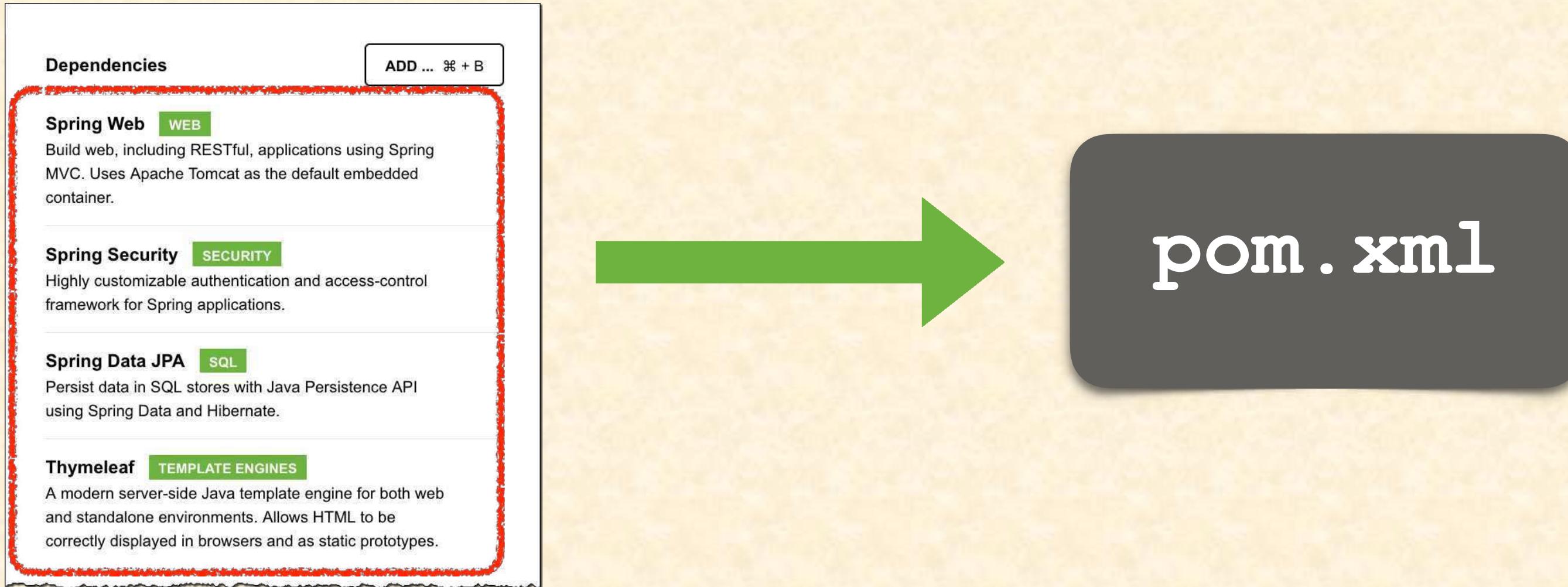
Spring Initializr

- In Spring Initializr, simply select **Web** dependency
- You automatically get **spring-boot-starter-web** in **pom.xml**



Spring Initializr

- If we are building a Spring app that needs: Web, Security, ...
- Simply select the dependencies in the Spring Initializr
- It will add the appropriate Spring Boot starters to your **pom.xml**



Spring Initializr

Dependencies

ADD ... ⌘ + B

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Security SECURITY

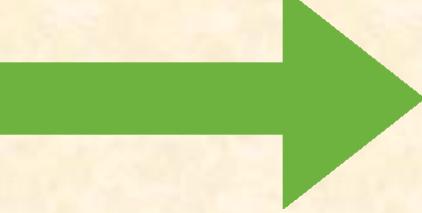
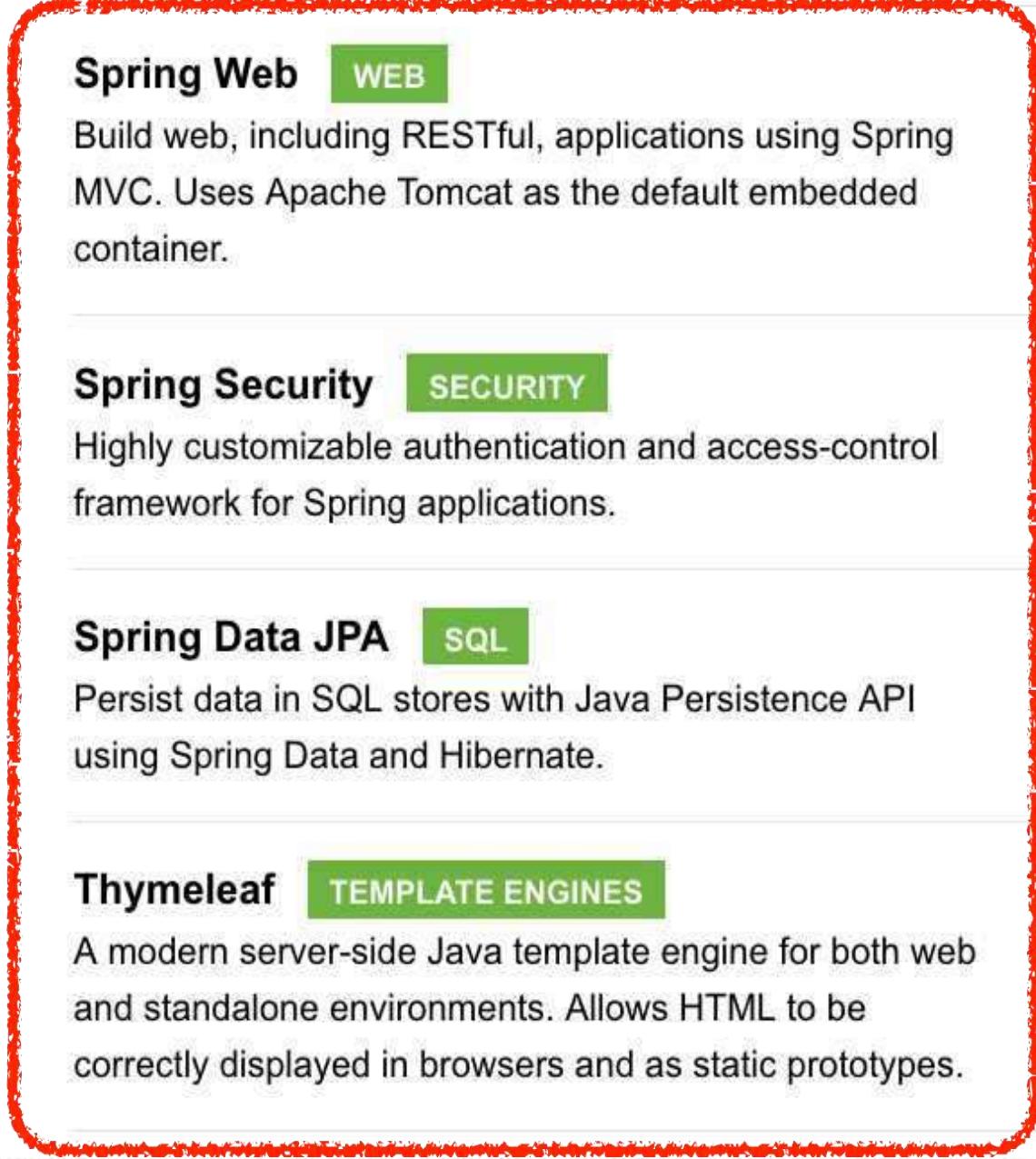
Highly customizable authentication and access-control framework for Spring applications.

Spring Data JPA SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Thymeleaf TEMPLATE ENGINES

A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.



File: pom.xml

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

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

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

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

Spring Boot Starters

- There are 30+ Spring Boot Starters from the Spring Development team

Name	Description
spring-boot-starter-web	Building web apps, includes validation, REST. Uses Tomcat as default embedded server
spring-boot-starter-security	Adding Spring Security support
spring-boot-starter-data-jpa	Spring database support with JPA and Hibernate
...	

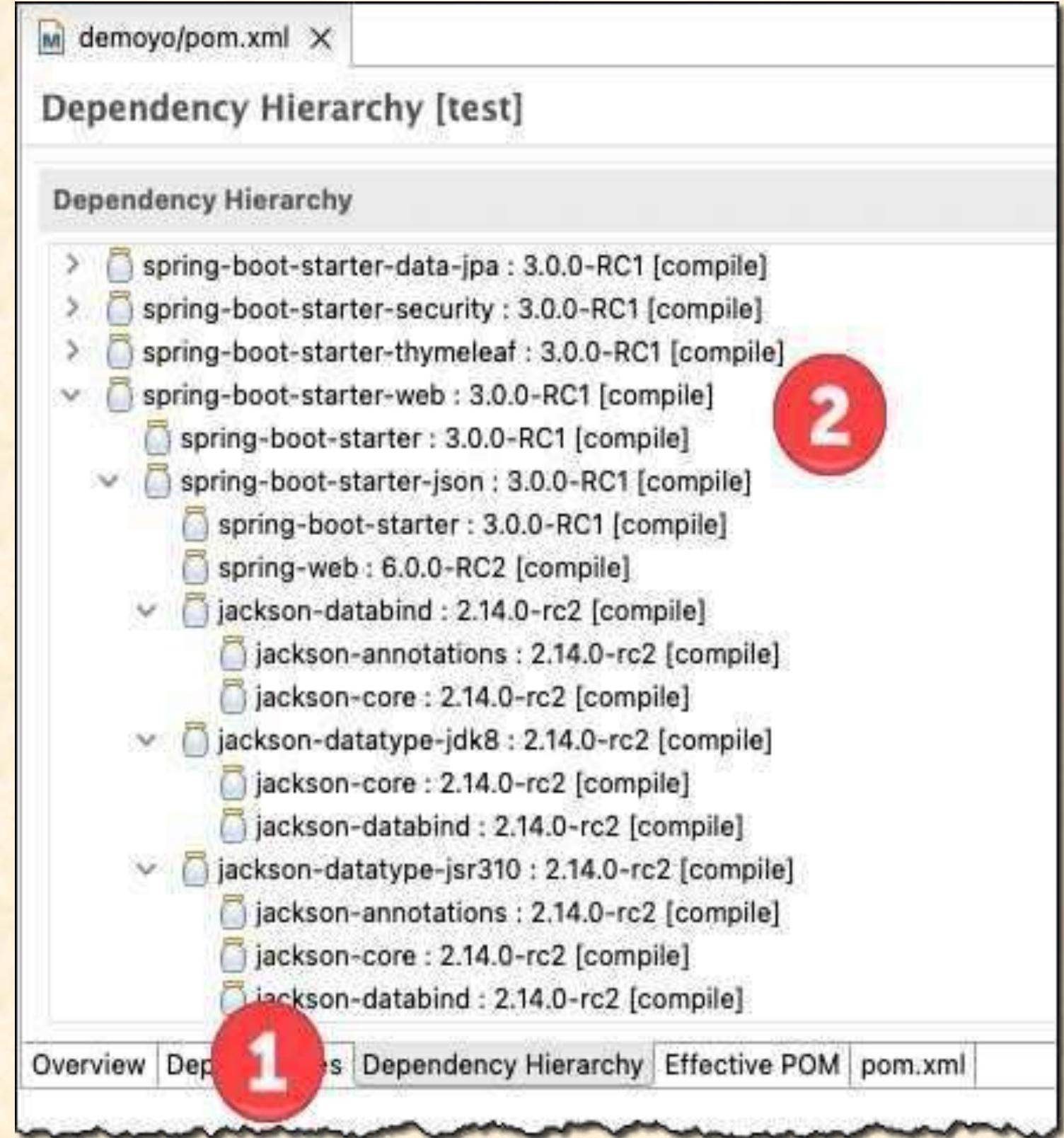
What Is In the Starter?

- FAQ: What is in **spring-boot-starter-xyz**?
- View the starter's POM file
 - Normally references other starters ... so you will need to dig a bit
 - Somewhat cumbersome ...

What Is In the Starter?



- For Eclipse Users
- Open the pom.xml
- Select the tab **Dependency Hierarchy**
- Expand the desired starter



Spring Boot Starter Parent

Spring Boot Starter Parent

- Spring Boot provides a "Starter Parent"
- This is a special starter that provides Maven defaults

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.0.0-RC1</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Included in pom.xml
when using
Spring Initializr

Spring Boot Starter Parent

- Maven defaults defined in the Starter Parent
 - Default compiler level
 - UTF-8 source encoding
 - *Others* ...

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.0.0-RC1</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Spring Boot Starter Parent

- To override a default, set as a property



Specify your
Java version

```
<properties>  
    <java.version>17</java.version>  
</properties>
```

Spring Boot Starter Parent

- For the **spring-boot-starter-*** dependencies, no need to list version

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.0.0-RC1</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Specify version of
Spring Boot

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

Inherit version from
Starter Parent

No need to list individual versions
Great for maintenance!

Spring Boot Starter Parent

- Default configuration of Spring Boot plugin

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

  </plugins>
</build>
```

Benefits of the Spring Boot Starter Parent

- Default Maven configuration: Java version, UTF-encoding etc
- Dependency management
 - Use version on parent only
 - **spring-boot-starter-*** dependencies inherit version from parent
- Default configuration of Spring Boot plugin

Spring Boot Dev Tools

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

The Problem

- When running Spring Boot applications
 - If we make changes to your source code
 - Then we have to manually restart your application

Solution: Spring Boot Dev Tools

- **spring-boot-devtools** to the rescue!
- Automatically restarts your application when code is updated
- Simply add the dependency to your POM file
- No need to write any additional code !...hurray!!

Spring Boot Dev Tools

- Adding the dependency to your POM file

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

Automatically restarts your application when code is updated

Spring Boot (REST API, MVC and Microservices)

Spring Boot Actuator

Problem?

- How can I monitor and manage my application?
- How can I check the application health?
- How can I access application metrics?

Solution: Spring Boot Actuator

- Exposes endpoints to monitor and manage your application
- We easily get DevOps functionality out-of-the-box
- Simply add the dependency to your POM file
- REST endpoints are automatically added to your application

No need to write additional code!

We get new REST endpoints for FREE!

Spring Boot Actuator

- Adding the dependency to your POM file

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

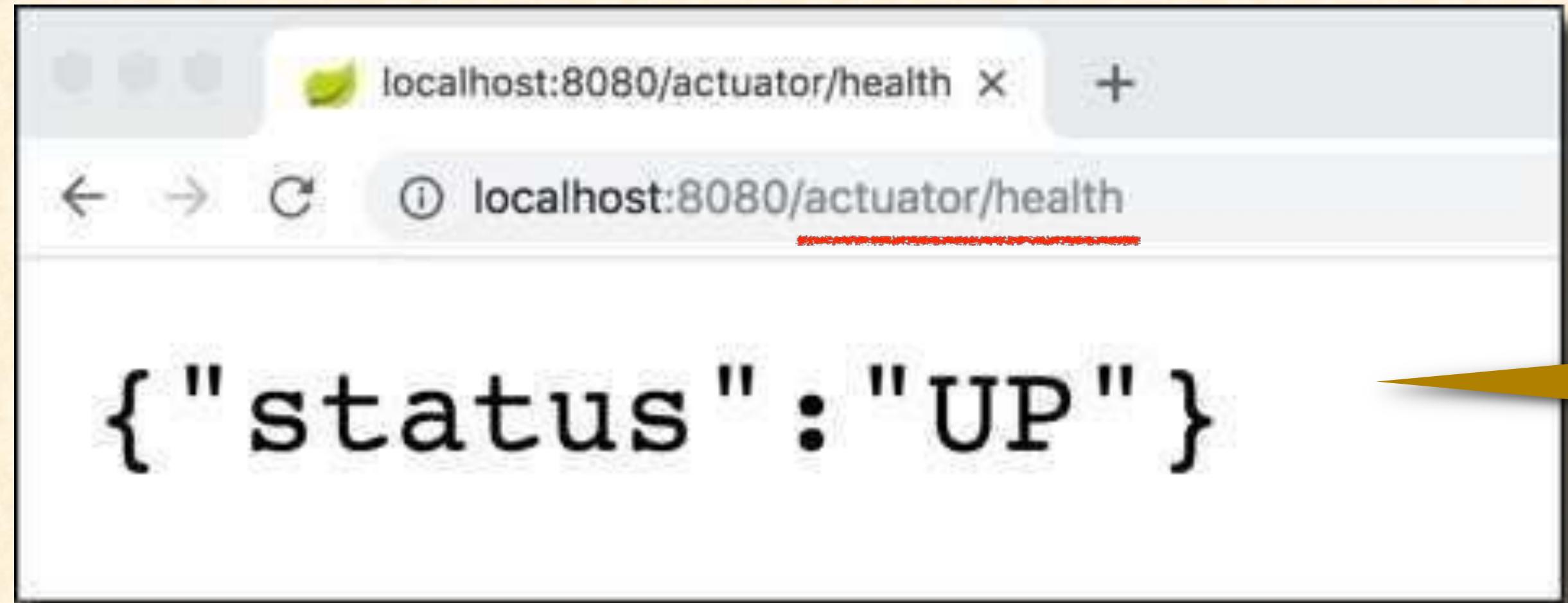
Spring Boot Actuator

- Automatically exposes endpoints for metrics out-of-the-box
- Endpoints are prefixed with: **/actuator**

Name	Description
/health	Health information about your application
...	

Health Endpoint

- **/health** checks the status of your application
- Normally used by monitoring apps to see if your app is up or down



A screenshot of a web browser window. The address bar shows "localhost:8080/actuator/health". The main content area displays the following JSON response:

```
{"status": "UP"}
```

Health status is customizable
based on
your own business logic

Exposing Endpoints

- By default, only `/health` is exposed
- The `/info` endpoint can provide information about your application
- To expose `/info`

File: `src/main/resources/application.properties`

```
management.endpoints.web.exposure.include=health,info  
management.info.env.enabled=true
```

Info Endpoint

- `/info` gives information about your application
- Default is empty

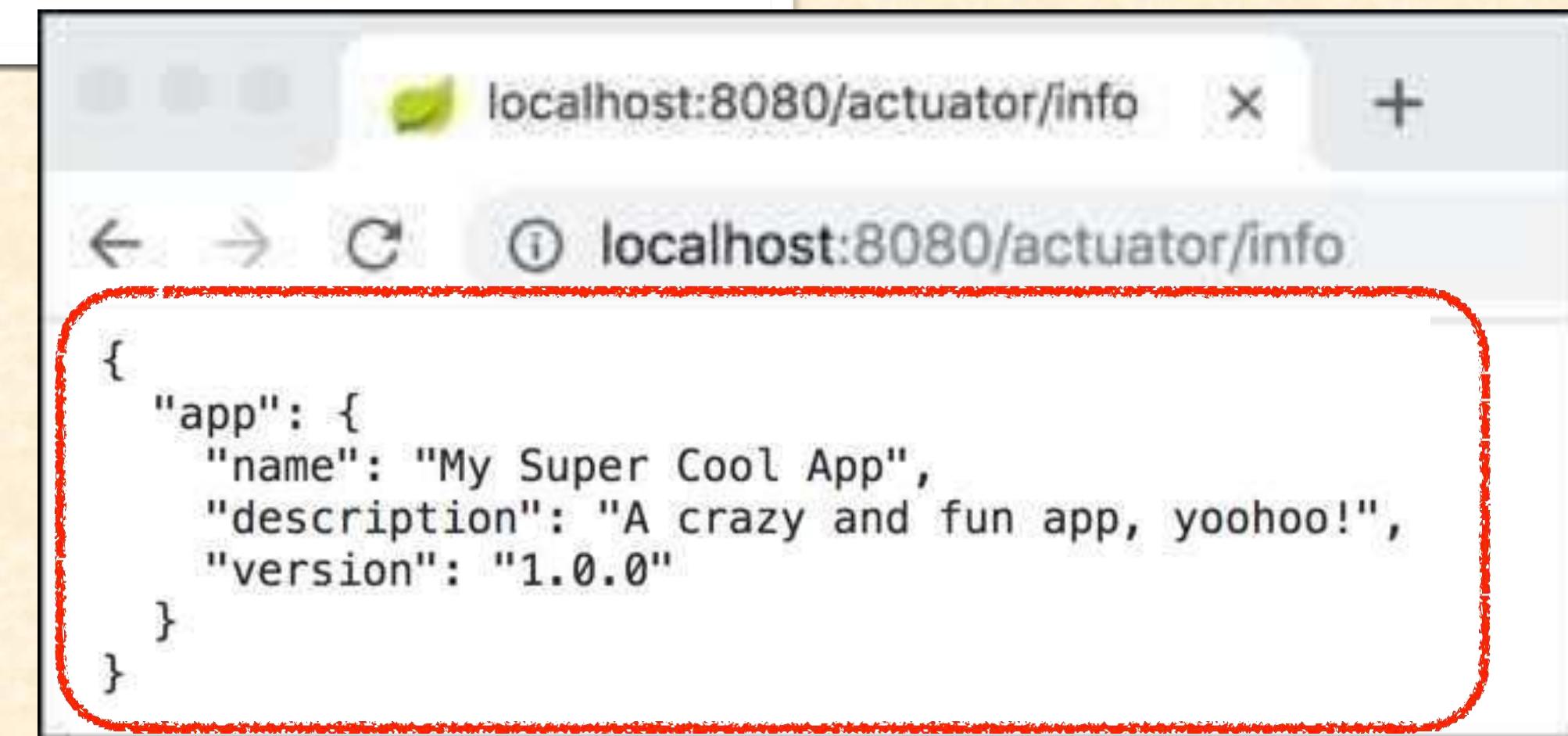
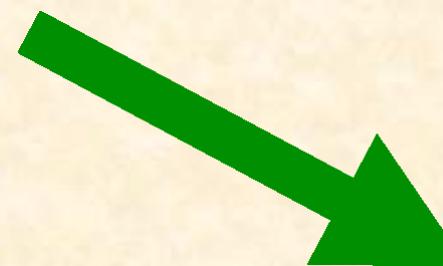


Info Endpoint

- Update **application.properties** with your app info

File: src/main/resources/application.properties

```
info.app.name=My Rest App  
info.app.description=A crazy and fun rest app  
info.app.version=1.0.0
```



Spring Boot Actuator Endpoints

- There are 10+ Spring Boot Actuator endpoints

Name	Description
/auditevents	Audit events for your application
/beans	List of all beans registered in the Spring application context
/mappings	List of all @RequestMapping paths
...	

Exposing Endpoints

- By default, only **/health** is exposed
- To expose all actuator endpoints over HTTP

File: `src/main/resources/application.properties`

```
# Use wildcard "*" to expose all endpoints
# Can also expose individual endpoints with a comma-delimited list
#
management.endpoints.web.exposure.include=*
```

Get A List of Beans

- Access `http://localhost:8080/actuator/beans`

```
{  
  "contexts": {  
    "application": {  
      "beans": {  
        "endpointCachingOperationInvokerAdvisor": {  
          "aliases": [],  
          "scope": "singleton",  
          "type": "org.springframework.boot.actuate.endpoint.invoker.cache.Cach  
          "resource": "class path resource [org/springframework/boot/actuate/au  
          "dependencies": [  
            "environment"  
          ]  
        },  
        "defaultServletHandlerMapping": {  
          "aliases": [],  
          "scope": "singleton",  
          "type": "org.springframework.web.servlet.HandlerMapping",  
          "resource": "class path resource [org/springframework/boot/autoconfig  
          "dependencies": []  
        },  
        "org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfigura  
        "aliases": [],  
        "scope": "singleton",  
        "type": "
```

What about security??

We'll discuss security in later videos

Spring Boot Actuator - Security

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

What about Security?

- You may NOT want to expose all of this information

```
{  
  "contexts": {  
    "application": {  
      "beans": {  
        "endpointCachingOperationInvokerAdvisor": {  
          "aliases": [],  
          "scope": "singleton",  
          "type": "org.springframework.boot.actuate.endpoint.invoker.cache.Cach  
          "resource": "class path resource [org/springframework/boot/actuate/au  
          "dependencies": [  
            "environment"  
          ]  
        },  
        "defaultServletHandlerMapping": {  
          "aliases": [],  
          "scope": "singleton",  
          "type": "org.springframework.web.servlet.HandlerMapping",  
          "resource": "class path resource [org/springframework/boot/autoconfig  
          "dependencies": []  
        },  
        "org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfigura  
        "aliases": [],  
        "scope": "singleton",  
        "type": "
```

Secured Endpoints

- Now when you access: **/actuator/beans**
- Spring Security will prompt for login

The screenshot shows a Spring Security login interface. A purple callout box highlights the default user name "user". A red callout box points to the terminal window below, containing a log message about a generated security password.

Please sign in

Default user name: **user**

Username

Password

Sign in

```
11-02 21:05:57.074 INFO 24986 --- [main] .s.s.UserDetailsSe
Using generated security password: 78fd68a6-c190-421d-934b-df7852fc7dc2
```

Check console logs
for password

Spring Security configuration

- You can override default user name and generated password

File: src/main/resources/application.properties

```
spring.security.user.name=scott  
spring.security.user.password=tiger
```

Customizing Spring Security

- You can customize Spring Security for Spring Boot Actuator
 - Use a database for roles, encrypted passwords etc ...
- We will cover details of Spring Security later in the course

Excluding Endpoints

- To exclude `/health`

File: `src/main/resources/application.properties`

```
# Exclude individual endpoints with a comma-delimited list
#
management.endpoints.web.exposure.exclude=health
```

Custom Application Properties

Problem

- You need for your app to be configurable ... no hard-coding of values
- You need to read app configuration from a properties file

Solution: Application Properties file

- By default, Spring Boot reads information from a standard properties file
 - Located at: **src/main/resources/application.properties**
- You can define ANY custom properties in this file
- Your Spring Boot app can access properties using **@Value**

Standard Spring Boot
file name

No additional coding
or configuration required

Step 1: Define custom application properties

File: src/main/resources/application.properties

```
#  
# Define custom properties  
#  
coach.name=Mickey Mouse  
team.name=The Mouse Club
```

You can give ANY
custom property names

Step 2: Inject Properties into Spring Boot app

```
@RestController  
public class FunRestController {  
  
    // inject properties for: coach.name and team.name  
  
    @Value("${coach.name}")  
    private String coachName;  
  
    @Value("${team.name}")  
    private String teamName;  
  
    ...  
}
```

No additional coding or configuration required

File: src/main/resources/application.properties

```
#  
# Define custom properties  
#  
coach.name=Mickey Mouse  
team.name=The Mouse Club
```

Spring Boot Properties

- Spring Boot can be configured in the **application.properties** file
- Server port, context path, actuator, security etc ...
- Spring Boot has 1,000+ properties ... wow!

Spring Boot Properties

- Don't let the 1,000+ properties overwhelm you
- The properties are roughly grouped into the following categories

Core

Web

Security

Data

Actuator

Integration

DevTools

Testing

Spring Boot Properties

We'll review some of the properties ...

Web Properties

Web

File: src/main/resources/application.properties

```
# HTTP server port  
server.port=7070  
  
# Context path of the application  
server.servlet.context-path=/my-rest-app  
  
# Default HTTP session time out  
server.servlet.session.timeout=15m  
...
```

http://localhost:7070/my-silly-app/fortune

15 minutes

Actuator Properties

Actuator

File: src/main/resources/application.properties

```
# Endpoints to include by name or wildcard
management.endpoints.web.exposure.include=*

# Endpoints to exclude by name or wildcard
management.endpoints.web.exposure.exclude=beans,mapping

# Base path for actuator endpoints
management.endpoints.web.base-path=/actuator

...
```

Security Properties

Security

File: src/main/resources/application.properties

```
# Default user name
spring.security.user.name=admin

# Password for default user
spring.security.user.password=topsecret
...
```

Data Properties

Data

File: src/main/resources/application.properties

```
# JDBC URL of the database
spring.datasource.url=jdbc:mysql://localhost:3306/ecommerce

# Login username of the database
spring.datasource.username=scott

# Login password of the database
spring.datasource.password=tiger
...
```

More on this
in later videos

Spring Boot (REST API, MVC and Microservices)

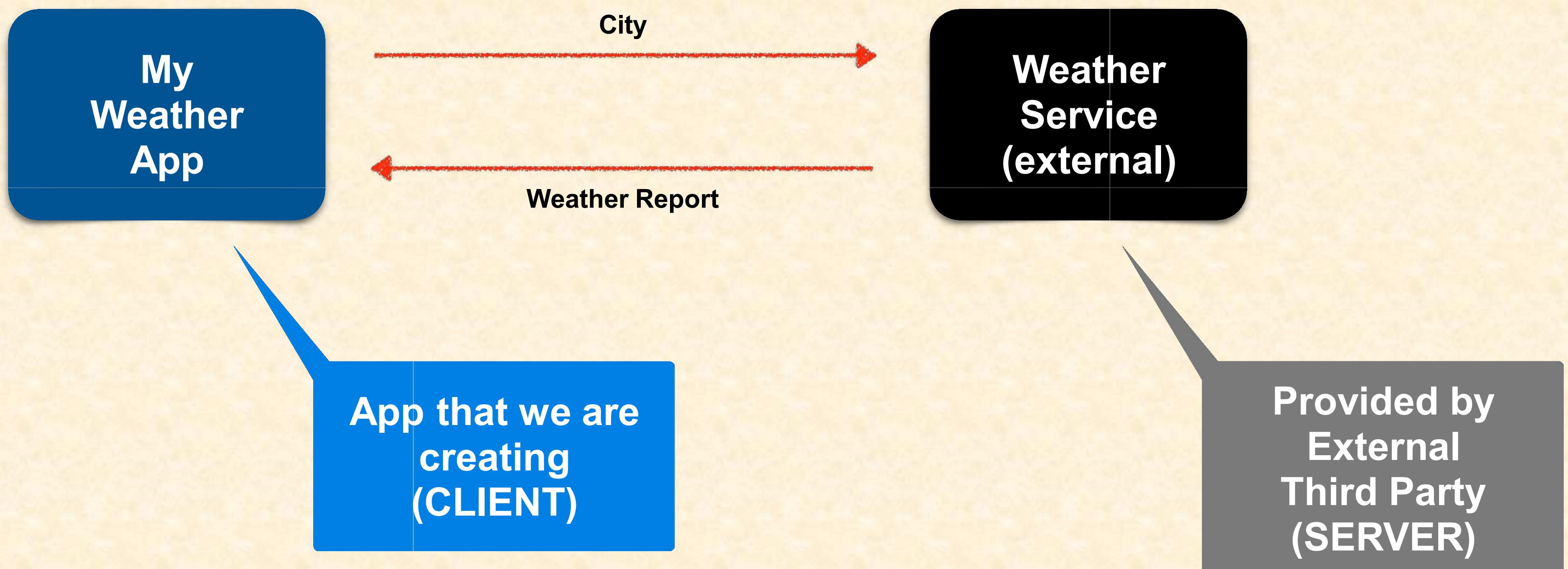
REST Web Services/APIs



Business Problem

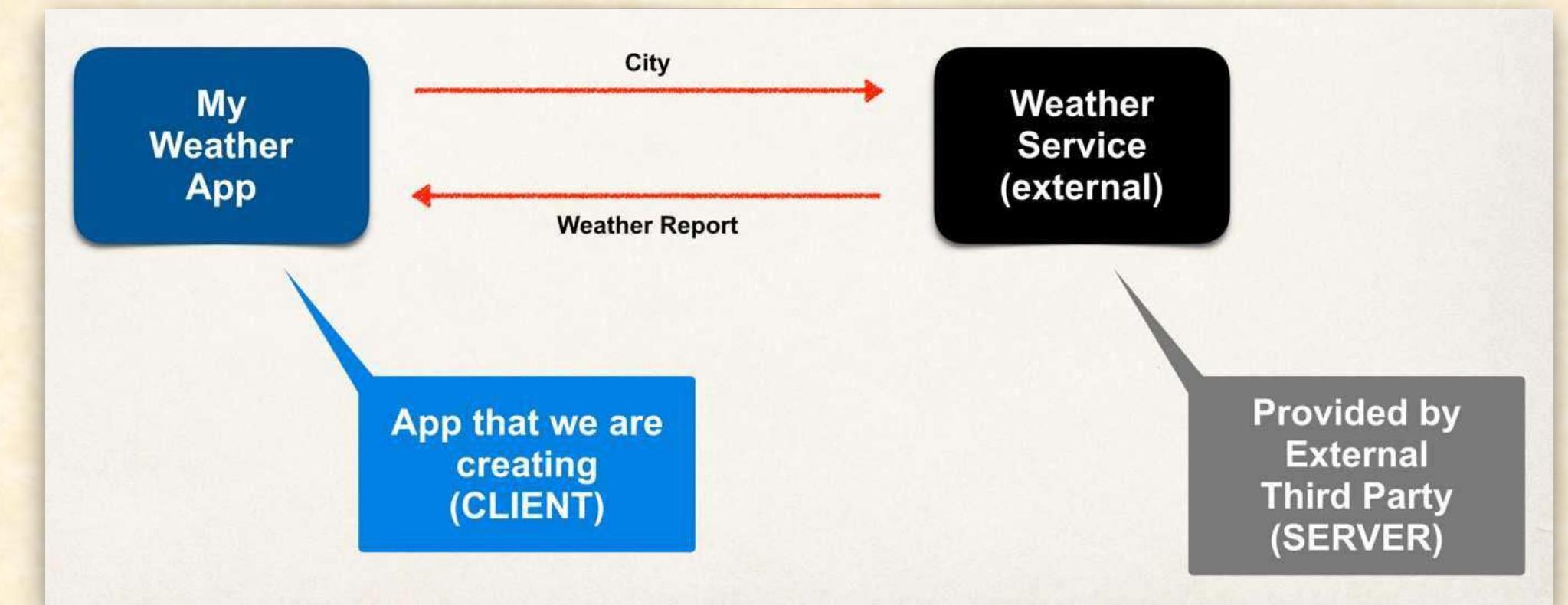
- Build a client app that provides the weather report for a city
- So, we need to get weather data from an external service

Application Architecture



Questions...

1. How to connect to the Weather Service?
2. What programming languages to use?
3. What is the data format?



Answers...

1. How to connect to the Weather Service?

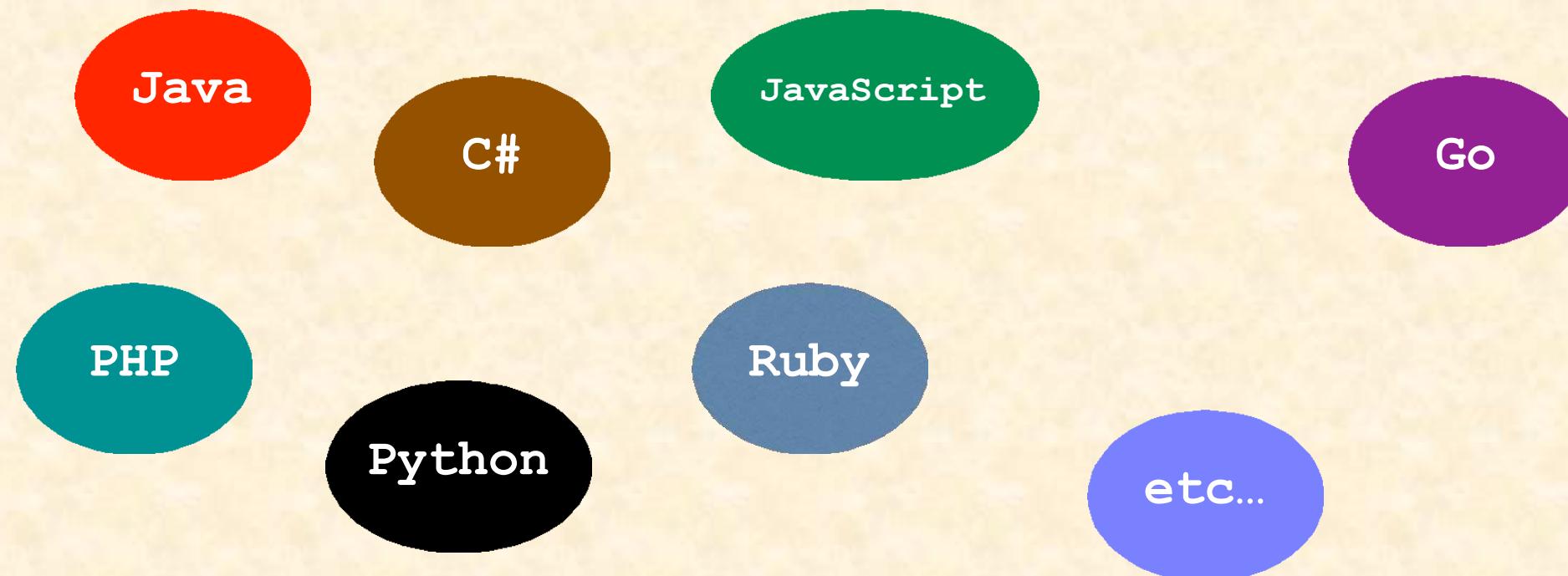
- We can make REST API calls over HTTP
- REST: REpresentational State Transfer
- Lightweight approach for communicating between applications



Answers...

2. What programming languages to use?

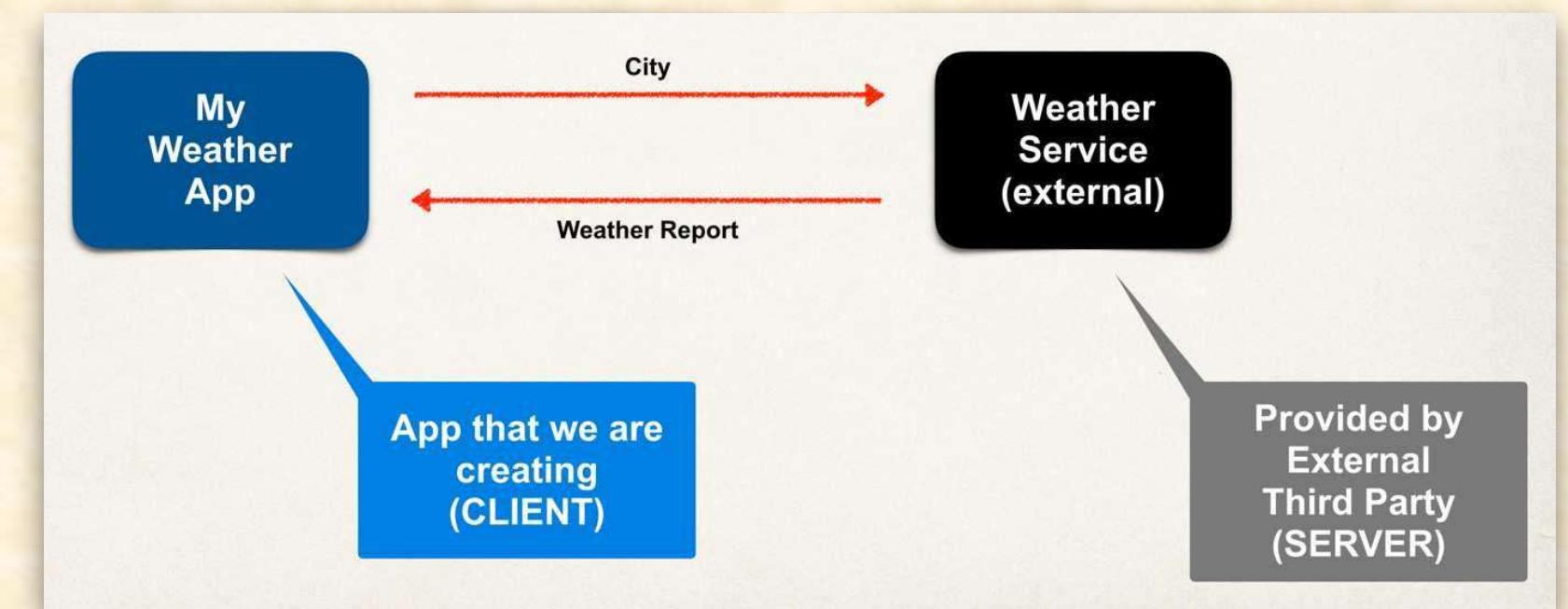
- REST is language independent
- The **client** application can use **ANY** programming language
- The **server** application can use **ANY** programming language



Answers...

3. What is the data format?

- REST applications can use any data format
- Commonly XML and JSON
- JSON is most popular and modern
- JavaScript Object Notation

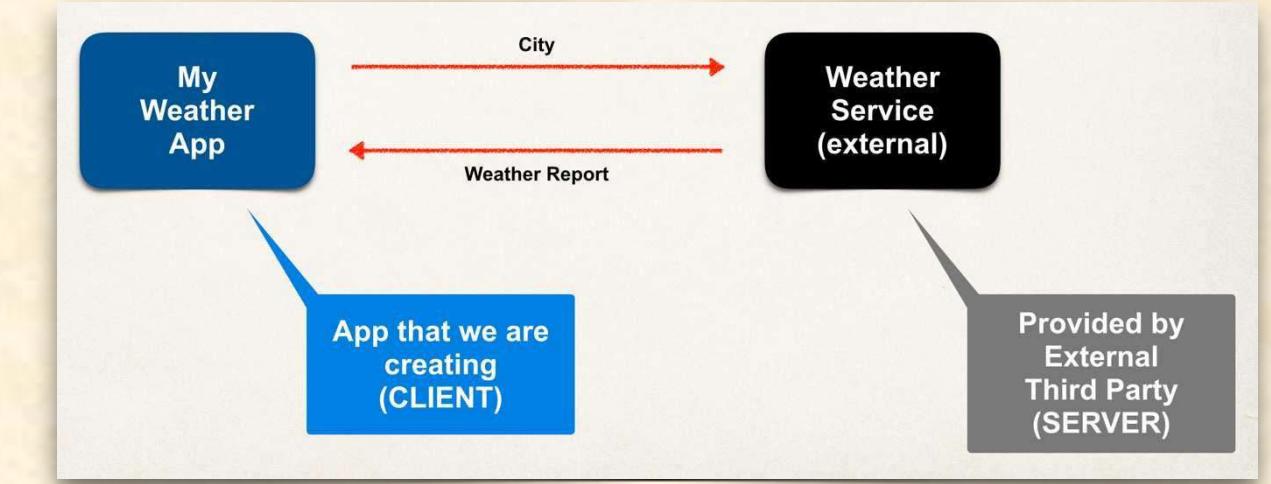


Possible Solution

- Use online Weather Service API provided by: openweathermap.org
- Provide weather data via an API
- Data is available in multiple formats: JSON, XML etc ...

Call Weather Service

- The API documentation gives us the following:
 - Pass in the city name



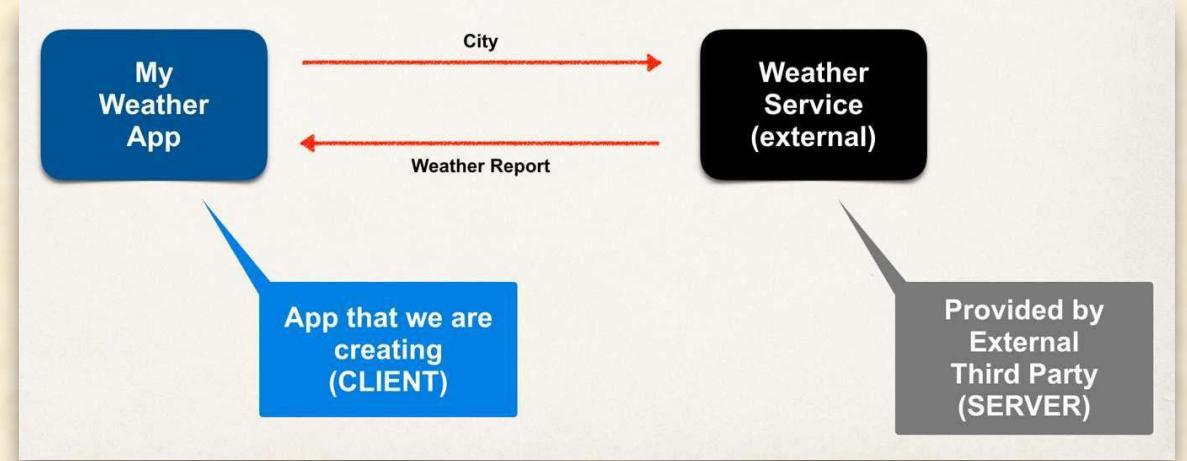
```
api.openweathermap.org/data/2.5/weather?q={city name}
```

OR

```
api.openweathermap.org/data/2.5/weather?q={city name}, {country code}
```

Response - Weather Report

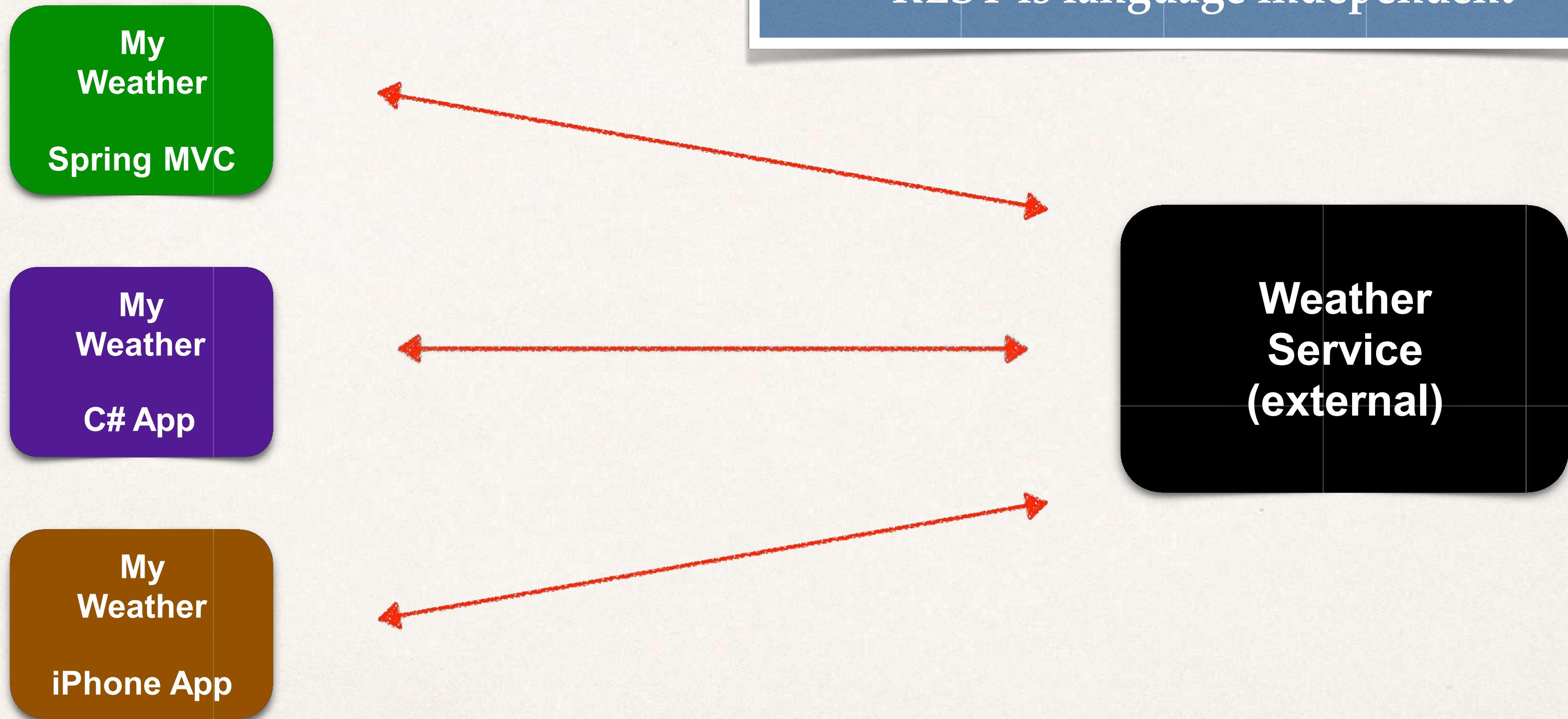
- The Weather Service responds with JSON



```
{  
    ...  
    "temp": 14,  
    "temp_min": 11,  
    "temp_max": 17,  
    "humidity": 81,  
    "name": "London",  
    ...  
}
```

Condensed
version

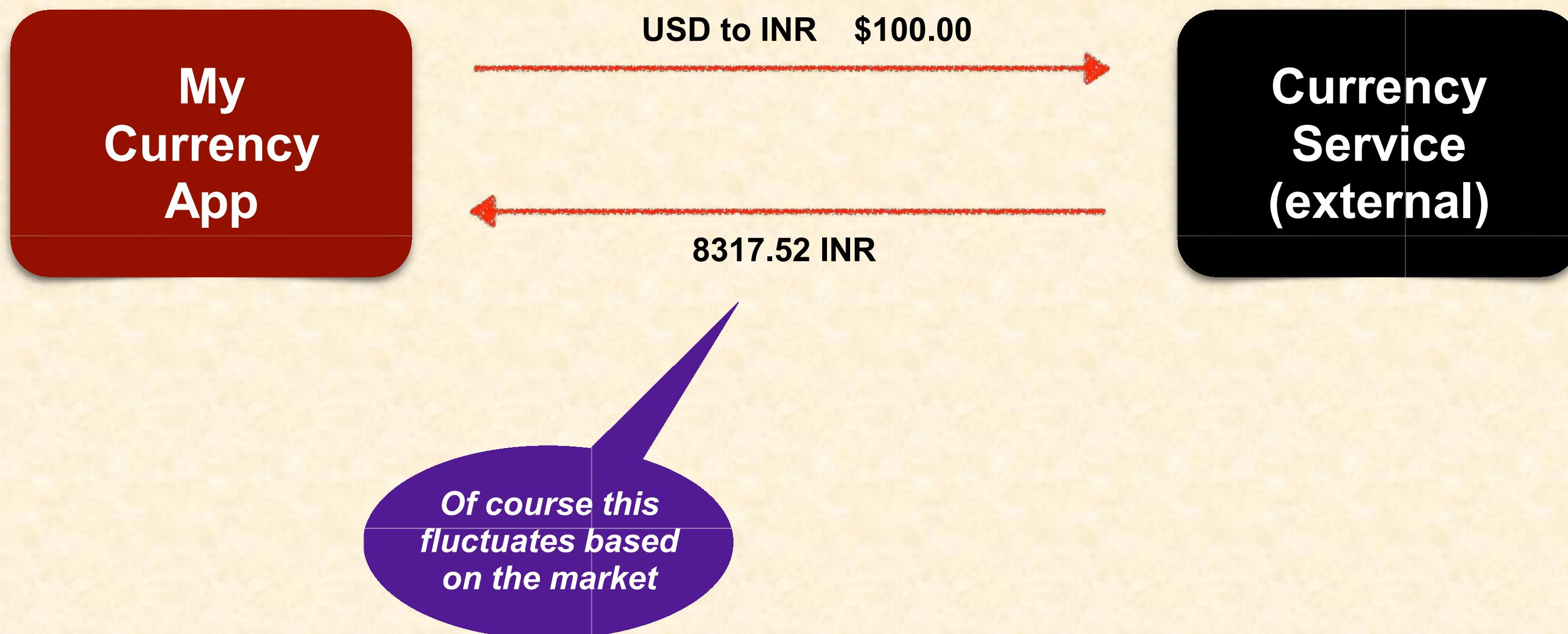
Multiple Client Apps



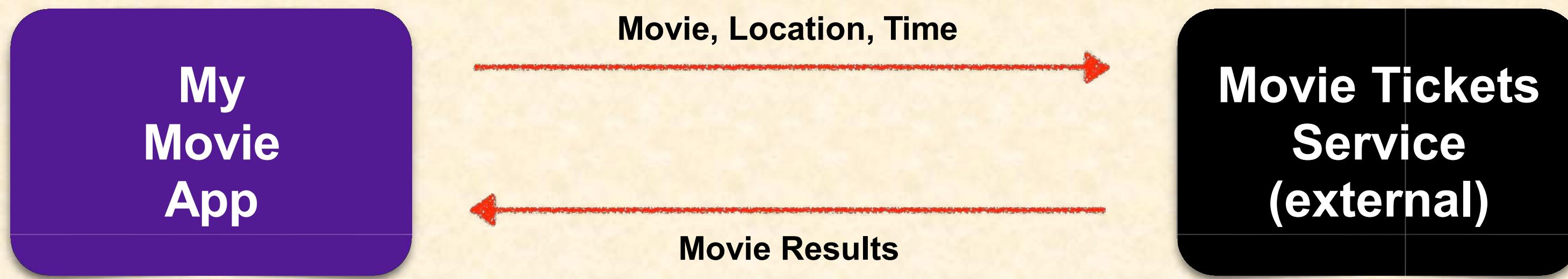
Remember:

REST calls can be made over HTTP
REST is language independent

Currency Converter App



Movie Tickets App



What do we call it?

REST API

REST
Web Services

REST Services

RESTful API

RESTful
Web Services

RESTful Services

Generally, its all mean the SAME thing

What do we call it?

REST API

REST
Web Services

REST Services

RESTful API

RESTful
Web Services

RESTful Services

Generally, all mean the SAME thing

Lets Revisit REST App

- Let's create a very simple REST Controller



Create REST Controller

Set up rest controller

```
@RestController  
public class MyRestController {  
  
    //expose "/" root that returns "Hello World"  
  
    @GetMapping("/")  
    public String sayHello() {  
        return "Hello World!";  
    }  
}
```

Handle HTTP GET requests

Spring Boot (REST API, MVC and Microservices)

Basics of JSON, HTTP and Postman

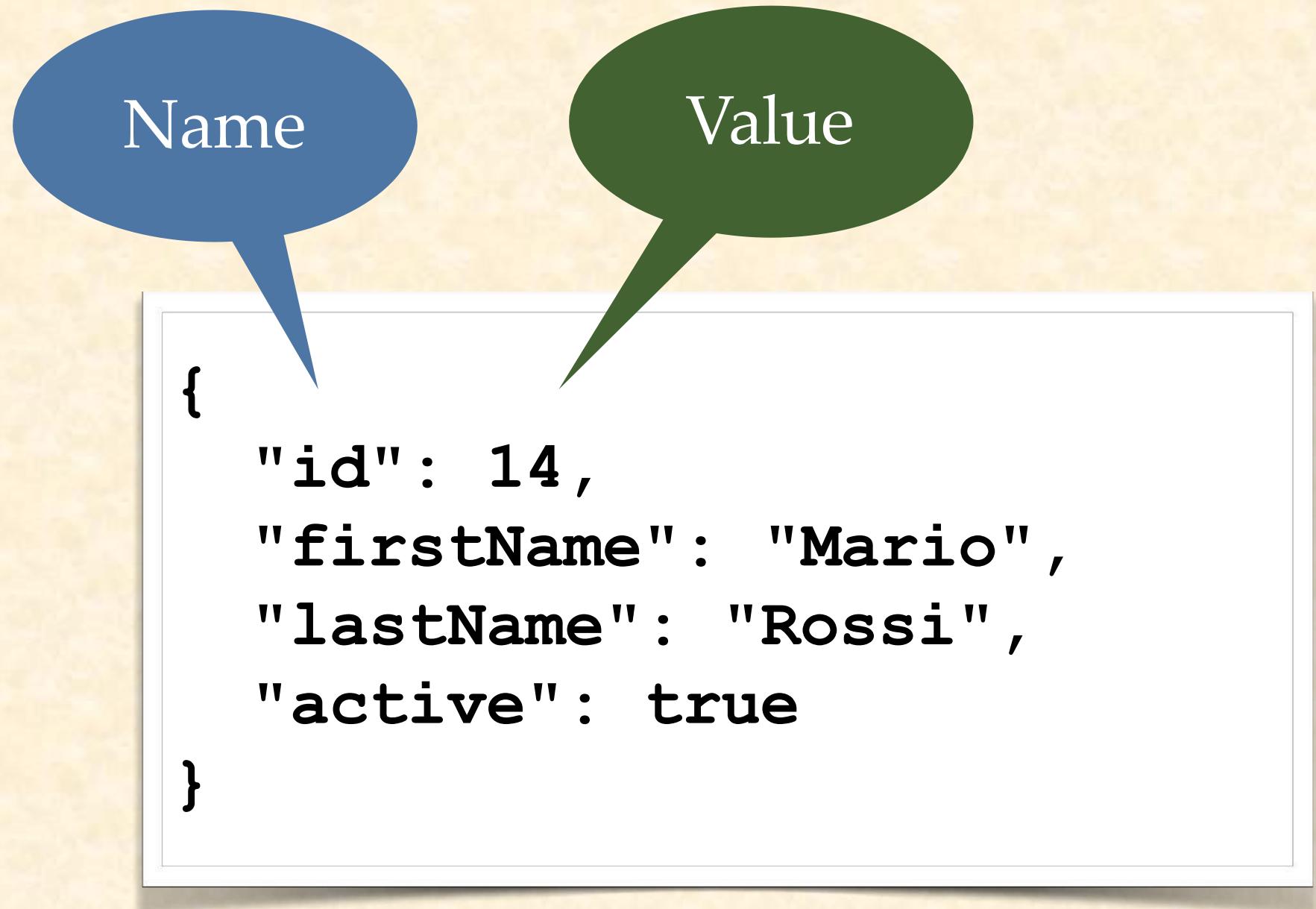
What is JSON?

- JavaScript Object Notation
- Lightweight data format for storing and exchanging data ... plain text
- Language independent ... not just for JavaScript
- We can use with any programming language: Java, C#, Python etc ...

JSON is just
plain text
data

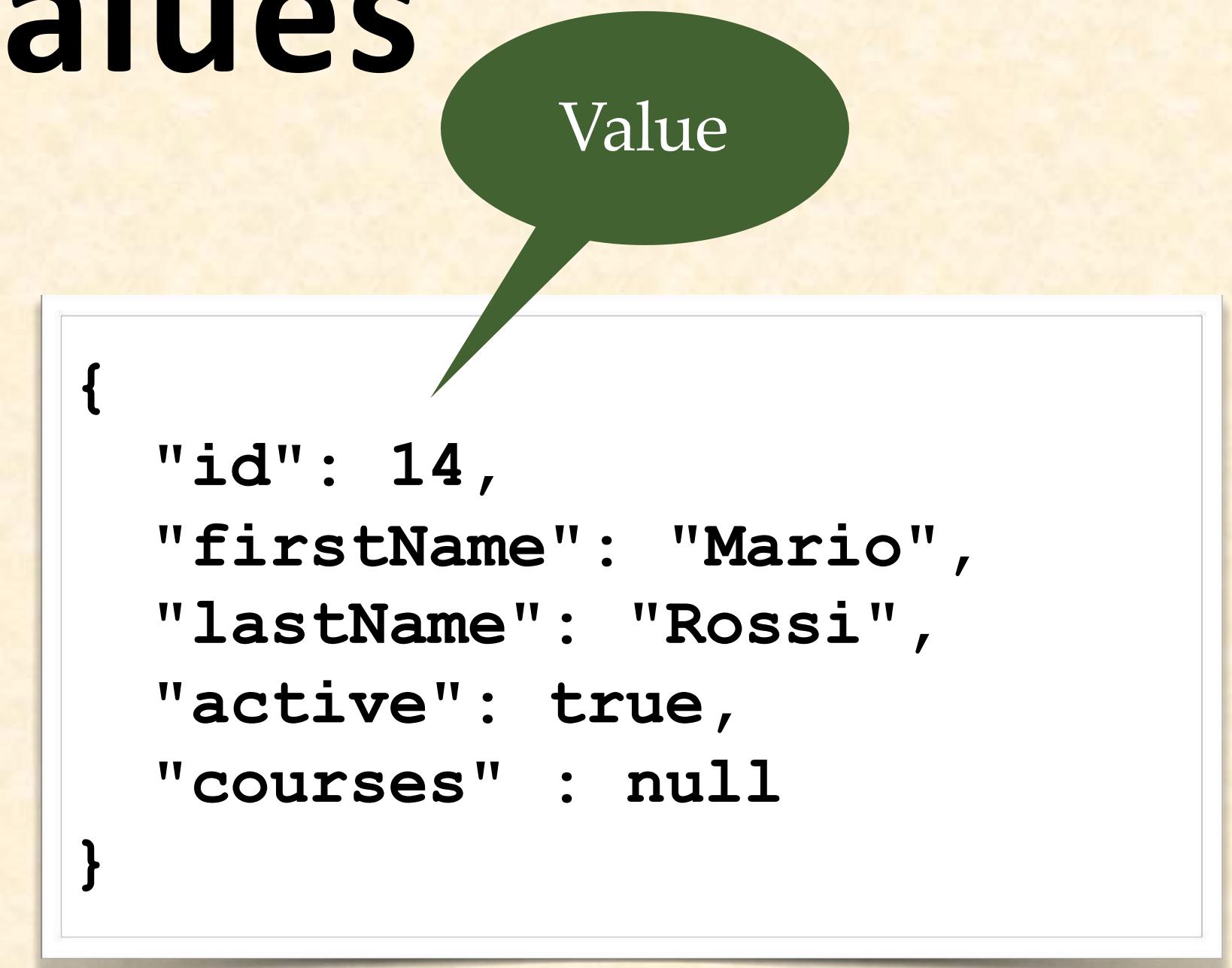
Simple JSON Example

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



JSON Values

- **Numbers:** no quotes
- **String:** in double quotes
- **Boolean:** true, false
- **Nested JSON object**
- **Array**



Nested JSON Objects

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

Nested

JSON Arrays

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

Array

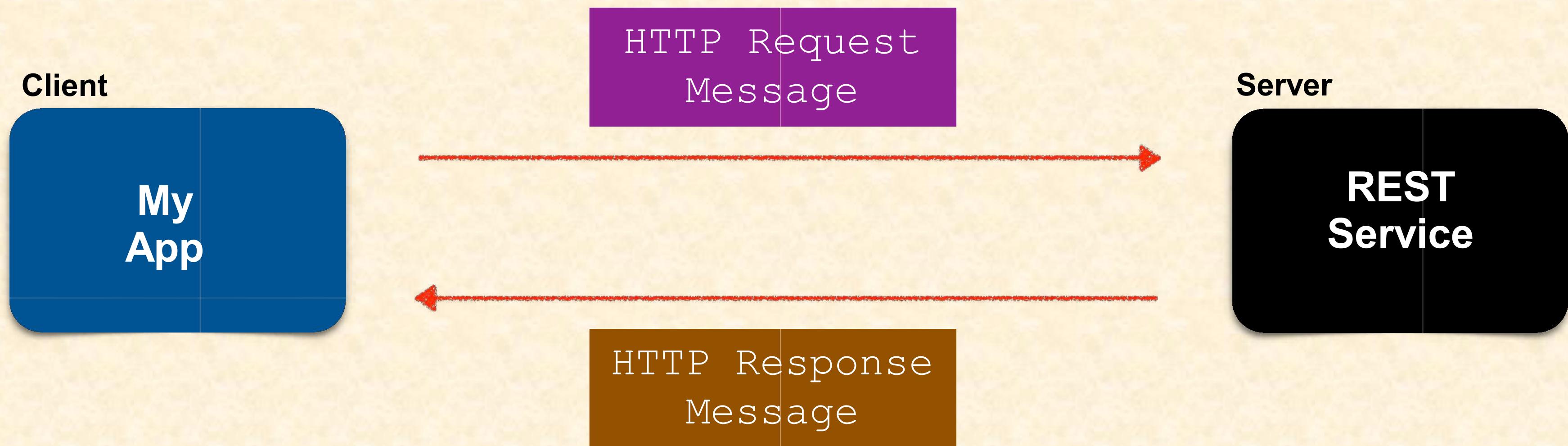
REST HTTP Basics

REST over HTTP

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

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

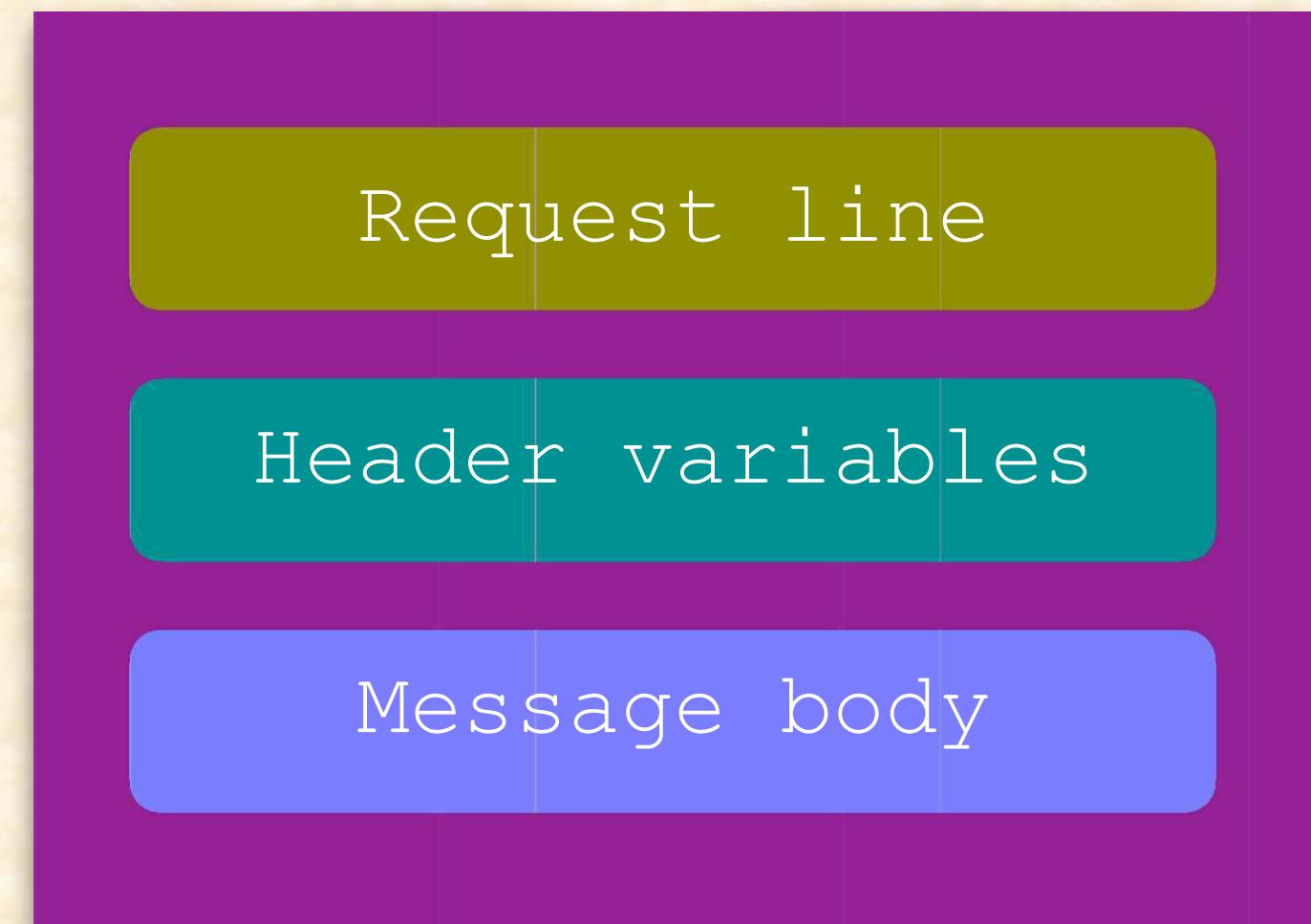
HTTP Messages



HTTP Request Message

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

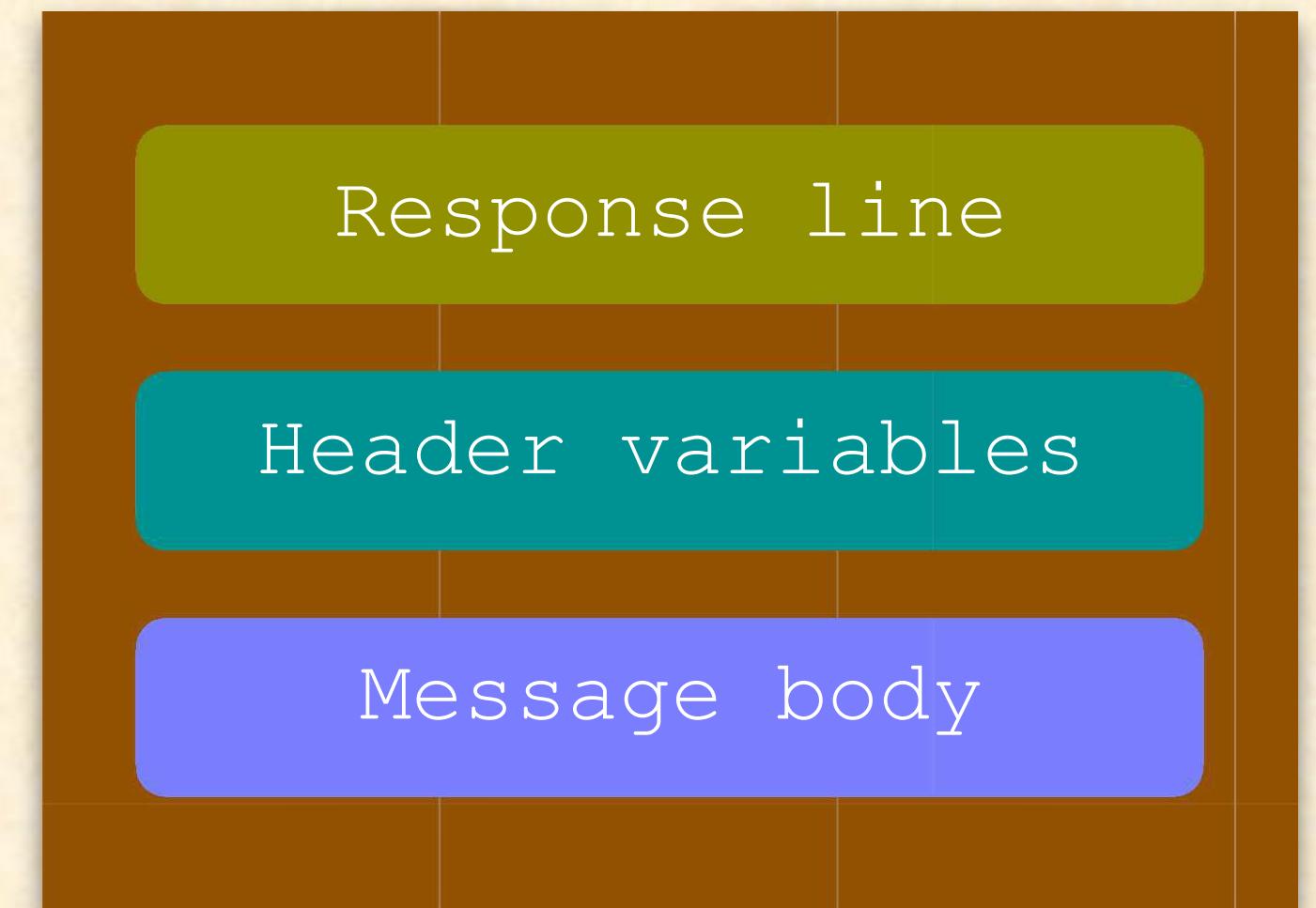
HTTP request Message



HTTP Response Message

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

HTTP response Message



HTTP Response - Status Codes

Code Range	Description
100 - 199	Informational
200 - 299	Successful
300 - 399	Redirection
400 - 499	Client error
500 - 599	Server error

The diagram consists of three arrows pointing from specific status codes to their corresponding descriptions in the table. One arrow points from '401 Authentication Required' to the '401' row. Another arrow points from '404 File Not Found' to the '404' row. A third arrow points from '500 Internal Server Error' to the '500' row.

MIME Content Types

- The message format is described by MIME content type
 - Multipurpose Internet Mail-Extension
- Basic Syntax: type/sub-type
- Examples
 - text/html, text/plain
 - application/json, application/xml, ...

Client Tool

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

Postman



www.getpostman.com

Install Postman Now

www.getpostman.com

Choose your platform:



Postman for Mac
for OS X Yosemite or later

[Download](#)



Postman for Windows
for Windows 7 or later

[x64 ▾](#) [Download](#)



Postman for Linux

[x64 ▾](#) [Download](#)

Postman Demo

The screenshot shows the Postman application interface. At the top, the URL bar contains `http://localhost:8080/`. To the right of the URL are buttons for creating a new request (`+`) and more options (`...`). Further right are dropdowns for selecting an environment (set to "No Environment") and icons for viewing and settings.

The main request configuration area shows a `GET` method selected, and the URL `http://localhost:8080/spring-rest-demo/test/hello` entered. To the right of the URL are buttons for "Params", "Send" (which is highlighted in blue), and "Save".

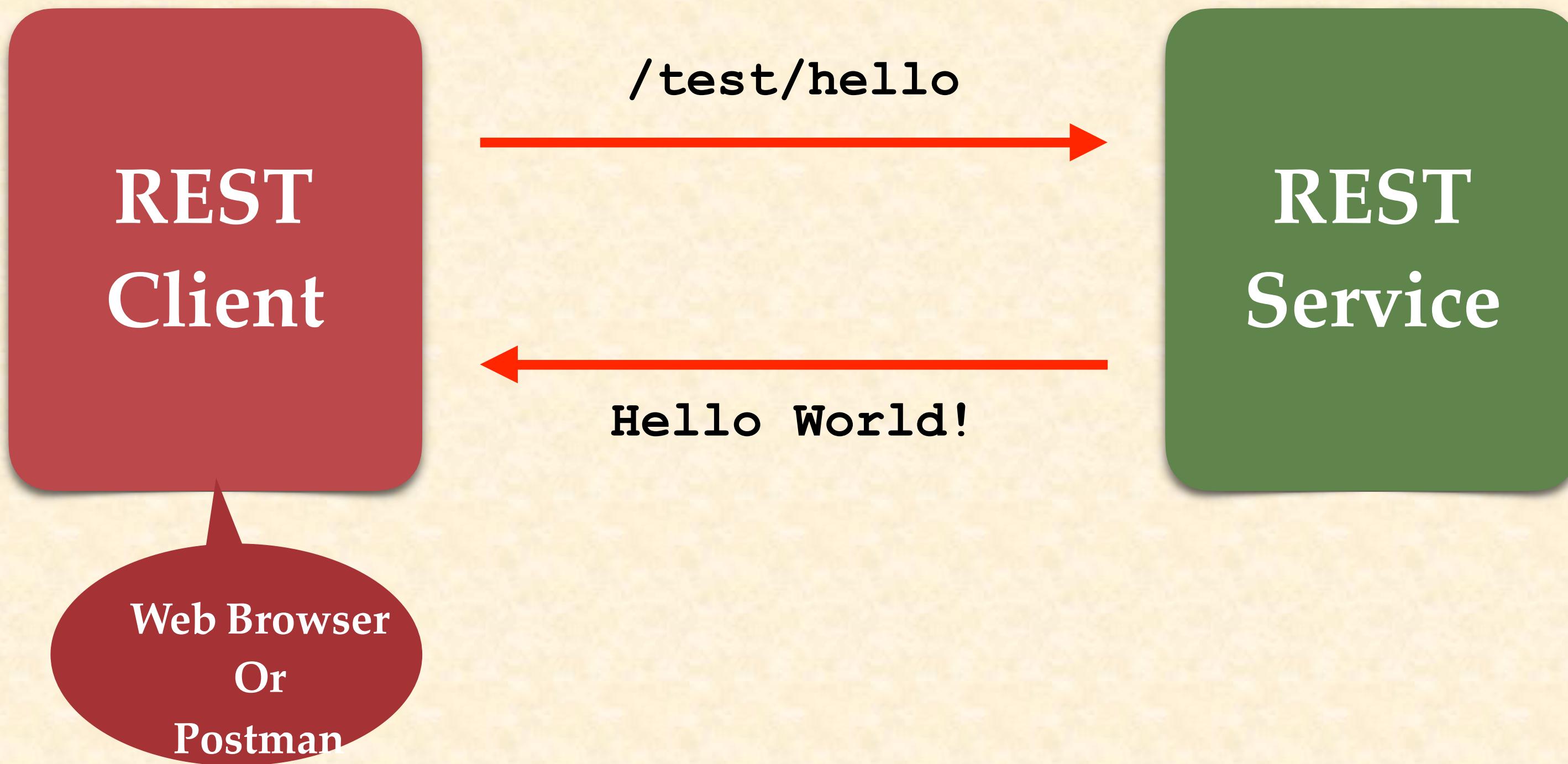
Below the request configuration are tabs for "Authorization", "Headers", "Body", "Pre-request Script", "Tests", "Cookies", and "Code". The "Body" tab is currently active. On the far right of this row are "Cookies" and "Code" buttons.

The response section below shows the status as `200 OK`, time taken as `53 ms`, and size of `133 B`. Below the status are buttons for "Pretty", "Raw", "Preview", and "Text" (with a dropdown arrow). To the right of these are icons for copy and search.

The "Body" panel displays the response content, which is a single line: `1 Hello World!`.

Spring REST Hello World

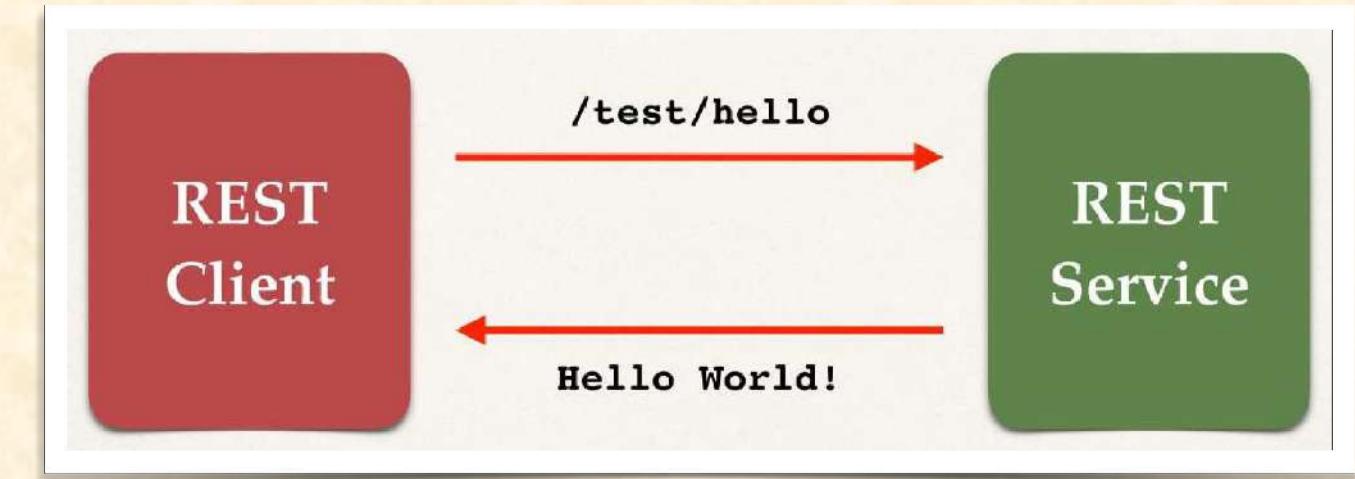
We will write
this code



Spring REST Controller

Adds REST support

```
@RestController  
@RequestMapping("/test")  
public class DemoRestController {  
  
    @GetMapping("/hello")  
    public String sayHello() {  
        return "Hello World!";  
    }  
  
}
```

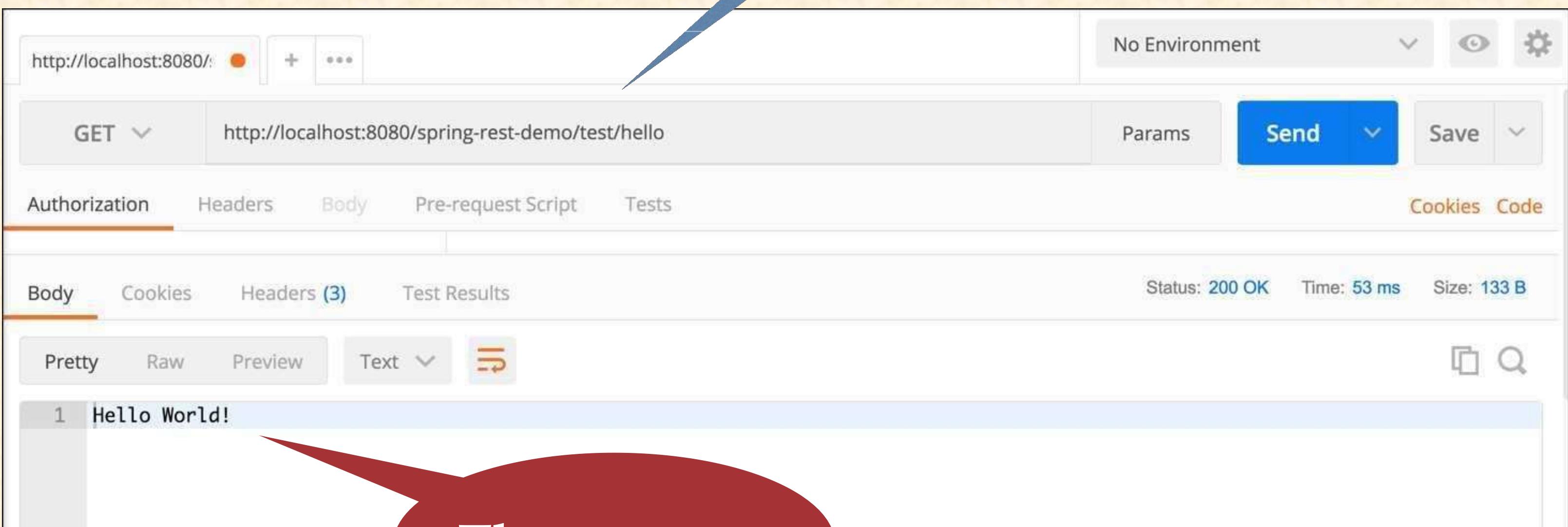


Access the REST endpoint at
/test/hello

Returns content to
client

Testing with REST Client - Postman

Access the REST endpoint at
`/test/hello`



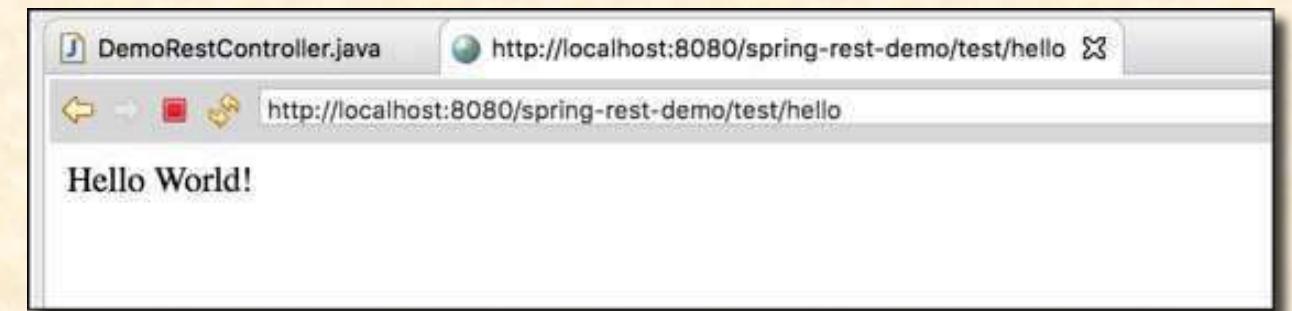
The response

Testing with REST Client - Web Browser



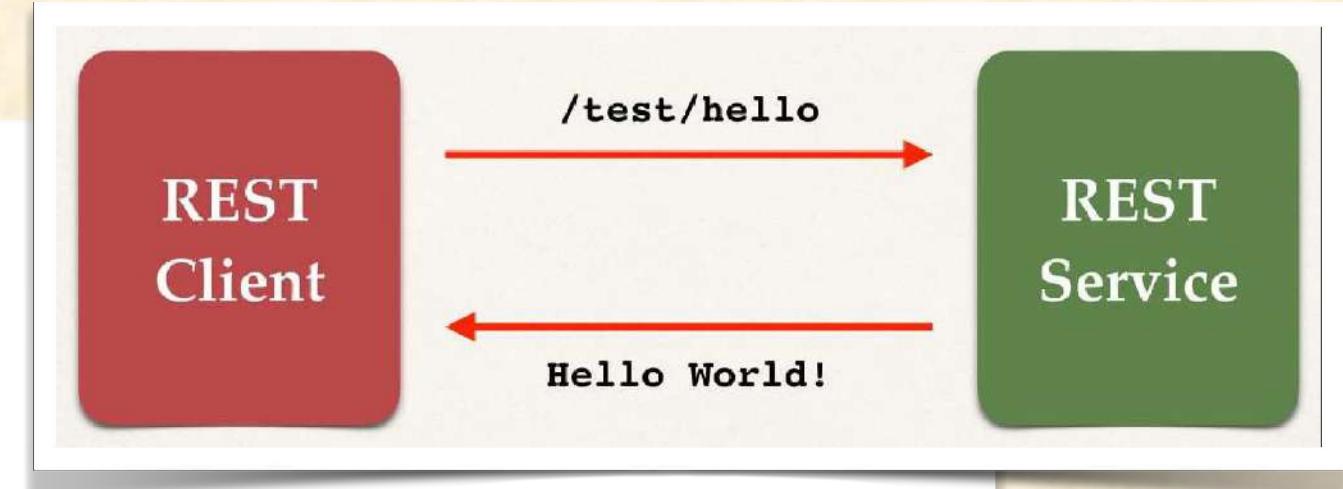
Web Browser vs Postman

- For simple REST testing for GET requests
 - Web Browser and Postman are similar
- However, for advanced REST testing: POST, PUT etc ...
 - Postman has much better support
 - POSTing JSON data, setting content type
 - Passing HTTP request headers, authentication etc ...



Create Spring REST Service

```
@RestController  
@RequestMapping("/test")  
public class DemoRestController {  
  
    @GetMapping("/hello")  
    public String sayHello() {  
        return "Hello World!";  
    }  
  
}
```



Handles HTTP GET requests

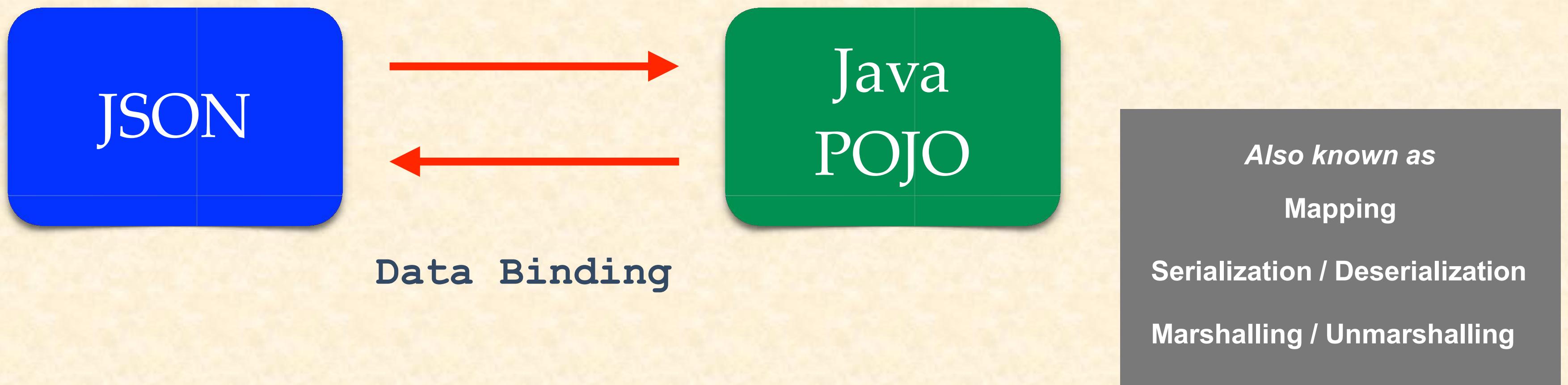
Spring Boot (REST API, MVC and Microservices)

Java -JSON Data Binding

Jackson Project

Java JSON Data Binding

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



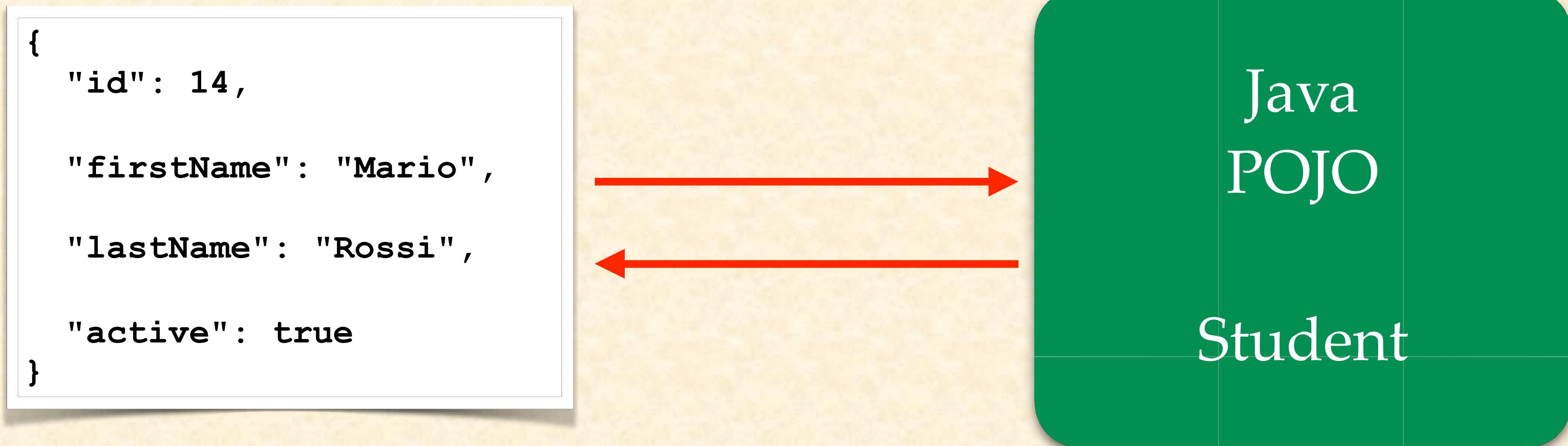
JSON Data Binding with Jackson

- 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

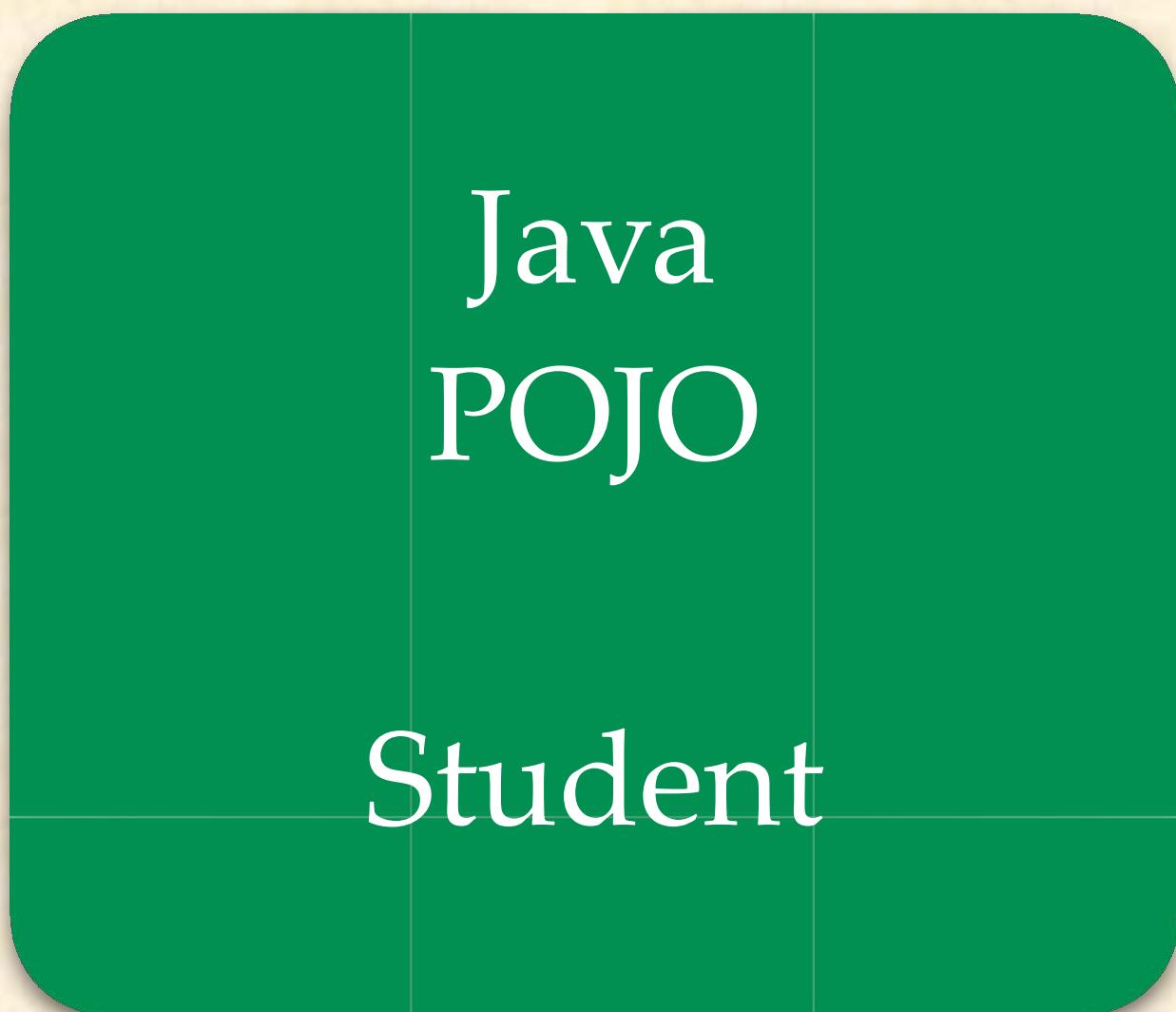
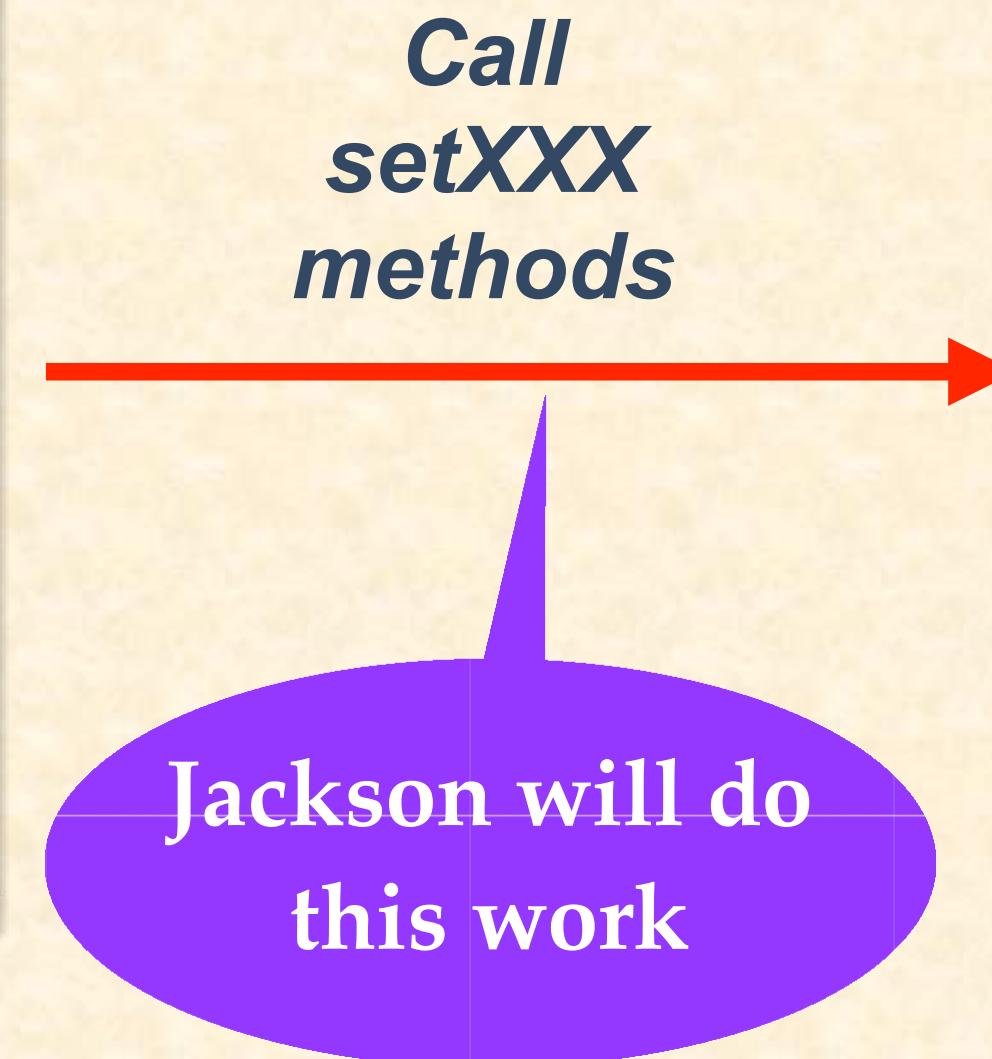
- By default, Jackson will call appropriate getter/setter method



JSON to Java POJO

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

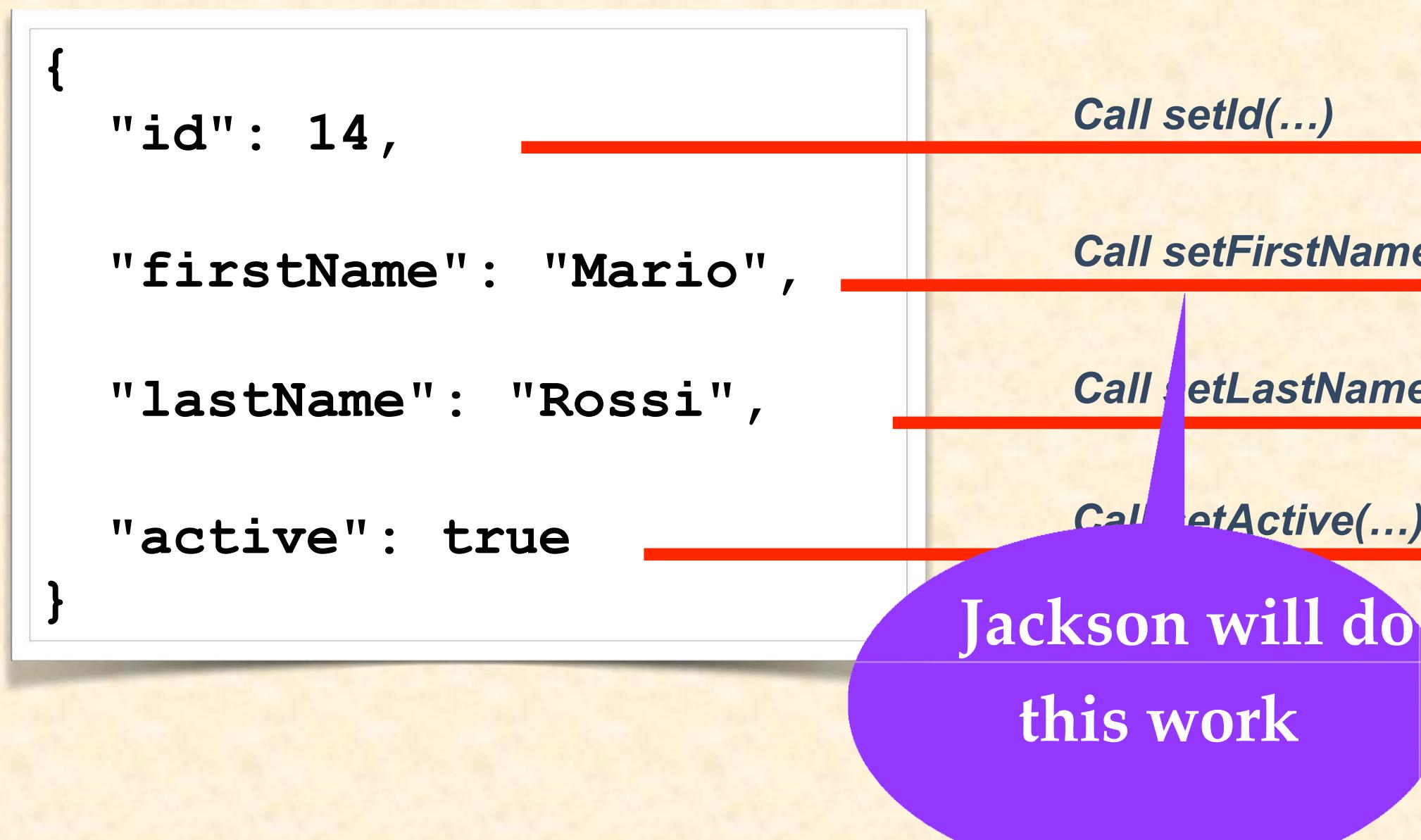
```
{  
  "id": 14,  
  
  "firstName": "Mario",  
  
  "lastName": "Rossi",  
  
  "active": true  
}
```



JSON to Java POJO

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

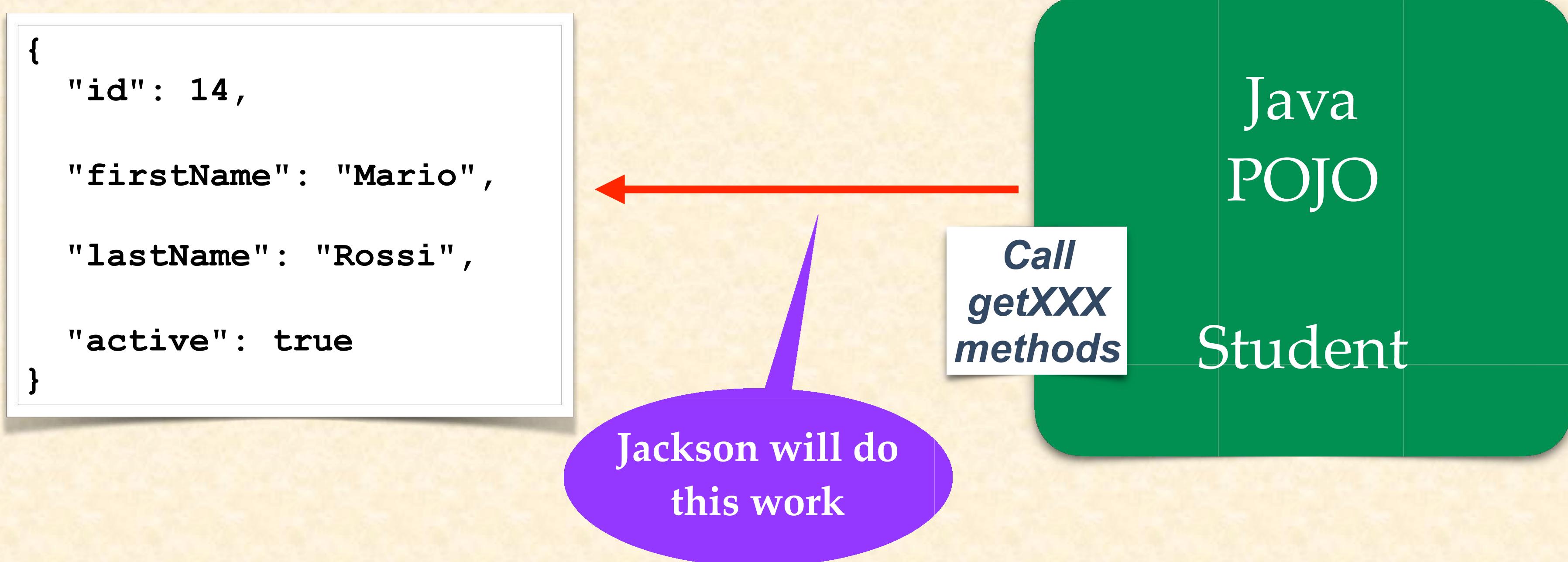
- Convert JSON to Java POJO ... call setter methods on POJO



```
public class Student {  
  
    private int id;  
    private String firstName;  
    private String lastName;  
    private boolean active;  
  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
  
    public void setActive(boolean active) {  
        this.active = active;  
    }  
  
    // getter methods  
}
```

Java POJO to JSON

- Now, let's go the other direction
- Convert Java POJO to JSON ... call getter methods on POJO



Spring and Jackson Support

- When building Spring REST applications
- Spring will automatically handle Jackson Integration
- JSON data being passed to REST controller is converted to POJO
- Java object being returned from REST controller is converted to JSON

Happens
automatically
behind the scenes

Spring REST Service - Students

Create a New Service

- Return a list of students

GET

/api/students

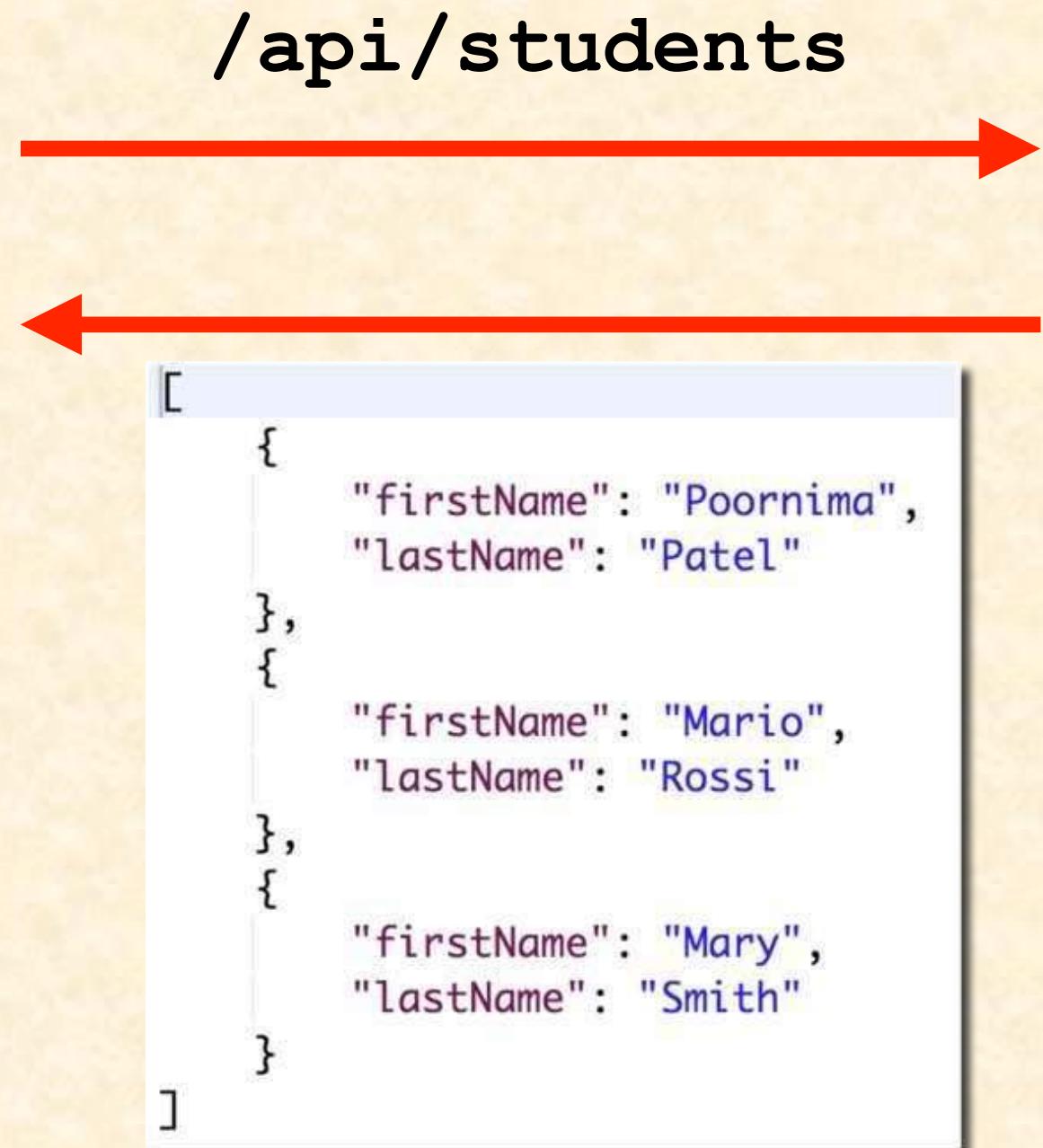
Returns a list of students

Spring REST Service

We will write
this code



Web Browser
Or
Postman



Convert Java POJO to JSON

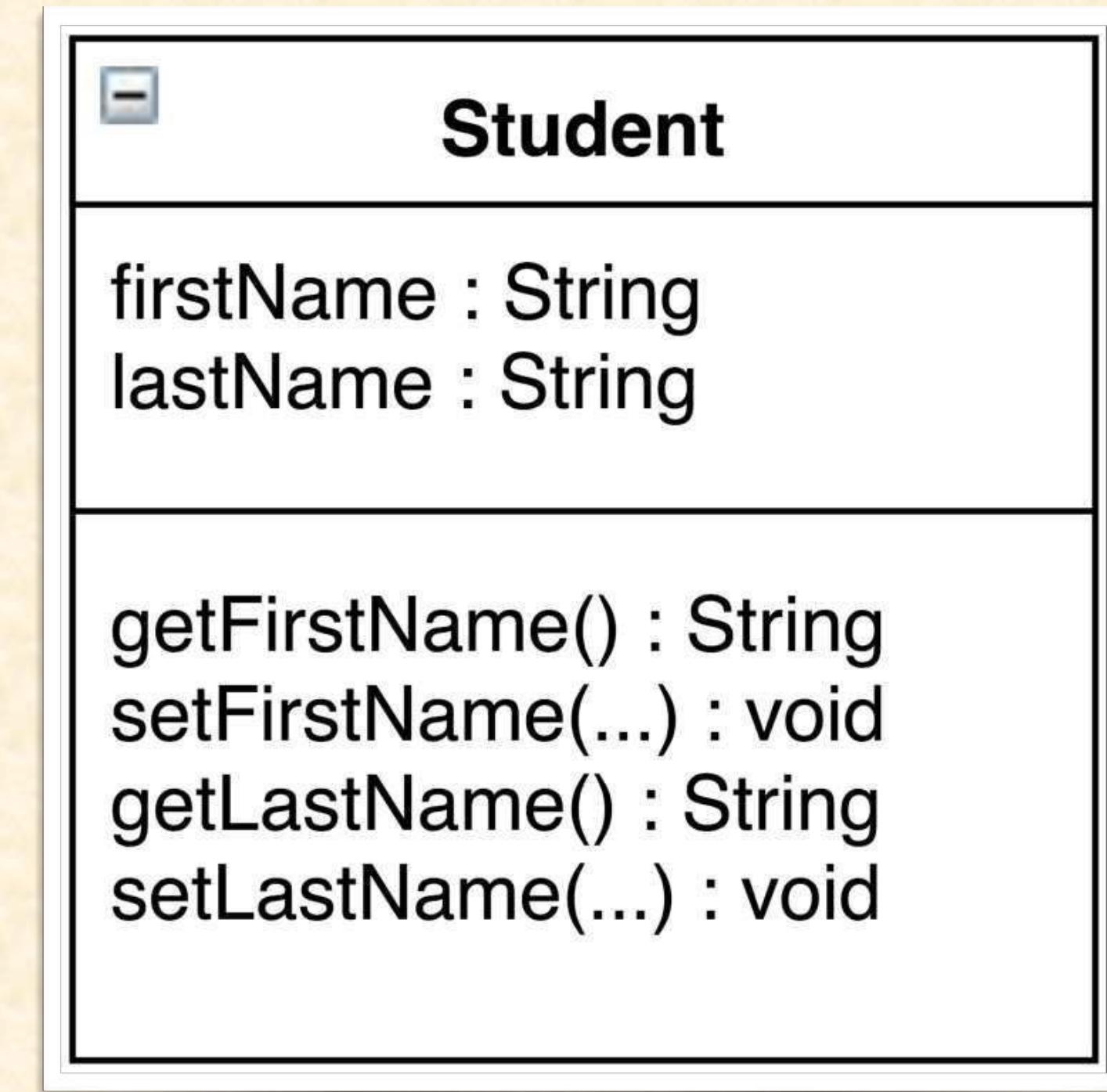
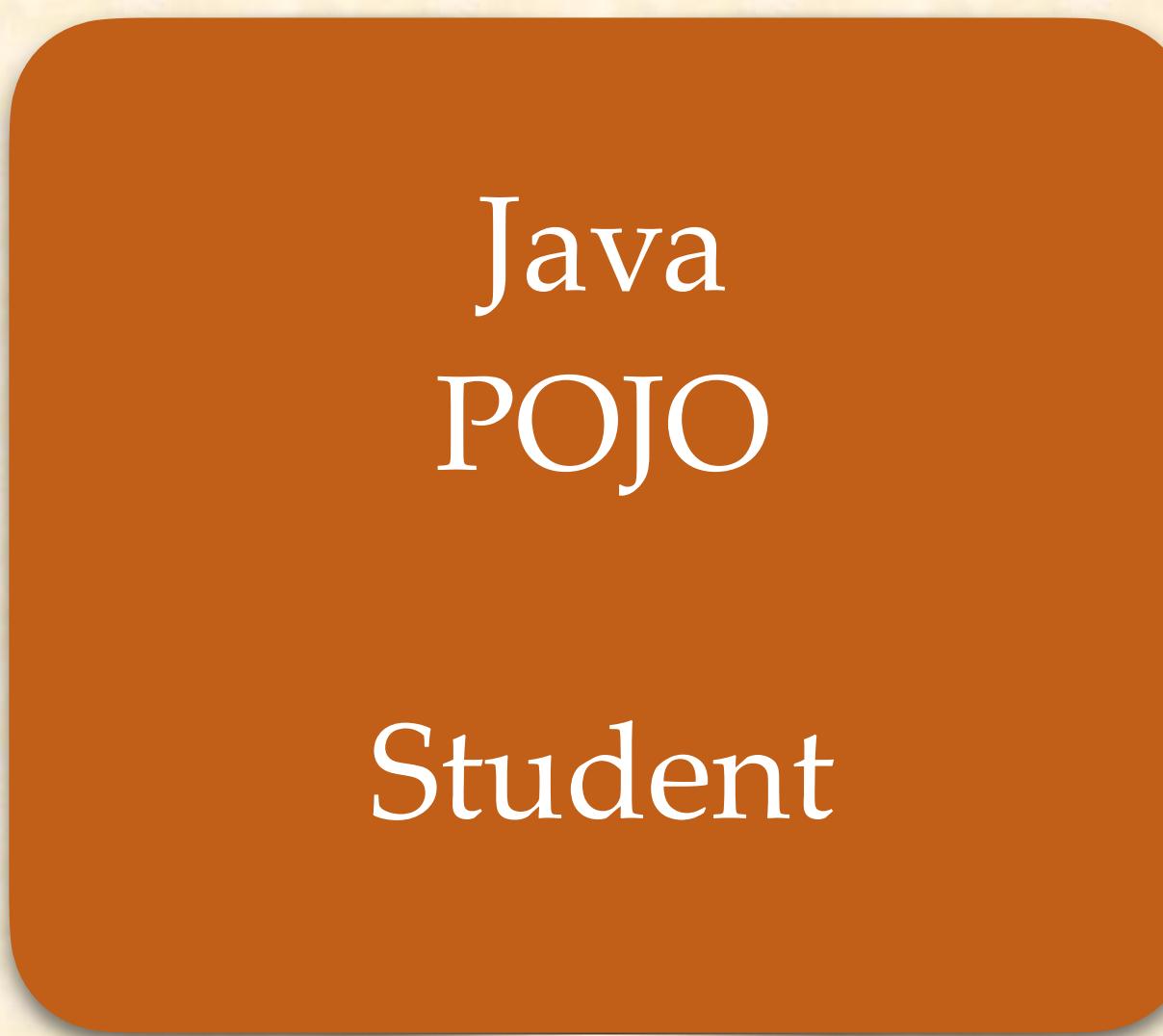
- Our REST Service will return **List<Student>**
- Need to convert **List<Student>** to JSON
- **Jackson** can help us out with this ...

Spring Boot and Jackson Support

Happens
automatically
behind the scenes

- Spring Boot will automatically handle **Jackson** integration
- JSON data being passed to REST controller is converted to Java POJO
- Java POJO being returned from REST controller is converted to JSON

Student POJO (class)

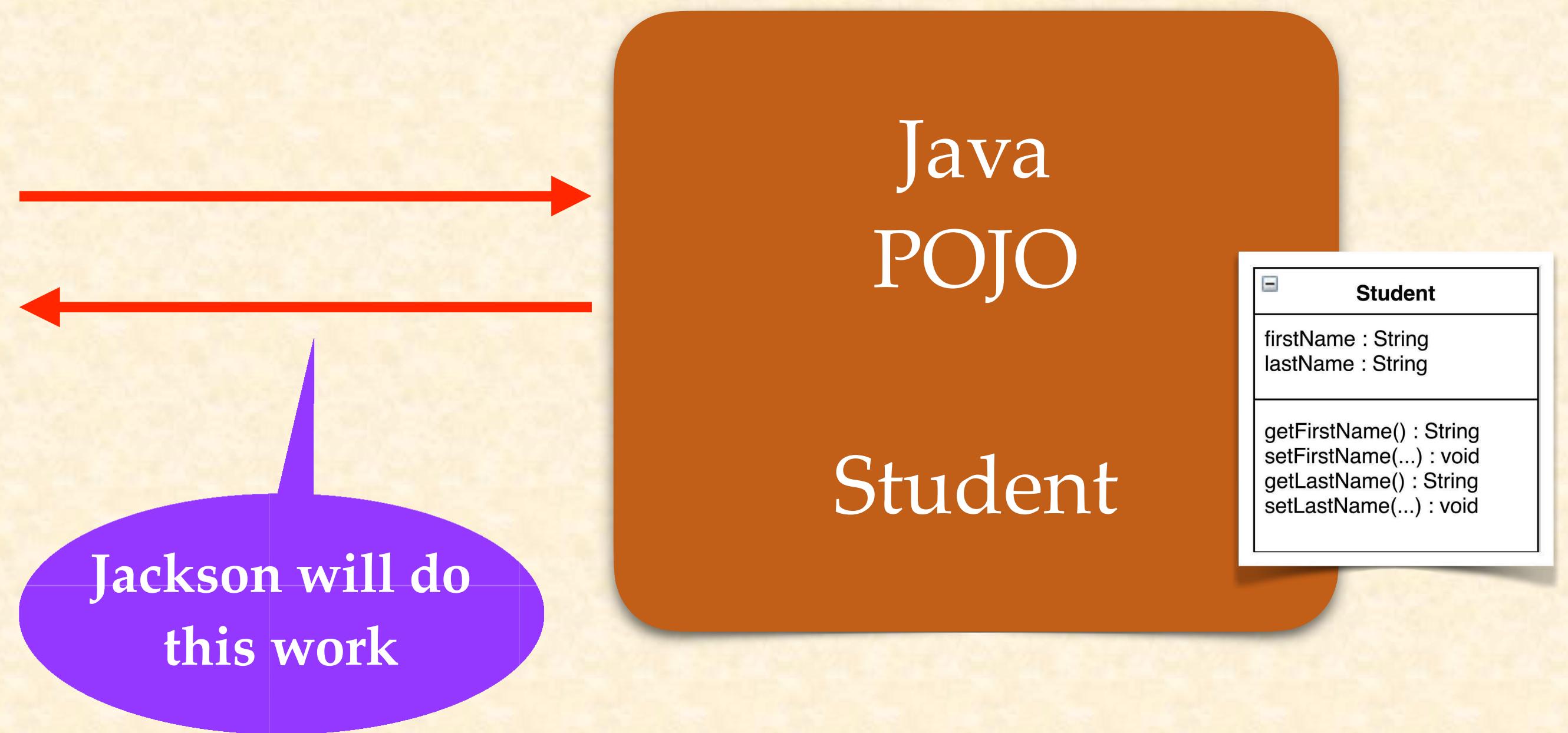


Jackson Data Binding

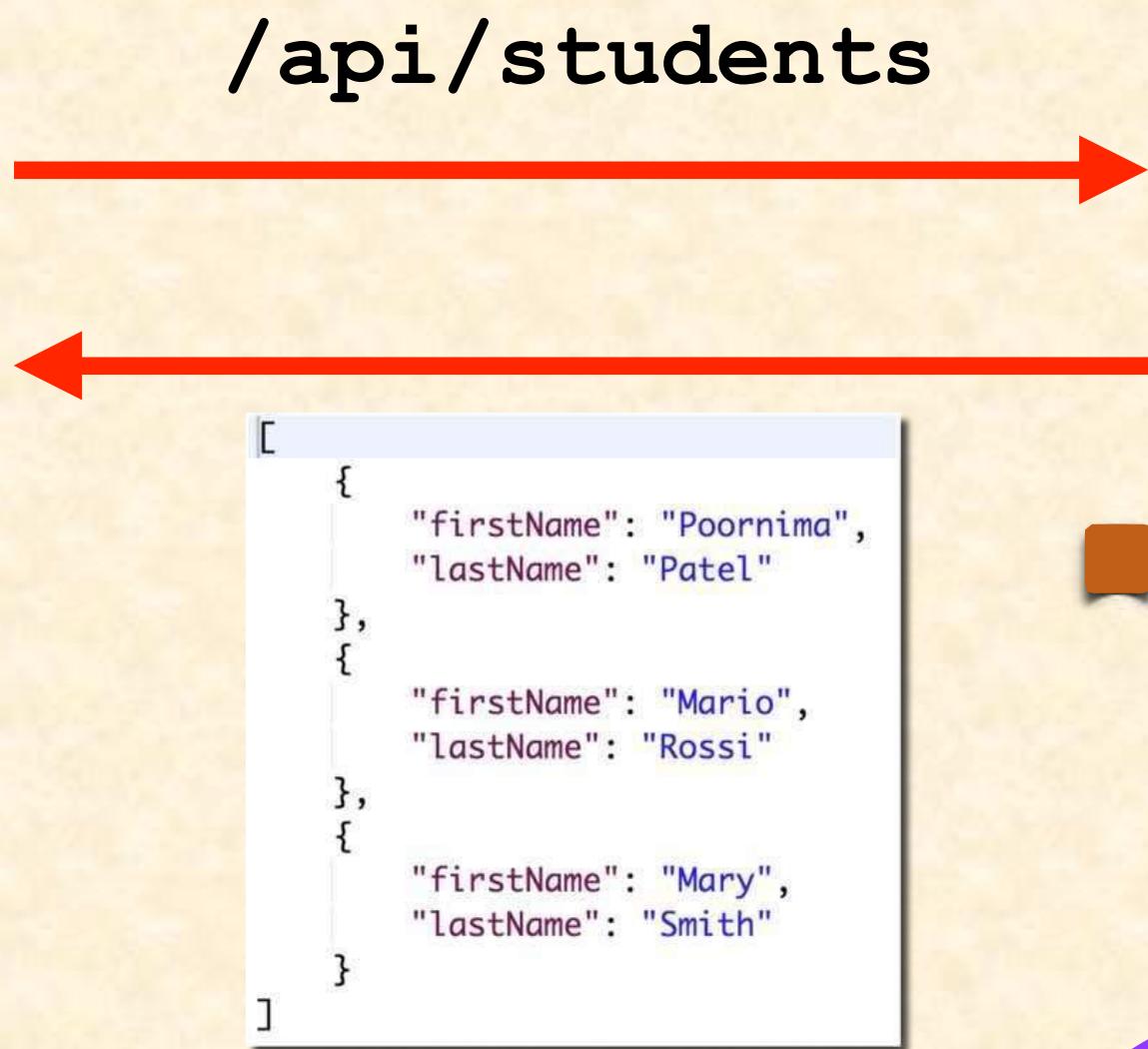
Remember

- Jackson will call appropriate getter/setter method

```
{  
    "id": 14,  
  
    "firstName": "Mario",  
  
    "lastName": "Rossi",  
  
    "active": true  
}
```



Spring REST Service



REST
Service

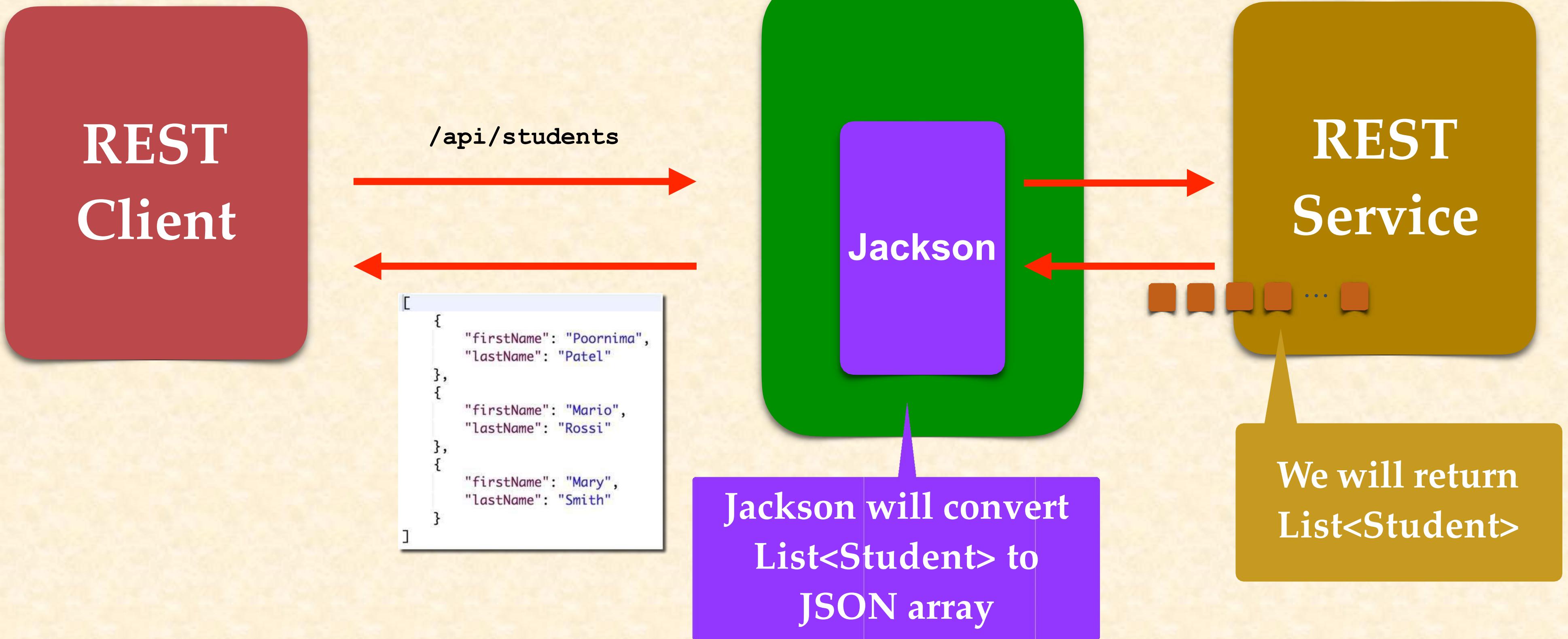


List<Student>

Jackson will
convert to
JSON array

We will write
this code

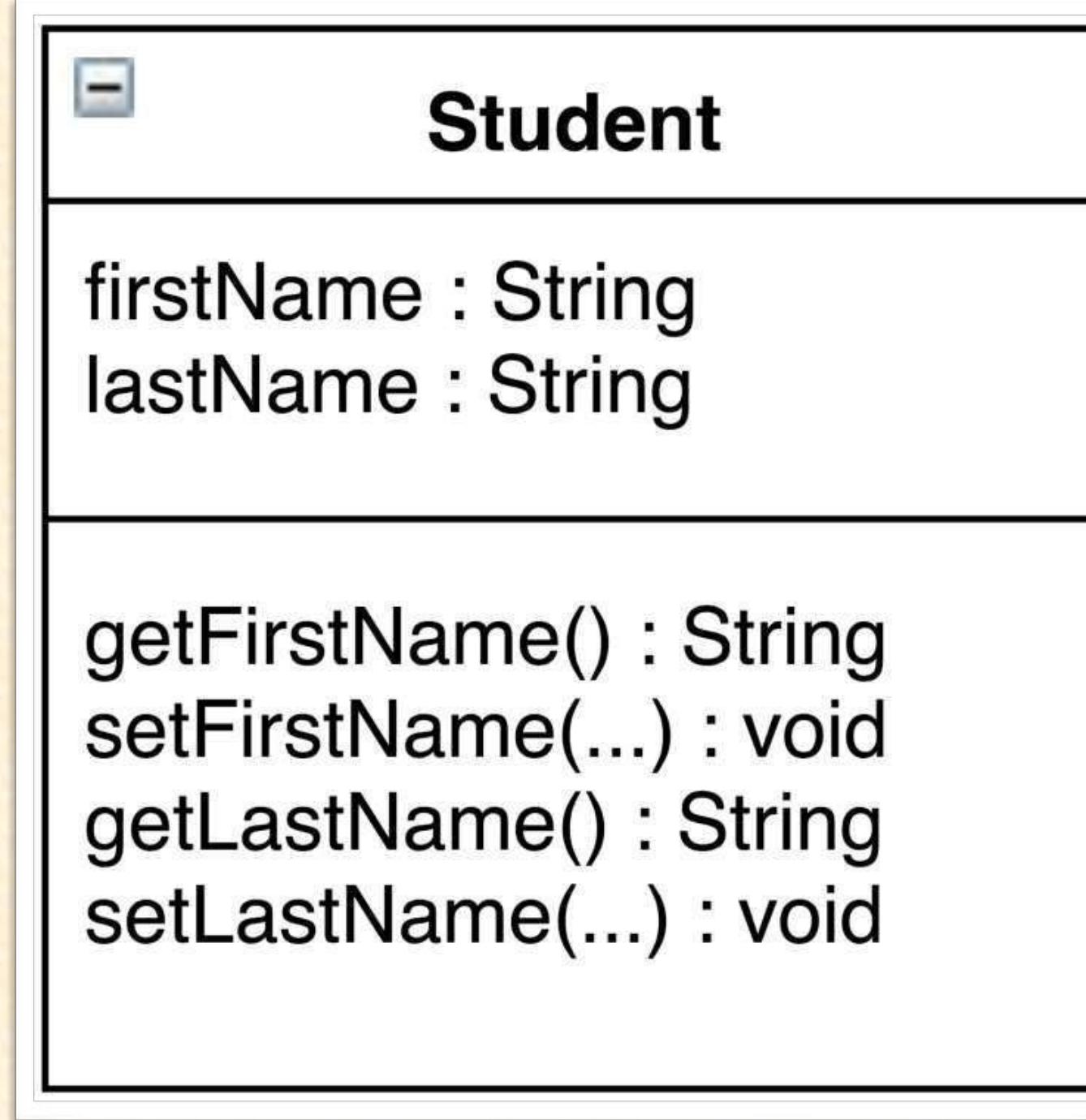
Behind the scenes



Development Process

1. Create Java POJO class for Student
2. Create Spring REST Service using **@RestController**

Step 1: Create Java POJO class for Student



File: Student.java

```
public class Student {  
  
    private String firstName;  
    private String lastName;  
  
    public Student() {  
    }  
  
    public Student(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
}
```

Fields
Constructors
Getter/Setters

Step 2: Create @RestController

File: StudentRestController.java

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

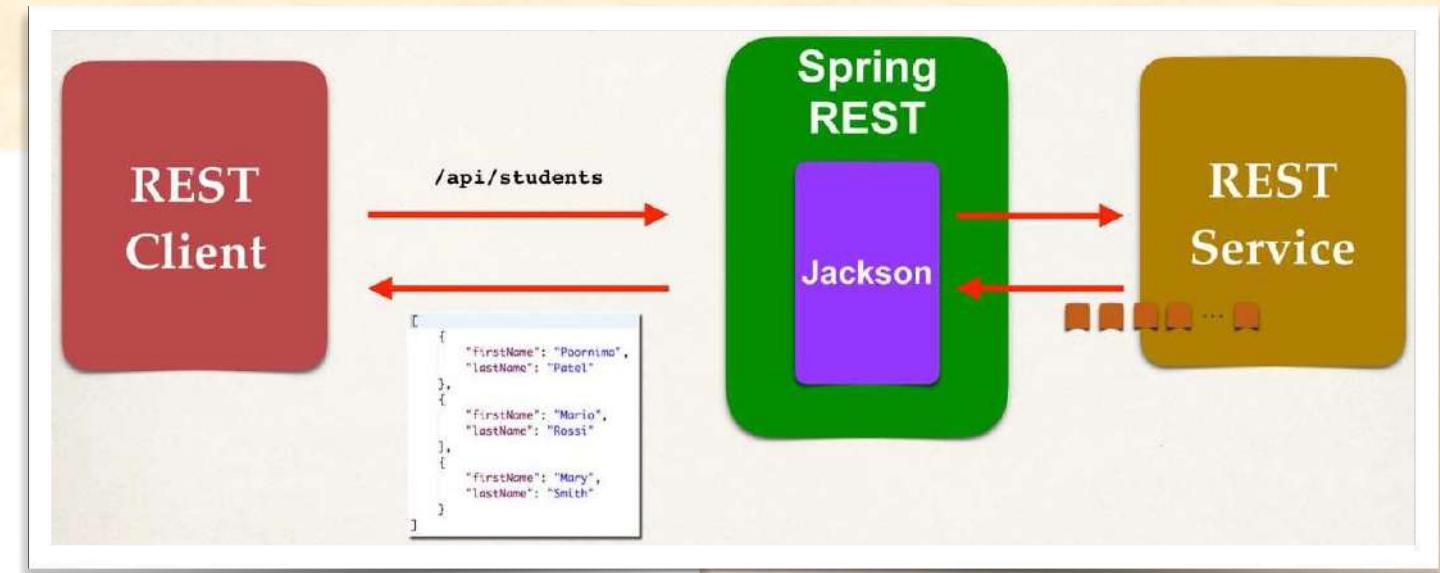
    // define endpoint for "/students" - return list of students

    @GetMapping("/students")
    public List<Student> getStudents() {

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

        theStudents.add(new Student("Poornima", "Patel"));
        theStudents.add(new Student("Mario", "Rossi"));
        theStudents.add(new Student("Mary", "Smith"));

        return theStudents;
    }
}
```



We'll hard code
for now ...
can add DB later ...

Jackson will convert
List<Student> to
JSON array

Spring Boot (REST API, MVC and Microservices)

REST API - Path Variables

Jackson Project

Path Variables

- Retrieve a single student by id

GET

/api/students/{**studentId**}

Retrieve a single student

/api/students/**0**

/api/students/**1**

/api/students/**2**



Known as a
"path variable"

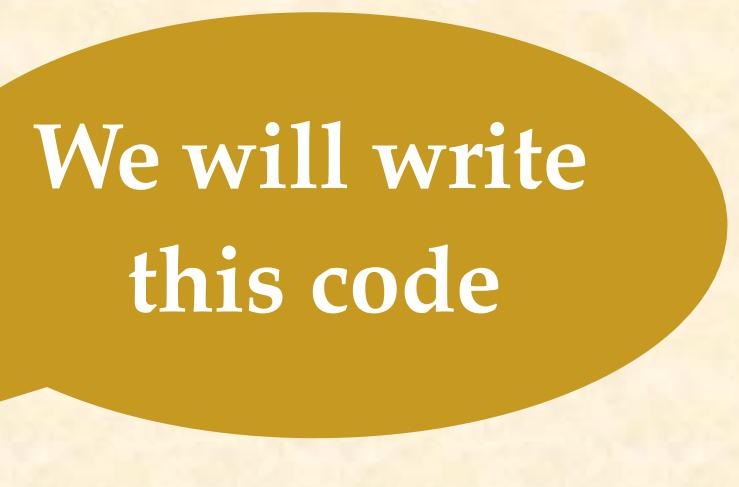
Spring REST Service



/api/students/{studentId}



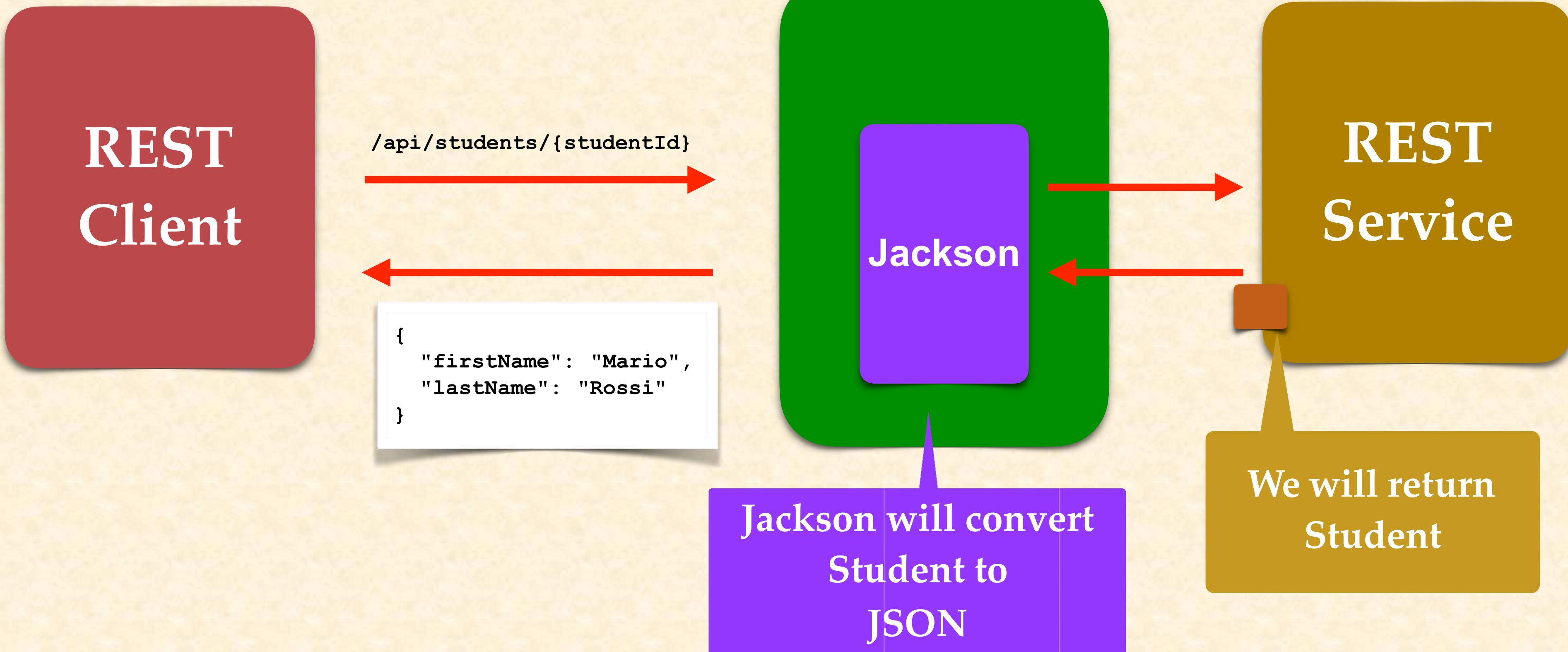
```
{  
    "firstName": "Mario",  
    "lastName": "Rossi"  
}
```



Student

Jackson will
convert to
JSON

Behind the scenes



Development Process

1. Add request mapping to Spring REST Service
 - Bind path variable to method parameter using `@PathVariable`

`@PathVariable` to extract values from the URI path

Step 1: Add Request Mapping

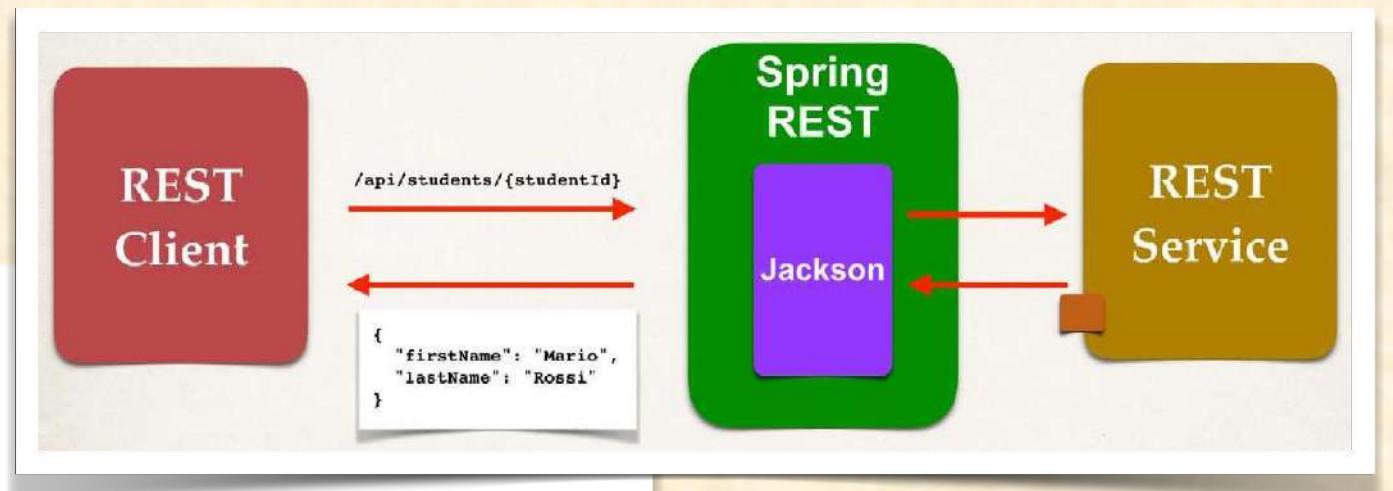
File: StudentRestController.java

```
@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);  
    }  
}
```

Keep it simple,
just index into the list
We'll do fancy DB stuff later

Bind the path variable
(by default, must match)

Jackson will convert
Student to JSON



Spring Boot (REST API, MVC and Microservices)

REST API - Exception Handling

Revisit the problem?

- Bad student id of 9999...

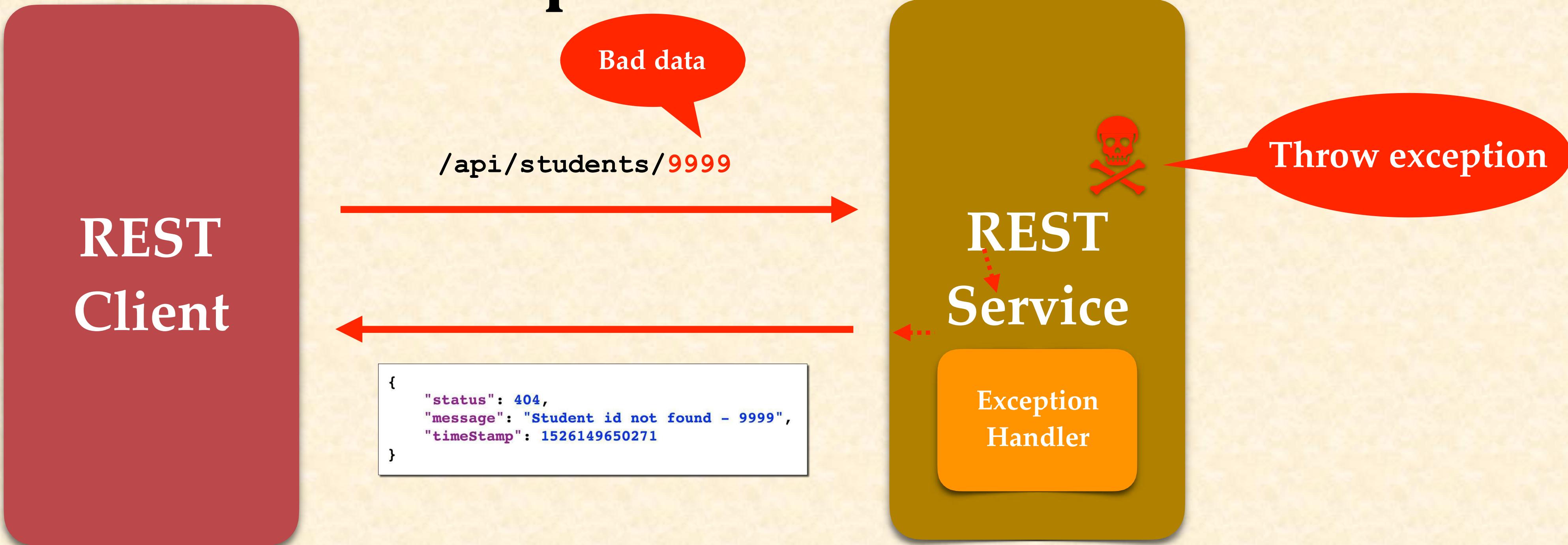


We really want like this ...

- Handle the exception and return error as JSON

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

Spring REST Exception Handling

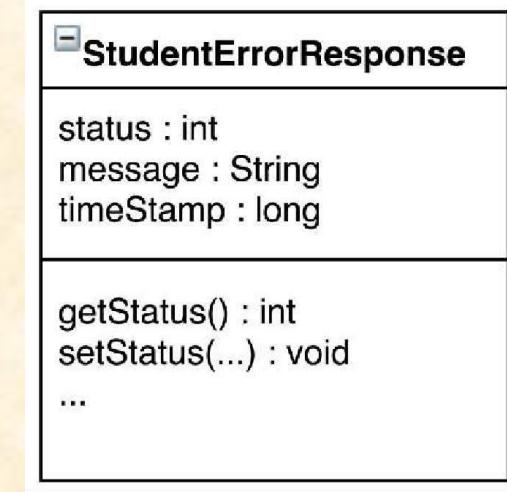


How to?

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



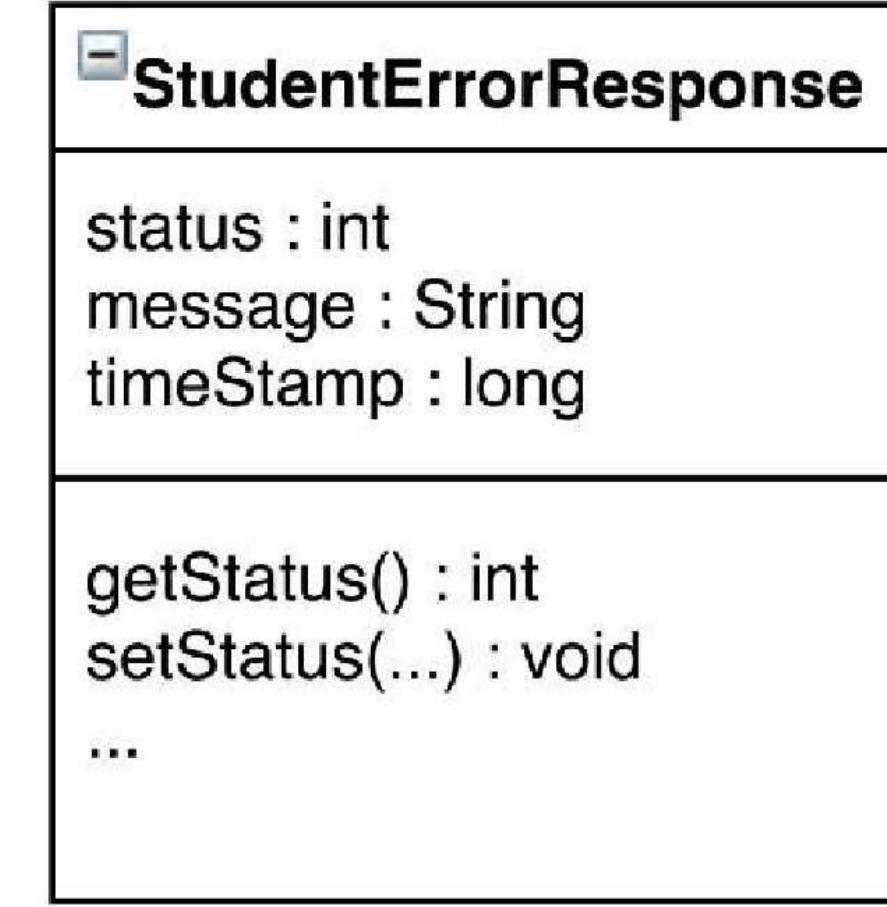
We can define any
custom fields that
you want to track

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

Step 1: Create custom error response class

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 be 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

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);  
    }  
    ...  
}
```

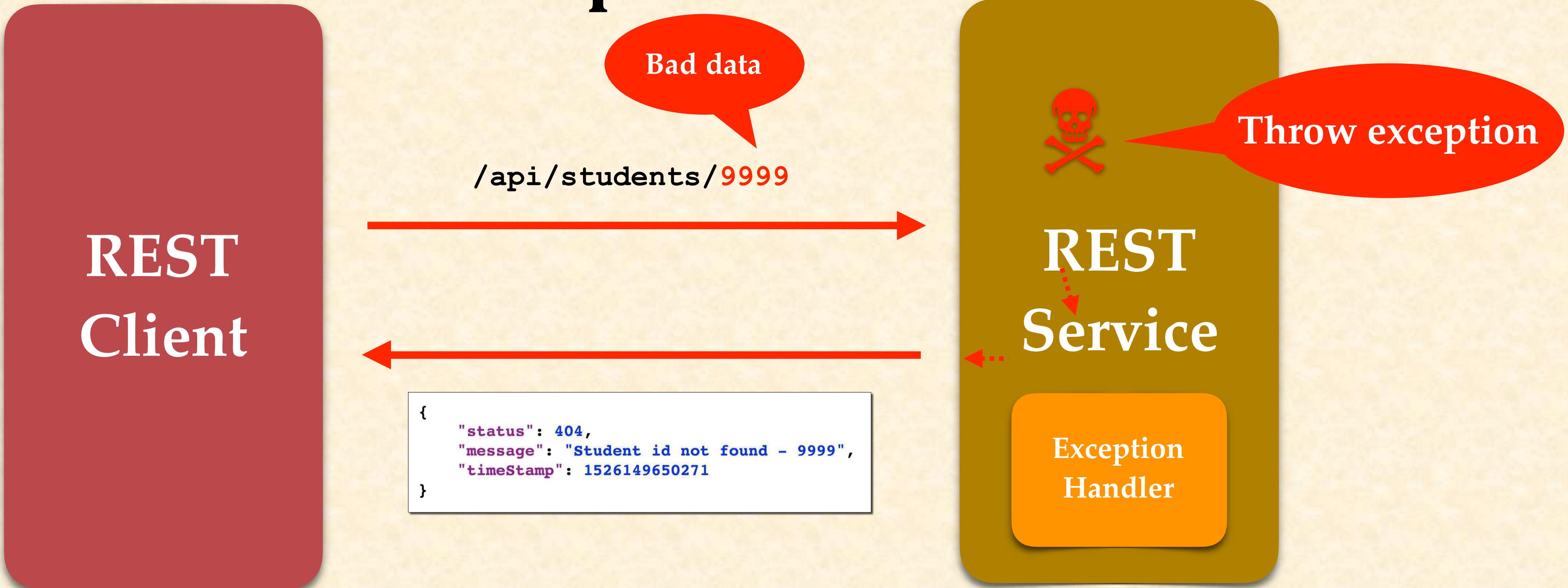
Happy path



Throw exception

Could also
check results
from DB

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

StudentRestController.java
Exception handler method
@RestController

```
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);  
    }  
}
```

Type of the response body

Exception type to handle / catch

Body

Status code

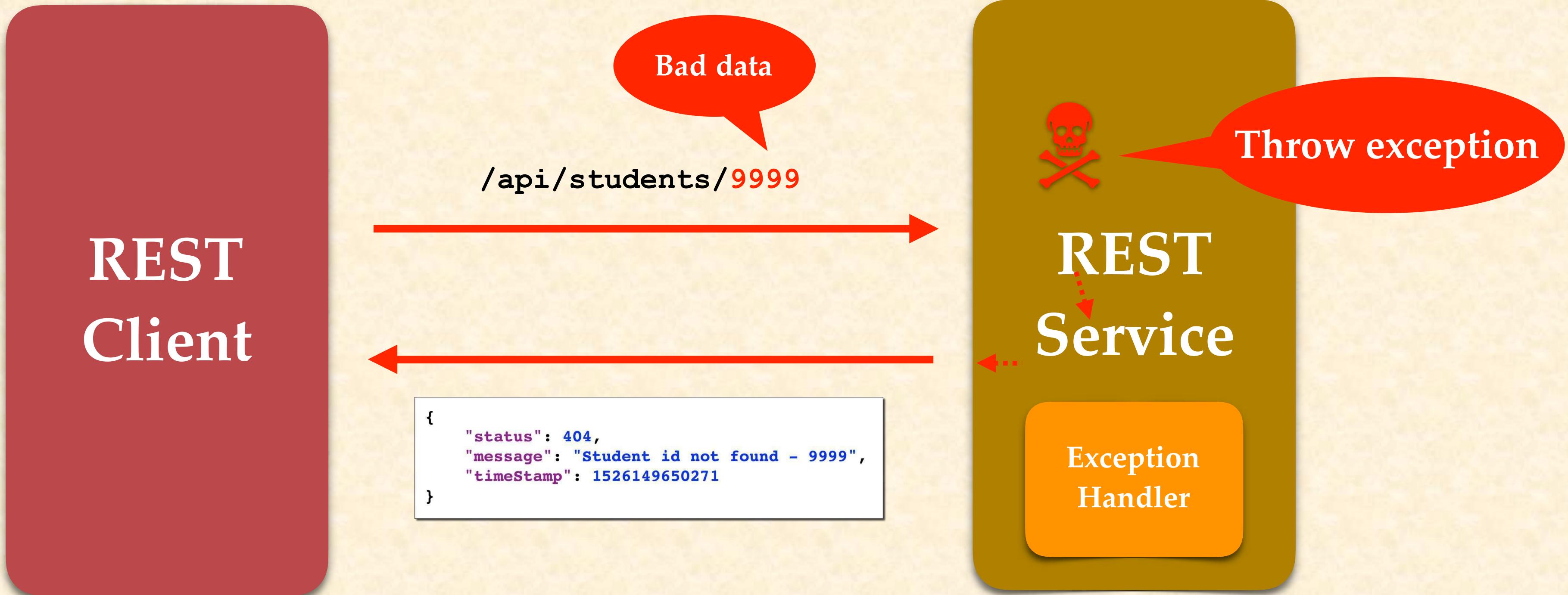
```
{  
    "status": 404,  
    "message": "Student id not found - 9999",  
    "timeStamp": 1526149650271  
}
```

StudentErrorResponse

status : int
message : String
timeStamp : long

getStatus() : int
setStatus(...) : void
...

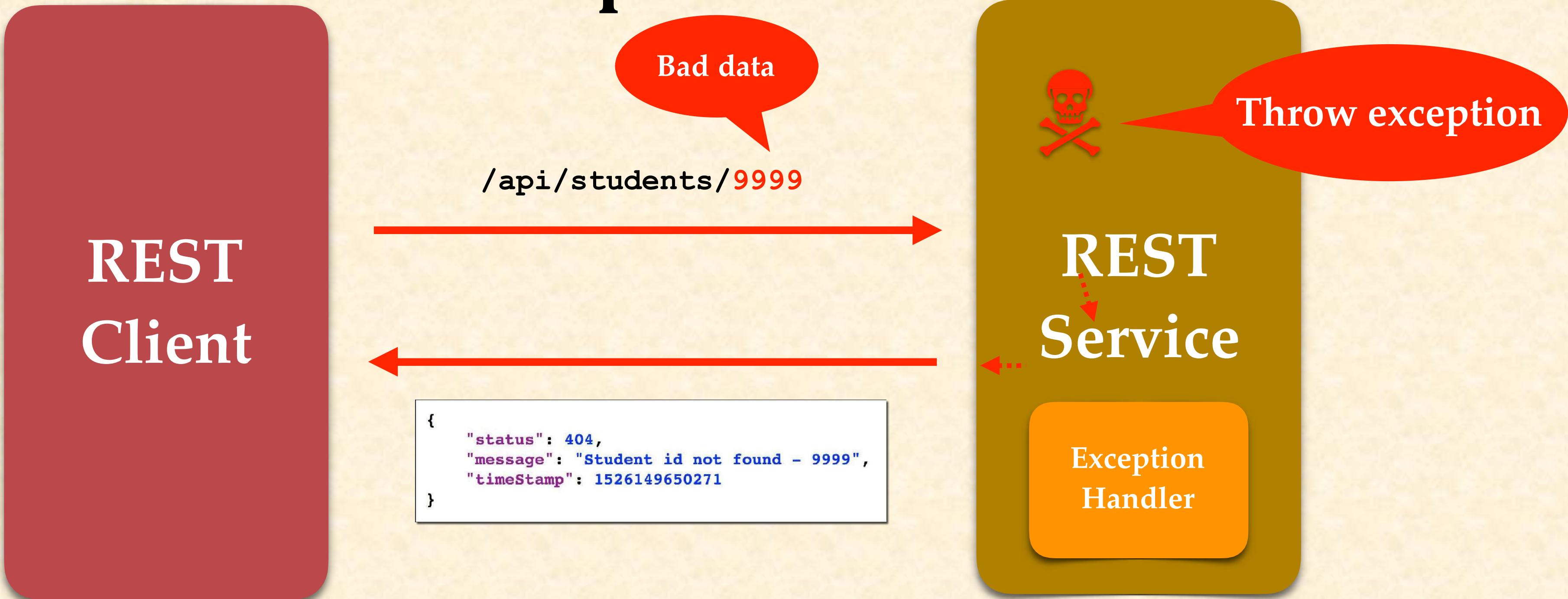
Spring REST Exception Handling



Spring Boot (REST API, MVC and Microservices)

REST API - Global Exception Handling

Spring REST Exception Handling



It works, but ...

- Exception handler code is only for the specific REST controller
- Can't be reused by other controllers
- We need **global** exception handlers
 - Promotes reuse
 - Centralizes exception handling



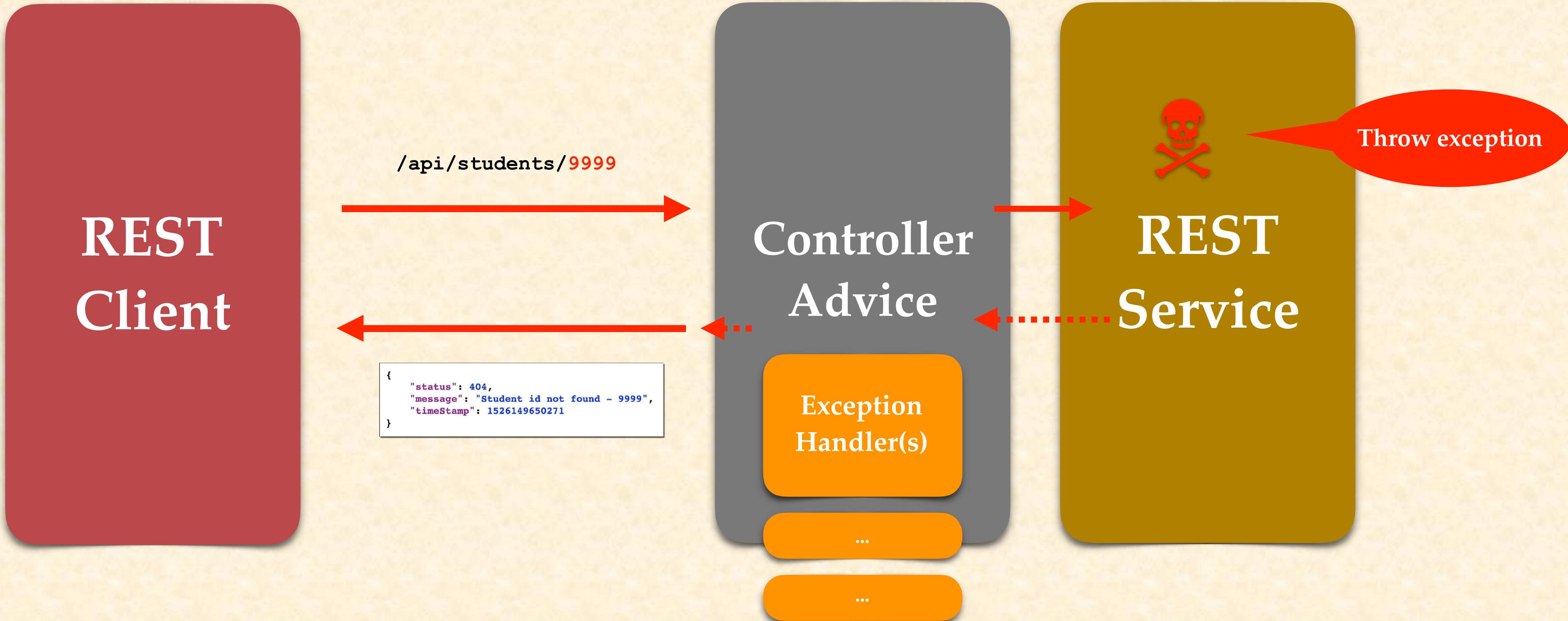
Large projects
will have
multiple controllers

Spring @ControllerAdvice

- @ControllerAdvice is similar to an interceptor / filter
- Pre-process requests to controllers
- Post-process responses to handle exceptions
- Perfect for global exception handling

Real-time
use of
AOP

Spring REST Exception Handling



How to?

1. Create new @ControllerAdvice
2. Refactor REST service ... remove exception handling code
3. Add exception handling code to @ControllerAdvice

Step 1: Create new @ControllerAdvice

StudentRestExceptionHandler.java

```
@ControllerAdvice  
public class StudentRestExceptionHandler {  
  
    ...  
  
}
```

Step 2: Refactor - remove exception handling

StudentRestController.java

```
@RestController
@RequestMapping("/api")
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);
    }
}
```

Remove
this code

Step 3: Add exception handler to @ControllerAdvice

StudentRestExceptionHandler.java

```
@ControllerAdvice
public class StudentRestExceptionHandler {

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

        StudentErrorResponse error = new StudentErrorResponse();

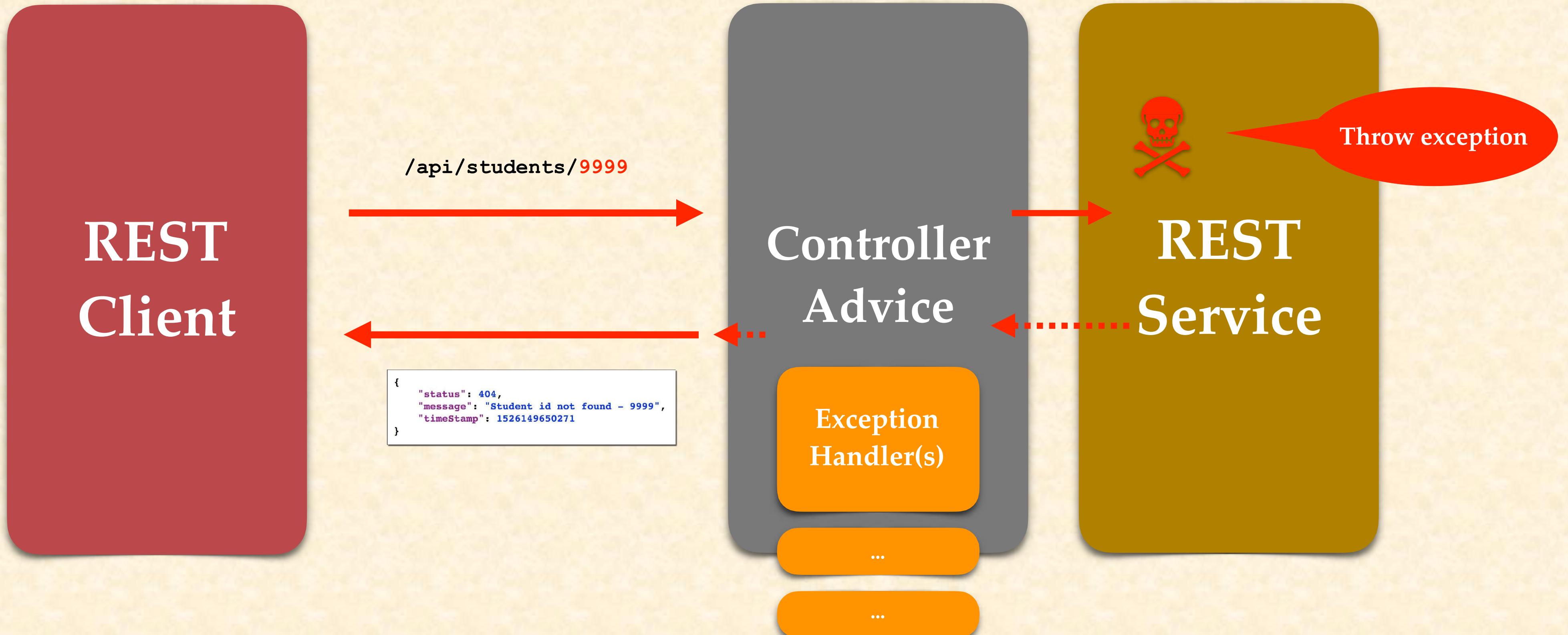
        error.setStatus(HttpStatus.NOT_FOUND.value());
        error.setMessage(exc.getMessage());
        error.setTimestamp(System.currentTimeMillis());

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

}
```

Same code
as before

Spring REST Exception Handling



Spring Boot (REST API, MVC and Microservices)

REST API Design

REST API Design

- For real-time projects, who will use our API?
- Also, how will they use our API?
- Design the API based on requirements

API Design Process

1. Review API requirements
2. Identify main resource / entity
3. Use HTTP methods to assign action on resource

1. API Requirements

Create a REST API for the Employee Directory

REST clients should be able to

- ✓ Get a list of employees
- ✓ Get a single employee by id
- ✓ Add a new employee
- ✓ Update an employee
- ✓ Delete an employee

2. Identify main resource / entity

- To identify main resource / entity, look for the most prominent "noun"
- For this project, it is "employee"
- Convention is to use plural form of resource / entity: **employees**

/api/employees

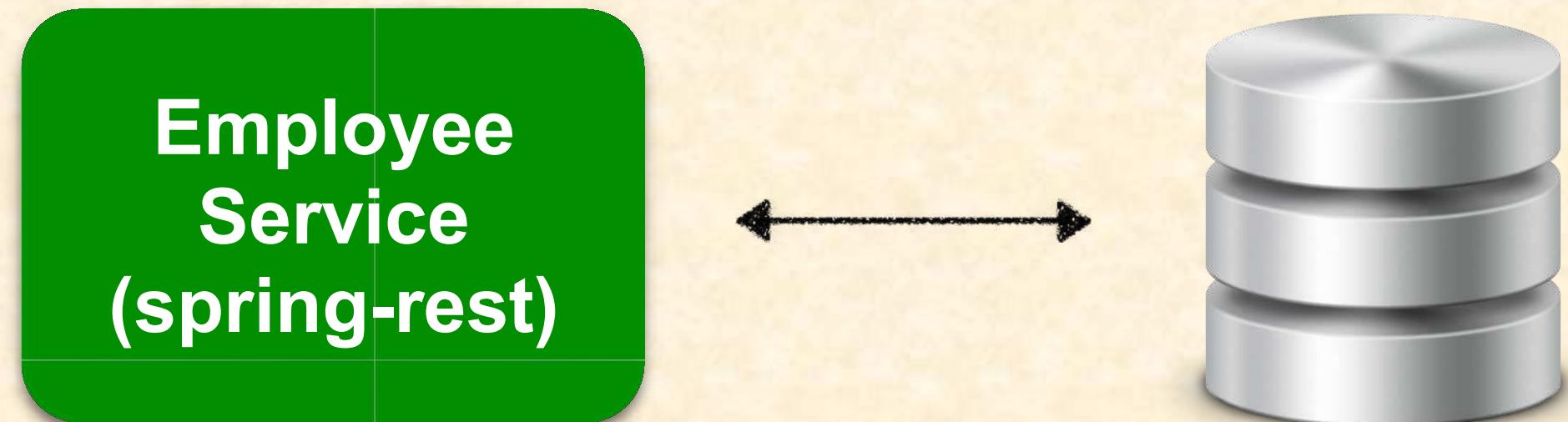
3. Use HTTP methods to assign action on resource

HTTP Method	CRUD Action
POST	<u>Create a new entity</u>
GET	<u>Read a list of entities or single entity</u>
PUT	<u>Update an existing entity</u>
DELETE	<u>Delete an existing entity</u>

Full CRUD

Employee Real-Time Project

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



Anti-Patterns

- DO NOT DO THIS ... these are REST anti-patterns, bad practice



/api/employeesList
/api/deleteEmployee
/api/addEmployee
/api/updateEmployee

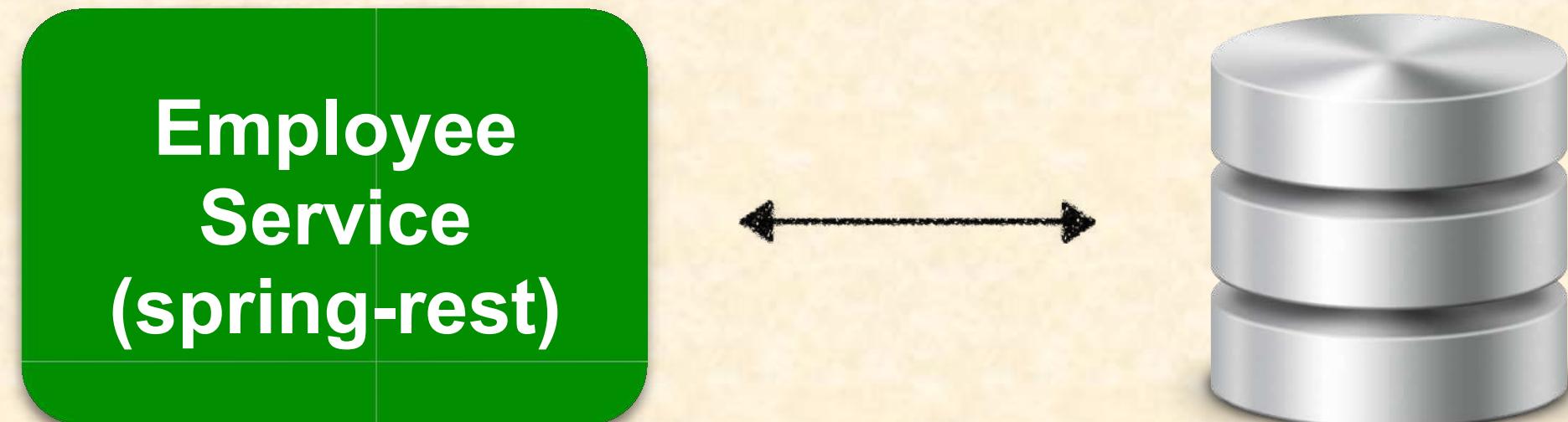
Don't include actions in the endpoint



Instead, use
HTTP methods
to assign actions

Employee Real-Time Project

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



PayPal

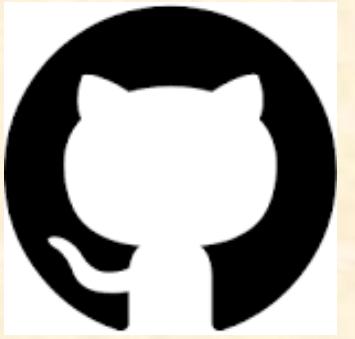


- PayPal Invoicing API
 - <https://developer.paypal.com/docs/api/invoicing/>

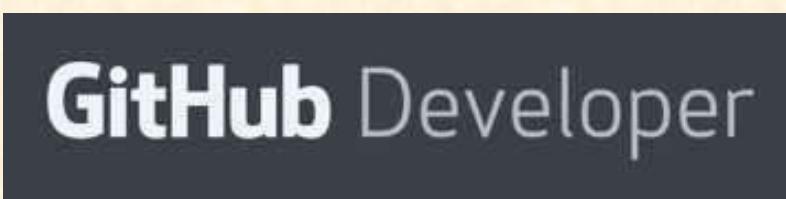
The screenshot shows the PayPal Developer API documentation for the Invoicing API. The top navigation bar includes links for 'Developer', 'Docs', 'APIs', and 'Support'. Below the navigation, there are five main sections: 'Create draft invoice' (POST /v1/invoicing/invoices), 'Update invoice' (PUT /v1/invoicing/invoices/{invoice_id}), 'Delete draft invoice' (DELETE /v1/invoicing/invoices/{invoice_id}), 'Show invoice details' (GET /v1/invoicing/invoices/{invoice_id}), and 'List invoices' (GET /v1/invoicing/invoices). Each section contains a colored button indicating the HTTP method and the corresponding endpoint URL.

- Create draft invoice**
POST /v1/invoicing/invoices
- Update invoice**
PUT /v1/invoicing/invoices/{invoice_id}
- Delete draft invoice**
DELETE /v1/invoicing/invoices/{invoice_id}
- Show invoice details**
GET /v1/invoicing/invoices/{invoice_id}
- List invoices**
GET /v1/invoicing/invoices

GitHub



- GitHub Repositories API
 - <https://developer.github.com/v3/repos/#repositories>



Create a new repository

POST /user/repos

Delete a repository

DELETE /repos/:owner/:repo

List your repositories

GET /user/repos

Get a repository

GET /repos/:owner/:repo

Spring Boot (REST API, MVC and Microservices)

Lombok



Lombok

What is Lombok framework?

- Lombok is a bean management framework
- ✓ It lets us write beans (POJO, classes) with our standard bean methods like getters, setters, `toString`, `equals`, `hashCode`, and more with EASE!
- ✓ Simply, the Lombok framework will generate any and all boilerplate bean code - our beans require!

Lombok Annotations

- There are a number of annotations available through Lombok
 - @Getter and @Setter are amazingly useful!
 - @ToString if you like that sort of thing!
 - @EqualsAndHashCode to save a TON of time
 - @NoArgsConstructor to create a constructor with no params
 - @AllArgsConstructor to create a constructor with all params
 - @RequiredArgsConstructor to create a constructor with just params we need

Lombok Annotations

And one annotation to rule them all!

- **@Data** to shortcut an `@ToString`, an `@EqualsAndHashCode`, `@RequiredArgsConstructor`, an `@Getter`, and an `@Setter`, all in one shot!
- For most things we will be using
`@Data`,
`@NoArgsConstructor`, and
`@AllArgsConstructor` in some combination

Lombok

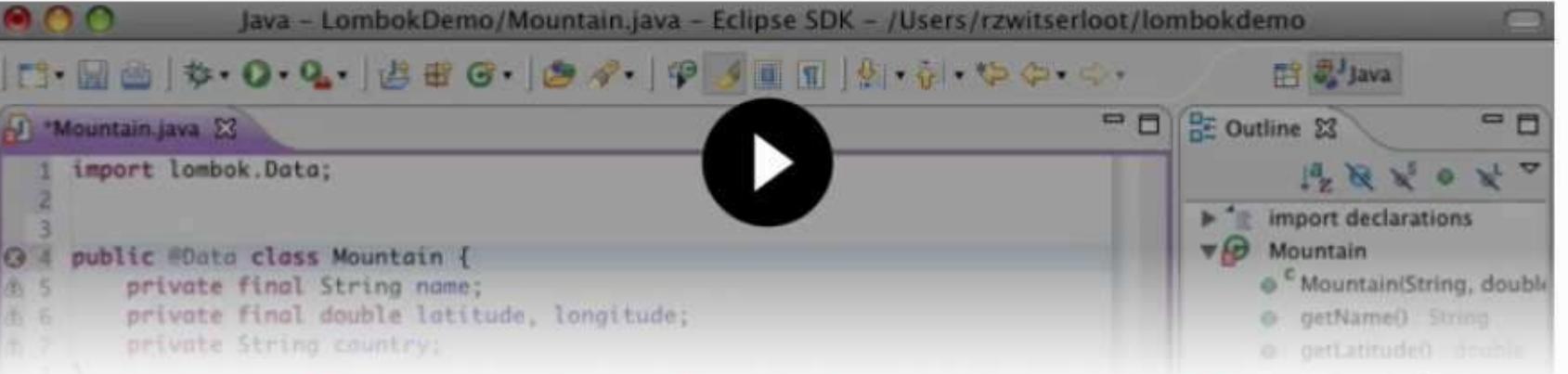
```
<dependency>
<groupId>org.projectlombok
</groupId>
<artifactId>Lombok
</artifactId>
<optional>true</optional>
</dependency>
```

projectlombok.org

Project Lombok Features▼ Community▼ Order / Donate Install▼ Download

Project Lombok

Project Lombok is a java library that automatically plugs into your editor and build tools, spicing up your java. Never write another getter or equals method again, with one annotation your class has a fully featured builder, Automate your logging variables, and much more.



Show me a text and images based explanation and tutorial instead!

Project Lombok is **powered by:**

Alex Myronenko 

Jens Hartlep 

I want to support Project Lombok too!

Spring Boot (REST API, MVC and Microservices)

Introduction to Hibernate / JPA



Agenda

- What is Hibernate?
- Benefits of Hibernate
- What is JPA?
- Benefits of JPA
- Setting up a database table
- Code Snippets for CRUD using Hibernate/JPA

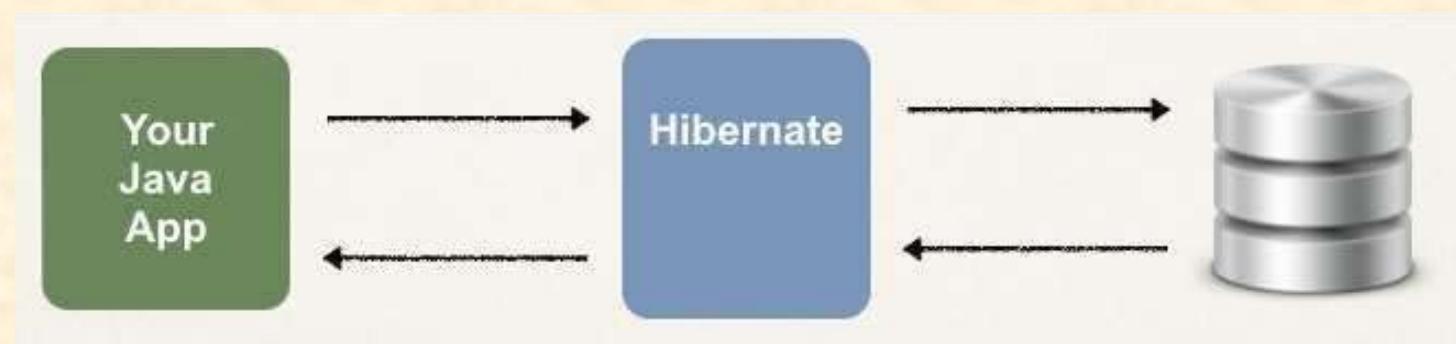
What is Hibernate?

- A framework for persisting / saving Java objects in a database
- www.hibernate.org/orm



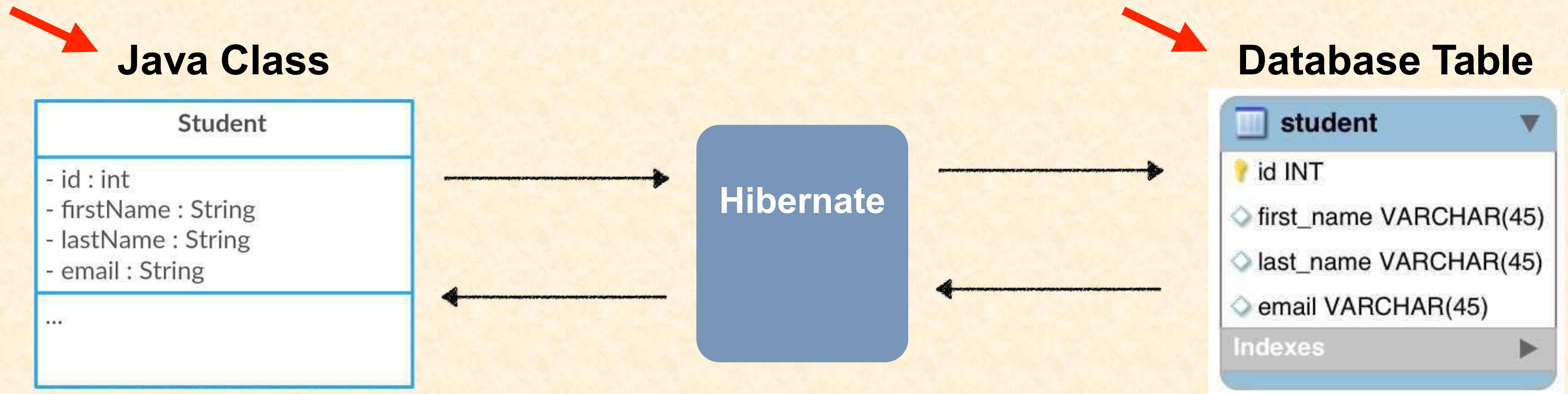
Benefits of Hibernate

- Hibernate handles all of the low-level SQL
- Minimizes the amount of JDBC code we have to develop
- Hibernate provides the Object-to-Relational Mapping (ORM)



Object-To-Relational Mapping (ORM)

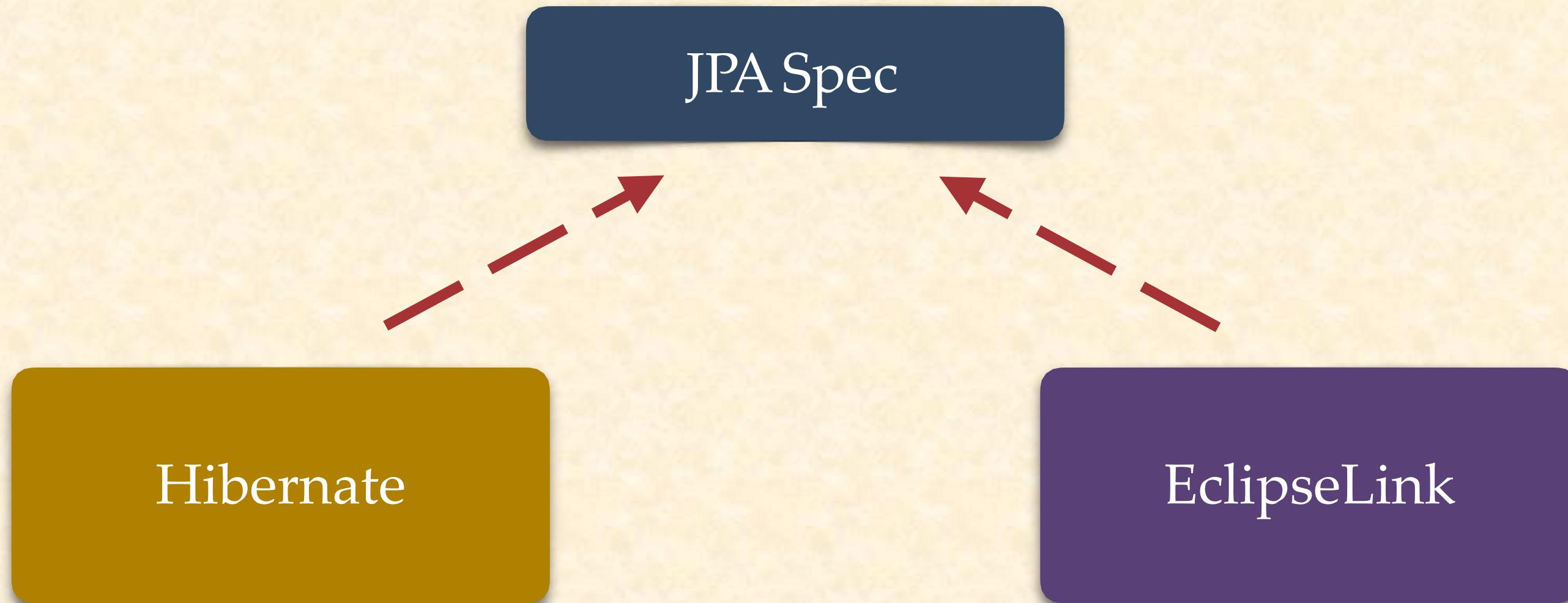
- Mapping between Java class and database table



What is JPA?

- Jakarta Persistence API (JPA) ... *previously known as Java Persistence API*
 - Standard API for Object-to-Relational-Mapping (ORM)
- Its only a specification
 - Defines a set of interfaces
 - Requires an implementation to be usable

JPA - Vendor Implementations



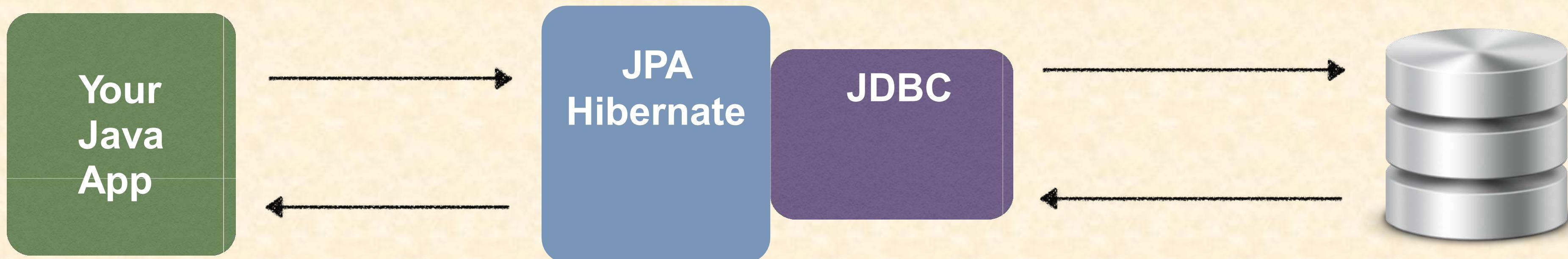
What are the Benefits of JPA?

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code by coding to JPA spec (interfaces)
- switch / migrate vendor implementations
 - For example, if Vendor ABC stops supporting their product
 - You could switch to Vendor XYZ without vendor lock in

How does Hibernate / JPA relate to JDBC?

Hibernate / JPA and JDBC

- Hibernate / JPA uses JDBC for all database communications



MySQL Server

- The MySQL Server is one of the main databases (RDBMS)
 - SQL Compliant DBMS
- Stores data for the database (tables – rows and columns)
- Supports CRUD features on the data

MySQL Workbench

- MySQL Workbench is a client GUI for interacting with the database
- Create database schemas and tables
- Execute SQL queries to retrieve data
- Perform insert, updates and deletes on data
- Handle administrative functions such as creating users
- Others ...

Install the MySQL Server

- Step 1: Install MySQL Database Server
 - <https://dev.mysql.com/downloads/mysql>
- Step 2: Install MySQL Workbench
 - <https://dev.mysql.com/downloads/workbench>

Install MySQL Server

Setup of Database Table

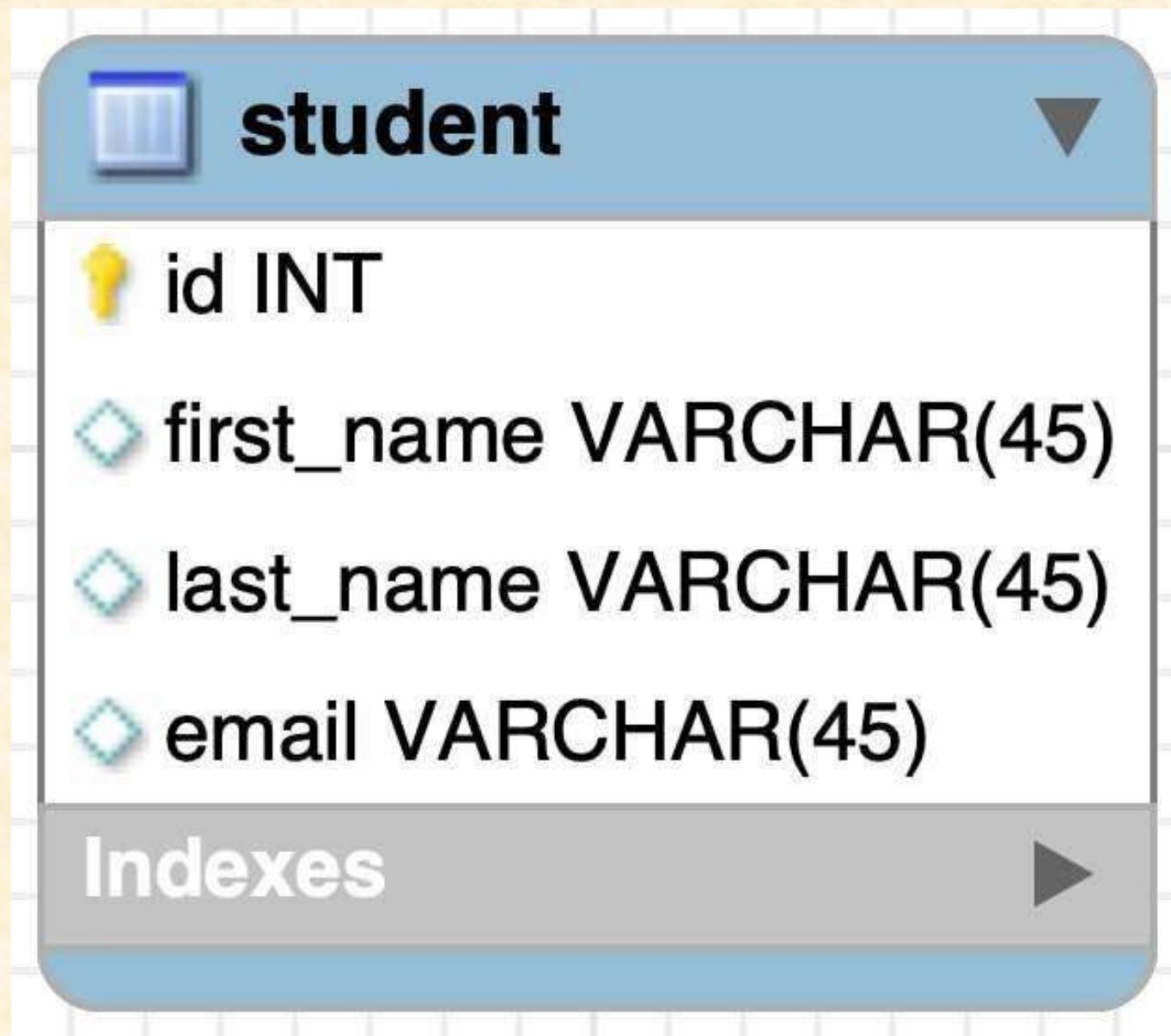
1. Create a new MySQL user or Use the default **root** user

user id: **root**

password: **changeme**

Setup of Database Table

1. Create a new database and a table: **student**



MySQL – student table

```
CREATE TABLE student (
    id int NOT NULL AUTO_INCREMENT,
    first_name varchar(45) DEFAULT NULL,
    last_name varchar(45) DEFAULT NULL,
    email varchar(45) DEFAULT NULL,
    PRIMARY KEY (id)
)
```

Automatic Data Source Configuration

- In Spring Boot, Hibernate is the default implementation of JPA
- **EntityManager** is the main component for creating queries etc ...
- **EntityManager** is from Jakarta Persistence API (JPA)

Automatic Data Source Configuration

- Based on configs, Spring Boot will automatically create the beans:
 - **DataSource**, **EntityManager**, ...
- We can then inject these into our app, for example in our DAOs

Spring Boot (REST API, MVC and Microservices)

Setting up Hibernate / JPA Project

Setting up Project with Spring Initializr

- At Spring Initializr website, start.spring.io
- Add dependencies
 - MySQL Driver: **mysql-connector-j**
 - Spring Data JPA: **spring-boot-starter-data-jpa**
 - **Spring web, Spring Dev tools and Lombok as usual**

Spring Boot - Auto configuration

- Spring Boot will automatically configure our data source for us
- Based on entries from Maven **pom** file
 - JDBC Driver: **mysql-connector-j**
 - Spring Data (ORM): **spring-boot-starter-data-jpa**
 - DB connection info from **application.properties**

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/studentdb  
spring.datasource.username=root  
spring.datasource.password=changeme
```

No need to give JDBC driver class name
Spring Boot will automatically detect it based on URL

JPA Development Process – How to?

1. Annotate Java Class to map to database table
2. Develop Java Code to perform database operations

Terminology

Entity Class

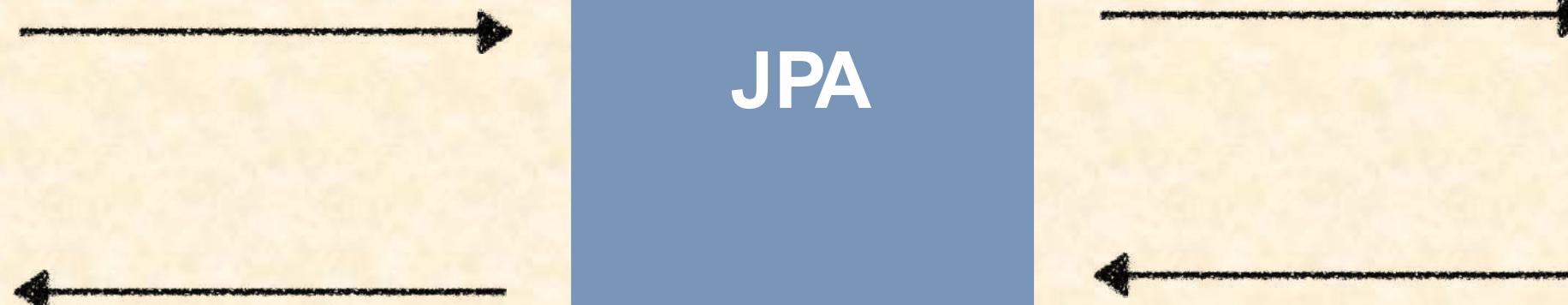
Java class that is mapped to a database table

Object-to-Relational Mapping (ORM)

Java Class

Student	
- id : int	
- firstName : String	
- lastName : String	
- email : String	
...	

JPA



Database Table

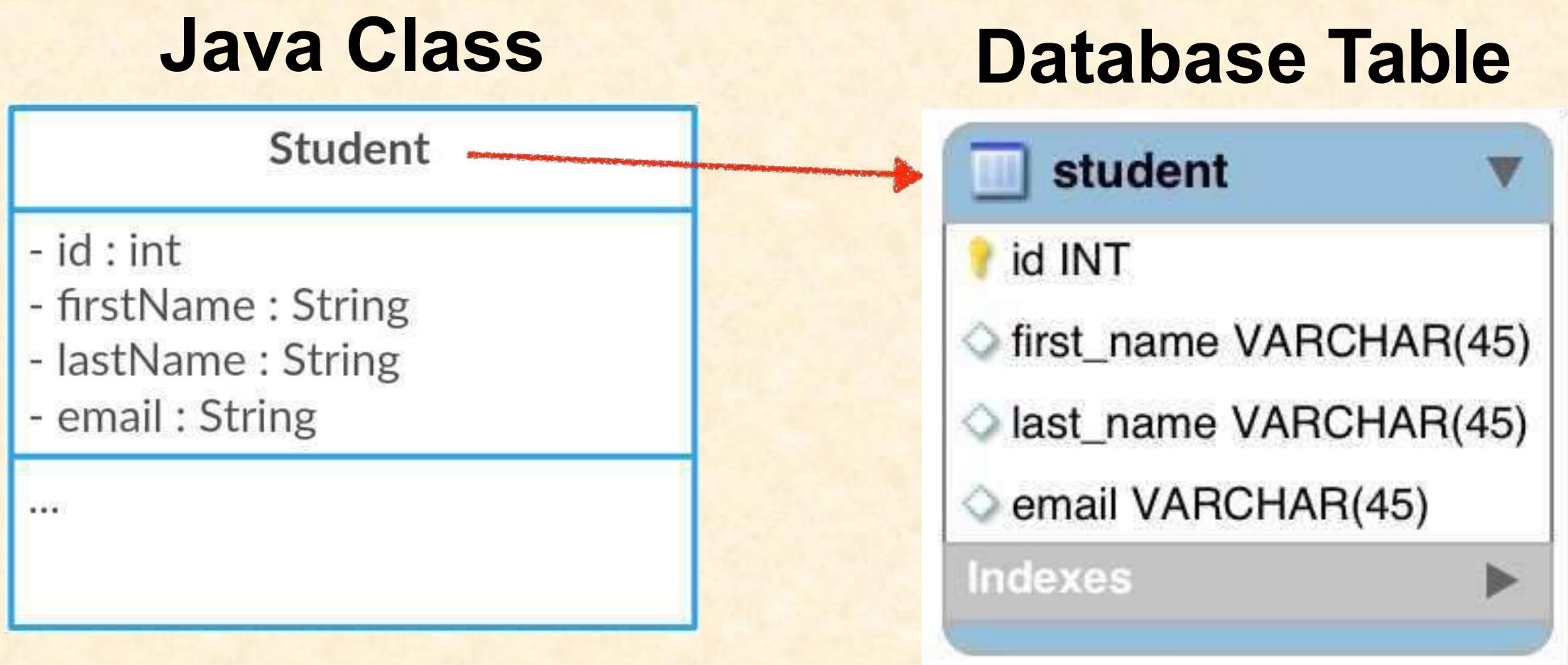
student	
!	id INT
◆	first_name VARCHAR(45)
◆	last_name VARCHAR(45)
◆	email VARCHAR(45)
Indexes	

Entity Class

- At a minimum, the Entity class
 - Must be annotated with `@Entity`
 - Must have a public or protected no-argument constructor*
 - The class can have other constructors

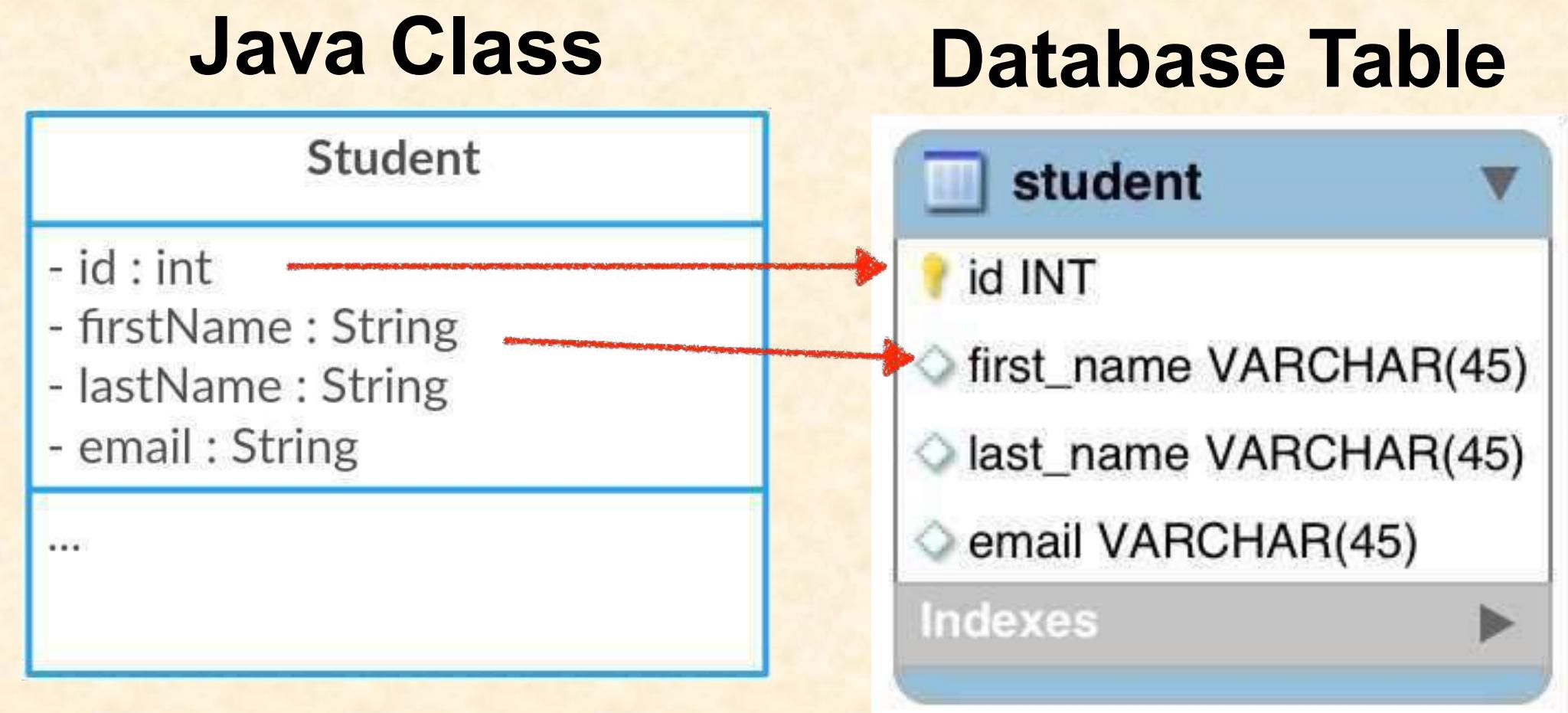
Step 1: Map class to database table

```
@Entity  
@Table(name="student")  
public class Student {  
  
    ...  
  
}
```



Step 2: Map fields to database columns

```
@Entity  
@Table(name="student")  
public class Student {  
  
    @Id  
    @Column(name="id")  
    private int id;  
  
    @Column(name="first_name")  
    private String firstName;  
    ...  
}
```



@Column - Optional

- Actually, the use of @Column is optional
- If not specified, the column name is the same name as Java field
- Same applies to @Table, database table name is same as the class

Terminology

Primary Key

Uniquely identifies each row in a table

Must be a unique value

Cannot contain NULL values

MySQL - Auto Increment

```
CREATE TABLE student (
    id int NOT NULL AUTO_INCREMENT,
    first_name varchar(45) DEFAULT NULL,
    last_name varchar(45) DEFAULT NULL,
    email varchar(45) DEFAULT NULL,
    PRIMARY KEY (id)
)
```

JPA Identity - Primary Key

```
@Entity  
@Table(name="student")  
public class Student {  
  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    @Column(name="id")  
    private int id;  
  
    ...  
}
```

ID Generation Strategies

GenerationType.AUTO	Pick an appropriate strategy for the particular database
GenerationType.IDENTITY	Assign primary keys using database identity column
GenerationType.SEQUENCE	Assign primary keys using a database sequence
GenerationType.TABLE	Assign primary keys using an underlying database table to ensure uniqueness

Custom Generation

- You can define your own CUSTOM generation strategy
- Create implementation of
org.hibernate.id.IdentifierGenerator
- Override the method: **public Serializable generate (...)**

Spring Boot (REST API, MVC and Microservices)

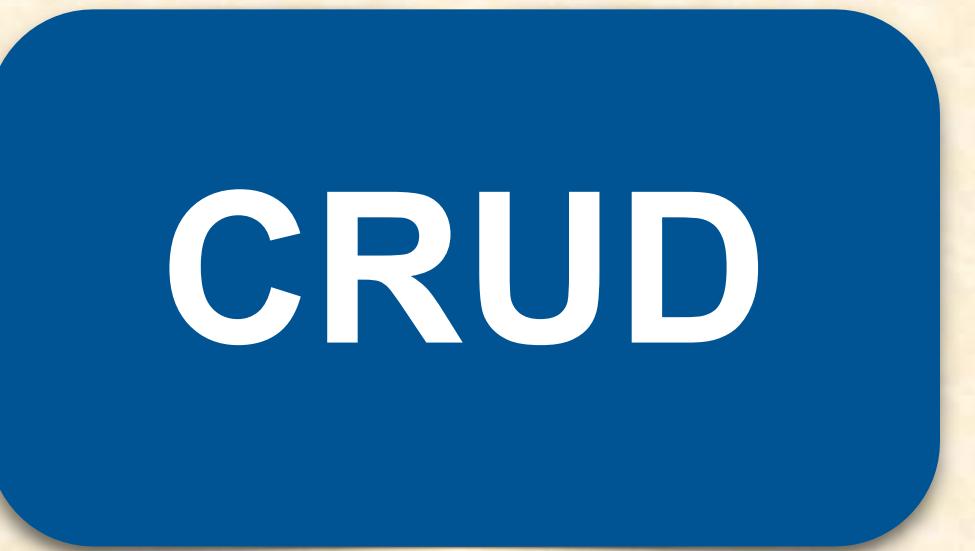
Hibernate / JPA CRUD – Create or Save

Creating / Saving an Object

JPA CRUD Operations

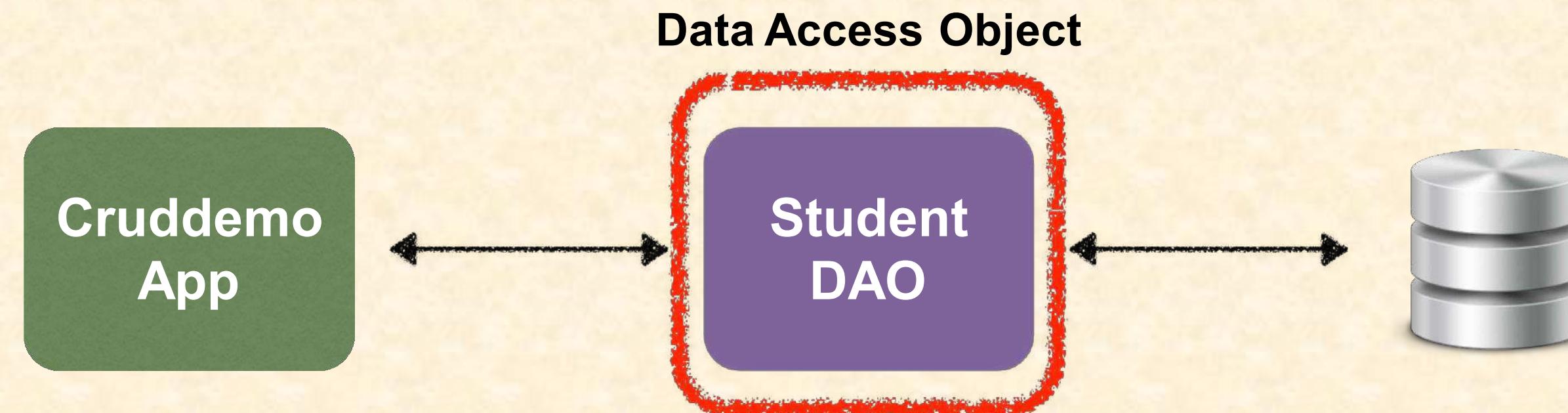
→ Create a new Student

- Read a Student
- Update a Student
- Delete a Student



Student Data Access Object (DAO)

- Responsible for interfacing with the database
- This is a common design pattern: **Data Access Object (DAO)**



Student Data Access Object

Methods

save(...)

findById(...)

findAll()

findByLastName(...)

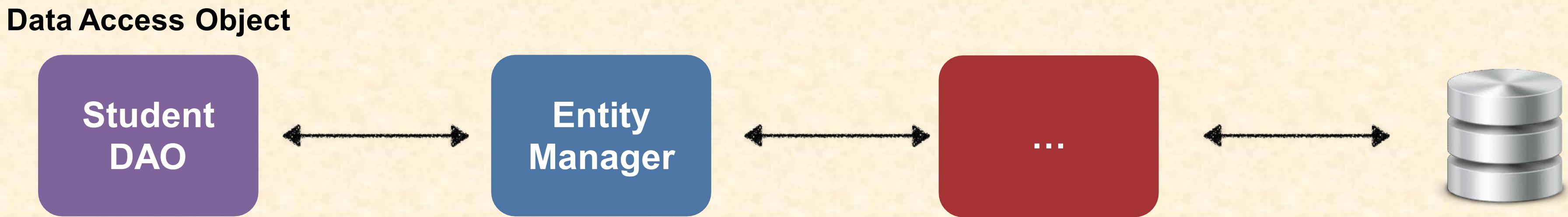
update(...)

delete(...)

deleteAll()

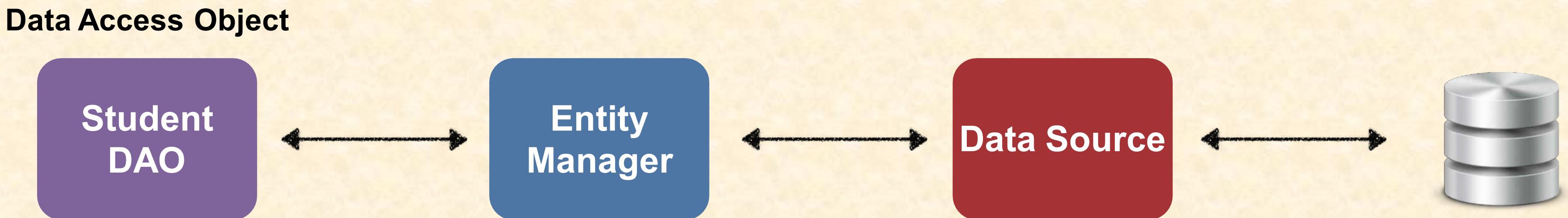
Student Data Access Object

- ⌘ Our DAO needs a JPA Entity Manager
- ⌘ JPA Entity Manager is the main component for saving/retrieving entities



JPA Entity Manager

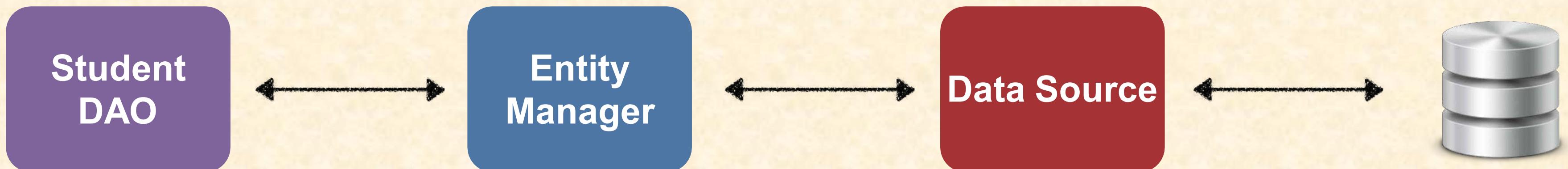
- Our JPA Entity Manager needs a Data Source
- The Data Source defines database connection info
- JPA Entity Manager and Data Source are automatically created by Spring Boot
 - Based on the file **application.properties** (JDBC URL, user id, password, etc ...)
- We can autowire/inject the JPA Entity Manager into our Student DAO



Student DAO- How to?

- Step 1: Define DAO interface
- Step 2: Define DAO implementation
 - Inject the entity manager

Data Access Object



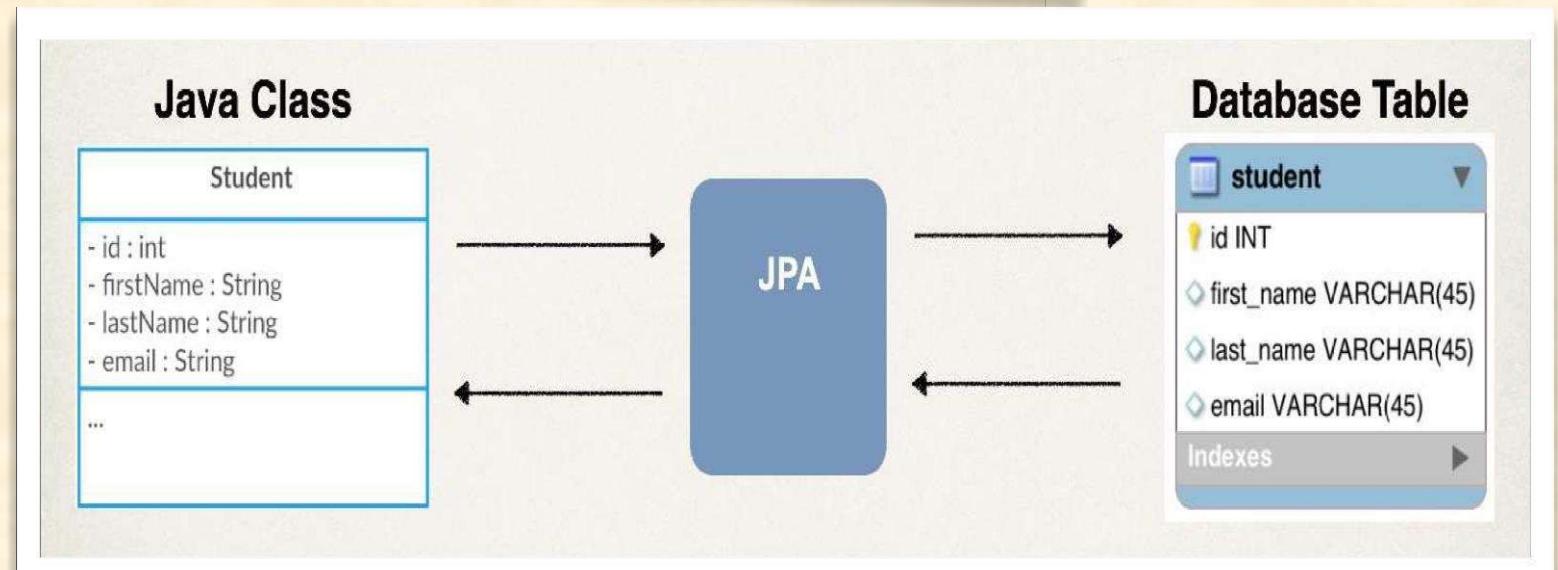
Step 1: Define DAO interface

```
import com.testcode.cruddemo.entity.Student;

public interface StudentDAO {

    → void save(Student theStudent);

}
```



Step 2: Define DAO implementation

```
import com.testcode.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;

public class StudentDAOImpl implements StudentDAO {

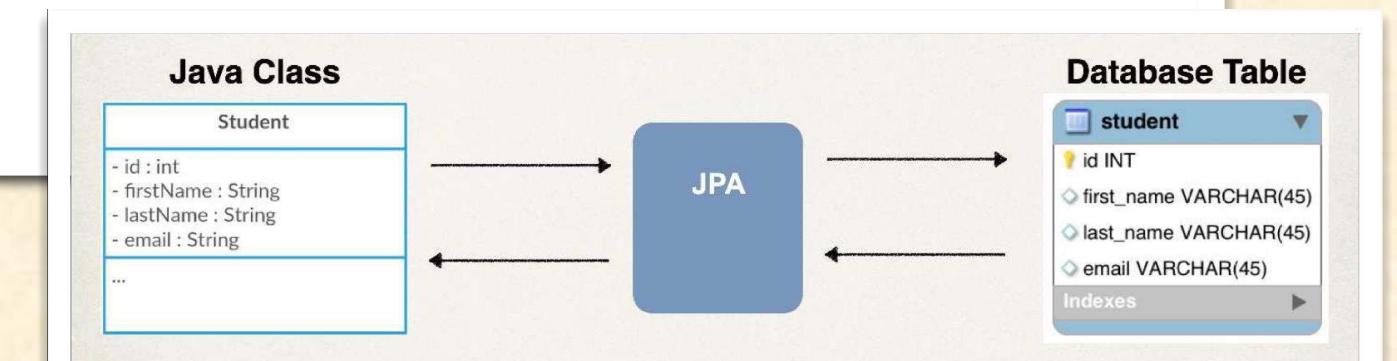
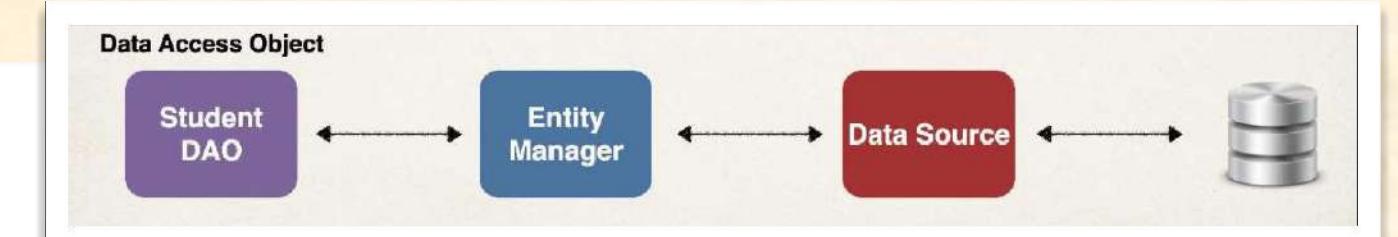
    private EntityManager entityManager;

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

    @Override
    public void save(Student theStudent) {
        entityManager.persist(theStudent);
    }
}
```

Inject the Entity Manager

Save the Java object



Spring @Transactional

- Spring provides an **@Transactional** annotation
- **Automagically** begin and end a transaction for your JPA code
 - No need for us to explicitly do this in our code
- This Spring **magic** happens behind the scenes

Step 2: Define DAO implementation

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.transaction.annotation.Transactional;

public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;

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

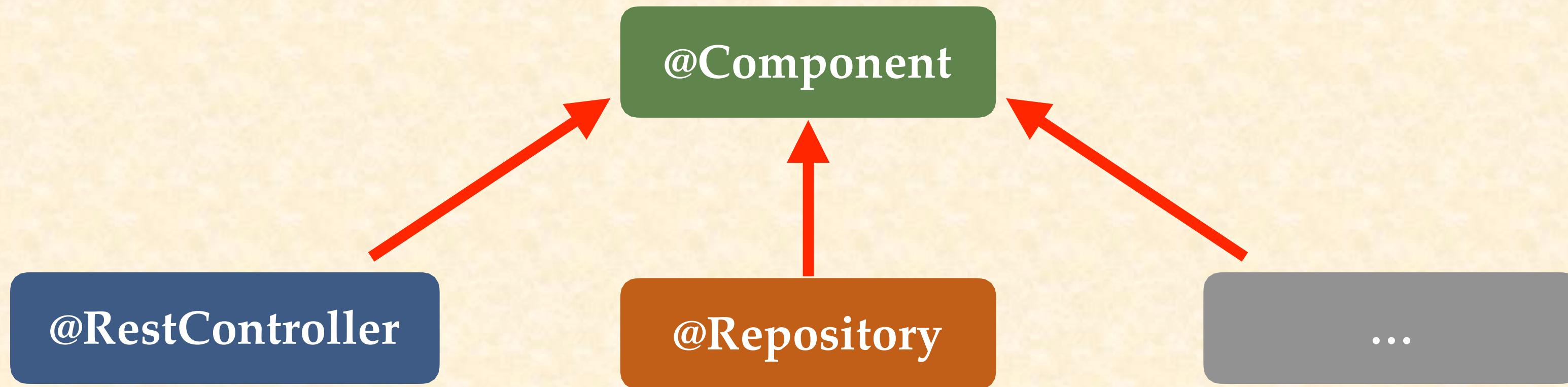
    @Override
    @Transactional
    public void save(Student theStudent) {
        entityManager.persist(theStudent);
    }

}
```

Handles transaction management

Specialized Annotation for DAOs

- Spring provides the `@Repository` annotation



`@Repository` indicates the class has capability of storage, retrieval, updating, deletion, and search.

Specialized Annotation for DAOs

- Applied to DAO implementations
- Spring will automatically register the DAO implementation
 - thanks to component-scanning
- Spring also provides translation of any JDBC related exceptions

Step 2: Define DAO implementation

Specialized annotation
for repositories

Supports component
scanning

Translates JDBC
exceptions

```
import com.testcode.cruddemo.entity.Student;
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 StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;

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

    @Override
    @Transactional
    public void save(Student theStudent) {
        entityManager.persist(theStudent);
    }

}
```

Spring Boot (REST API, MVC and Microservices)

Hibernate / JPA CRUD – Read

Retrieving an Object

JPA CRUD Operations

- Create objects
- Read objects
- Update objects
- Delete objects

Retrieving a Java Object with JPA

```
// retrieve/read from database using the primary key  
// in this example, retrieve Student with primary key: 1
```

```
Student myStudent = entityManager.find(Student.class, 1);
```

Entity class

Primary key

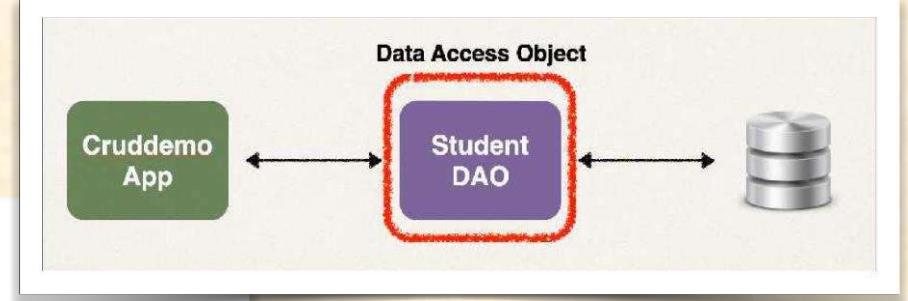
Development Process

1. Add new method to DAO interface
2. Add new method to DAO implementation

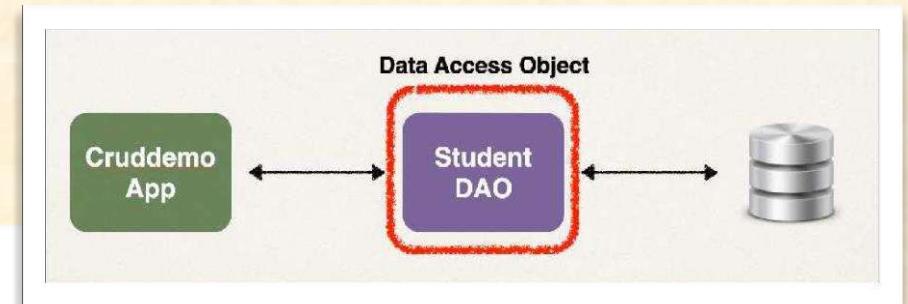
Step 1: Add new method to DAO interface

```
import com.testcode.cruddemo.entity.Student;

public interface StudentDAO {
    ...
    Student findById(Integer id);
}
```



Step 2: Define DAO implementation



```
import com.testcode.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
...
public class StudentDAOImpl implements StudentDAO {
    private EntityManager entityManager;
    ...
    @Override
    public Student findById(Integer id) {
        return entityManager.find(Student.class, id);
    }
}
```

No need to add `@Transactional` since we are doing a query

If not found, returns null

Entity class

Primary key

Querying Objects

JPQL

JPA Query Language (JPQL)

- Query language for retrieving objects
- Similar in concept to SQL
 - where, like, order by, join, in, etc...
- However, JPQL is based on **entity name** and **entity fields**

Retrieving all Students

Name of JPA Entity ...
the class name

```
TypedQuery<Student> theQuery = entityManager.createQuery("FROM Student", Student.class);  
  
List<Student> students = theQuery.getResultList();
```

Note: this is NOT the name of the database table

All JPQL syntax is based on
entity name and entity fields

Java Class	
Student	
- id : int	
- firstName : String	
- lastName : String	
- email : String	
...	

Retrieving Students: lastName = 'Doe'

Field of JPA Entity

```
TypedQuery<Student> theQuery = entityManager.createQuery(  
        "FROM Student WHERE lastName='Doe'" , Student.class);  
  
List<Student> students = theQuery.getResultList();
```

Java Class

Student
- id : int - firstName : String - lastName : String - email : String
...

Retrieving Students using OR predicate:

```
TypedQuery<Student> theQuery = entityManager.createQuery(  
    "FROM Student WHERE lastName='Doe' OR firstName='Daffy' ", Student.class);  
  
List<Student> students = theQuery.getResultList();
```

Field of JPA Entity

Field of JPA Entity

Java Class

Student
- id : int
- firstName : String
- lastName : String
- email : String
...

Retrieving Students using LIKE predicate:

```
TypedQuery<Student> theQuery = entityManager.createQuery(  
        "FROM Student WHERE email LIKE '%testcode.com'", Student.class);  
  
List<Student> students = theQuery.getResultList();
```

Match of email addresses
that ends with
testcode . com

Java Class

Student
- id : int
- firstName : String
- lastName : String
- email : String
...

JPQL - Named Parameters

```
public List<Student> findByLastName(String theLastName) {  
  
    TypedQuery<Student> theQuery = entityManager.createQuery(  
        "FROM Student WHERE lastName=:theData", Student.class);  
  
    theQuery.setParameter("theData", theLastName);  
  
    return theQuery.getResultList();  
}
```

JPQL Named Parameters are prefixed with a colon :

Think of this as a placeholder
that is filled in later

Java Class

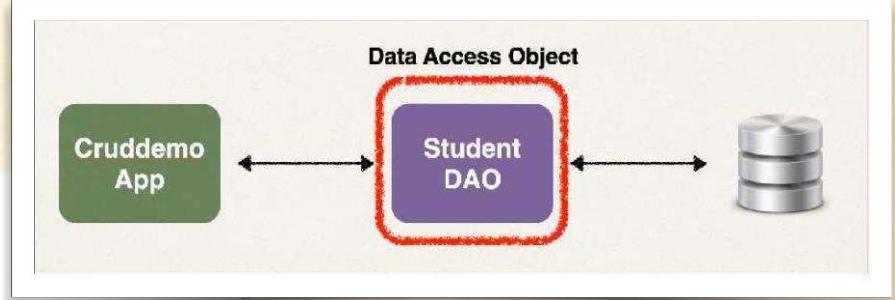
Student
- id : int
- firstName : String
- lastName : String
- email : String
...

Development Process

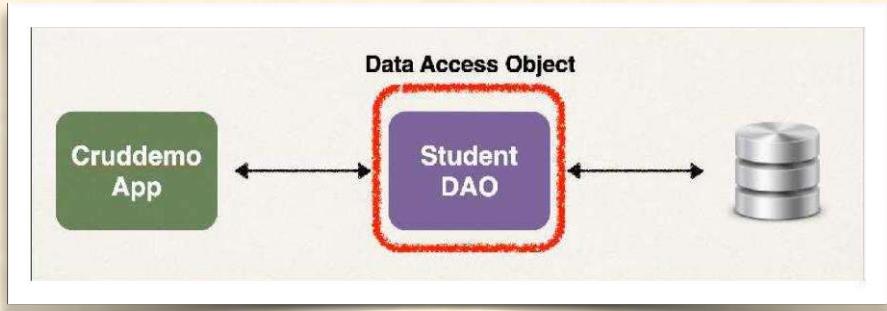
1. Add new method to DAO interface
2. Add new method to DAO implementation

Step 1: Add new method to DAO interface

```
import com.testcode.cruddemo.entity.Student; import  
java.util.List;  
  
public interface StudentDAO {  
    ...  
  
    → List<Student> findAll();  
}
```



Step 2: Define DAO implementation



```
import com.testcode.cruddemo.entity.Student  
import jakarta.persistence.EntityManager;  
import jakarta.persistence.TypedQuery;  
import java.util.List;  
  
...  
  
public class StudentDAOImpl implements StudentDAO {  
  
    private EntityManager entityManager;  
  
    ...  
  
    @Override  
    public List<Student> findAll() {  
        TypedQuery<Student> theQuery = entityManager.createQuery("FROM Student", Student.class);  
        return theQuery.getResultList();  
    }  
}
```

No need to add `@Transactional`
since we are doing a query

Name of JPA Entity

Spring Boot (REST API, MVC and Microservices)

Hibernate / JPA CRUD – Update & Delete

Updating an Object

JPA CRUD Operations

- Create objects
- Read objects
- Update objects

- Delete objects

Update a Student

```
Student theStudent = entityManager.find(Student.class, 1);  
  
// change first name to "Scooby"  
theStudent.setFirstName("Scooby");  
  
entityManager.merge(theStudent);
```

Update the entity

Update last name for all students

```
int numRowsUpdated = entityManager.createQuery(  
    "UPDATE Student SET lastName='Tester' ")  
    .executeUpdate();
```

Return the number
of rows updated

Execute this
statement

Field of JPA Entity

Name of JPA Entity ...
the class name

Java Class

Student
- id : int
- firstName : String
- lastName : String
- email : String
...

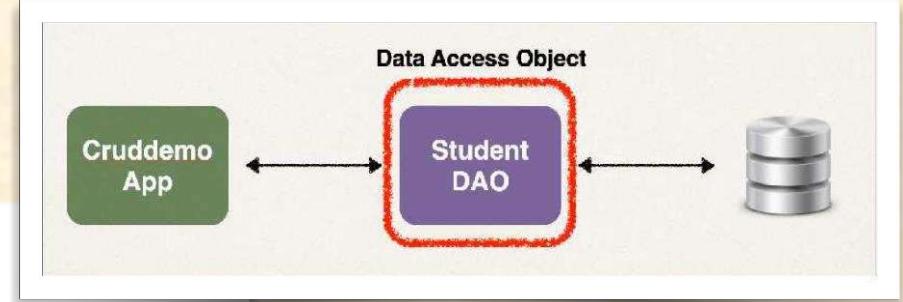
Development Process

1. Add new method to DAO interface
2. Add new method to DAO implementation

Step 1: Add new method to DAO interface

```
import com.testcode.cruddemo.entity.Student;

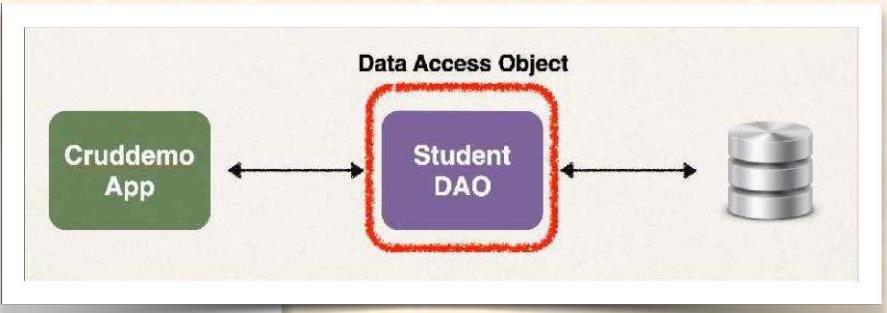
public interface StudentDAO {
    ...
    → void update(Student theStudent);
}
```



Step 2: Define DAO implementation

```
import com.testcode.cruddemo.entity.Student;  
import jakarta.persistence.EntityManager;  
import org.springframework.transaction.annotation.Transactional;  
  
...  
  
public class StudentDAOImpl implements StudentDAO {  
  
    private EntityManager entityManager;  
    ...  
  
    @Override  
    @Transactional  
    public void update(Student theStudent) {  
        entityManager.merge(theStudent);  
    }  
  
}
```

Add `@Transactional` since we are performing an update



Deleting an Object

JPA CRUD Operations

- Create objects
- Read objects
- Update objects
- Delete objects

Delete a Student

```
// retrieve the student
int id = 1;
Student theStudent = entityManager.find(Student.class, id);

// delete the student
entityManager.remove(theStudent);
```

Delete based on a condition

Field of JPA Entity

```
int numRowsDeleted = entityManager.createQuery(  
    "DELETE FROM Student WHERE lastName='Smith'")  
    .executeUpdate();
```

Return the number of rows deleted

Execute this statement

Name of JPA Entity ...
the class name

Java Class

Student
- id : int
- firstName : String
- lastName : String
- email : String
...

Delete All Students

```
int numRowsDeleted = entityManager
    .createQuery("DELETE FROM Student")
    .executeUpdate();
```

Java Class

Student
- id : int
- firstName : String
- lastName : String
- email : String
...

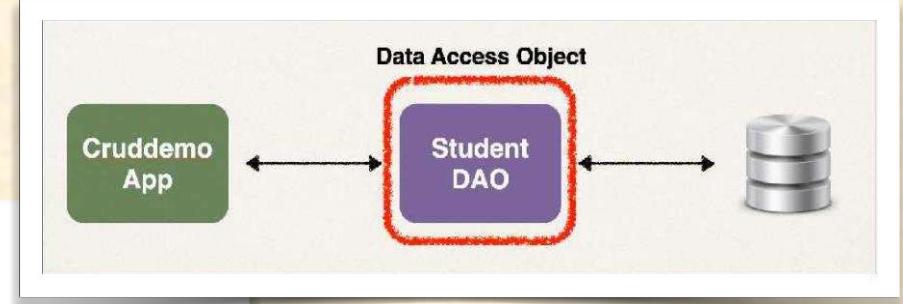
Development Process

1. Add new method to DAO interface
2. Add new method to DAO implementation

Step 1: Add new method to DAO interface

```
import com.testcode.cruddemo.entity.Student;

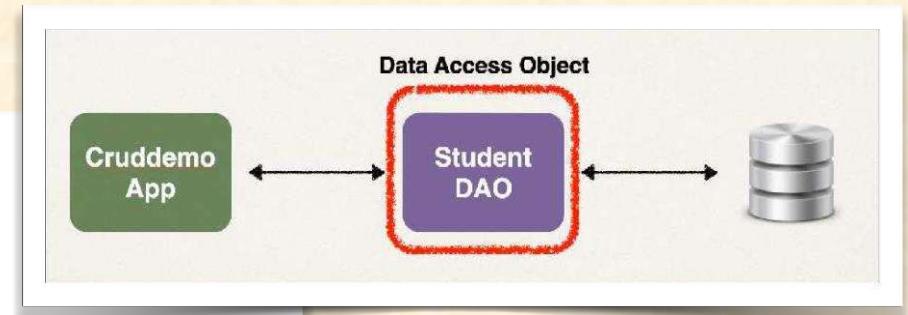
public interface StudentDAO {
    ...
    → void delete(Integer id);
}
```



Step 2: Define DAO implementation

```
import com.testcode.cruddemo.entity.Student;  
import jakarta.persistence.EntityManager;  
import org.springframework.transaction.annotation.Transactional;  
  
...  
  
public class StudentDAOImpl implements StudentDAO {  
  
    private EntityManager entityManager;  
    ...  
  
    @Override  
    @Transactional  
    public void delete(Integer id) {  
        Student theStudent = entityManager.find(Student.class, id);  
        entityManager.remove(theStudent);  
    }  
}
```

Add `@Transactional` since we are performing a delete



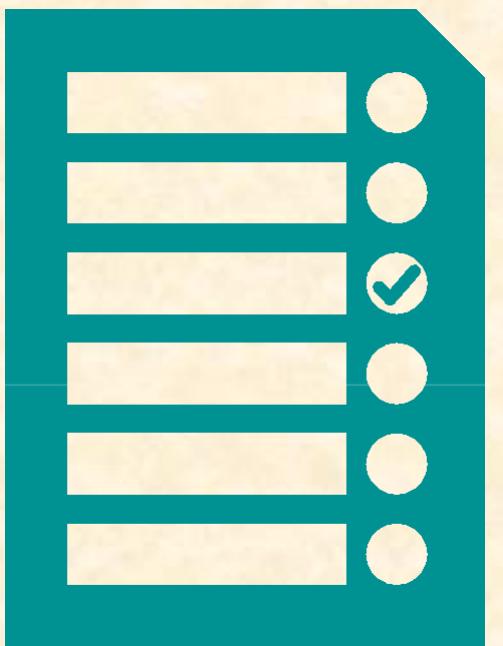
Spring Boot (REST API, MVC and Microservices)

Hibernate / JPA – Creating Tables from Java Code

Create Database Tables from Java Code

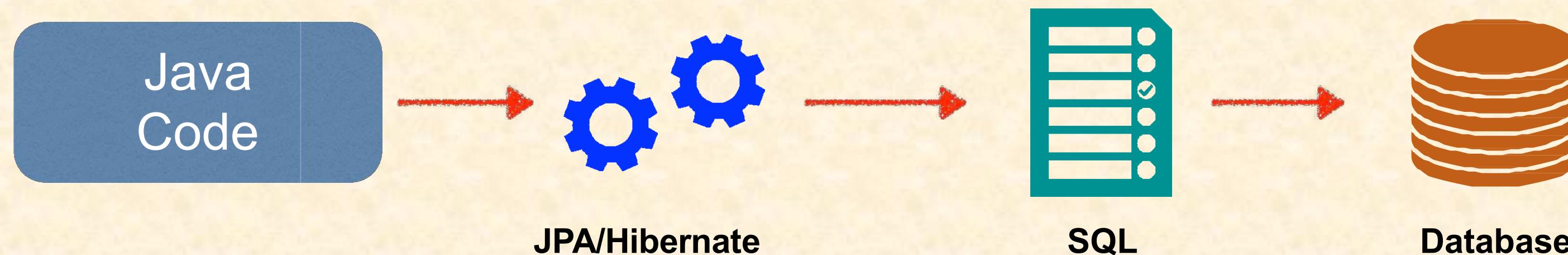
Create database tables: student

- Previously, we created database tables by using a SQL script from MySQL console



Create database tables: student

- JPA/Hibernate provides an option to automagically create database tables
- Creates tables based on Java code with JPA/Hibernate annotations
- Useful for development and testing



Configuration

- In Spring Boot configuration file: **application.properties**

```
spring.jpa.hibernate.ddl-auto=create
```

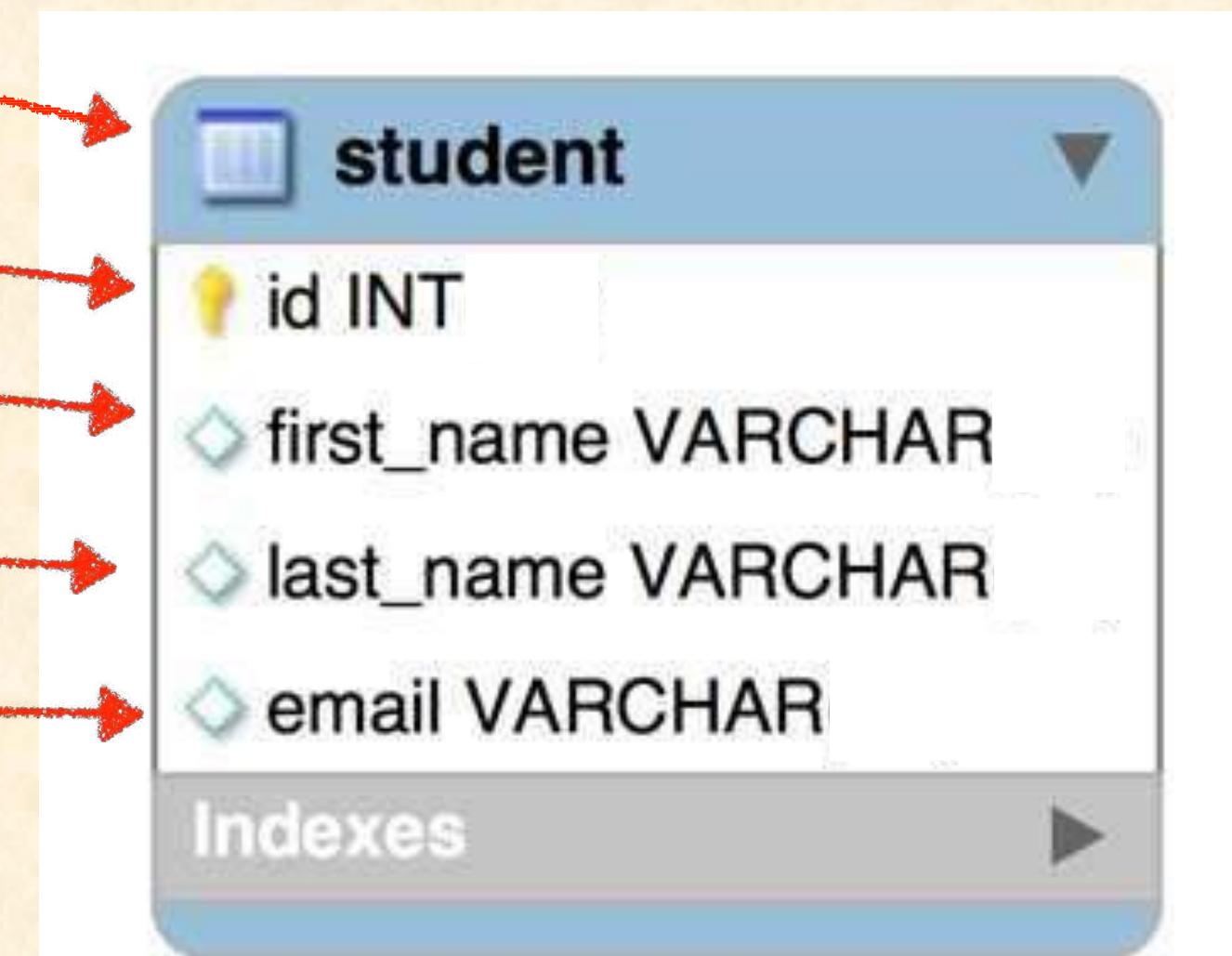
- When you run your app, JPA/Hibernate will drop tables then create them
- Based on the JPA/Hibernate annotations in our Java code

Creating Tables based on Java Code

Hibernate will generate and execute this

```
1 @Entity  
2 @Table(name="student")  
public class Student {  
  
    @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  
}
```

create table student (id integer not null auto_increment, first_name varchar(255), last_name varchar(255), email varchar(255), primary key (id))



Configuration - application.properties

`spring.jpa.hibernate.ddl-auto=PROPERTY VALUE`

Property Value	Property Description
<code>none</code>	No action will be performed
<code>create-only</code>	Database tables are only created
<code>drop</code>	Database tables are dropped
<code>create</code>	Database tables are dropped followed by database tables creation
<code>create-drop</code>	Database tables are dropped followed by database tables creation. On application shutdown, drop the database tables
<code>validate</code>	Validate the database tables schema
<code>update</code>	Update the database tables schema

When database tables are dropped,
all data is lost

Basic Projects

- For basic projects, can use auto configuration

```
spring.jpa.hibernate.ddl-auto=create
```

- Database tables are dropped first and then created from scratch

Note:

When database tables are dropped, all data is lost

Basic Projects

- If you want to create tables once ... and then keep data, use: update

```
spring.jpa.hibernate.ddl-auto=update
```

- However, will ALTER database schema based on latest code updates
- Be VERY careful here ... only use for basic projects



`spring.jpa.hibernate.ddl-auto=create`

Use Case

- Automatic table generation is useful for
 - Database integration testing with in-memory databases
 - Basic, small hobby projects

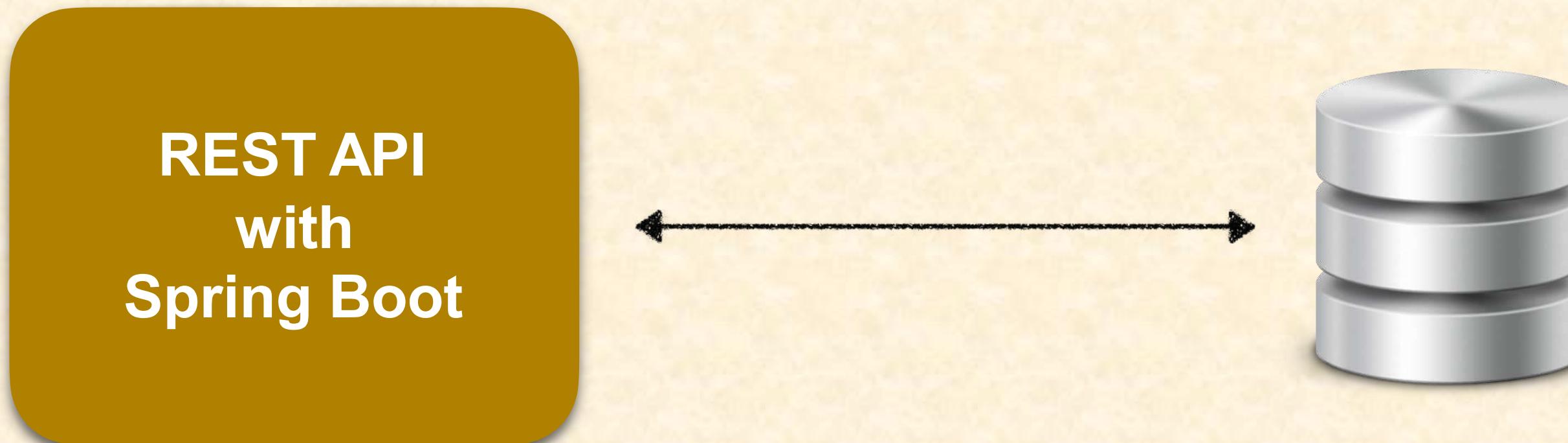
Spring Boot (REST API, MVC and Microservices)

REST API with CRUD Operations – EMS Part - I



Time to Code a Real-Time Project -EMS

REST API with Spring Boot
that connects to a database
and CRUD operations



API Requirements

Create a REST API for the Employee Directory

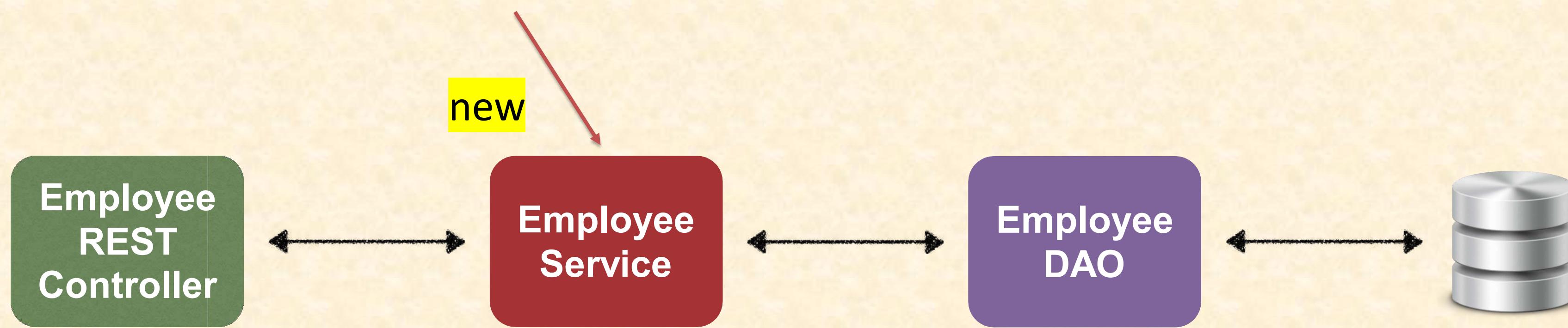
REST clients should be able to:

- Get a list of employees
- Get a single employee by id
- Add a new employee
- Update an employee
- Delete an employee

Development Process

1. Set up Database Development 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

Application Architecture



1. Set up Database Table in MySQL

1. Create a new database table: **employee**
2. Load table with sample data

```
CREATE TABLE `employee` (  
`id` int NOT NULL AUTO_INCREMENT,  
`first_name` varchar(20) DEFAULT NULL,  
`last_name` varchar(20) DEFAULT NULL,  
`email` varchar(30) DEFAULT NULL,  
PRIMARY KEY (`id`)
```

employee	
!	id INT(11)
◆	first_name VARCHAR(45)
◆	last_name VARCHAR(45)
◆	email VARCHAR(45)
Indexes	

```
mysql> select * from employee;  
+----+-----+-----+-----+  
| id | first_name | last_name | email      |  
+----+-----+-----+-----+  
|  1 | liam       | nessom    | liam@nessom.com |  
|  2 | bruce      | willis    | bruce@willis.com |  
|  3 | dengel     | washington | dengel@washington.com |  
+----+-----+-----+-----+
```

Development Process: How to?

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

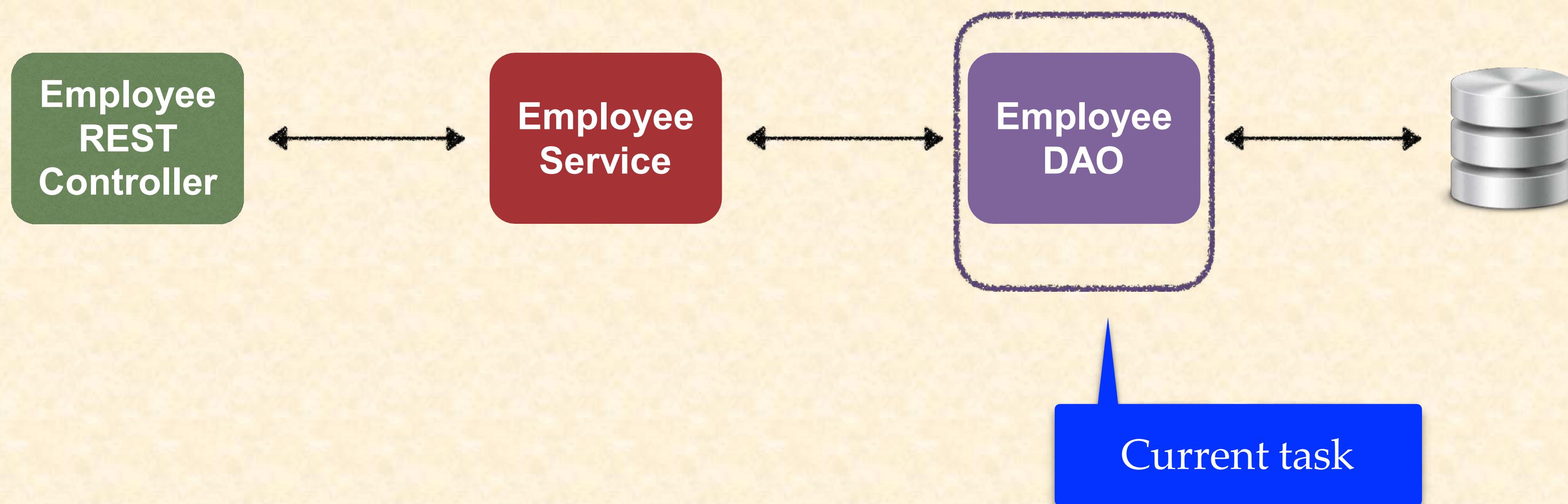
7. Delete an existing employee



Dependencies to be added

Spring Web
Spring Dev Tools
Spring Data JPA
MySQL Driver
Lombok - optional

Application Architecture



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



**Let's build a
DAO layer for this**

DAO Interface

```
public interface EmployeeDAO {  
  
    List<Employee> findAll();  
    Employee findById(int empId);  
  
    Employee save(Employee emp); // save and update  
  
    void deleteById(int empId);  
}
```

DAO Impl

Same interface for
consistent API

```
@Repository  
public class EmployeeDAOJpaImpl implements EmployeeDAO {  
  
    private EntityManager entityManager;  
  
    @Autowired  
    public EmployeeDAOJpaImpl(EntityManager theEntityManager) {  
        entityManager = theEntityManager;  
    }  
  
    ...  
}
```

Automatically created
by Spring Boot

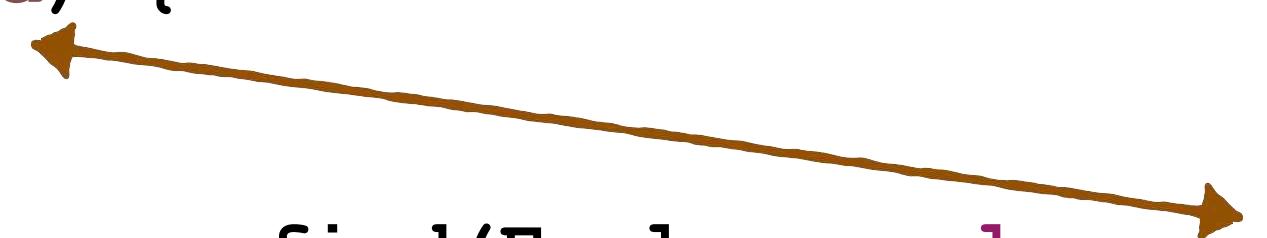
Constructor
injection

Get a list of employees

```
@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;  
}
```

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);  
}
```



Next.....

Define Services with @Service

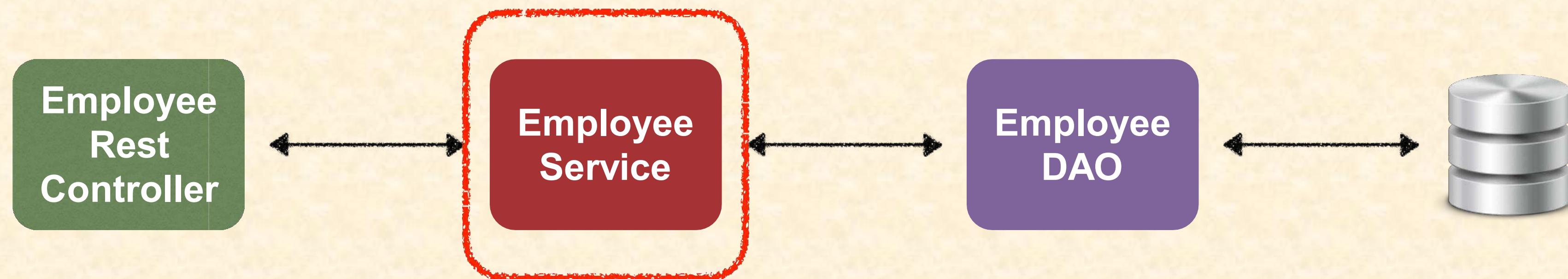
Spring Boot (REST API, MVC and Microservices)

REST API with CRUD Operations – EMS Part - II



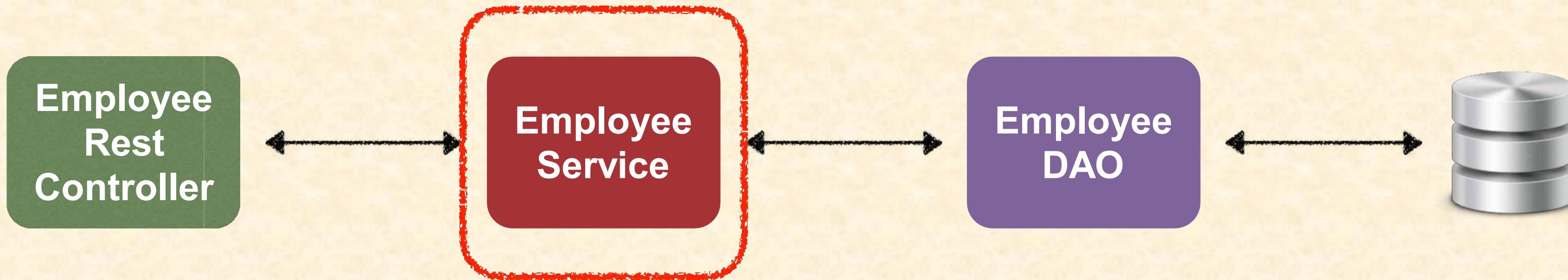
Define Services with @Service

Refactor: Add a Service Layer

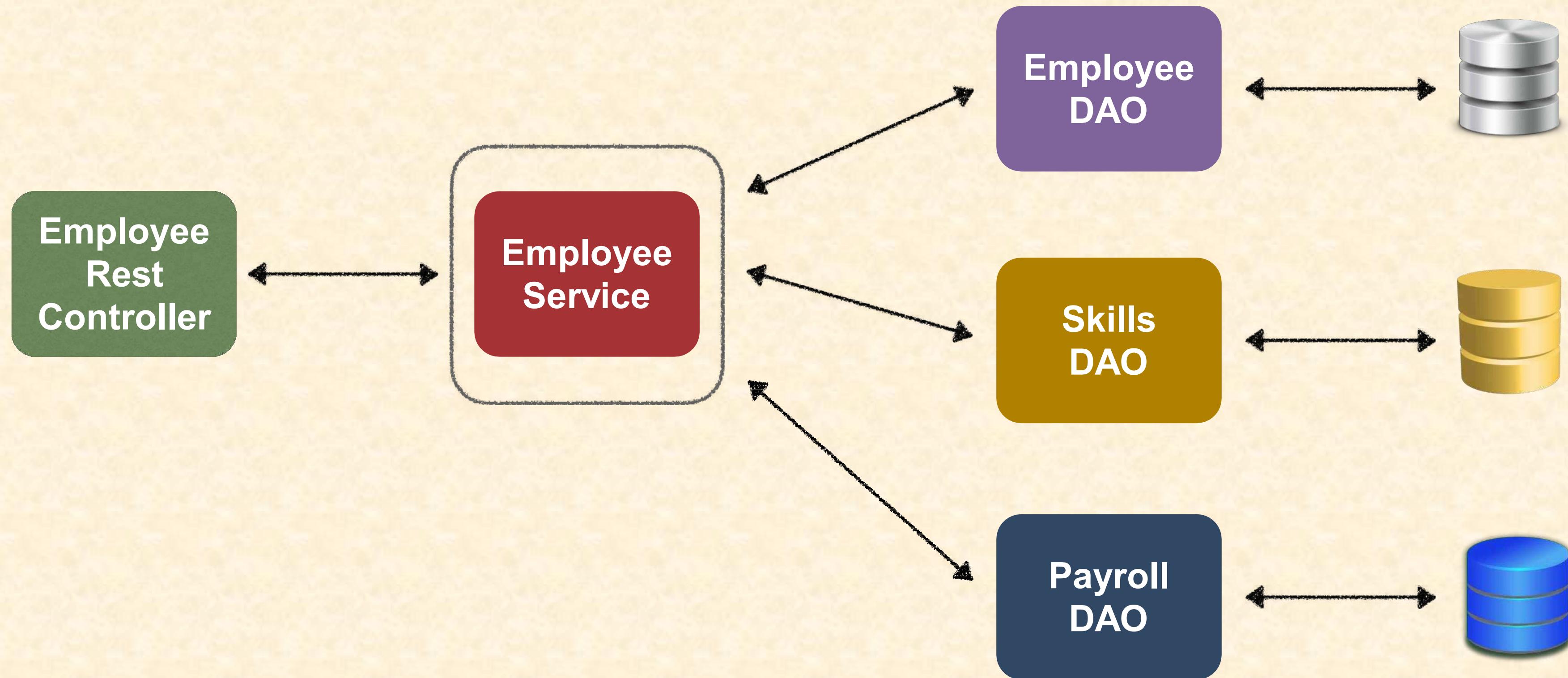


Purpose of Service Layer

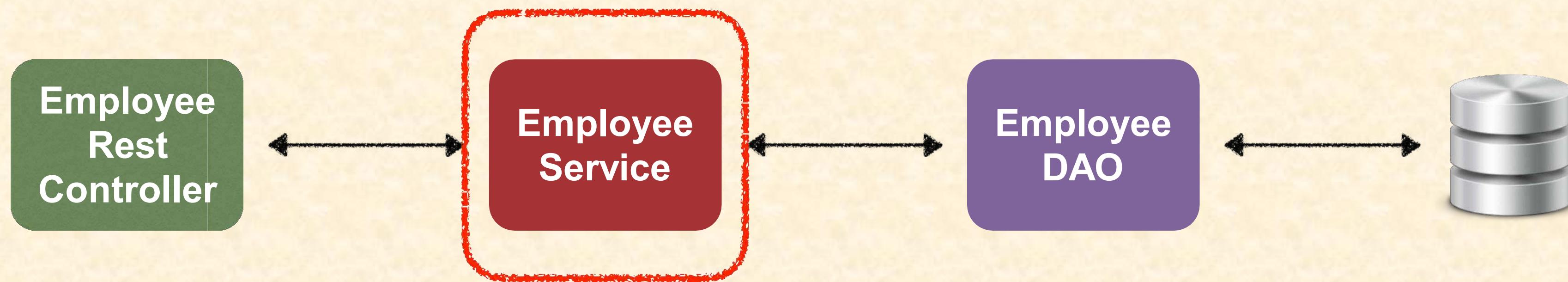
- ⌘ Service Facade design pattern
- ⌘ Intermediate layer for custom business logic
- ⌘ Integrate data from multiple sources (DAO/repositories)



Integrate Multiple Data Sources

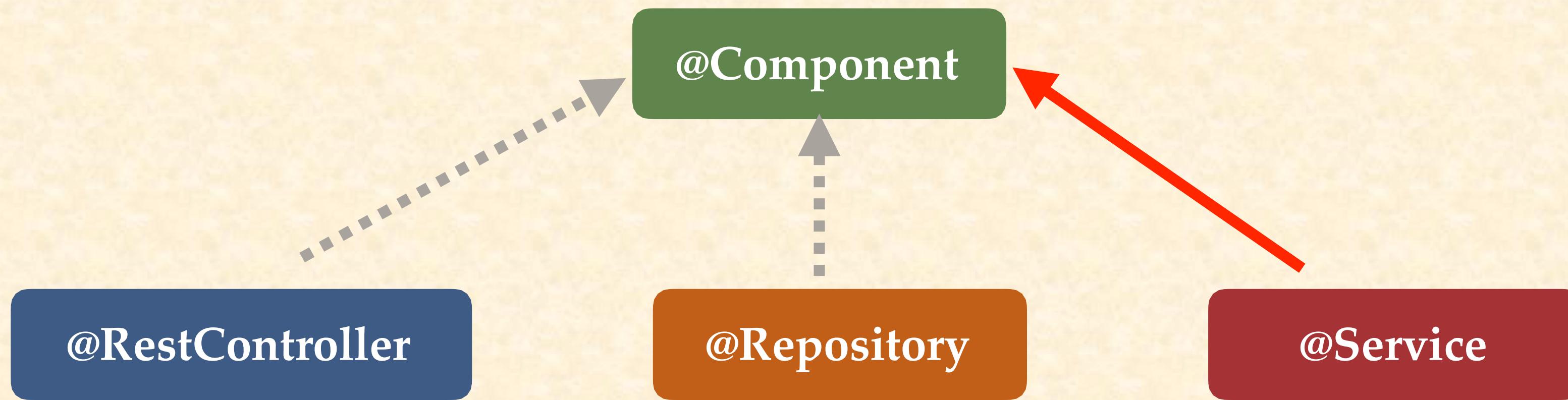


Most Times - Delegate Calls



Specialized Annotation for Services

- Spring provides the `@Service` annotation

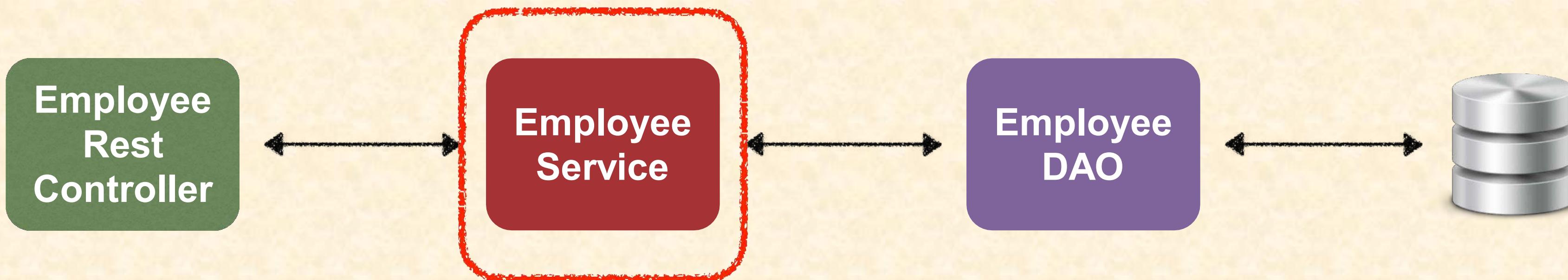


Specialized Annotation for Services

- `@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();  
    Employee findById(int empId);  
    Employee save(Employee emp); // save and update  
    void deleteById(int empId);  
}
```

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();

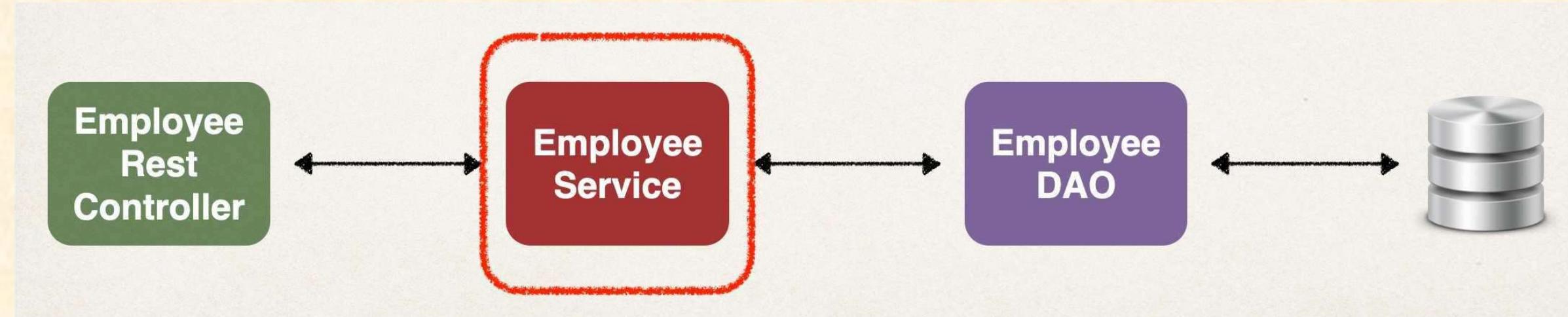
}

}

```
@Service  
@Transactional  
public class EmployeeServiceImpl implements EmployeeService{  
    @Autowired  
    private EmployeeDao employeeDao;  
  
    @Override  
    public List<Employee> findAll() {  
        return employeeDao.findAll();  
    }  
  
    @Override  
    public Employee findById(int id) {  
        return employeeDao.findById(id);  
    }  
  
    @Override  
    public Employee save(Employee employee) {  
        return employeeDao.save(employee);  
    }  
  
    @Override  
    public void deleteById(int id) {  
        employeeDao.deleteById(id);  
    }  
}
```

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



Rest Controller Methods - Find and Add

```
@PostMapping("/employees")
public Employee addEmployee(@RequestBody Employee employee) {
    employee.setId(0);
    Employee dbEmployee = employeeService.save(employee);
    return dbEmployee;
}
```

```
@GetMapping("/employees")
public List<Employee> getEmployees() {
    return employeeService.findAll();
}
```

```
@GetMapping("/employees/{employeeId}")
public Employee getEmployee(@PathVariable int employeeId) {
    return employeeService.findById(employeeId);
}
```

```
@PutMapping("/employees")
public Employee updateEmployee(@RequestBody Employee employee) {
    Employee dbEmployee = employeeService.save(employee);
    return dbEmployee;
}
```

```
@DeleteMapping("employees/{employeeId}")
public String deleteEmployee(@PathVariable int employeeId) {
    Employee tempEmployee = employeeService.findById(employeeId);
    if(tempEmployee == null) {
        throw new RuntimeException("Employee ID not found: "+employeeId);
    }
    else
        employeeService.deleteById(employeeId);

    return "Employee with ID:"+employeeId+" deleted";
}
```

Read a Single Employee

REST Client



Employee
REST
Controller

Create a New Employee

Since new employee,
we are not
passing id / primary key

REST
Client

POST

/api/employees

```
{  
    "firstName": "Angelina",  
    "lastName": "Jolie",  
    "email": "angelina@jolie.net"  
}
```



Employee
REST
Controller

```
{  
    "id": 5,  
    "firstName": "Angelina",  
    "lastName": "Jolie",  
    "email": "angelina@jolie.net"  
}
```

Response contains
new id / primary key

Update Employee

ID of employee to update
With updated info

PUT

/api/employees

```
{  
    ...  
    "id": 5,  
    "firstName": "Angelina",  
    "lastName": "Jolie",  
    "email": "angelina@jolie.com"  
}
```

REST
Client

Employee
REST
Controller

Response contains
updated info (echoed)

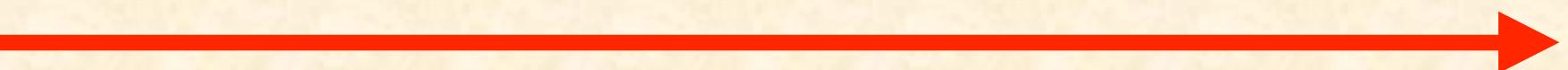
```
{  
    "id": 5,  
    "firstName": "Angelina",  
    "lastName": "Jolie",  
    "email": "angelina@jolie.com"  
}
```

Delete Employee

REST
Client

DELETE

/api/employees/{employeeId}



Deleted employee id - {employeeId}

Employee with ID:5 deleted

Employee
REST
Controller

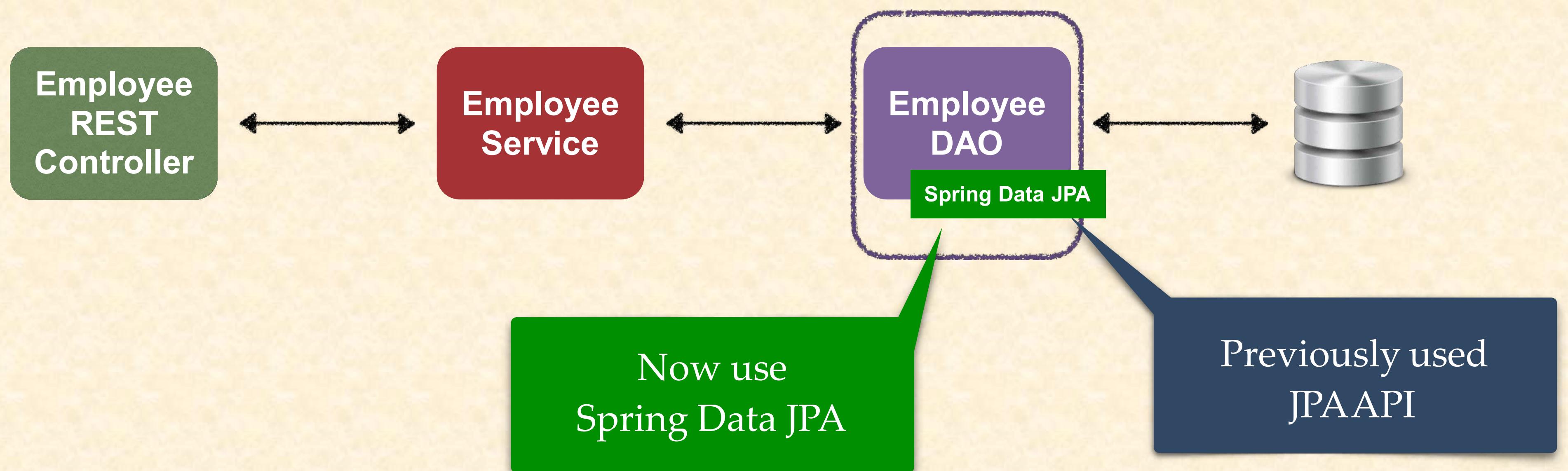
Spring Boot (REST API, MVC and Microservices)

REST API – Spring Data JPA



Spring Data JPA in Spring Boot

Application Architecture



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 Employee findById(int theId) {  
  
    // get data  
    Employee theData = entityManager.find(Employee.class, theId);  
  
    // return data  
    return theData;  
}
```

Only difference is the entity type and primary key

Most of the code is the same

Entity type

Primary key

Our Wish

- We wish we could tell Spring:

Create a DAO for me

Plug in my entity type and primary key

Give me all of the basic CRUD features for free

Our Wish Diagram

Entity: Employee

Primary key: Integer

findAll()
findById(...)
save(...)
deleteById(...)
... others ...

CRUD methods

Spring Data JPA - Solution

- Spring Data JPA is the solution!! <https://spring.io/projects/spring-data-jpa>
- Creates a DAO and just plug in our entity type and primary key
- Spring will give us a CRUD implementation for FREE like MAGIC!!
- Helps to minimize boiler-plate DAO code ...great!!!

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)

Entity: Employee

Primary key: Integer

findAll()
findById(...)
save(...)
deleteById(...)
... others ...



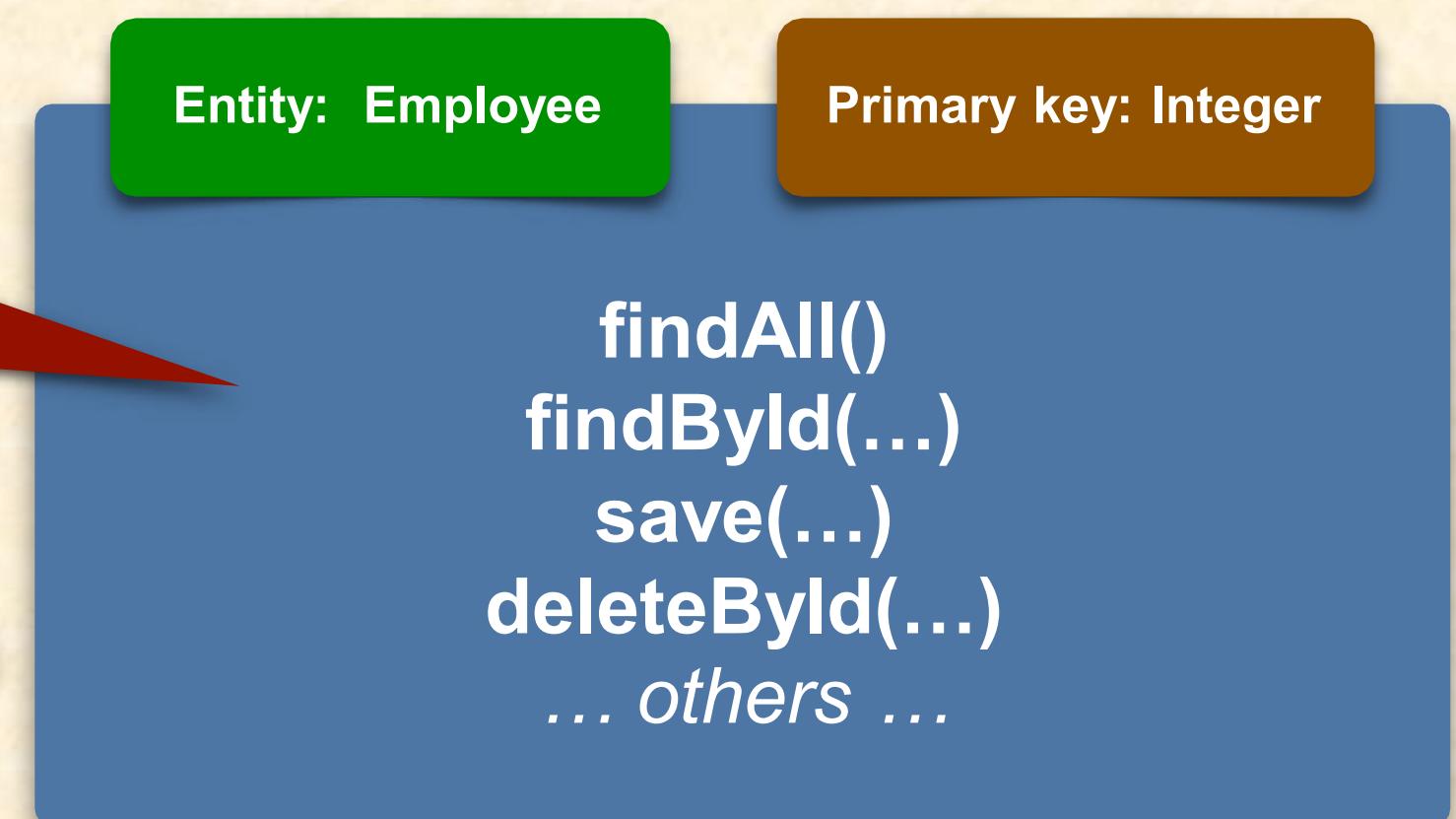
Development Process

1. Extend **JpaRepository** interface
2. Use the Repository in our app

No need for
implementation class

Step 1: Extend JpaRepository interface

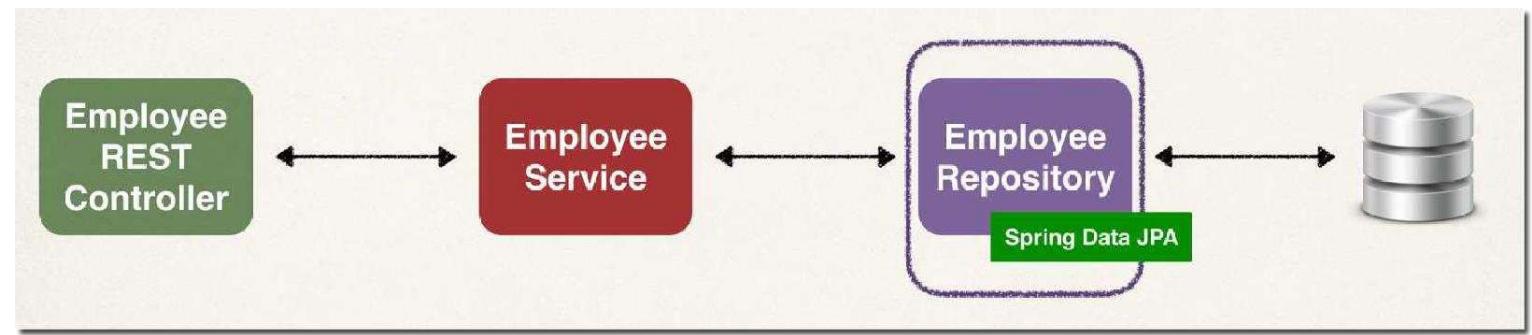
```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {  
    // that's all ... no need to write any code!  
}
```



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

Before Spring Data JPA

```
public interface EmployeeDAO {  
  
    public List<Employee> findAll();  
  
    public Employee findById(int theId);  
  
    public void deleteEmployee(Employee theEmployee);  
  
    public void createEmployee(Employee theEmployee);  
}  
  
@Repository  
public class EmployeeJpaDAO implements EmployeeDAO {  
  
    private EntityManager entityManager;  
  
    @Autowired  
    public EmployeeJpaDAO(EntityManager theEntityManager) {  
        entityManager = theEntityManager;  
    }  
  
    @Override  
    public List<Employee> findAll() {  
        // create query  
        TypedQuery<Employee> theQuery = entityManager.createQuery("from Employee");  
  
        // execute query  
        List<Employee> employeeList = theQuery.getResultList();  
        return employeeList;  
    }  
  
    @Override  
    public Employee findById(int theId) {  
        // get employee  
        Employee theEmployee = entityManager.find(Employee.class, theId);  
  
        // return employee  
        return theEmployee;  
    }  
}
```



2 Files
30+ lines of code

After Spring Data JPA

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {  
  
    // that's it ... no need to write any code LOL!  
}
```

1 File
3 lines of code!

No need for
implementation class



Ready for

Spring Data JPA Demo?

Spring Boot (REST API, MVC and Microservices)

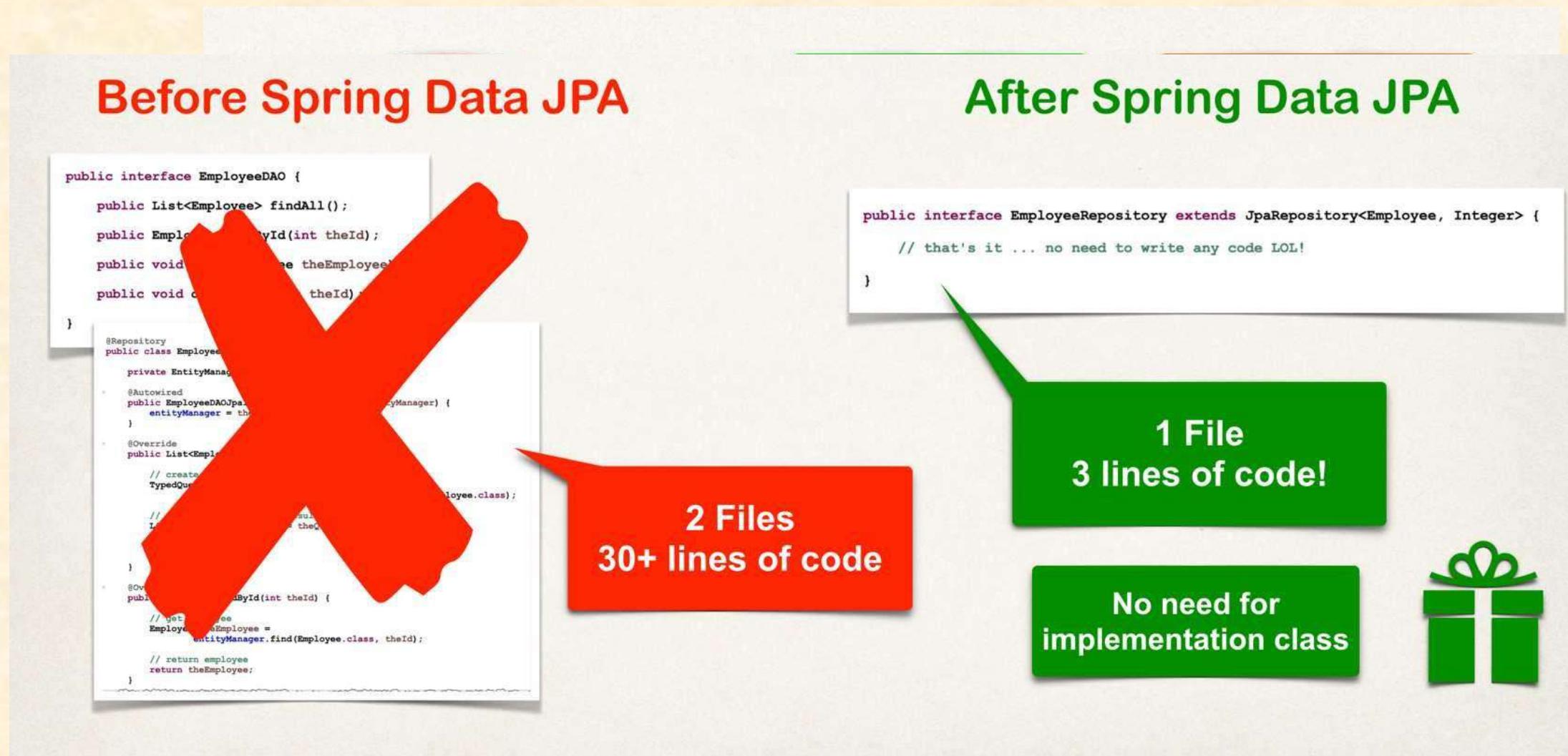
REST API – Spring Data REST



Spring Data REST in Spring Boot

Spring Data JPA

- Earlier, we saw the magic of Spring Data JPA
- This helped to eliminate boilerplate code

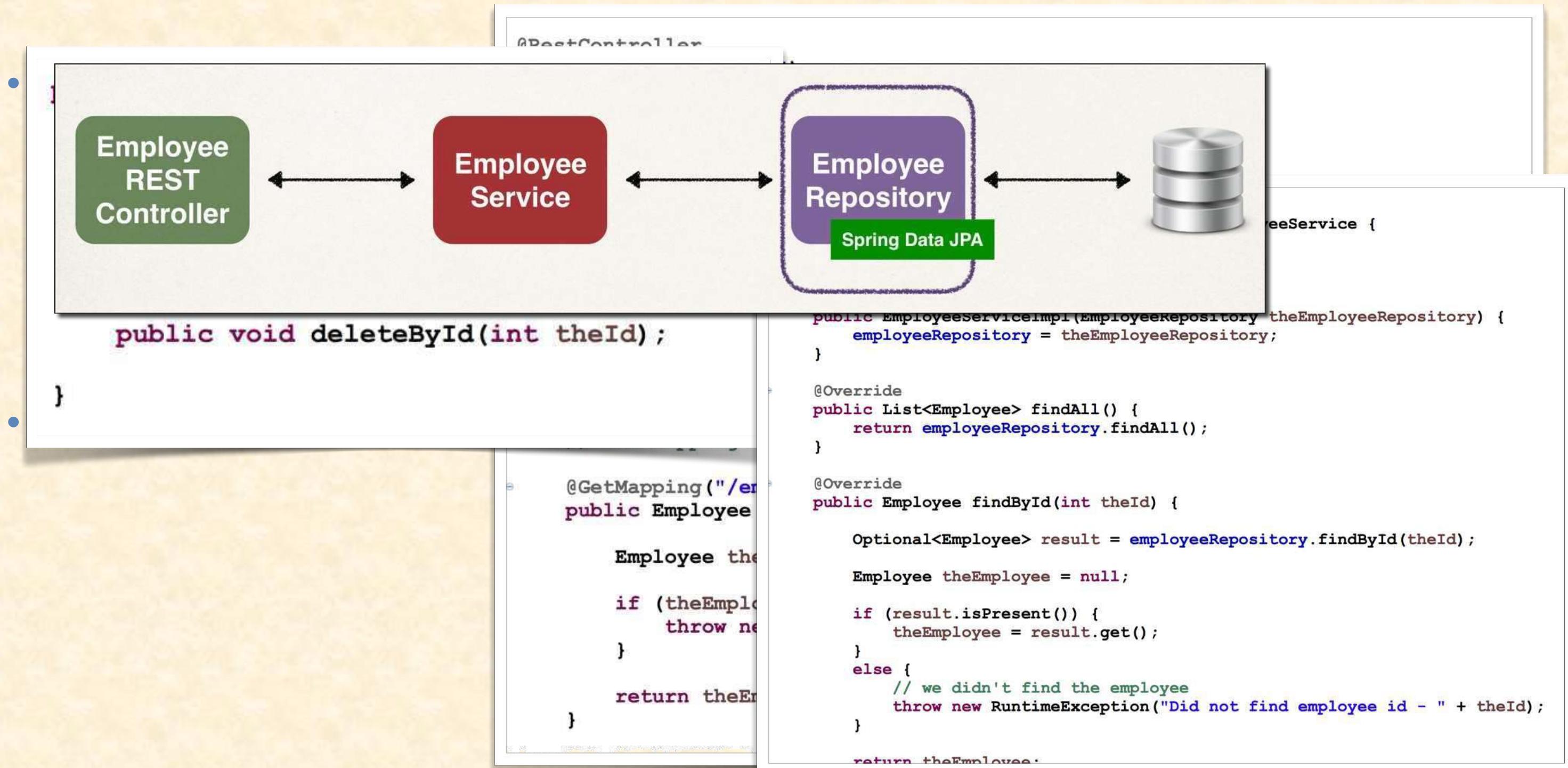


Hmmm ...

Can this apply to REST APIs?

The Problem

- We saw how to create a REST API for Employee



Our Wish

- I wish we could tell Spring:

Create a REST API for me

Use my existing JpaRepository (entity, primary key)

Give me all of the basic REST API CRUD features for free

Spring Data REST - Solution

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

<https://spring.io/projects/spring-data-rest>



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

*More on
plural forms in
Upcoming
videos*

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

/employees

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



In A Nutshell

For Spring Data REST, you only need 3 items

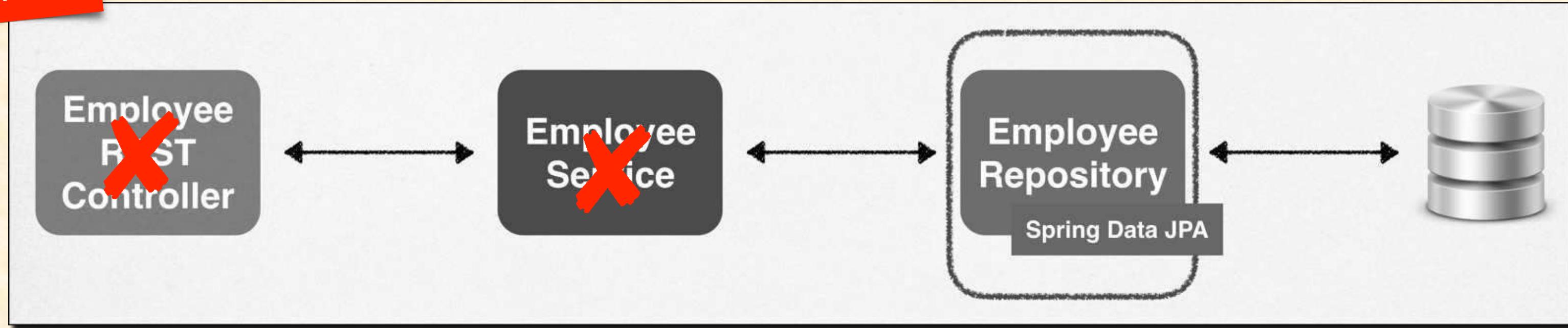
1. Your entity: **Employee**
2. JpaRepository: **EmployeeRepository extends JpaRepository**
3. Maven POM dependency for: **spring-boot-starter-data-rest**

We already have these two

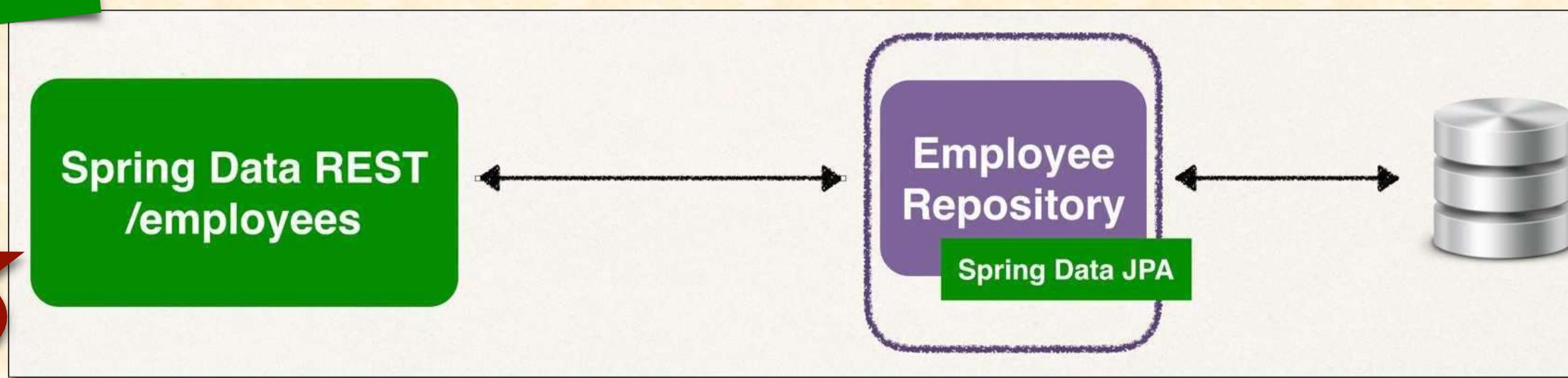
Only item that is new

Application Architecture

Before

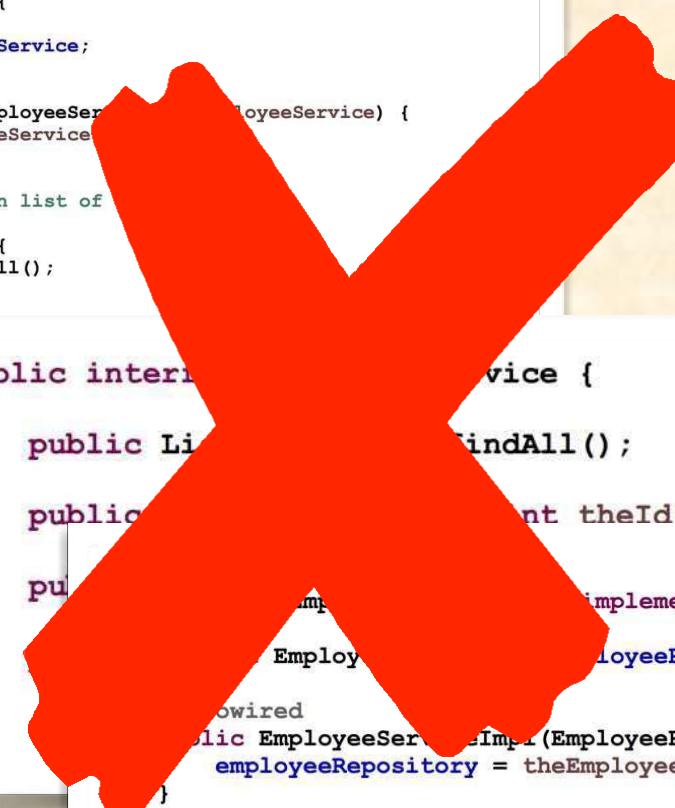


After



Minimized Boilerplate Code

Before Spring Data REST



```
@RestController
@RequestMapping("/api")
public class EmployeeRestController {
    private EmployeeService employeeService;
    @Autowired
    public EmployeeRestController(EmployeeService theEmployeeService) {
        employeeService = theEmployeeService;
    }
    // expose "/employees" and return list of employees
    @GetMapping("/employees")
    public List<Employee> findAll() {
        return employeeService.findAll();
    }
    // add mapping for GET /employee/{id}
    @GetMapping("/employees/{id}")
    public Employee getEmployee(@PathVariable int theId) {
        Employee theEmployee = employeeService.findById(theId);
        if (theEmployee == null)
            throw new RuntimeException("Did not find employee id - " + theId);
        return theEmployee;
    }
    @Override
    public List<Employee> findAll() {
        return employeeRepository.findAll();
    }
    @Override
    public Employee findById(int theId) {
        Employee result = employeeRepository.findById(theId);
        if (result == null)
            throw new RuntimeException("Did not find employee id - " + theId);
        return result;
    }
}
```

**3 files
100+ lines of code**

After Spring Data REST



```
<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



Ready for

Spring Data Rest Demo?

Spring Boot (REST API, MVC and Microservices)

REST API – HATEOAS

HATEOAS

- Spring Data REST endpoints are HATEOAS compliant
 - **HATEOAS:** Hypermedia As the Engine Of Application State
- Hypermedia-driven sites provide information to access REST interfaces
 - Think of it as meta-data for REST data

<https://spring.io/understanding/HATEOAS>

HATEOAS

- Spring Data REST response using HATEOAS
- For example REST response from: **GET /employees/3**

Get single
employee

Response

```
{  
  "firstName": "Avani",  
  "lastName": "Gupta",  
  "email": "avani@testcode.com",  
  "_links": {  
    "self": {  
      "href": "http://localhost:8080/employees/3"  
    },  
    "employee": {  
      "href": "http://localhost:8080/employees/3"  
    }  
  }  
}
```

Employee data

**Response meta-data
Links to data**

The diagram illustrates a REST response object with annotations. A red dashed box encloses the main data object, which contains properties like firstName, lastName, email, and a _links object. A purple callout labeled 'Employee data' points to the main object. A yellow callout labeled 'Response meta-data Links to data' points to the _links field, specifically highlighting the self link.

HATEOAS

- For a collection, meta-data includes page size, total elements, pages etc
- For example REST response from: **GET /employees**

Response

```
{  
  "_embedded": {  
    "employees": [  
      {  
        "firstName": "Leslie",  
        ...  
      },  
      ...  
    ]  
  },  
  "page": {  
    "size": 20,  
    "totalElements": 5,  
    "totalPages": 1,  
    "number": 0  
  }  
}
```

Get list of employees

JSON Array of employees

Response meta-data
Information about the page

More on
configuring page
sizes later

The diagram illustrates a REST response JSON object. A red dashed box highlights the `_embedded` and `page` fields. An arrow points from the `employees` array to a purple box labeled "JSON Array of employees". Another arrow points from the `page` field to a gold box labeled "Response meta-data Information about the page". A callout bubble on the right says "Get list of employees".

HATEOAS

- For details on HATEOAS, see

<https://spring.io/understanding/HATEOAS>

- HATEOAS uses Hypertext Application Language (HAL) data format
- For details on HAL, see

https://en.wikipedia.org/wiki/Hypertext_Application_Language

Advanced Features

- Spring Data REST advanced features
 - Pagination, sorting and searching

<https://spring.io/projects/spring-data-rest>

Spring Data REST

Configuration, Pagination and Sorting

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> {  
}
```

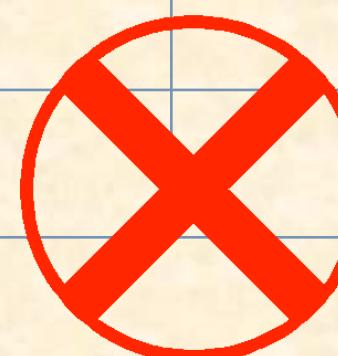


/employees

Pluralized Form

- Spring Data REST pluralized form is VERY simple
 - Just adds an "s" to the entity
- The English language is VERY complex!
 - Spring Data REST does NOT handle

Singular	Plural
Goose	Geese
Person	People
Syllabus	Syllabi
...	...



Problem

- Spring Data REST does not handle complex pluralized forms
 - In this case, you need to specify plural name
- What if we want to expose a different resource name?
 - Instead of `/employees` ... use `/members`

Solution

- Specify plural name / path with an annotation

```
@RepositoryRestResource(path="members")
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {
}
```

<http://localhost:8080/members>

Pagination

- By default, Spring Data REST will return the first 20 elements
 - Page size = 20
- You can navigate to the different pages of data using query param

```
http://localhost:8080/employees?page=0
```

```
http://localhost:8080/employees?page=1
```

...



Pages are
zero-based

Spring Data REST Configuration

- Following properties available: application.properties

Name	Description
<code>spring.data.rest.base-path</code>	Base path used to expose repository resources
<code>spring.data.rest.default-page-size</code>	Default size of pages
<code>spring.data.rest.max-page-size</code>	Maximum size of pages
...	...

`spring.data.rest.*`

Sample Configuration

http://localhost:8080/magic-api/employees

File: application.properties

```
spring.data.rest.base-path=/magic-api
```

```
spring.data.rest.default-page-size=50
```

Returns 50
elements per page

Sorting

- You can sort by the property names of your entity
 - In our Employee example, we have: **firstName**, **lastName** and **email**
- Sort by last name (ascending is default)
`http://localhost:8080/employees?sort=lastName`
- Sort by first name, descending
`http://localhost:8080/employees?sort=firstName,desc`
- Sort by last name, then first name, ascending
`http://localhost:8080/employees?sort=lastName,firstName,asc`

Spring Boot (REST API, MVC and Microservices)

Spring Security Overview



Agenda

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

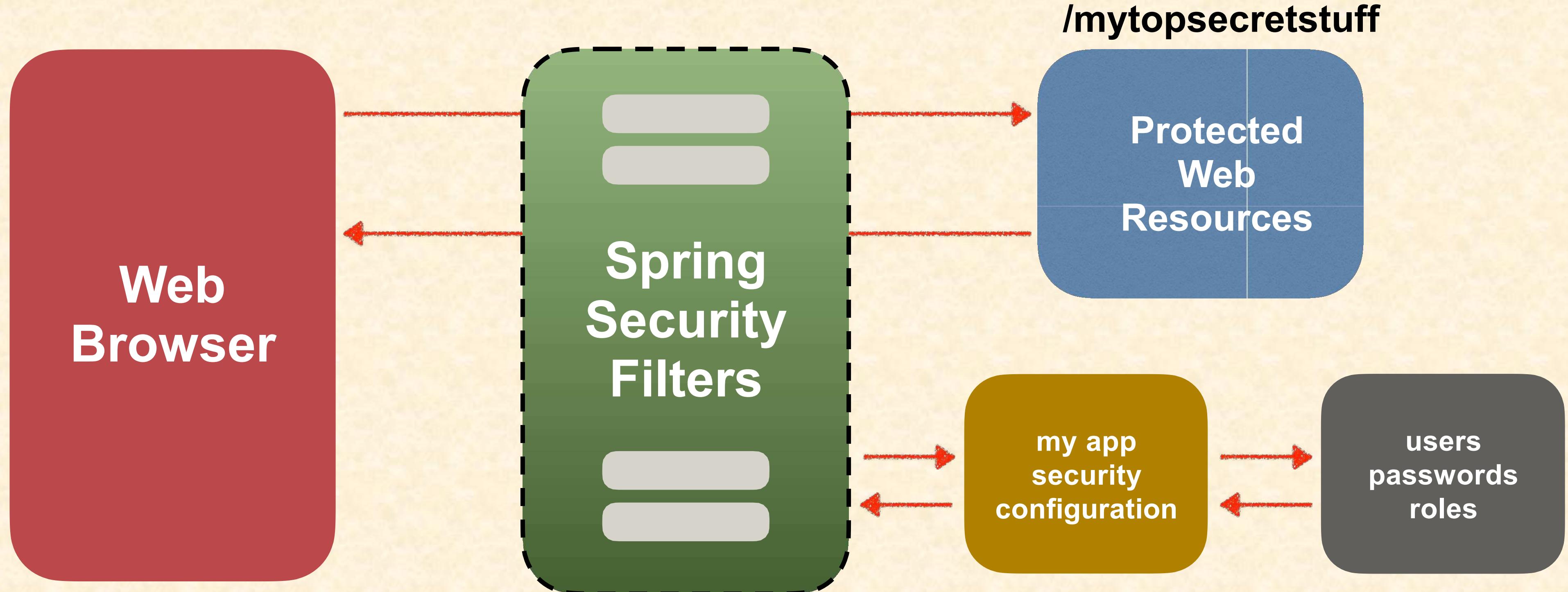
Spring Security Model

- Spring Security defines a framework for security
- Implemented using Servlet filters in the background

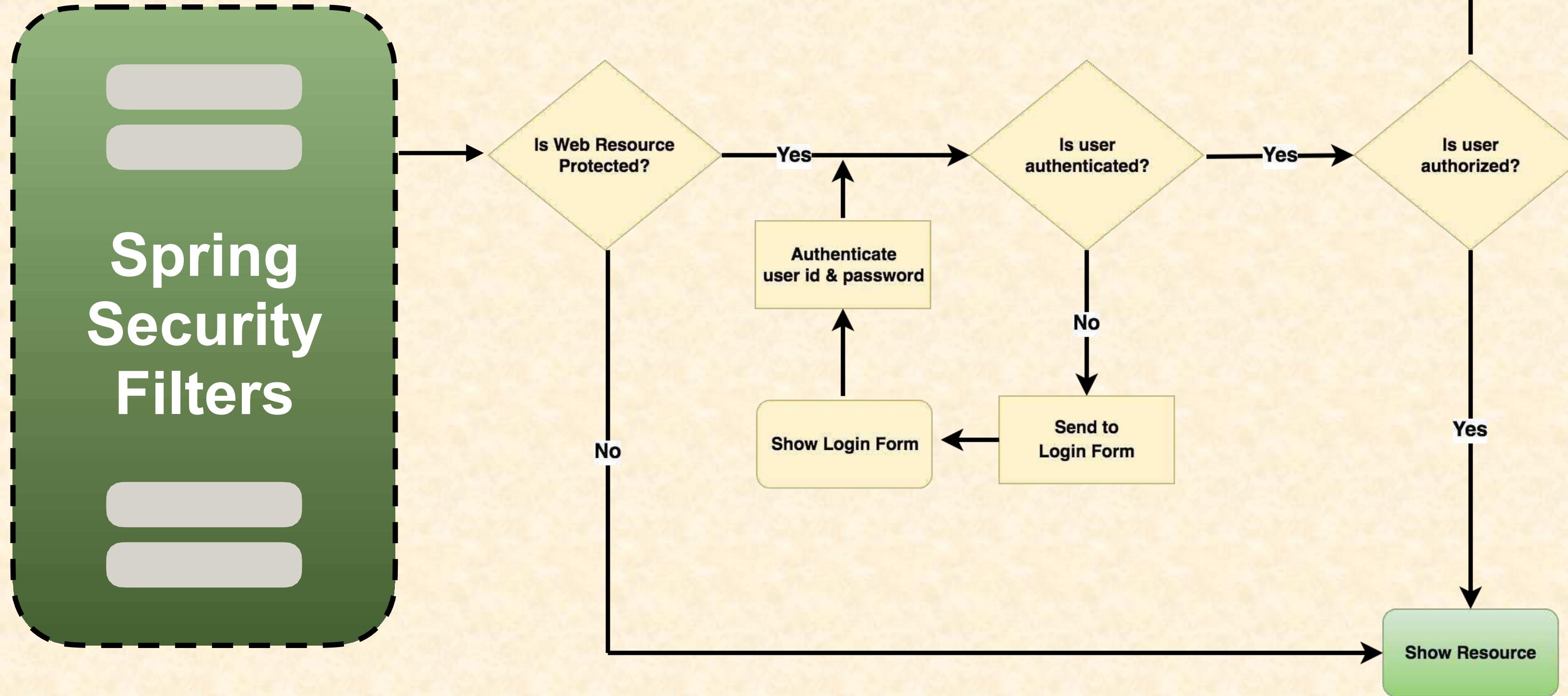
Spring Security with Servlet Filters

- Servlet Filters are used to pre-process / post-process web requests
- Servlet Filters can route web requests based on security logic
- Spring provides a bulk of security functionality with servlet filters

Spring Security Overview



Spring Security - Flow of 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

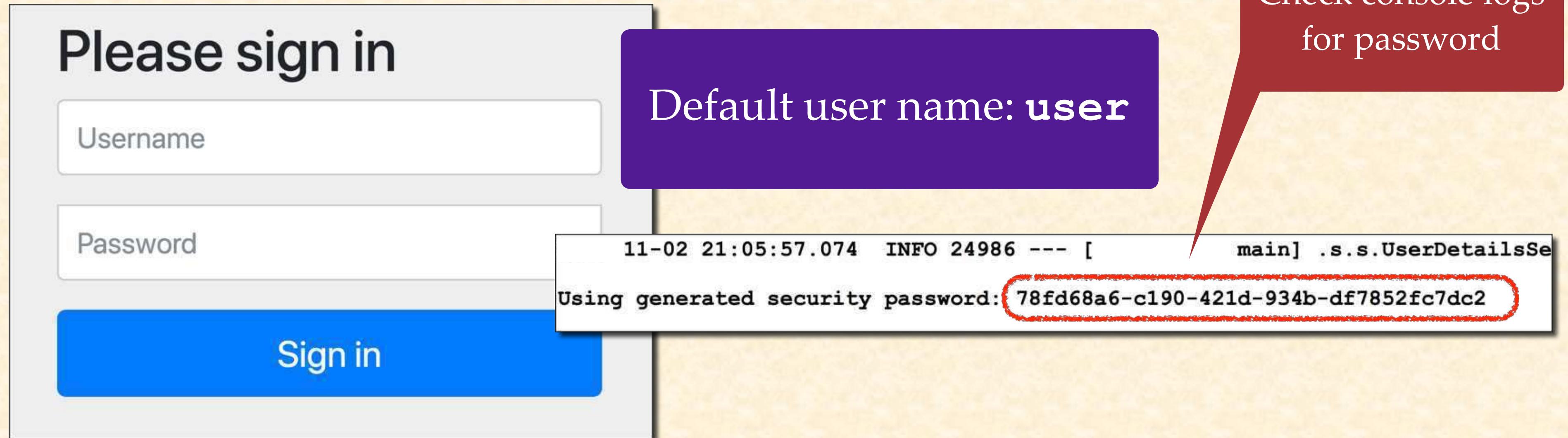
1. Edit `pom.xml` and add `spring-boot-starter-security`

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

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

Secured Endpoints

- Now when you access your application
- Spring Security will prompt for login



Spring Security configuration

- You can override default user name and generated password

src/main/resources/application.properties

```
spring.security.user.name=scott  
spring.security.user.password=test123
```

Authentication and Authorization

- In-memory
- JDBC
- LDAP
- Custom / Pluggable
- *others* ...



We will cover password storage in DB as plain-text AND encrypted

Demo

Spring Boot (REST API, MVC and Microservices)

REST API - Configuring Basic Security

Configuring Basic Security

Our Users

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE , MANAGER
susan	test123	EMPLOYEE , MANAGER , ADMIN

We can give ANY names
for user roles

Development Process

1. Create Spring Security Configuration (@Configuration)
2. Add users, passwords and roles

Step 1: Create Spring Security Configuration

DemoSecurityConfig.java

```
import org.springframework.context.annotation.Configuration;

@Configuration
public class DemoSecurityConfig {

    // add our security configurations here ...

}
```

Spring Security Password Storage

- In Spring Security, passwords are stored using a specific format

```
{ id}encodedPassword
```

ID	Description
noop	Plain text passwords
bcrypt	BCrypt password hashing
...	...

Password Example

The encoding
algorithm id

The password

{noop}test123

Let's Spring Security
know the passwords are
stored as plain text (noop)

Step 2: Add users, passwords and roles

DemoSecurityConfig.java

```
@Configuration
public class DemoSecurityConfig {

    @Bean
    public InMemoryUserDetailsManager userDetailsService() {

        UserDetails john = User.builder()
            .username("john")
            .password("{noop}test123")
            .roles("EMPLOYEE")
            .build();

        UserDetails mary = User.builder()
            .username("mary")
            .password("{noop}test123")
            .roles("EMPLOYEE", "MANAGER")
            .build();

        UserDetails susan = User.builder()
            .username("susan")
            .password("{noop}test123")
            .roles("EMPLOYEE", "MANAGER", "ADMIN")
            .build();

        return new InMemoryUserDetailsManager(john, mary, susan);
    }
}
```

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

We will add DB support
later
(plaintext and encrypted)

Restrict Access Based on Roles

Our Example

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	Read all	EMPLOYEE
GET	/api/employees/{employeeId}	Read single	EMPLOYEE
POST	/api/employees	Create	MANAGER
PUT	/api/employees	Update	MANAGER
DELETE	/api/employees/{employeeId}	Delete employee	ADMIN

Restricting Access to Roles

- General Syntax

Restrict access to a
given path
“/api/employees”

```
requestMatchers(<< add path to match on >>)
    .hasRole(<< authorized role >>)
```

Single role

“ADMIN”

Restricting Access to Roles

Specify HTTP method:
GET, POST, PUT, DELETE ...

Restrict access to a
given path
“/api/employees”

```
requestMatchers(<< add HTTP METHOD to match on >>, << add path to match on >>)  
.hasRole(<< authorized roles >>)
```

Single role

Restricting Access to Roles

```
requestMatchers(<< add HTTP METHOD to match on >>, << add path to match on  
>>)  
    .hasAnyRole(<< list of authorized roles >>)
```

Any role

Comma-delimited list

Authorize Requests for EMPLOYEE role

```
requestMatchers(HttpServletRequest.GET, "/api/employees").hasRole("EMPLOYEE")
requestMatchers(HttpServletRequest.GET, "/api/employees/**").hasRole("EMPLOYEE")
```

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	<u>Read all</u>	EMPLOYEE
GET	/api/employees/{employeeId}	<u>Read single</u>	EMPLOYEE
POST	/api/employees	<u>Create</u>	MANAGER
PUT	/api/employees	<u>Update</u>	MANAGER
DELETE	/api/employees/{employeeId}	<u>Delete employee</u>	ADMIN

The ****** syntax:
match on all sub-paths

Authorize Requests for MANAGER role

```
requestMatchers(HttpServletRequest.POST, "/api/employees").hasRole("MANAGER")
requestMatchers(HttpServletRequest.PUT, "/api/employees").hasRole("MANAGER")
```

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	<u>Read</u> all	EMPLOYEE
GET	/api/employees/{employeeId}	<u>Read</u> single	EMPLOYEE
POST	/api/employees	<u>Create</u>	MANAGER
PUT	/api/employees	<u>Update</u>	MANAGER
DELETE	/api/employees/{employeeId}	<u>Delete</u> employee	ADMIN

Authorize Requests for ADMIN role

```
requestMatchers(HttpMethod.DELETE, "/api/employees/**").hasRole("ADMIN")
```

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	<u>Read all</u>	EMPLOYEE
GET	/api/employees/{employeeId}	<u>Read single</u>	EMPLOYEE
POST	/api/employees	<u>Create</u>	MANAGER
PUT	/api/employees	<u>Update</u>	MANAGER
DELETE	/api/employees/{employeeId}	<u>Delete employee</u>	ADMIN

Pull It Together

@Bean

```
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {  
  
    http.authorizeHttpRequests(configurer ->  
        configurer  
            .requestMatchers(HttpMethod.GET, "/api/employees").hasRole("EMPLOYEE")  
            .requestMatchers(HttpMethod.GET, "/api/employees/**").hasRole("EMPLOYEE")  
            .requestMatchers(HttpMethod.POST, "/api/employees").hasRole("MANAGER")  
            .requestMatchers(HttpMethod.PUT, "/api/employees").hasRole("MANAGER")  
            .requestMatchers(HttpMethod.DELETE, "/api/employees/**").hasRole("ADMIN"));  
  
    // use HTTP Basic authentication  
    http.httpBasic(Customizer.withDefaults());  
  
    return http.build();  
}
```

Use HTTP Basic Authentication

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN



HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	<u>Read all</u>	EMPLOYEE
GET	/api/employees/{employeeId}	<u>Read single</u>	EMPLOYEE
POST	/api/employees	<u>Create</u>	MANAGER
PUT	/api/employees	<u>Update</u>	MANAGER
DELETE	/api/employees/{employeeId}	<u>Delete employee</u>	ADMIN

Pull It Together

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(configurer ->
        configurer
            .requestMatchers(HttpMethod.GET, "/api/employees").hasRole("EMPLOYEE")
            .requestMatchers(HttpMethod.GET, "/api/employees/**").hasRole("EMPLOYEE")
            .requestMatchers(HttpMethod.POST, "/api/employees").hasRole("MANAGER")
            .requestMatchers(HttpMethod.PUT, "/api/employees").hasRole("MANAGER")
            .requestMatchers(HttpMethod.DELETE, "/api/employees/**").hasRole("ADMIN"));

    // use HTTP Basic authentication
    http.httpBasic(Customizer.withDefaults());

    // disable Cross Site Request Forgery (CSRF)
    http.csrf(csrf -> csrf.disable());

    return http.build();
}
```

In general, CSRF is not required for stateless REST APIs that use POST, PUT, DELETE and/or PATCH

Cross-Site Request Forgery (CSRF)

- Spring Security can protect against CSRF attacks
- Embed additional authentication data/token into all HTML forms
- On subsequent requests, web app will verify token before processing
- Primary use case is traditional web applications (HTML forms etc ...)

When to use CSRF Protection?

- The Spring Security team recommends
 - Use CSRF protection for any normal browser web requests
 - Traditional web apps with HTML forms to add/modify data
- If you are building a REST API for non-browser clients
 - you *may* want to disable CSRF protection
- In general, not required for stateless REST APIs
 - That use POST, PUT, DELETE and/or PATCH

Demo

Spring Boot (REST API, MVC and Microservices)

Spring Security - User Accounts Stored in DB



Spring Security

User Accounts Stored in Database

Database Access

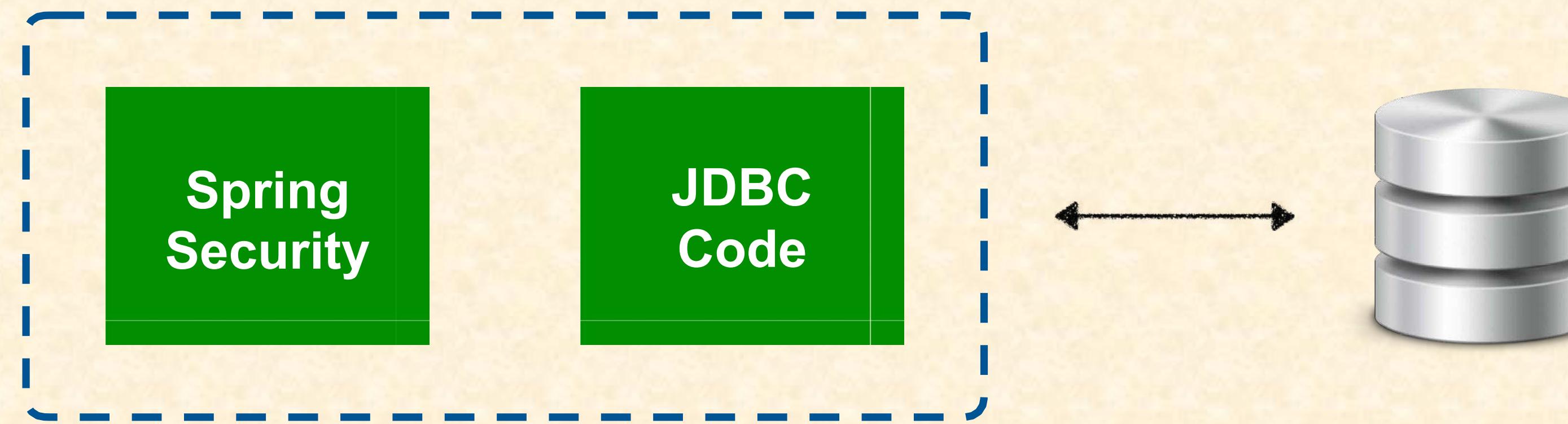
- So far, our user accounts were hard coded in Java source code
- We want to add database access

Recall Our User Roles

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE , MANAGER
susan	test123	EMPLOYEE , MANAGER , ADMIN

Database Support in Spring Security

- Spring Security can read user account info from database
- By default, we have to follow Spring Security's predefined table schemas

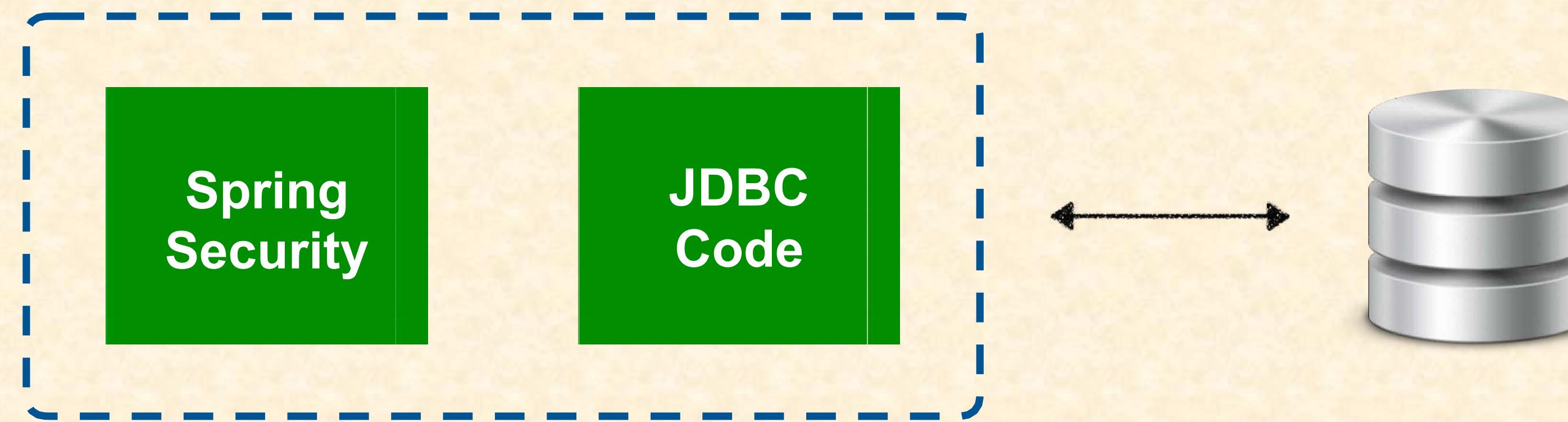


Customize Database Access with Spring Security

- We can also customize the table schemas
- Useful if we have custom tables specific to our project
- We will be responsible for developing the code to access the data
 - JDBC, JPA/Hibernate etc ...

Database Support in Spring Security

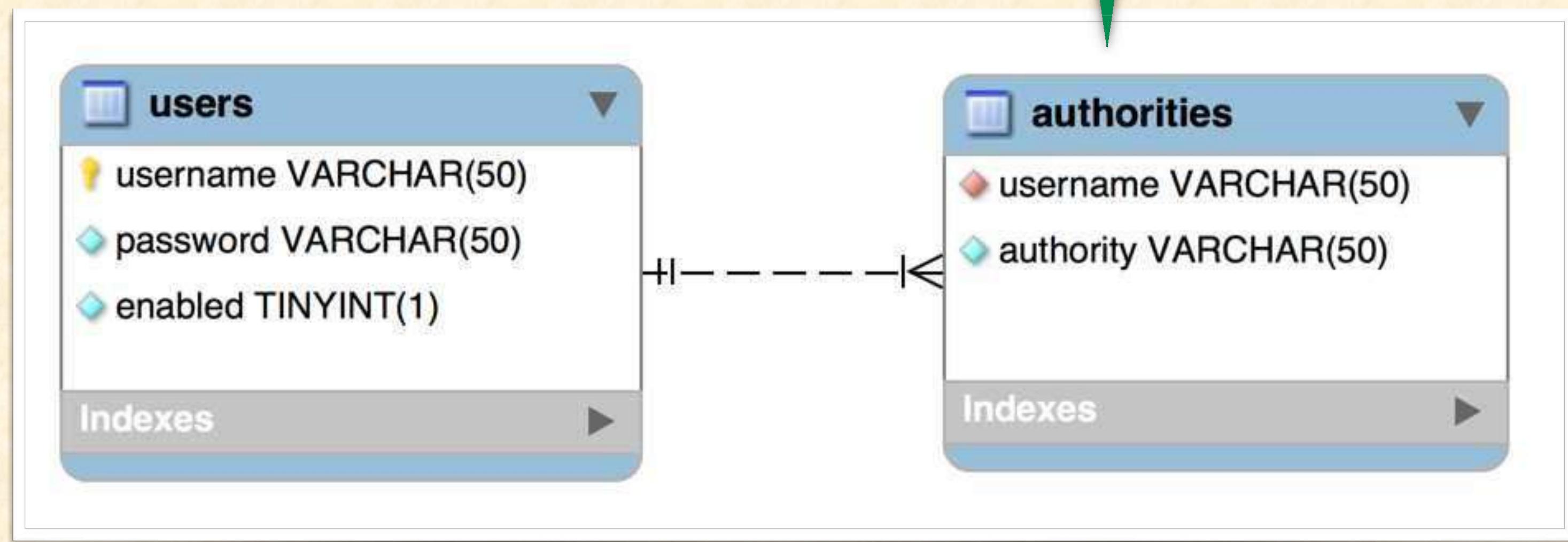
- Follow Spring Security's predefined table schemas



Development Process

1. Develop SQL Script to set up database tables
2. Add database support to Maven POM file
3. Create JDBC properties file
4. Update Spring Security Configuration to use JDBC

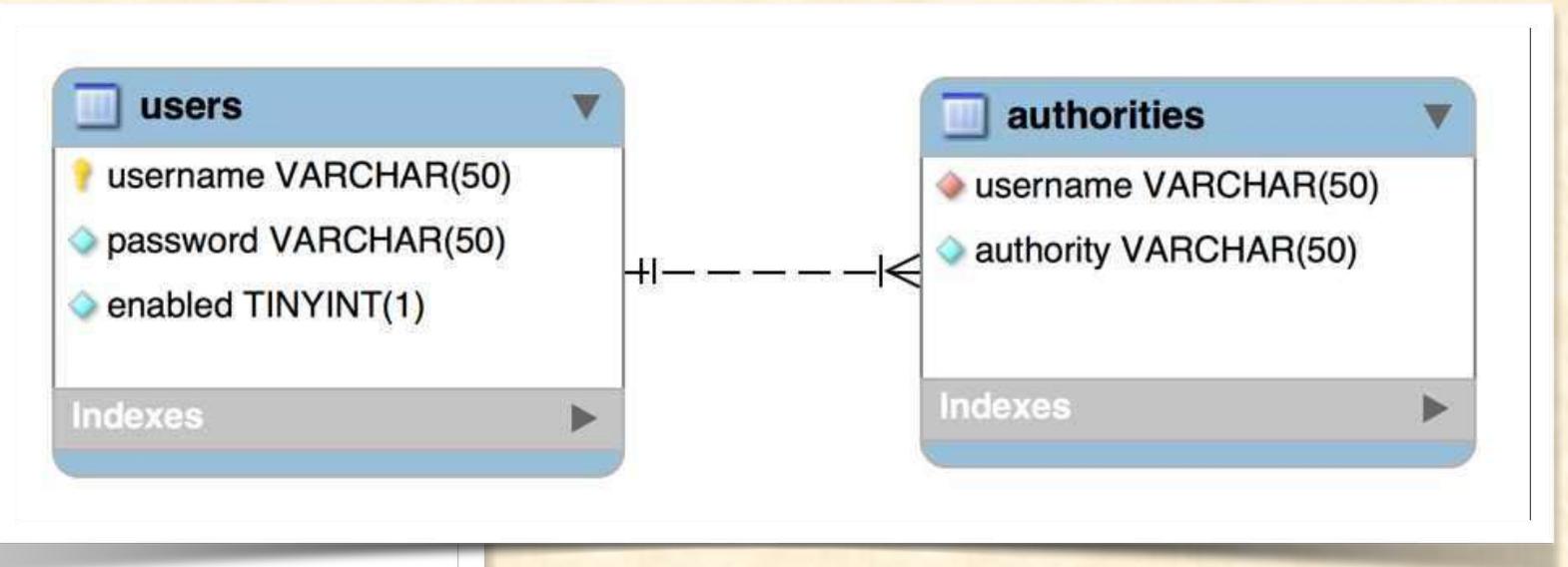
Default Spring Security Database Schema



“authorities” same as “roles”

Step 1: Develop SQL Script to setup database tables

```
CREATE TABLE `users` (
  `username` varchar(50) NOT NULL,
  `password` varchar(50) NOT NULL,
  `enabled` tinyint NOT NULL,
  PRIMARY KEY (`username`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```



Step 1: Develop SQL Script to setup database tables

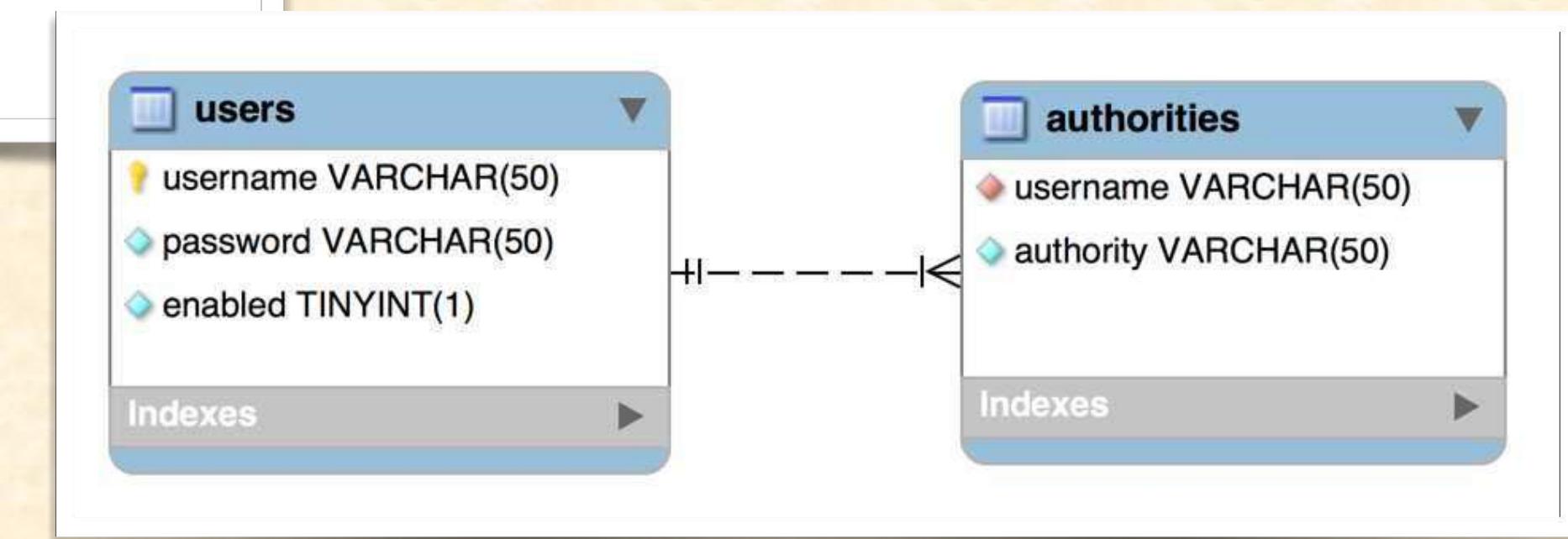
The encoding algorithm id

```
INSERT INTO `users`  
VALUES  
('john', 'noop{test123}', 1),  
('mary', 'noop{test123}', 1),  
('susan', 'noop{test123}', 1);
```

The password

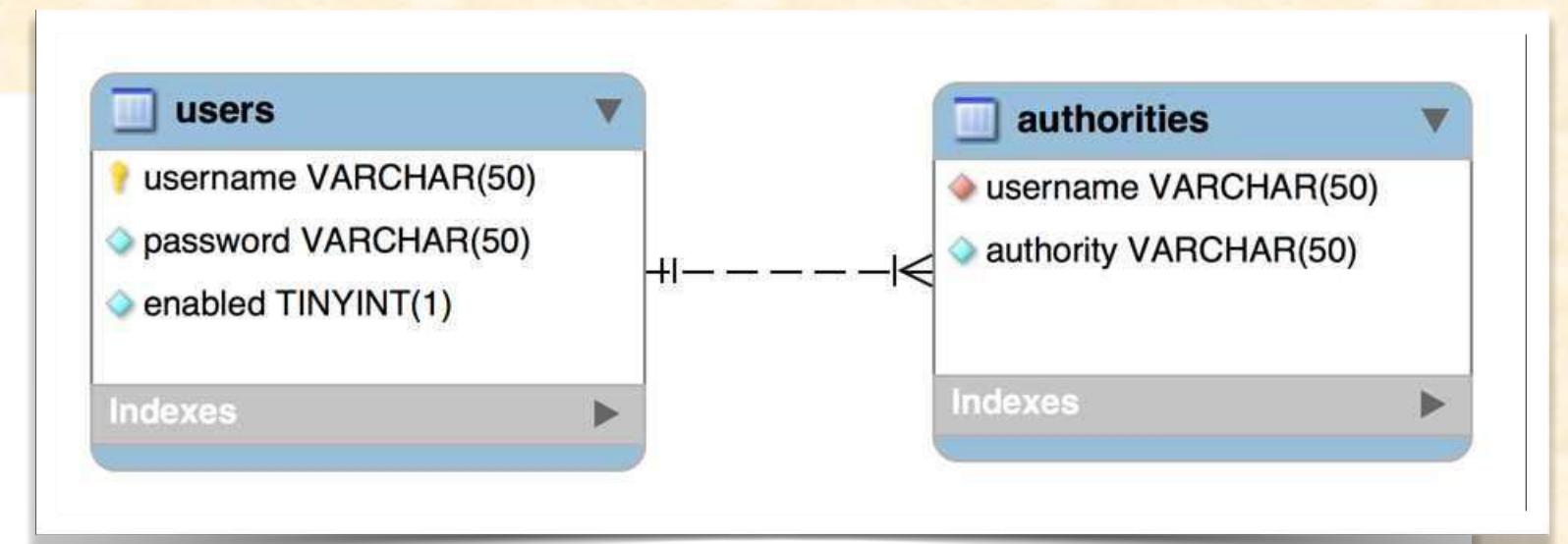
User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

Let's Spring Security know the passwords are stored as plain text (noop)



Step 1: Develop SQL Script to setup database tables

```
CREATE TABLE `authorities` (
  `username` varchar(50) NOT NULL,
  `authority` varchar(50) NOT NULL,
  UNIQUE KEY `authorities_idx_1` (`username`, `authority`),
  CONSTRAINT `authorities_ibfk_1`
  FOREIGN KEY (`username`)
  REFERENCES `users` (`username`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```



Step 1: Develop SQL Script to setup database tables

“authorities” same as “roles”

```
INSERT INTO `authorities`  
VALUES  
('john', 'ROLE_EMPLOYEE'),  
('mary', 'ROLE_EMPLOYEE'),  
('mary', 'ROLE_MANAGER'),  
('susan', 'ROLE_EMPLOYEE'),  
('susan', 'ROLE_MANAGER'),  
('susan', 'ROLE_ADMIN');
```

Internally Spring Security uses
“ROLE_” prefix

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN



Step 2: Add Database Support to Maven POM file

```
<!-- MySQL -->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

JDBC Driver

Step 3: Create JDBC Properties File

File: application.properties

```
#  
# JDBC connection properties  
  
spring.datasource.url=jdbc:mysql://localhost:3306/employee_directory  
spring.datasource.username=springstudent  
spring.datasource.password=springstudent
```

Step 4: Update Spring Security to use JDBC

```
@Configuration  
public class DemoSecurityConfig {  
  
    @Bean  
    public UserDetailsService userDetailsManager(DataSource dataSource) {  
  
        return new JdbcUserDetailsManager(dataSource);  
    }  
    ...  
}
```

Tell Spring Security to use
JDBC authentication
with our data source

Inject data source
Auto-configured by Spring Boot

No longer
hard-coding users :-)

Demo

Spring Boot (REST API, MVC and Microservices)

Spring Security – Password Encryption



Spring Security Password Encryption

Password Storage

- So far, our user passwords are stored in plaintext ☺

username	password	enabled
john	{noop}test123	1
mary	{noop}test123	1
susan	{noop}test123	1

- Its ok for getting started, but not for production / real-time project

Password Storage - Best Practice

- The best practice is store passwords in an encrypted format

username	password	enabled
john	{bcrypt}\$2a\$10\$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQI9YeuspUdgF.q	1
mary	{bcrypt}\$2a\$10\$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQI9YeuspUdgF.q	1
susan	{bcrypt}\$2a\$10\$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQI9YeuspUdgF.q	1

Encrypted version of password

Spring Security Recommendation

- Spring Security recommends using the popular **bcrypt** algorithm
- **bcrypt**
 - Performs one-way encrypted hashing
 - Adds a random salt to the password for additional protection
 - Includes support to defeat brute force attacks

How to Get a Bcrypt password?

We have a plaintext password and we want to encrypt using bcrypt

- Option 1: Use a website utility to perform the encryption
- Option 2: Write Java code to perform the encryption

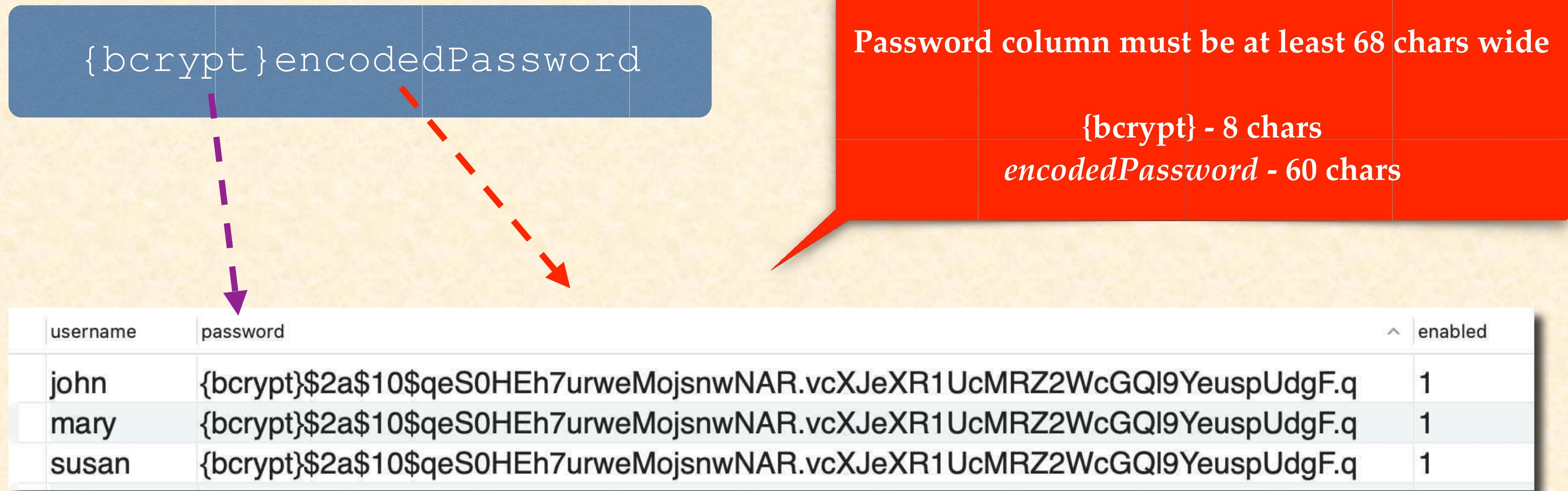
Development Process

1. Run SQL Script that contains encrypted passwords
 - Modify schema for password field, length should be 68

THAT'S IT ... no need to change Java source code

Spring Security Password Storage

- In Spring Security, passwords are stored using a specific format



Modify Users Schema for Password Field

```
CREATE TABLE `users` (
  `username` varchar(50) NOT NULL,
  `password` char(68) NOT NULL,
  `enabled` tinyint NOT NULL,
  PRIMARY KEY (`username`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Password column must be at least 68 chars wide

{bcrypt} - 8 chars

encodedPassword - 60 chars

Step 1: Develop SQL Script to setup database tables

The encoding algorithm id

```
INSERT INTO `users`  
VALUES  
( 'john' , '{bcrypt}'$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQ19YeuspUdgF.q' , 1)  
,
```

The encrypted password: fun123

Let's Spring Security know the passwords are stored as encrypted passwords: bcrypt

Step 1: Develop SQL Script to setup database tables

The encoding
algorithm id

```
INSERT INTO `users`  
VALUES  
('john' , '{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQ19YeuspUdgF.q' ,1)  
,  
('mary' , '{bcrypt}$2a$04$eFytJDGtjbThXa80FyOOBuFdK2IwjyWefYkMpIBEF1pBwDH.5PM0K' ,1),  
('susan' , '{bcrypt}$2a$04$eFytJDGtjbThXa80FyOOBuFdK2IwjyWefYkMpIBEF1pBwDH.5PM0K' ,1)  
;
```

The encrypted password: fun123

Spring Security Login Process



Spring Security Login Process



Note:
The password from db is
NEVER decrypted

1. Retrieve password from db for the user
2. Read the encoding algorithm id (bcrypt etc)
3. For the case of bcrypt, encrypt the plaintext password from login form (using salt)
4. Compare encrypted password from login form **WITH** encrypted password from db
5. If there is a match, login successful
6. If no match, login **NOT** successful

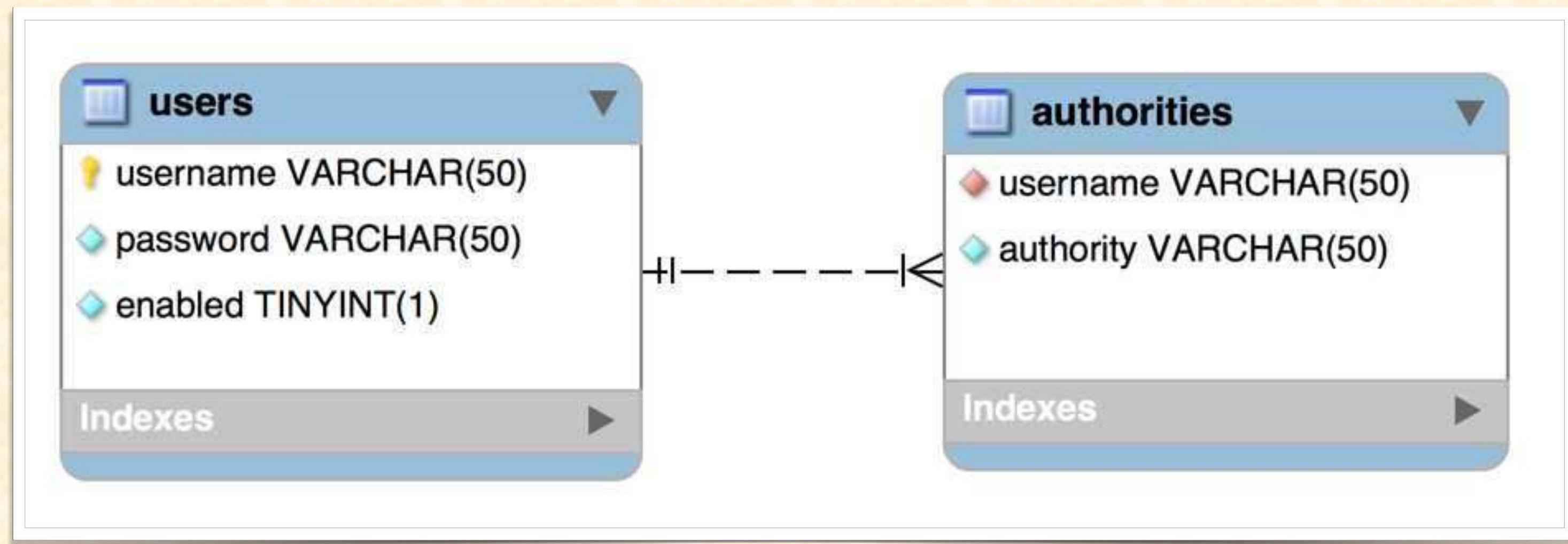
Because bcrypt is a
one-way
encryption algorithm

Spring Boot (REST API, MVC and Microservices)

Spring Security – Custom Tables

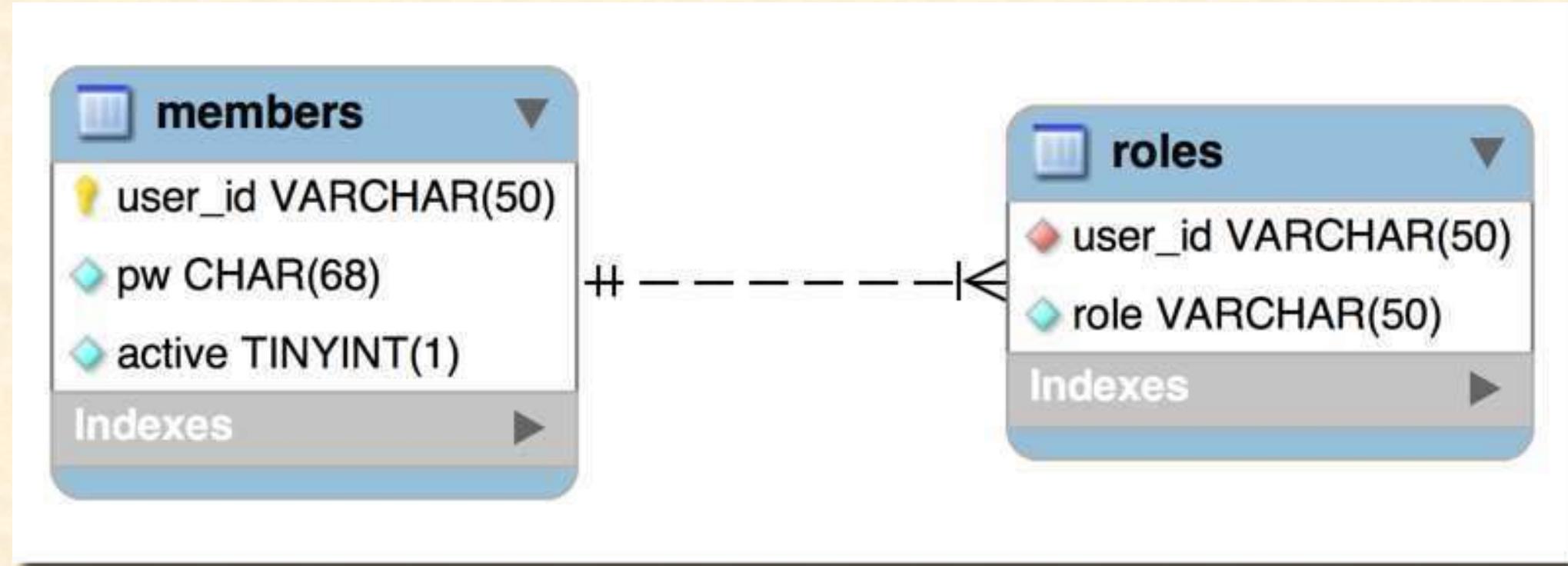
Spring Security Custom Tables

Default Spring Security Database Schema



Custom Tables

- What if we have our own custom tables?
- Our own custom column names?



This is all custom
Nothing matches with default Spring Security table schema

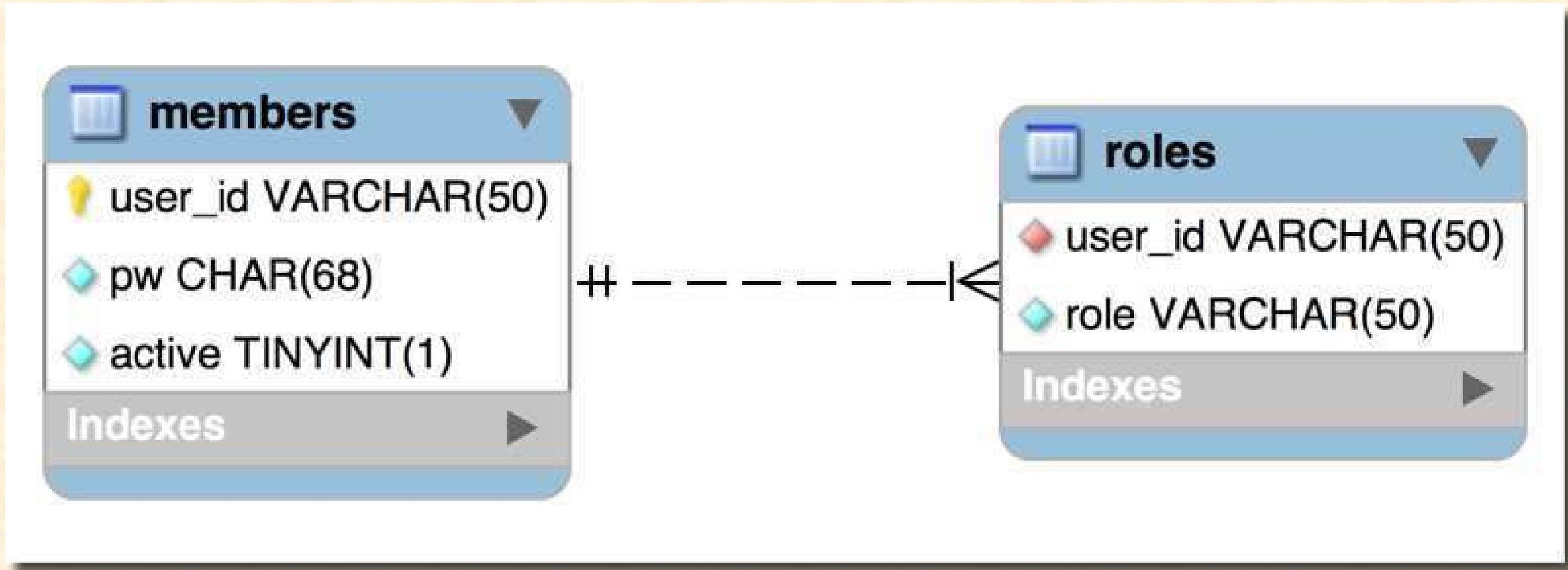
For Security Schema Customization

- Tell Spring how to query our custom tables
- Provide query to find user by user name
- Provide query to find authorities / roles by user name

Development Process

1. Create our custom tables with SQL
2. Update Spring Security Configuration
 - Provide query to find user by user name
 - Provide query to find authorities / roles by user name

Step 1: Create our custom tables with SQL



This is all custom
Nothing matches with default Spring Security table schema

Step 2: Update Spring Security Configuration

```
@Configuration
public class DemoSecurityConfig {

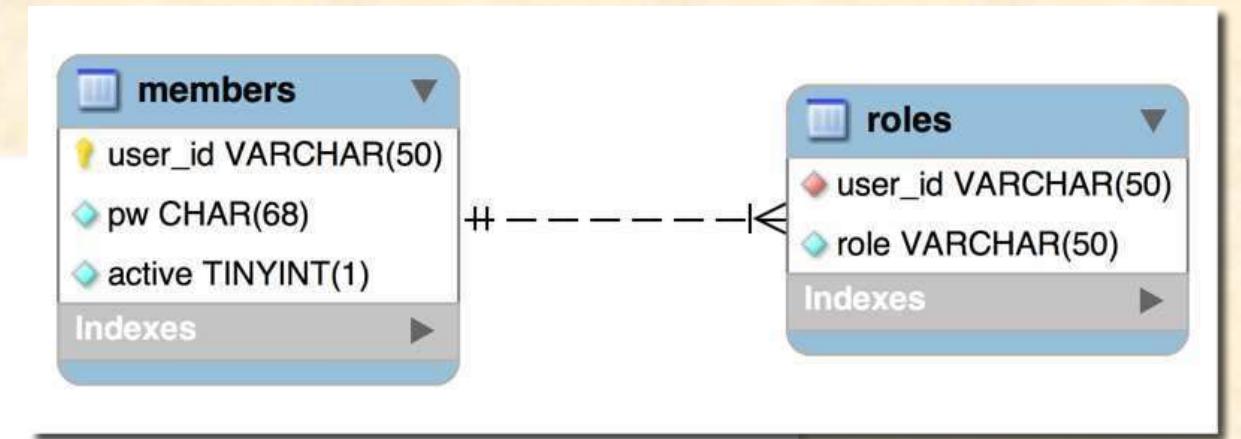
    @Bean
    public UserDetailsService userDetailsService(DataSource dataSource) {
        JdbcUserDetailsService theUserDetailsService = new JdbcUserDetailsService(dataSource);

        theUserDetailsService
            .setUsersByUsernameQuery("select user_id, pw, active from members where user_id=?");

        theUserDetailsService
            .setAuthoritiesByUsernameQuery("select user_id, role from roles where user_id=?");

        return theUserDetailsService;
    }

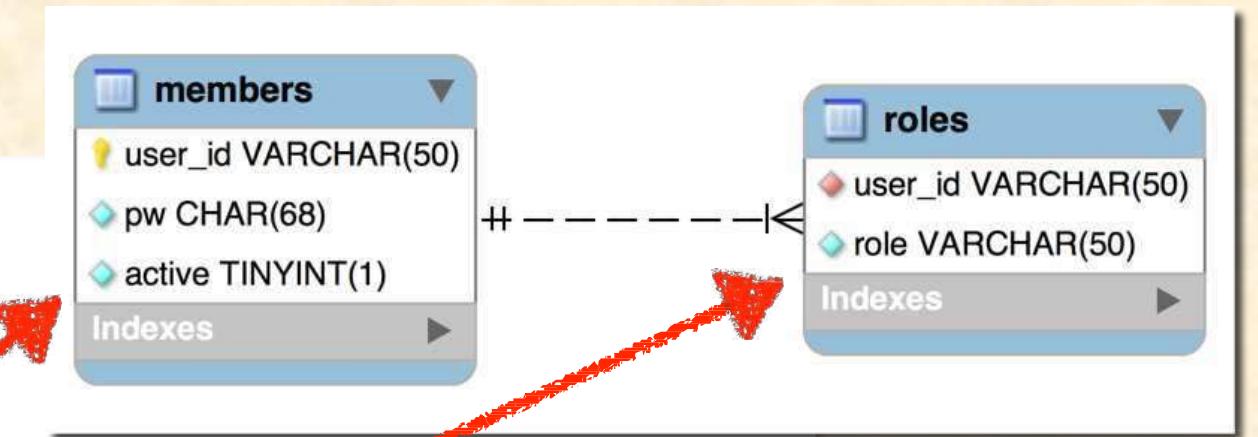
    ...
}
```



Step 2: Update Spring Security Configuration

```
@Configuration  
public class DemoSecurityConfig {  
  
    @Bean  
    public UserDetailsService userDetailsManager(DataSource dataSource) {  
        JdbcUserDetailsManager theUserDetailsManager = new JdbcUserDetailsManager(dataSource);  
  
        theUserDetailsManager  
            .setUsersByUsernameQuery("select user_id, pw, active from members where user_id=?");  
  
        theUserDetailsManager  
            .setAuthoritiesByUsernameQuery("select user_id, role from roles where user_id=?");  
  
        return theUserDetailsManager;  
    }  
  
    ...  
}
```

Question mark “?”
Parameter value will be the
user name from login



How to find users

How to find roles

Demo

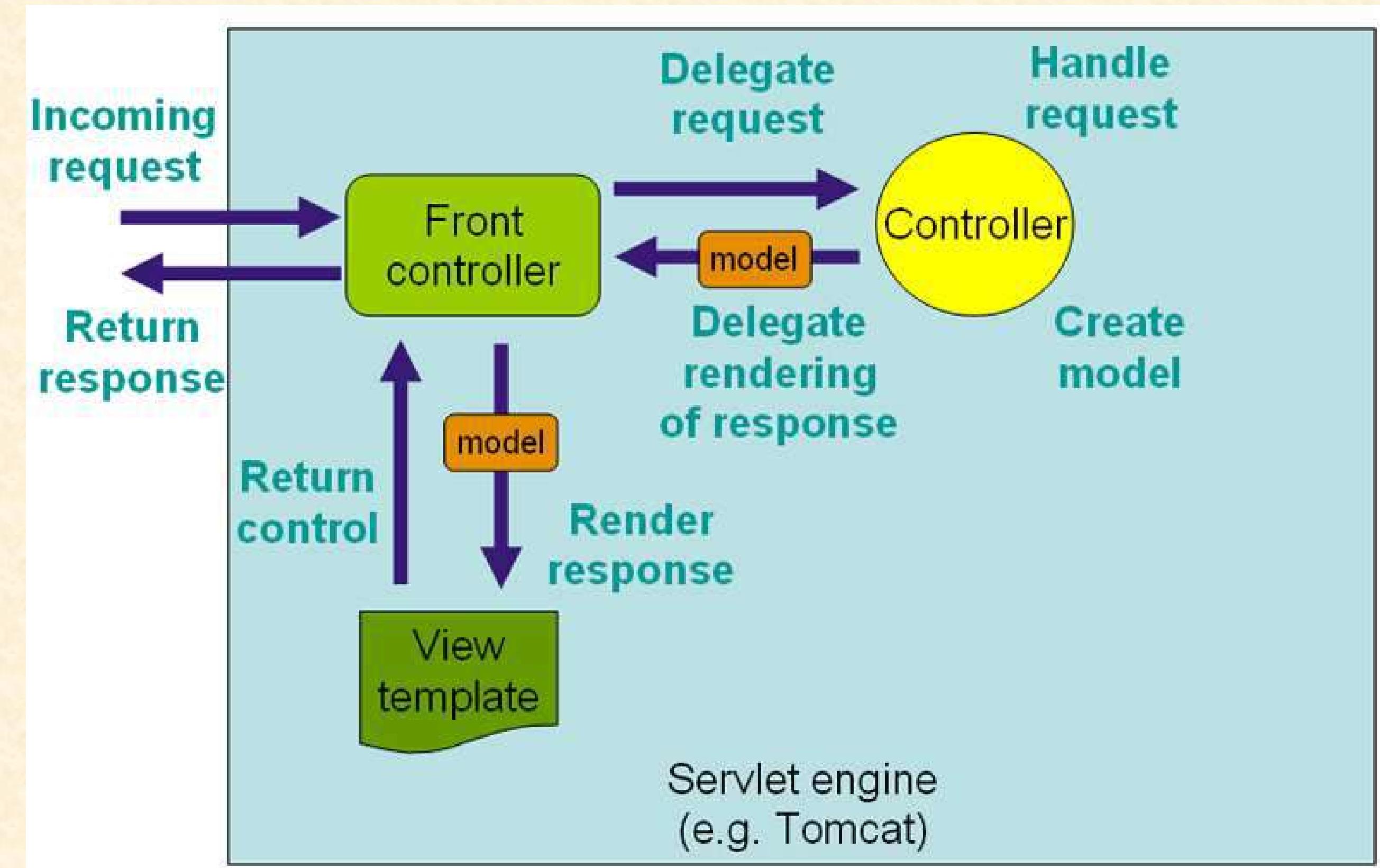
Spring Boot (REST API, MVC and Microservices)

Introduction to Spring MVC

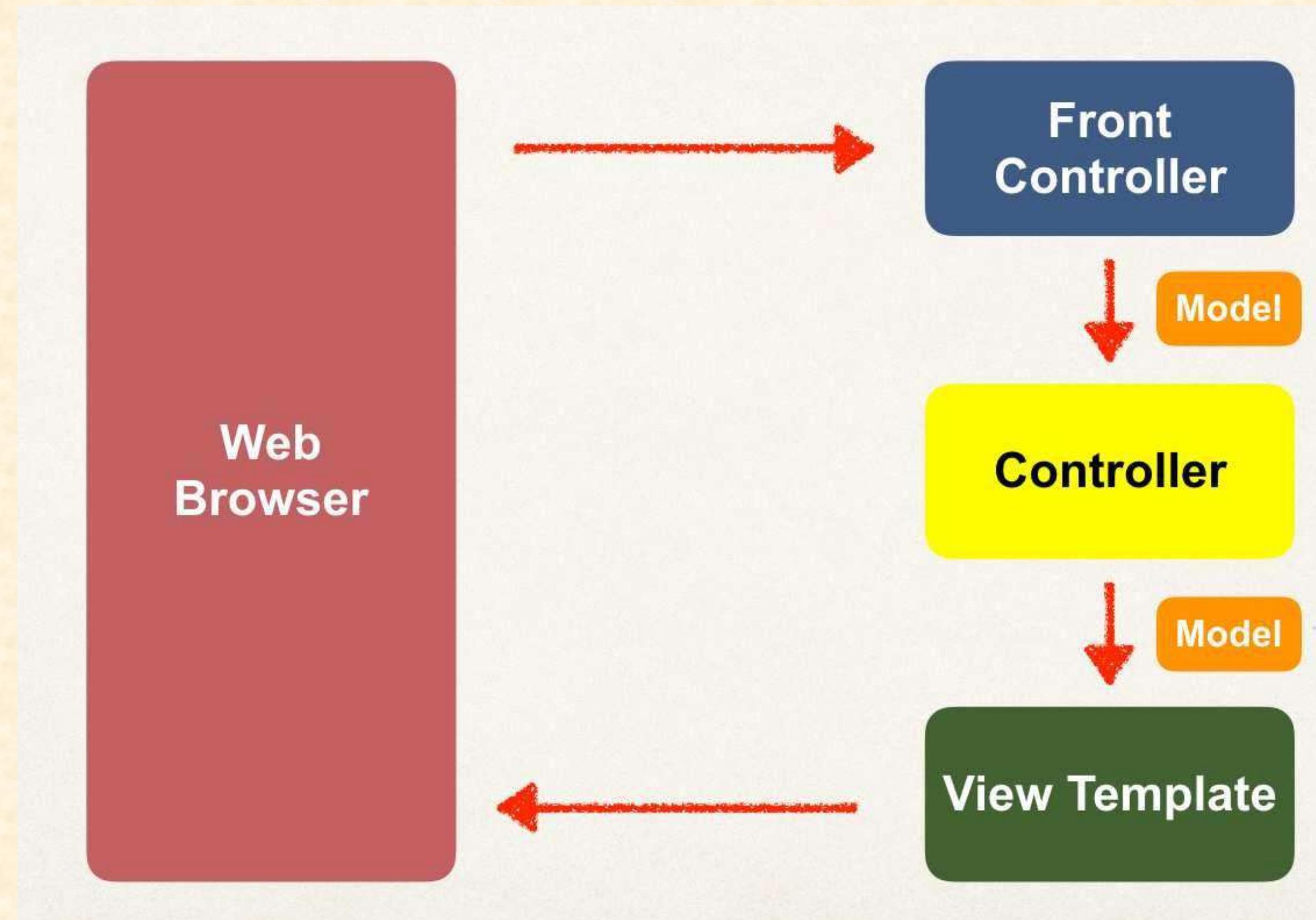


Spring MVC

Spring's web MVC framework is request-driven, designed around a central Servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications

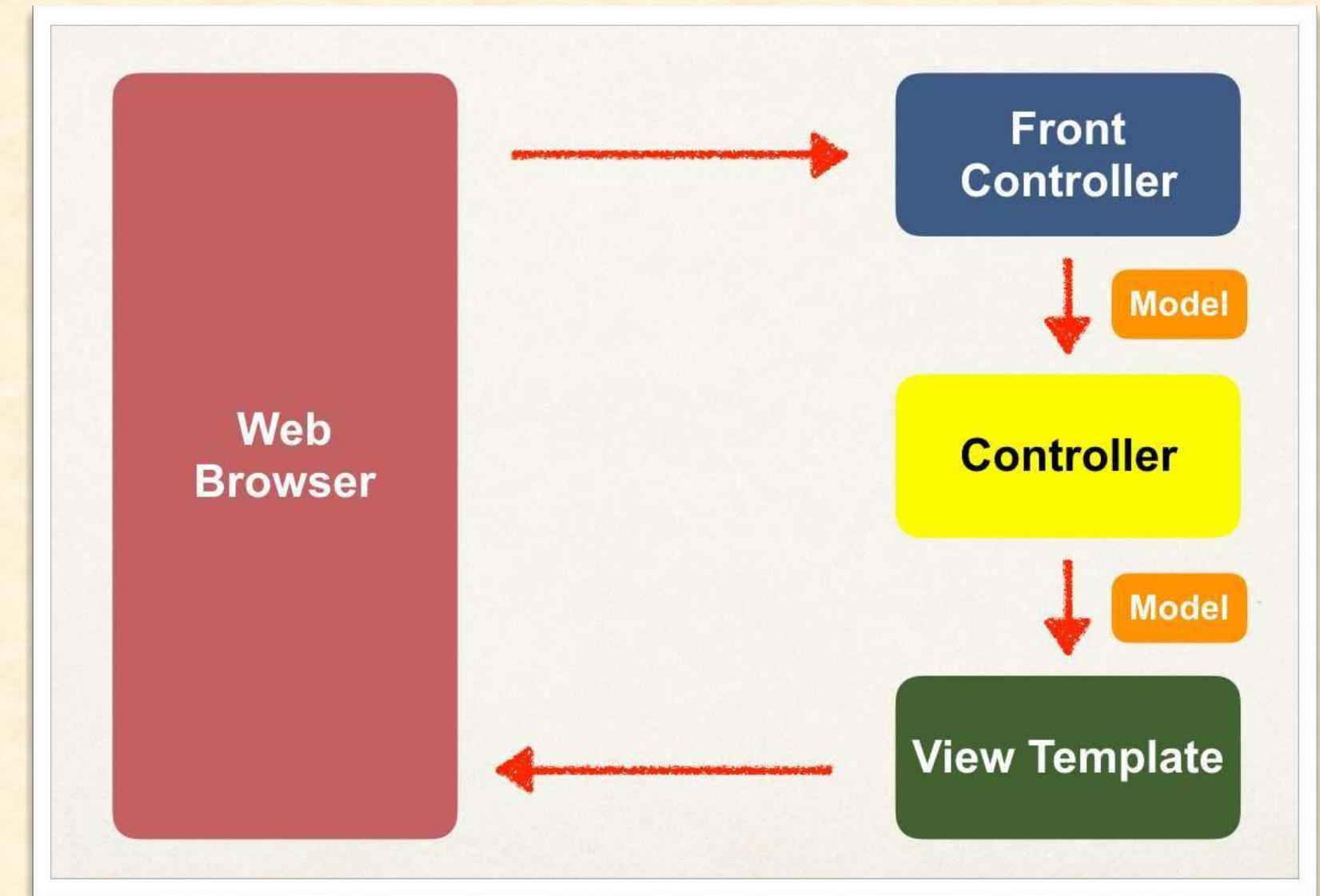


How Spring MVC Works Behind the Scenes?



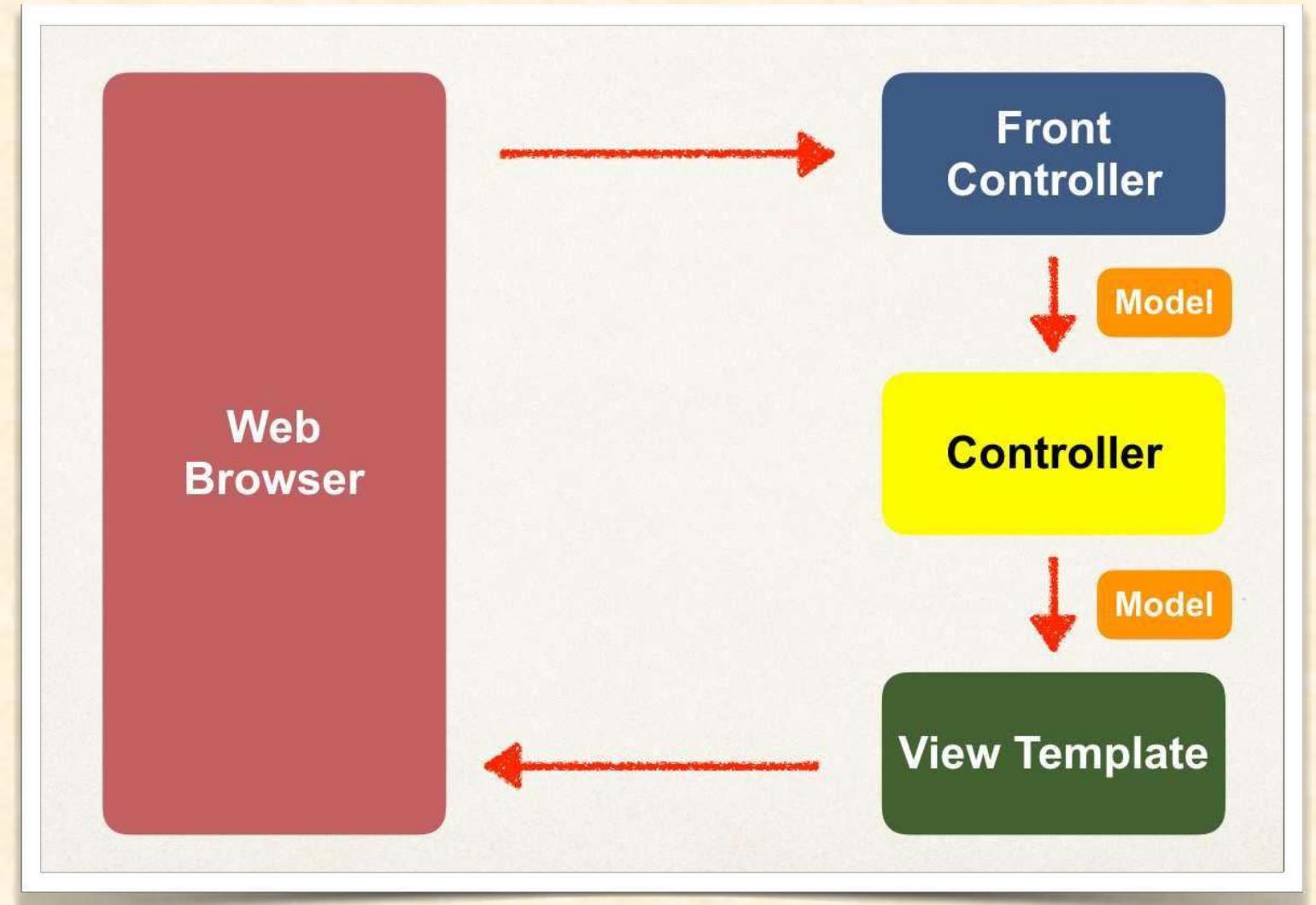
Spring MVC Front Controller

- Front controller known as **DispatcherServlet**
 - Part of the Spring Framework
 - Already developed by Spring Dev Team
- We will create
 - Model objects (orange)
 - View templates (dark green)
 - Controller classes (yellow)



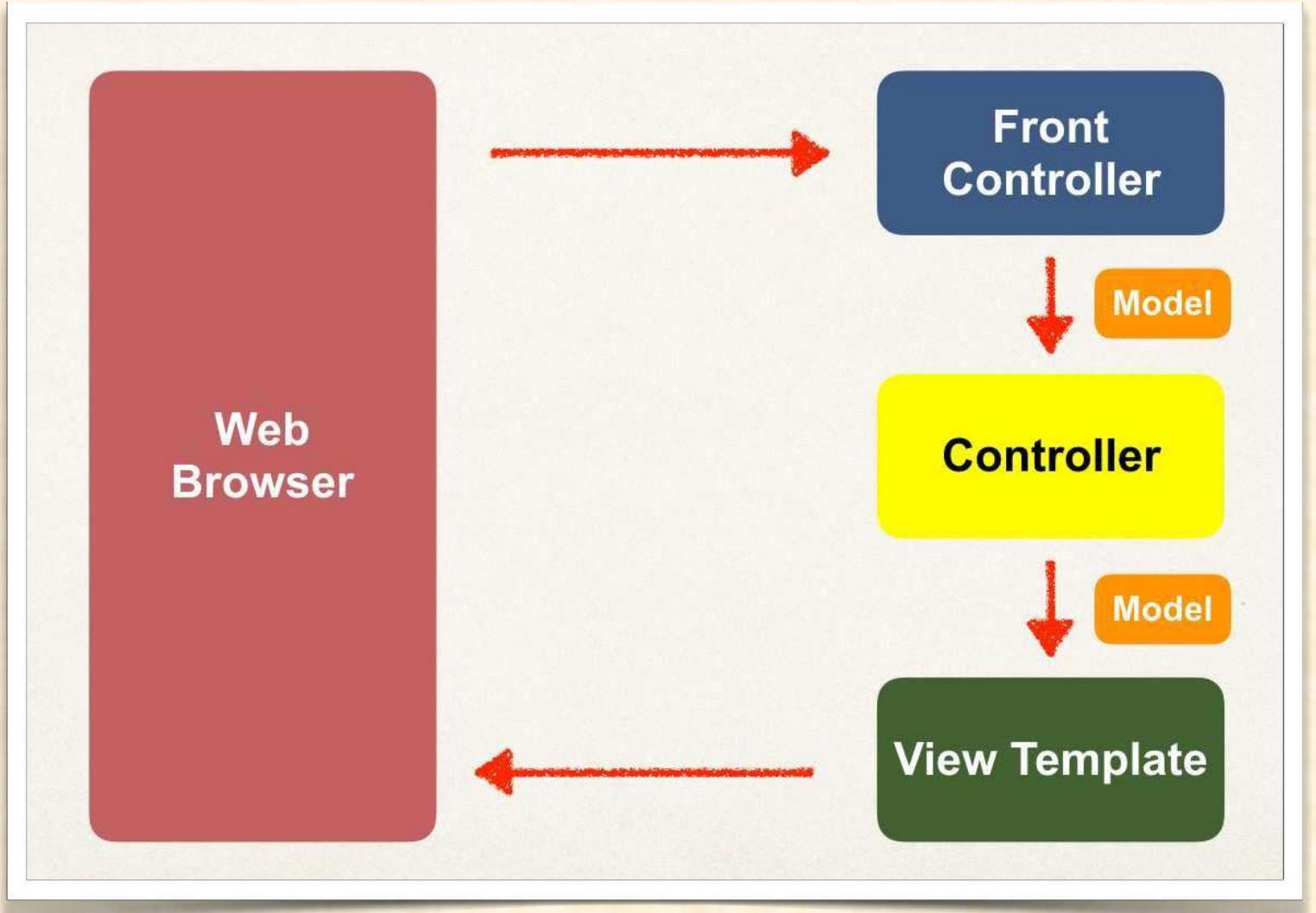
Controller

- Code created by developer
- Contains your business logic
 - Handle the request
 - Store/retrieve data (db, web service...)
 - Place data in model
- Send to appropriate view template



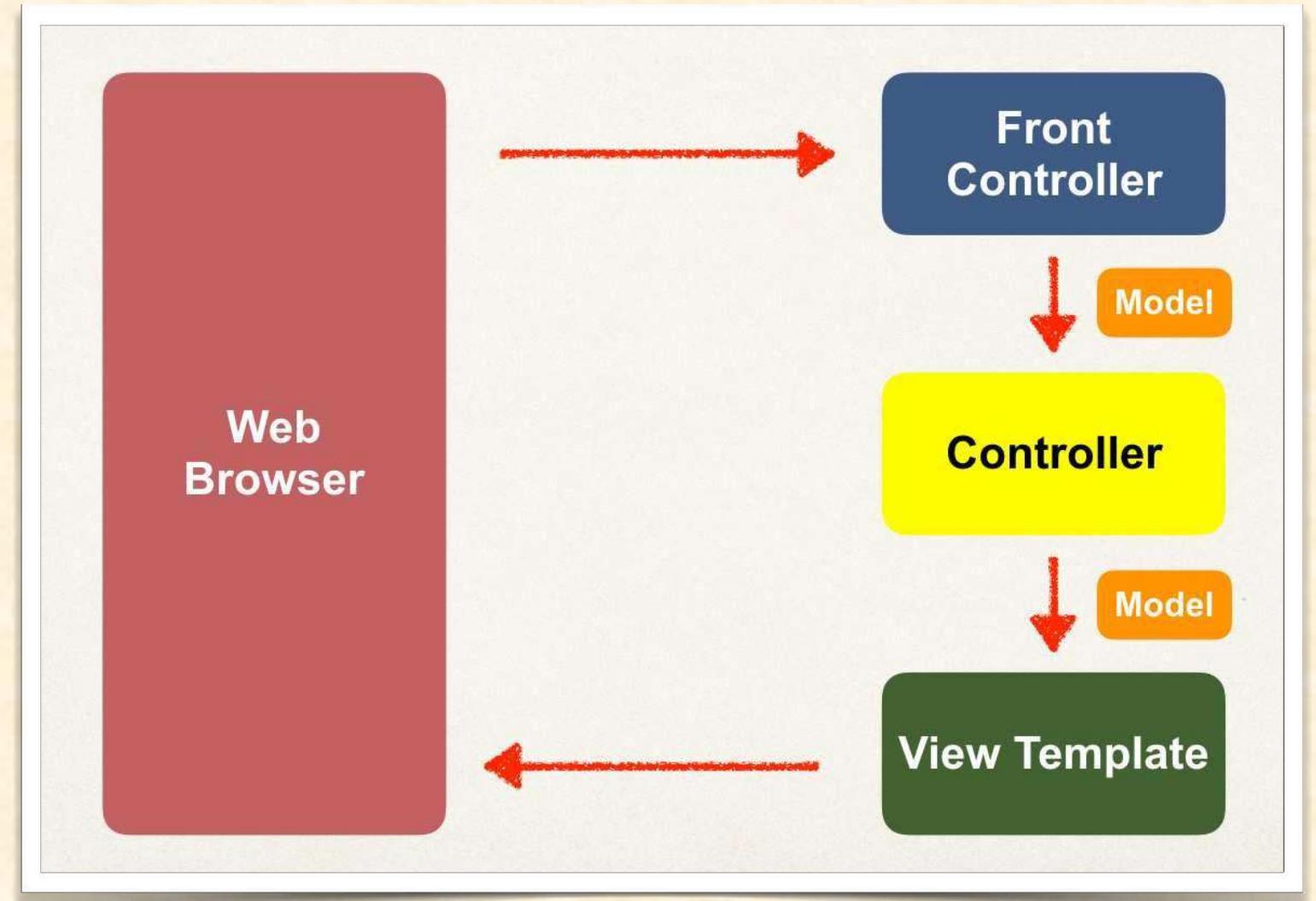
Model

- Model: contains your data
- Store/retrieve data via backend systems
 - database, web service, etc...
 - Use a Spring bean if required
- Place our data in the model
 - Data can be any Java object/collection



View Template

- Spring MVC is flexible
 - Supports many view templates
- Recommended: Thymeleaf
- Developer creates a page
 - Displays data



Components of a Spring MVC Application

- A set of web pages to layout UI components
- A collection of Spring beans (controllers, services, etc...)
- Spring configuration (XML, Annotations or Java)

Web
Pages

Beans

Spring
Configuration

Spring Boot (REST API, MVC and Microservices)

Thymeleaf with Spring Boot

What is Thymeleaf?



Thymeleaf

www.thymeleaf.org

- Thymeleaf is a Java template engine
- Commonly used to generate the HTML views for web apps
- However, it is a general purpose templating engine
 - We can use Thymeleaf outside of web apps

Separate project
Unrelated to spring.io

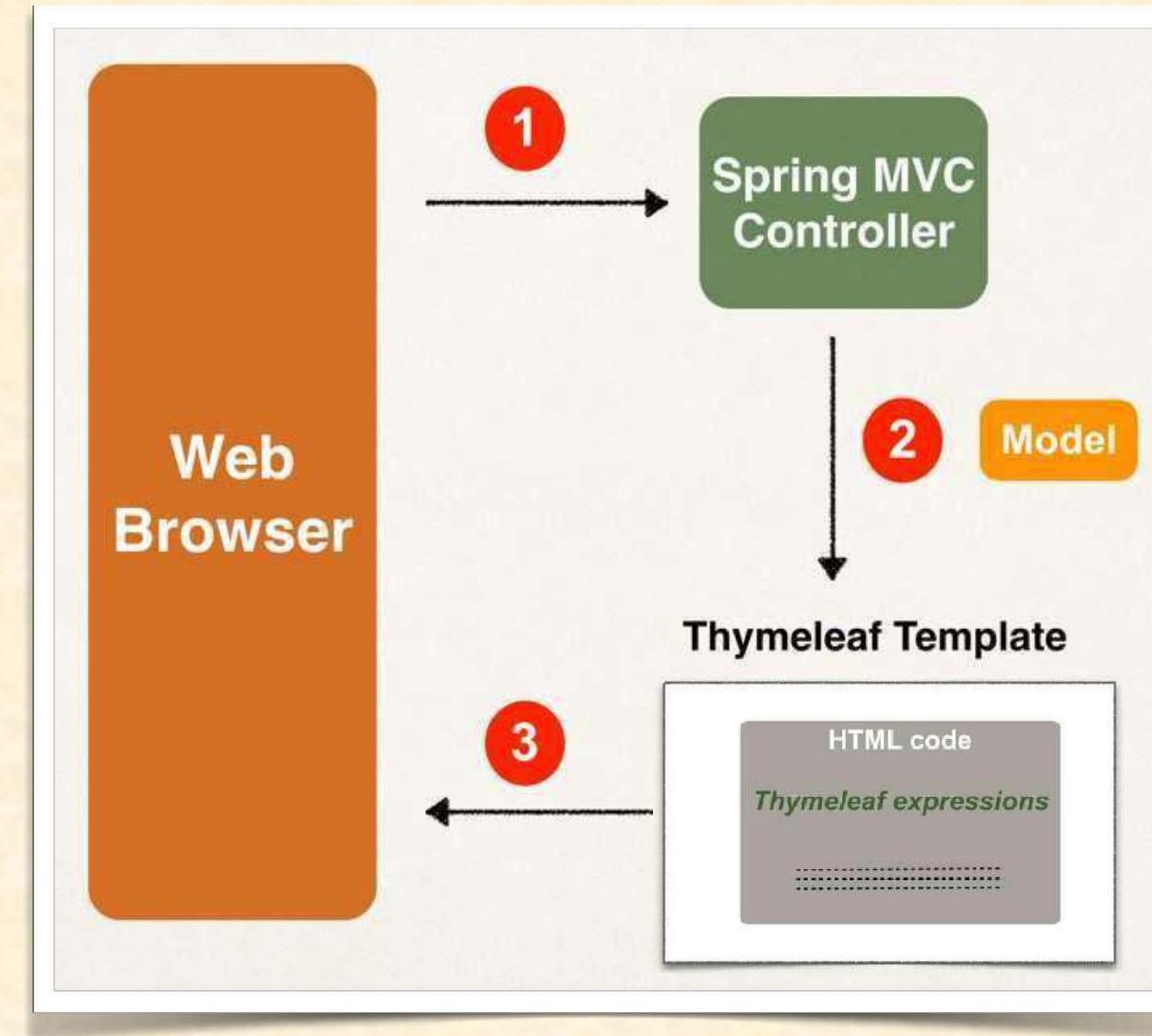
What is a Thymeleaf template?

- An HTML page with some Thymeleaf expressions
- Include dynamic content from Thymeleaf expressions

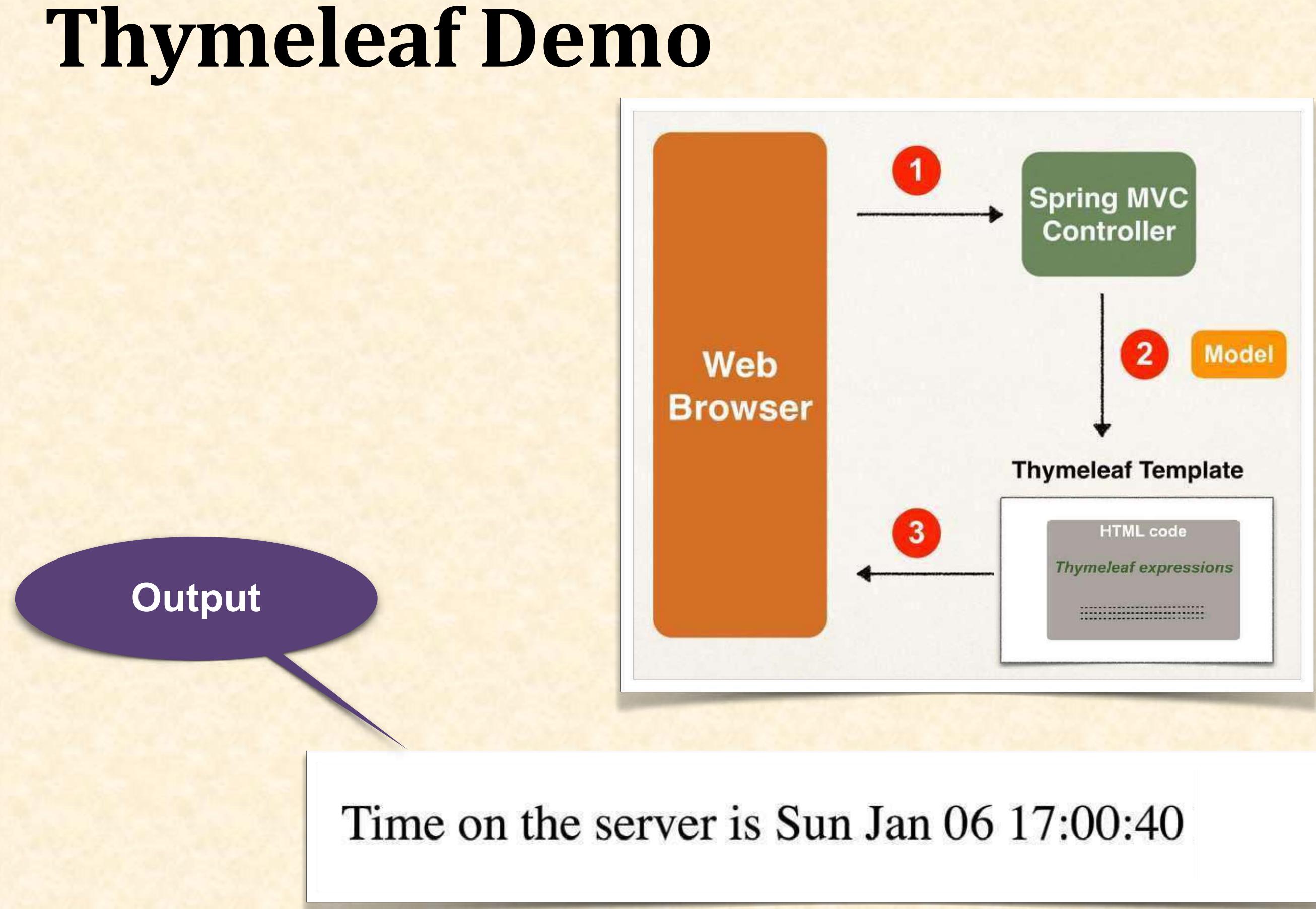


Where is the Thymeleaf template processed?

- In a web app, Thymeleaf is processed on the server
- Results included in HTML returned to browser



Thymeleaf Demo



Development Process

1. Add Thymeleaf to Maven POM file
2. Develop Spring MVC Controller
3. Create Thymeleaf template

Step 1: Add Thymeleaf to Maven pom file

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

Based on this,
Spring Boot will auto configure to
use Thymeleaf templates

Dependencies

Spring Web WEB

Build web, including RESTful, applications using Spring MVC.
Uses Apache Tomcat as the default embedded container.

Thymeleaf TEMPLATE ENGINES

A modern server-side Java template engine for both web and
standalone environments. Allows HTML to be correctly displayed
in browsers and as static prototypes.

Step 2: Develop Spring MVC Controller

File: DemoController.java

```
@Controller  
public class DemoController {  
    @GetMapping("/")  
    public String sayHello(Model theModel) {  
  
        theModel.addAttribute("theDate", new java.util.Date());  
  
        return "helloworld";  
    }  
}
```

src/main/resources/templates/helloworld.html

Where to place Thymeleaf template?

- In Spring Boot, your Thymeleaf template files go in
 - `src/main/resources/templates`
- For web apps, Thymeleaf templates have a `.html` extension

Step 3: Create Thymeleaf template

Thymeleaf accesses "theDate"
from the
Spring MVC Model

File: src/main/resources/templates/helloworld.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head> ... </head>

<body>
    <p th:text="Time on the server is ' + ${theDate} " />
</body>

</html>
```

Thymeleaf
expression

File: DemoController.java

```
@Controller
public class DemoController {

    @GetMapping("/")
    public String sayHello(Model theModel) {

        theModel.addAttribute("theDate", new
        java.util.Date());
        return "helloworld";
    }
}
```

1

2

Time on the server is Sun Jan 06 17:00:40

Additional Features

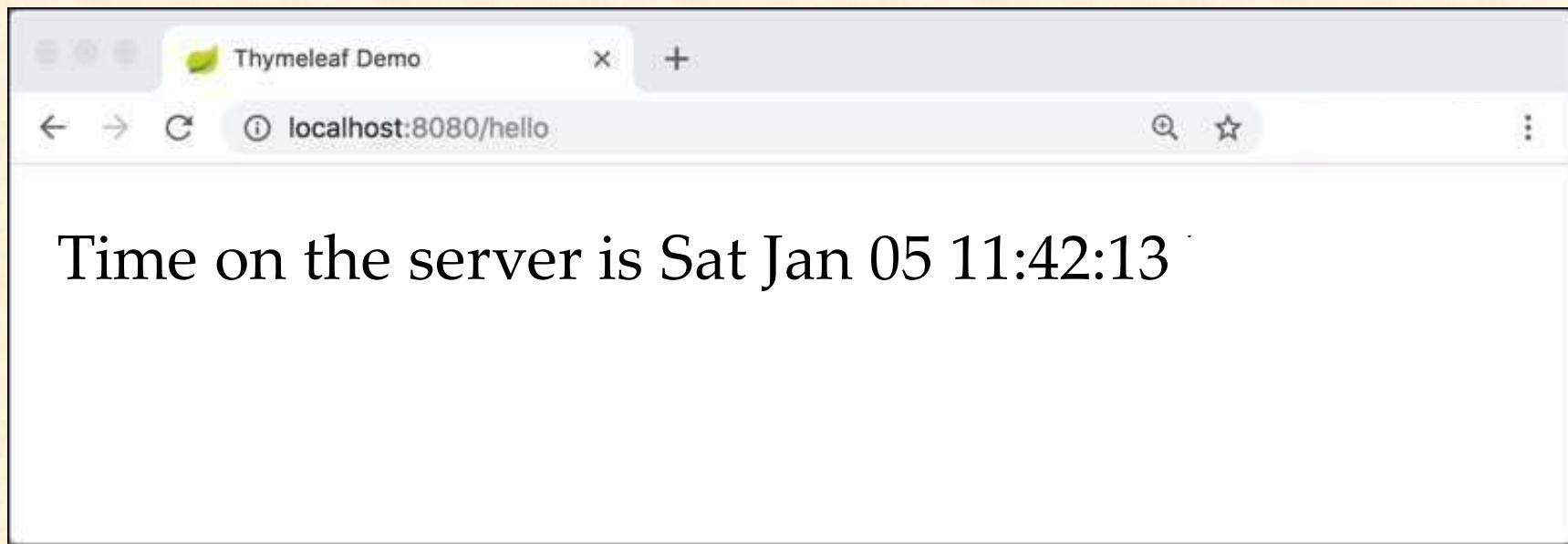
- Looping and conditionals
- CSS and JavaScript integration
- and many more

www.thymeleaf.org

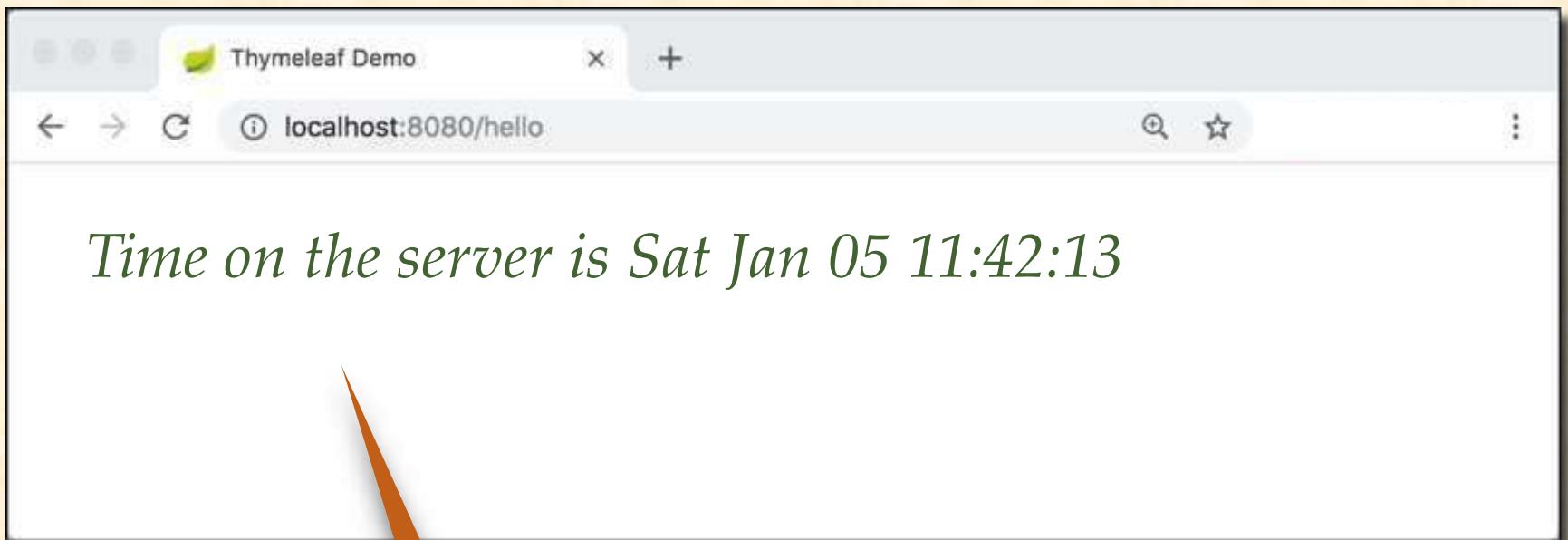
CSS and Thymeleaf

Let's Apply CSS Styles to our Page

Before



After



```
font-style: italic;  
color: green;
```

Using CSS with Thymleaf Templates

- You have the option of using
 - Local CSS files as part of your project
 - Referencing remote CSS files
- We'll cover both options here

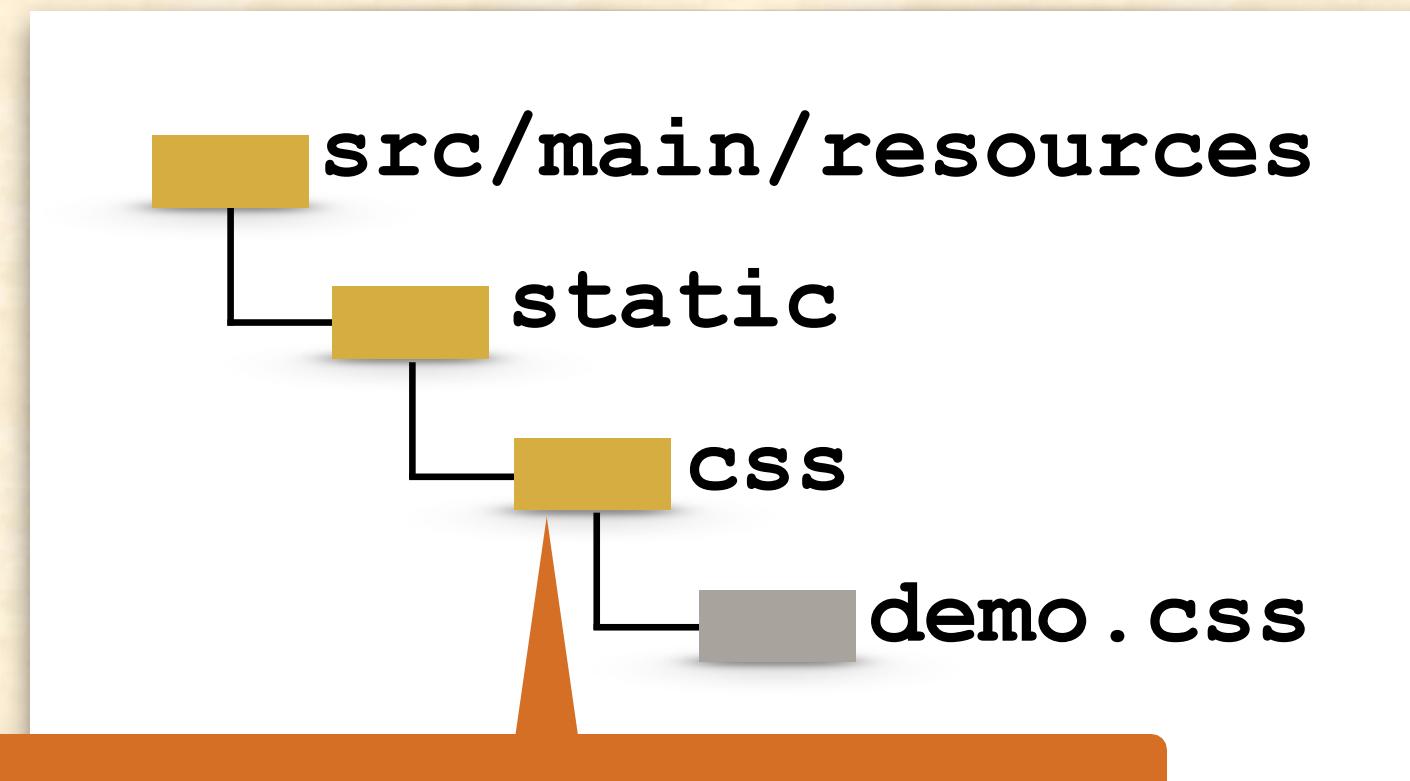
Development Process

1. Create CSS file
2. Reference CSS in Thymeleaf template
3. Apply CSS style

Step 1: Create CSS file

- Spring Boot will look for static resources in the directory

- src/main/resources/static**



Can be any sub-directory name

You can create your own custom sub-directories

static/css

static/images

static/js

etc ...

File: demo.css

```
.funny {  
    font-style: italic;  
    color: green;  
}
```

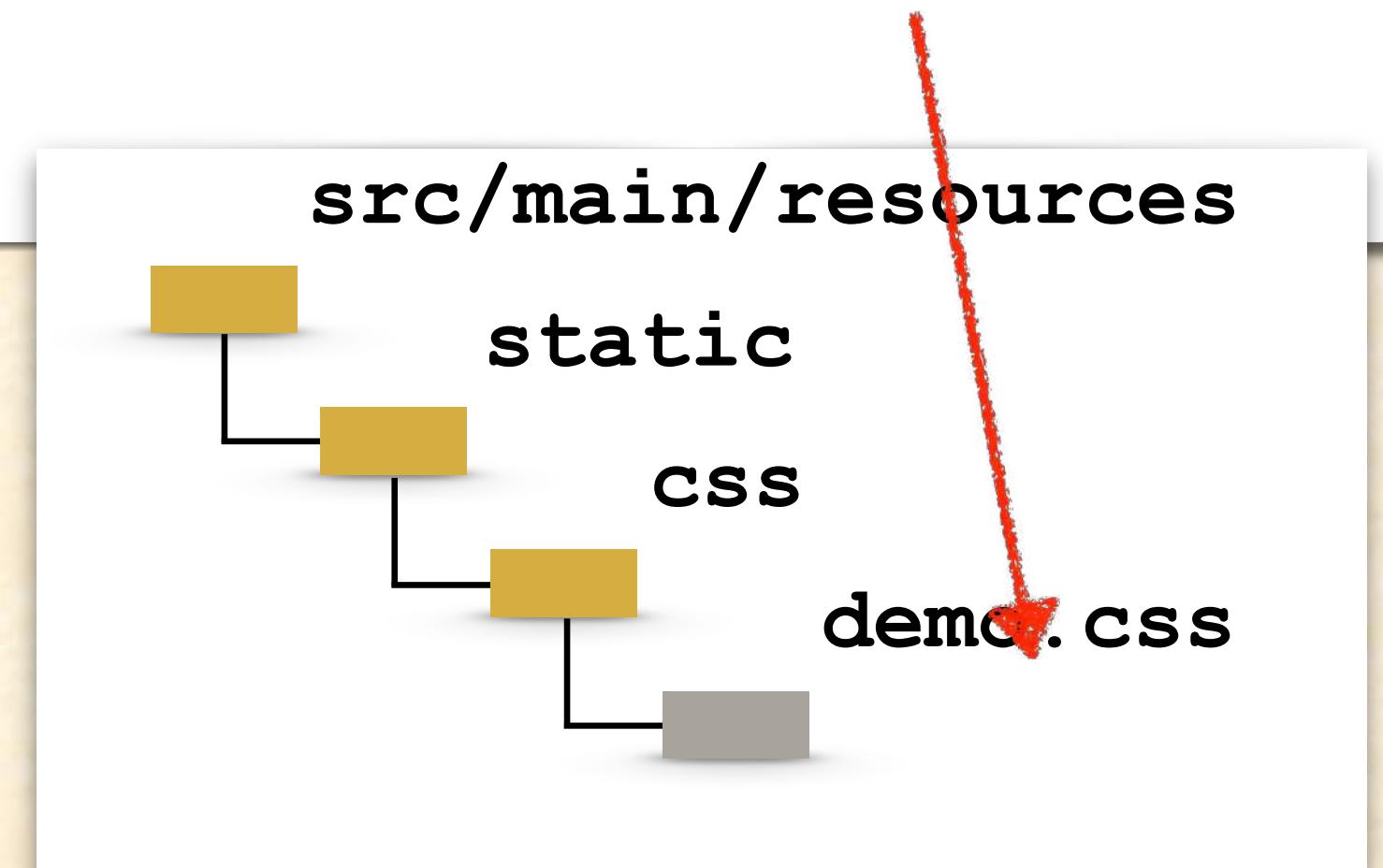
Step 2: Reference CSS in Thymeleaf template

File: helloworld.html

```
<head>
    <title>Thymeleaf Demo</title>

    <!-- reference CSS file -->
    <link rel="stylesheet" th:href="@{/css/demo.css}" />
</head>
```

@ symbol
Reference context path of your application
(app root)



Step 3: Apply CSS

File: helloworld.html

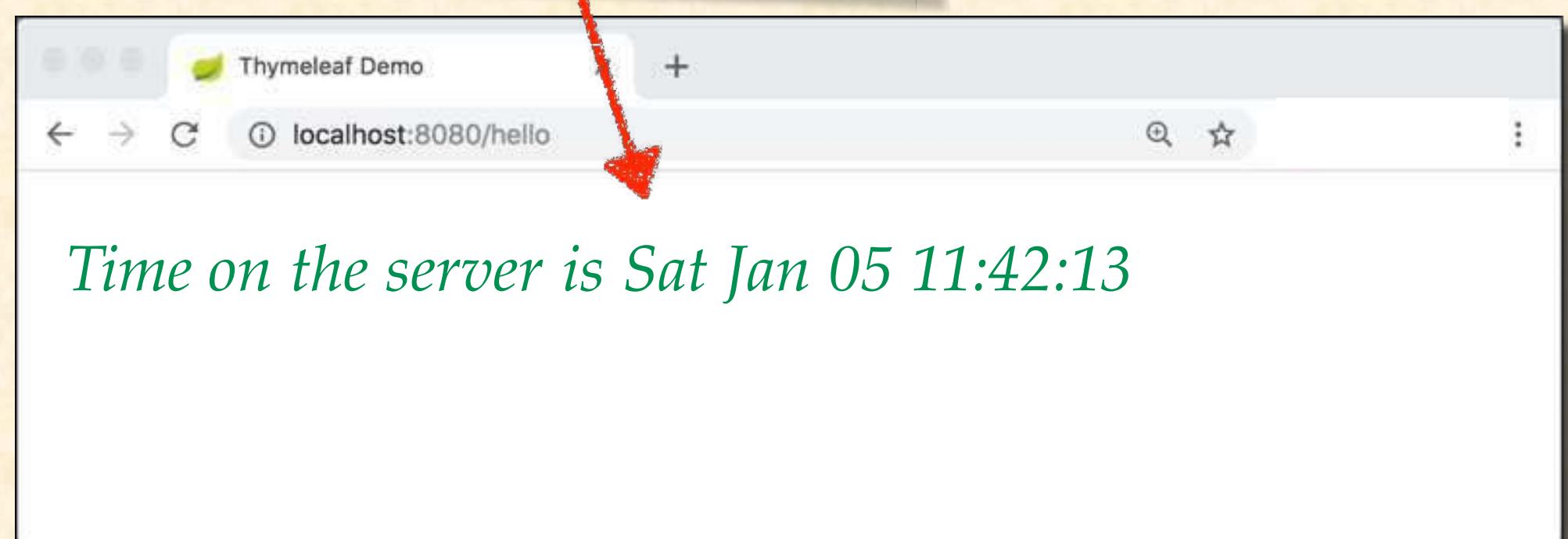
```
<head>
    <title>Thymeleaf Demo</title>

    <!-- reference CSS file -->
    <link rel="stylesheet" th:href="@{/css/demo.css}" />
</head>

<body>
    <p th:text=''Time on the server is ' + ${theDate}' class="funny" />
</body>
```

File: demo.css

```
.funny {
    font-style: italic;
    color: green;
}
```

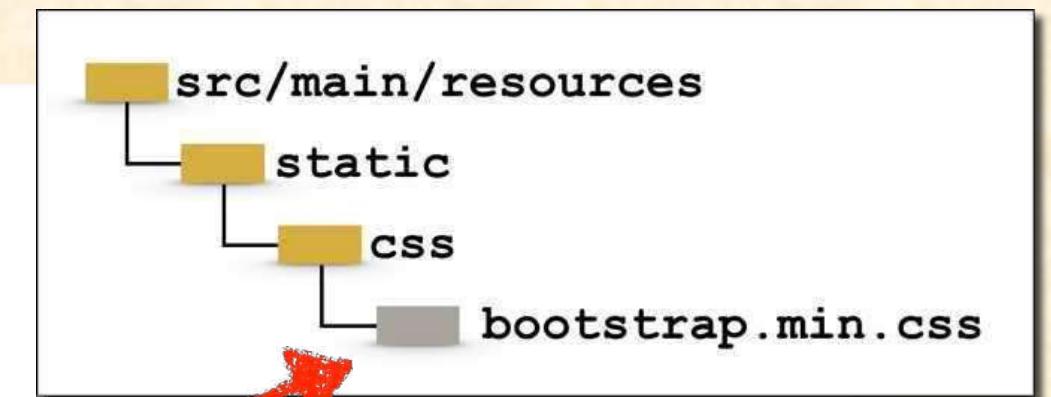


3rd Party CSS Libraries - Bootstrap

- Local Installation
- Download Bootstrap file(s) and add to **/static/css** directory

```
<head>
...
<!-- reference CSS file --&gt;
&lt;link rel="stylesheet" th:href="@{/css/bootstrap.min.css}" /&gt;

&lt;/head&gt;</pre>
```



3rd Party CSS Libraries - Bootstrap

- Remote Files

```
<head>
...
<!-- reference CSS file --&gt;
&lt;link rel="stylesheet"
      href="<u>https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css"
      />
...
</head>
```

Spring Boot (REST API, MVC and Microservices)

Reading Form Data with Spring MVC



High Level View

hello-form.html



What's your name? Submit Query



result.html

Hello World of Spring!

Student name: John Doe

Application Flow



Application Flow



Controller Class

```
@Controller  
public class HelloWorldController {
```

// need a controller method to show the initial HTML form

```
@RequestMapping("/showForm")  
public String showForm() {  
    return "hello-form";  
}
```

// need a controller method to process the HTML form

```
@RequestMapping("/processForm")  
public String processForm() {  
    return "result";  
}
```

```
<!DOCTYPE html>  
<html xmlns="http://www.thymeleaf.org">  
<head>  
<meta charset="ISO-8859-1">  
<title>Welcome to Thymeleaf</title>  
<link rel="stylesheet" th:href="@{/css/demo.css}">  
</head>  
<body>  
<h1>Form</h1>  
  
<form th:action="@{/processForm}" method="GET">  
<input type="text" name="studName" placeholder="What is ur  
name?"><br>  
<input type="submit">  
</form>  
</body>  
</html>  
  
<!DOCTYPE html>  
<html xmlns="http://www.thymeleaf.org">  
<head>  
<meta charset="ISO-8859-1">  
<title>Welcome to Thymeleaf</title>  
<link rel="stylesheet"  
href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.  
css" />  
</head>  
<body>  
<h1> Welcome to Spring Boot + Thymeleaf</h1>  
<br><br>  
<h1>Student Name: <span th:text="${param.studName}"></span></h1>  
</body>  
</html>
```

Development Process

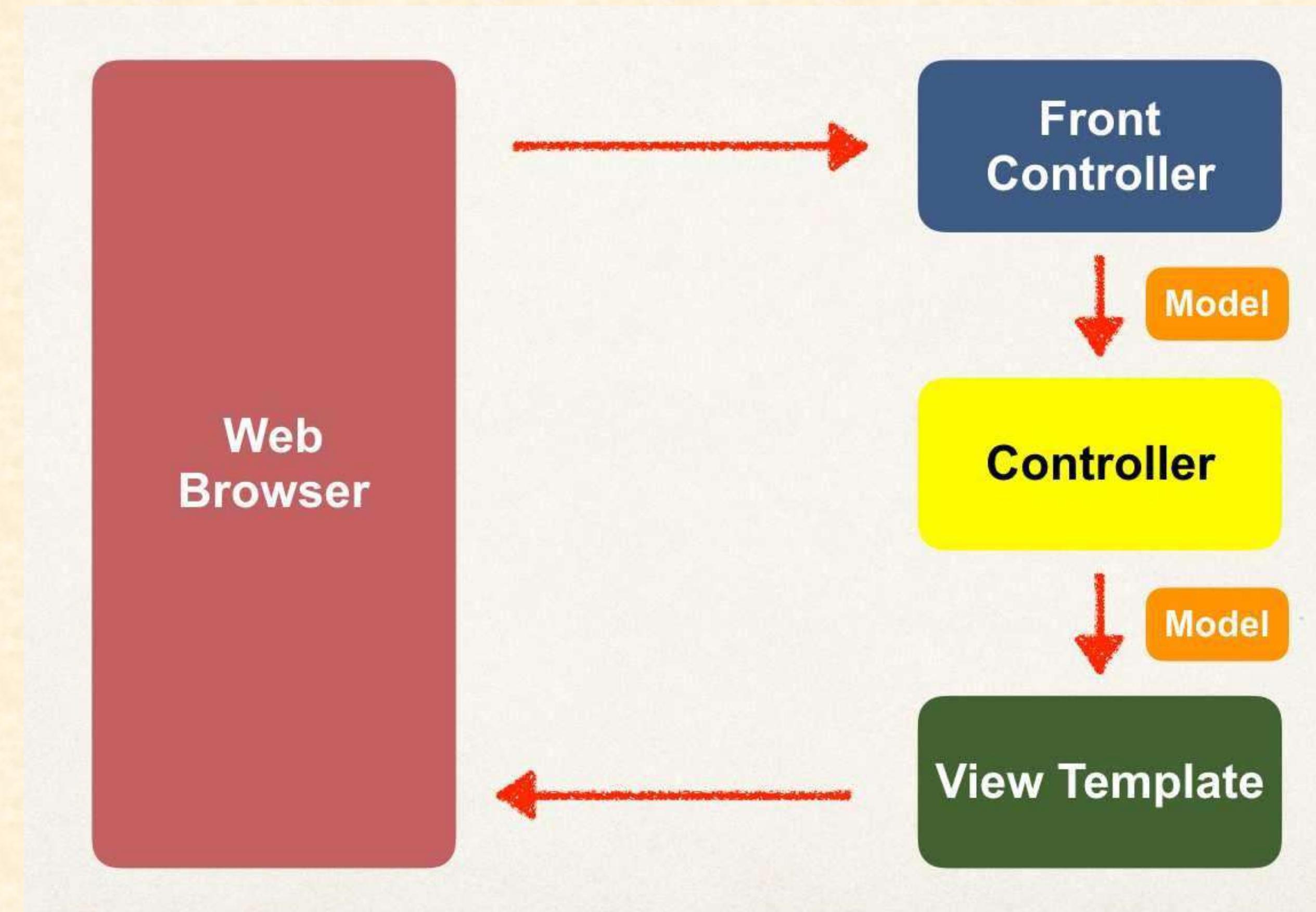
- 1. Create Controller class**

- 2. Show HTML form**
 - a. Create controller method to show HTML Form
 - b. Create View Page for HTML form

- 3. Process HTML Form**
 - a. Create controller method to process HTML Form
 - b. Develop View Page for Confirmation

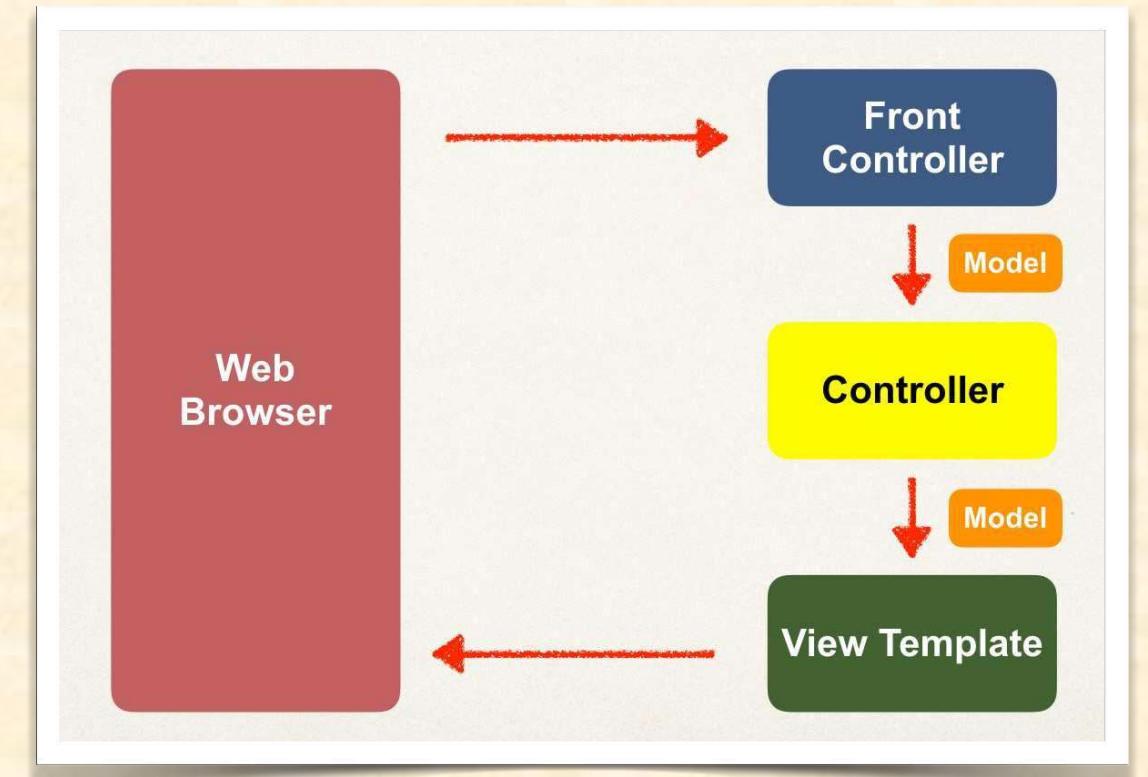
Adding Data to Spring Model

Focus on the Model



Spring Model

- The **Model** is a container for your application data
- In our Controller
 - We can put anything in the **model**
 - strings, objects, info from database, etc...
- Our View page can access data from the **model**



Reading HTML Form Data with @RequestParam Annotation

Code Example

- We want to create a new method to process form data
- Read the form data: student's name
- Convert the name to upper case
- Add the uppercase version to the model

Bind variable using @RequestParam Annotation

```
@RequestMapping("/processForm")
public String processForm(
    @RequestParam("studentName") String name,
    Model model) {
    // now we can use the variable: name
    model.addAttribute("name",
        name.toUpperCase());
}
```

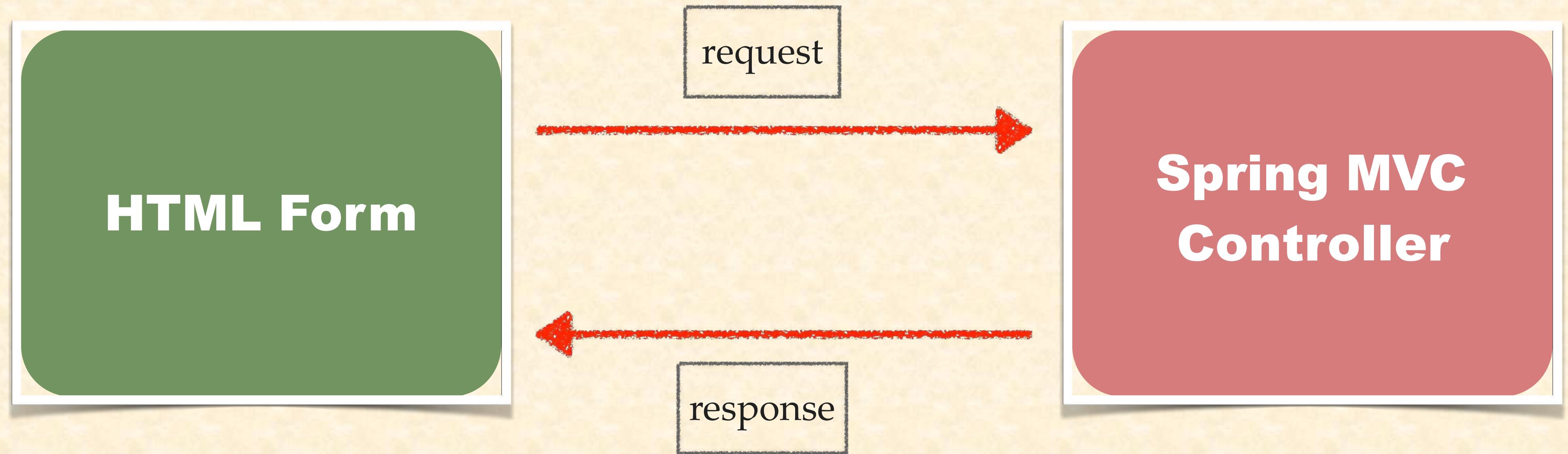
Behind the scenes:

Spring will read param from request: studentName

Bind it to the variable: name

@GetMapping and @PostMapping

HTTP Request / Response



Most Commonly Used HTTP Methods

Method	Description
GET	Requests data from given resource
POST	Submits data to given resource
<i>others</i>	...

Sending Data with GET method

```
<form th:action="@{/processForm}" method="GET" ...>  
...  
</form>
```

- Form data is added to end of URL as name/value pairs
 - **theUrl?field1=value1&field2=value2...**

Handling Form Submission

```
@RequestMapping("/processForm")
public String processForm(...) {
    ...
}
```

- This mapping handles ALL HTTP methods
 - GET, POST, etc ...

Constrain the Request Mapping - GET

```
@RequestMapping(path="/processForm", method=RequestMethod.GET)
public String processForm(...) {
    ...
}
```

- This mapping **ONLY** handles **GET** method
- Any other HTTP REQUEST method will get rejected

Annotation Short-Cut

```
@GetMapping("/processForm")
public String processForm(...) {
    ...
}
```

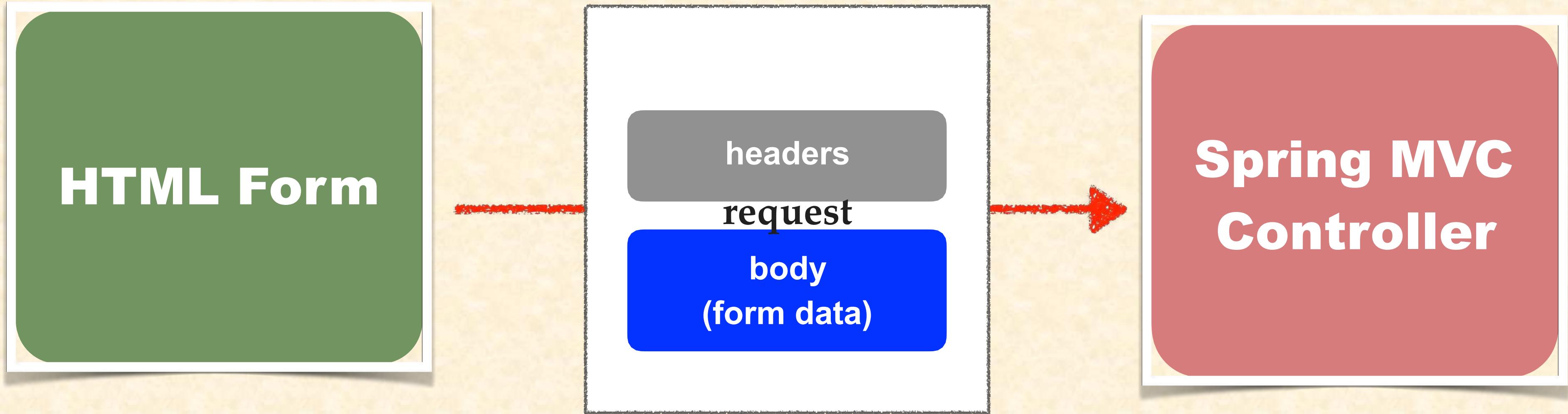
- @GetMapping: this mapping **ONLY** handles **GET** method
- Any other HTTP REQUEST method will get rejected

Sending Data with POST method

```
<form th:action="@{/processForm}" method="POST" ...>  
...  
</form>
```

- Form data is passed in the body of HTTP request message

Sending Data with POST method



Constrain the Request Mapping - POST

```
@RequestMapping(path="/processForm", method=RequestMethod.POST)
public String processForm(...) {
    ...
}
```

- This mapping **ONLY** handles **POST** method
- Any other HTTP REQUEST method will get rejected

Annotation Short-Cut

```
@PostMapping("/processForm")
public String processForm(...) {
    ...
}
```

- **@PostMapping:** This mapping **ONLY** handles **POST** method
- Any other HTTP REQUEST method will get rejected

Well which one???

GET

- Good for debugging
- Bookmark or email URL
- Limitations on data length

POST

- Can't bookmark or email URL
- No limitations on data length
- Can also send binary data

Spring Boot (REST API, MVC and Microservices)

Spring MVC Form Data Binding - Text Field

Review HTML Forms

- HTML Forms are used to get input from the user

Sign In

Email Address:

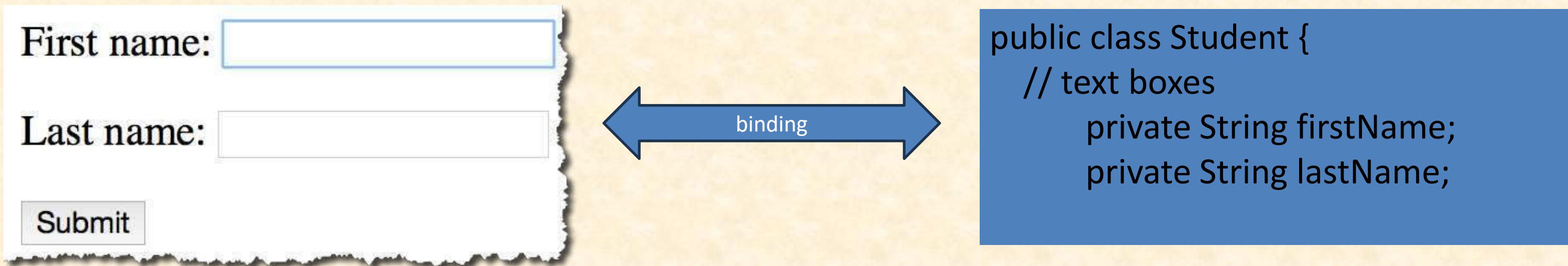
Password:

Remember me

Sign In

Data Binding

- Spring MVC forms can make use of data binding
- Automatically setting / retrieving data from a Java object / bean



Big Picture

student-form.html

First name:

Last name:

Submit

Student



**Student
Controller**

Student

student-confirmation.html

The student is confirmed: John Doe

```
public class Student {  
    // text boxes  
    private String firstName;  
    private String lastName;
```

Showing Form

In our Spring Controller

- Before you show the form, you must add a *model attribute*
- This is a bean that will hold form data for the *data binding*

Show Form - Add Model Attribute

Code snippet from Controller

```
@GetMapping("/showStudentForm")
public String showForm(Model theModel) {
    theModel.addAttribute("student", new Student());
    return "student-form";
}
```

```
public class Student {
    // text boxes
    private String firstName;
    private String lastName;
```

Setting up HTML Form - Data Binding

```
<form th:action="@{/processStudentForm}" th:object="${student}" method="POST">
```

First name: <input type="text" th:field="*{firstName}" />

Last name: <input type="text" th:field="*{lastName}" />

<input type="submit" value="Submit" />

</form>

Setting up HTML Form - Data Binding

Name of model attribute

```
<form th:action="@{/processStudentForm}" th:object="${student}" method="POST">
```

First name: <input type="text" th:field="*{firstName}" />

```
<br><br>
```

Last name: <input type="text" th:field="*{lastName}" />

```
<br><br>
```

```
<input type="submit" value="Submit" />
```

```
</form>
```

```
@GetMapping("/showStudentForm")
public String showForm(Model theModel) {
    theModel.addAttribute("student", new Student());
    return "student-form";
}
```

Setting up HTML Form - Data Binding

```
<form th:action="@{/processStudentForm}" th:object="${student}" method="POST">
```

First name: <input type="text" th:field="*{firstName}" />

*{ ... } is shortcut syntax for: \${student.firstName}

Last name: <input type="text" th:field="*{lastName}" />

*{ ... } is shortcut syntax for: \${student.lastName}

<input type="submit" value="Submit" />

</form>

First name:

Last name:

When Form is Loaded ... fields are populated

```
<form th:action="@{/processStudentForm}" th:object="${student}" method="POST">
```

First name: <input type="text" th:field="*{firstName}" />

Last name: <input type="text" th:field="*{lastName}" />

<input type="submit" value="Submit" />

</form>

When form is loaded,
Spring MVC will read student
from the model,
then call:

student.getFirstName()

...

student.getLastName()

When Form is submitted ... calls setter methods

```
<form th:action="@{/processStudentForm}" th:object="${student}" method="POST">
```

First name: <input type="text" th:field="*{firstName}" />

Last name: <input type="text" th:field="*{lastName}" />

<input type="submit" value="Submit" />

</form>

When form is submitted,
Spring MVC will
create a **new** Student instance
and add to the model,
then call:

student.setFirstName(...)

...

student.setLastName(...)

Handling Form Submission in the Controller

Code snippet from Controller

```
@PostMapping("/processStudentForm")
public String processForm(@ModelAttribute("student") Student theStudent) {

    // log the input data
    System.out.println("theStudent: " + theStudent.getFirstName()
        + " " + theStudent.getLastName());

    return "student-confirmation";
}
```

Confirmation page

```
<html>
```

```
<body>
```

The student is confirmed:

```
</body>
```

```
</html>
```

Calls student.getFirstName()

Calls student.getLastName()

The student is confirmed: John Doe

Pulling It All Together

student-form.html

First name:

Last name:

Submit

Student



**Student
Controller**

Student

```
public class Student {  
    // text boxes  
    private String firstName;  
    private String lastName;
```

student-confirmation.html

The student is confirmed: John Doe

Development Process

1. Create Student class (Student.java)
2. Create Student controller class (Student Controller)
 1. Create HTML form (student-form.html)
 2. Create form processing code (use model object and place data)
 3. Create confirmation page (student-confirmation.html)

Spring Boot (REST API, MVC and Microservices)

Spring MVC Form Data Binding - Drop Down List



Review of HTML <select> Tag

First name:

Last name:

Country: Brazil

```
<select name="country">
    <option value="Brazil">Brazil</option>
    <option value="France">France</option>
    <option value="Germany">Germany</option>
    <option value="India">India</option>
    ...
</select>
```

Value sent during
form submission

BR
FR
DE
IN

Displayed to user

Thymeleaf and <select> tag

```
<select th:field="*{country}">  
    <option th:value="Brazil">Brazil</option>  
    <option th:value="France">France</option>  
    <option th:value="Germany">Germany</option>  
    <option th:value="India">India</option>  
</select>
```

Value sent during
form submission

Displayed to user

```
public class Student {  
    // text boxes  
    private String firstName;  
    private String lastName;  
  
    private String country;
```



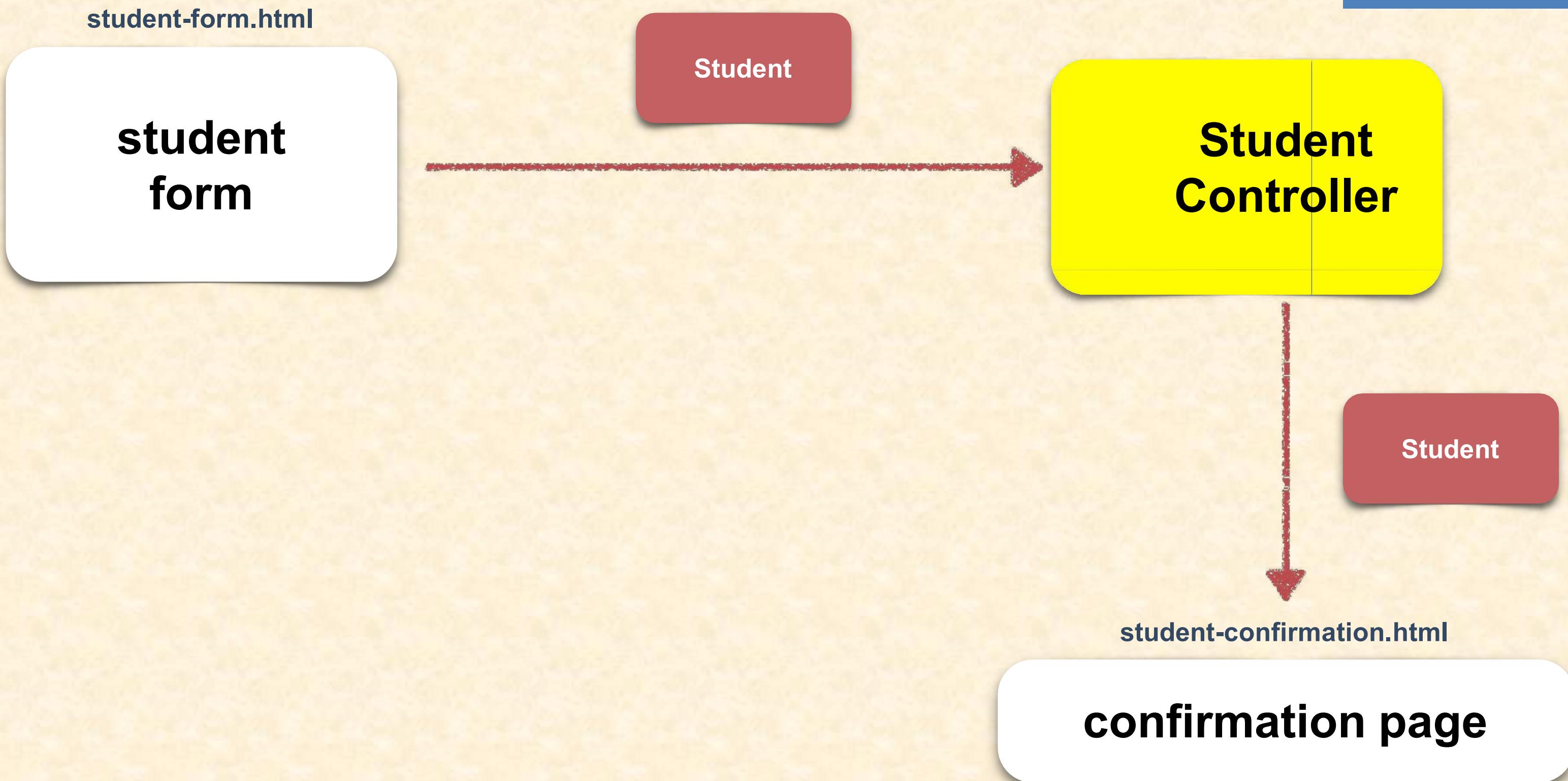
First name:

Last name:

Country:

Submit

Pulling It All Together



```
public class Student {  
    // text boxes  
    private String firstName;  
    private String lastName;  
    private String country;
```

Development Process

1. Update HTML form - adding a select box
2. Update Student class - add getter/setter for new property
3. Update confirmation page

```
Country:  
<select th:field="*{country}">  
<option th:each="tempCountry :${countries}" th:value="${tempCountry}" th:text="${tempCountry}" />  
</select>
```

```
application.properties:  
countries = India, Brazil, United Kingdom, United States od America, Australia
```

```
@Value("${countries}")  
private List<String> countries;
```

Spring Boot (REST API, MVC and Microservices)

Spring MVC Form Data Binding – Radio Buttons

Radio Buttons

Student Registration Form

First name:

Last name:

Country: ▾

Favorite Programming Language: Go Java Python

```
public class Student {  
    // text boxes  
    private String firstName;  
    private String lastName;  
    private String country;  
    private String favLanguage;
```



Code Example

Student Registration Form

First name:

Last name:

Country:

Favorite Programming Language: Go Java Python

Favorite Programming Language:

```
<input type="radio" th:field="*{favLanguage}" value="Go"/>  
<input type="radio" th:field="*{favLanguage}" value="Java"/>  
<input type="radio" th:field="*{favLanguage}" value="Python"/>
```

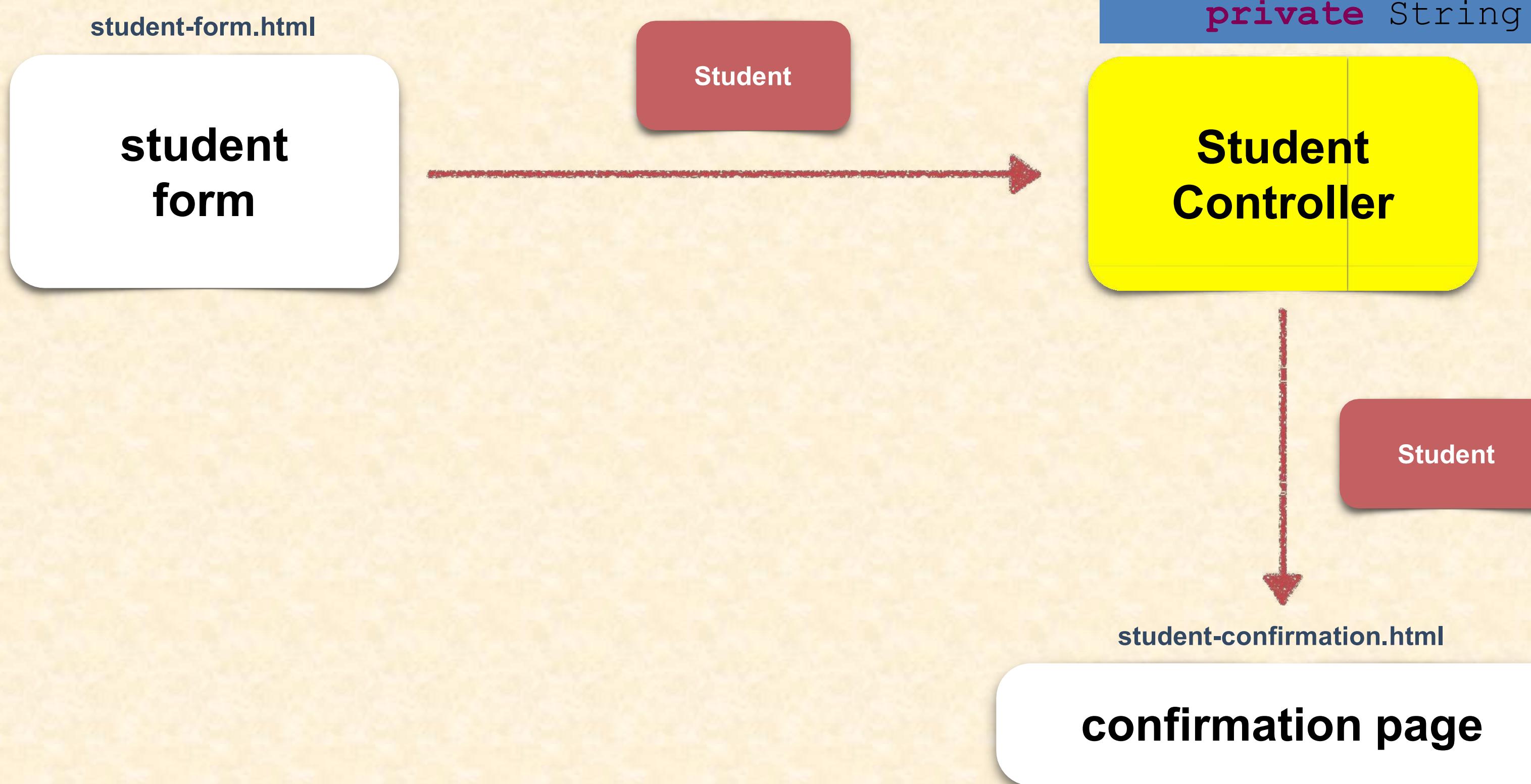
```
th:value="Go">Go</input>  
th:value="Java">Java</input>  
th:value="Python">Python</input>
```

Binding to property on
Student object

Value sent during
form submission

Displayed to user

Pulling It All Together



```
public class Student {  
    // text boxes  
    private String firstName;  
    private String lastName;  
    private String country;  
    private String favLanguage;
```

Development Process

1. Update HTML form
2. Update Student class - add getter/setter for new property
3. Update confirmation page

```
<input type="radio" th:field="*{favoriteLanguage}"  
       th:each="tempLang : ${languages}"  
       th:value="${tempLang}"  
       th:text="${tempLang}" />
```

Value sent during
form submission

```
<br><br>  
<input type="text" />
```

Displayed to user

```
countries=Brazil,France,Germany,India,Mexico,Spain,United States  
languages=Go,Java,Python,Rust,TypeScript
```

```
@Value("${languages}")  
private List<String> languages;
```

```
// add the list of languages to the model  
theModel.addAttribute(attributeName: "languages", languages);
```

Spring Boot (REST API, MVC and Microservices)

Spring MVC Form Data Binding – Check Box

Check Box - Pick Your Favorite Operating Systems

Favorite Operating Systems: Linux macOS Microsoft Windows

Submit

```
public class Student {  
    // text boxes  
    private String firstName;  
    private String lastName;  
    private String country;  
    private String favLanguage;  
    private String favOS;
```

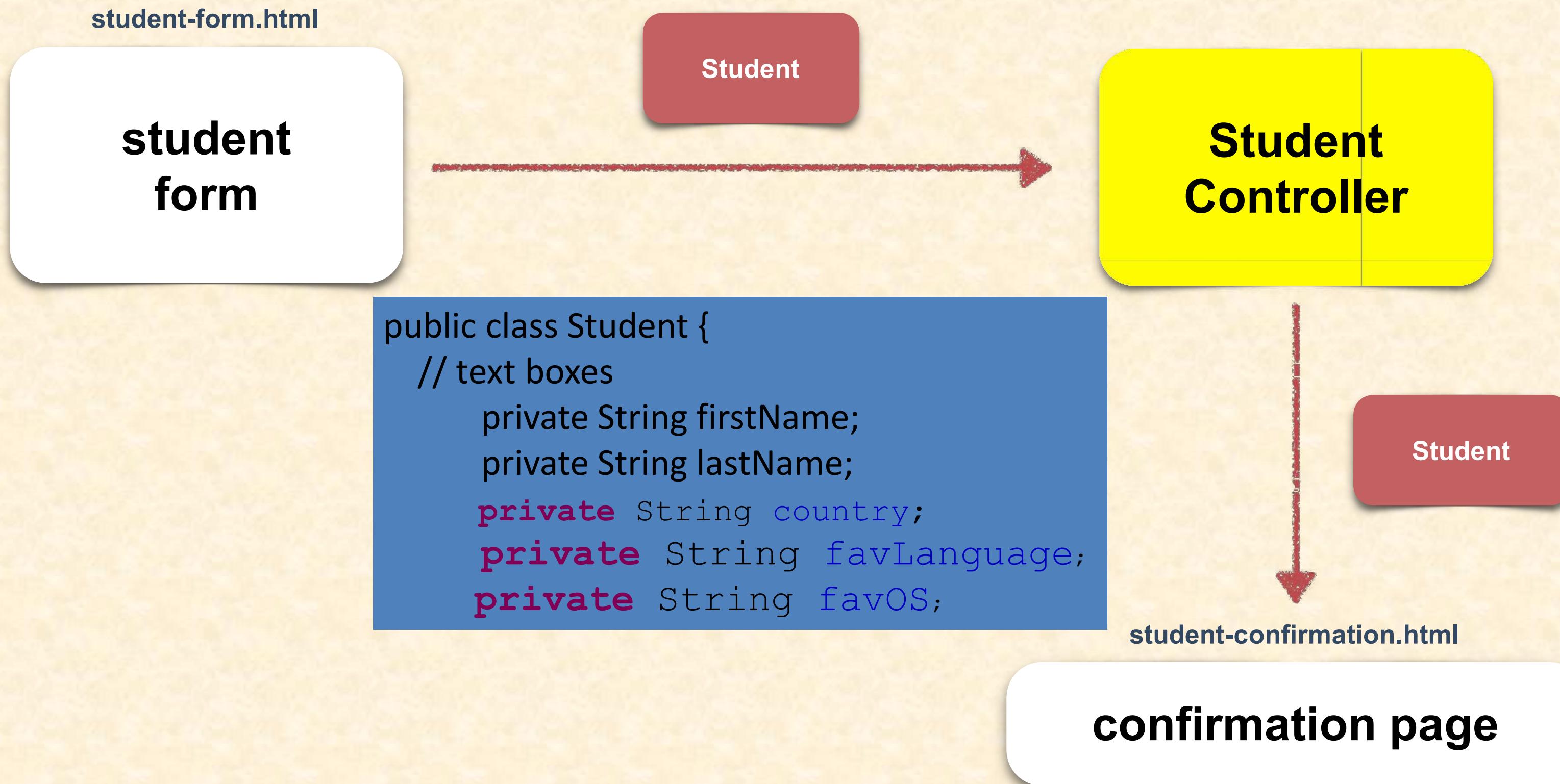
Code Example

Favorite Operating Systems: Linux macOS Microsoft Windows

Submit

```
<input type="checkbox" th:field="*{favOS}" th:value="Linux">Linux</input>
<input type="checkbox" th:field="*{favOS}" th:value="macOS">macOS</input>
<input type="checkbox" th:field="*{favOS}"
      th:value="MSWindows">MS Windows</input>
```

Pulling It All Together



Development Process

1. Update HTML form
2. Update Student class - add getter/setter for new property
3. Update confirmation page

```
<ul>
    <li th:each="tempSystem : ${student.favoriteSystems}" th:text="${tempSystem}" />
</ul>
```

Spring Boot (REST API, MVC and Microservices)

Spring MVC - Form Validation

The Need for Validation

Check the user input form for

- required fields
- valid numbers in a range
- valid format (postal code, etc.)
- custom business rule

Java's Standard Bean Validation API

- Java has a standard Bean Validation API
- Defines a metadata model and API for entity validation
- Spring Boot and Thymeleaf also support the Bean Validation API

<http://www.beanvalidation.org>

Bean Validation Features

Validation Feature
required
validate length
validate numbers
validate with regular expressions
custom validation

Validation Annotations

Annotation	Description
@NotNull	Checks that the annotated value is not null
@Min	Must be a number \geq value
@Max	Must be a number \leq value
@Size	Size must match the given size
@Pattern	Must match a regular expression pattern
@Future / @Past	Date must be in future or past of given date
others ...	

Our Road Map for form validation

1. Set up our development environment (Validation dependency)
2. required field, size
3. validate number range: min, max
4. validate using regular expression (regexp)

Spring MVC Form Validation

Required Fields

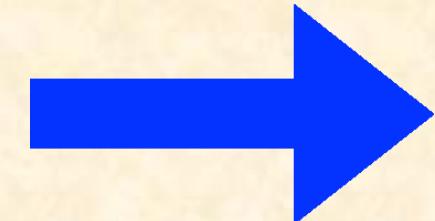
Required Fields

Fill out the form. Asterisk () means required.*

First name:

Last name (*):

Submit



Fill out the form. Asterisk () means required.*

First name:

Last name (*): **is required**

Submit

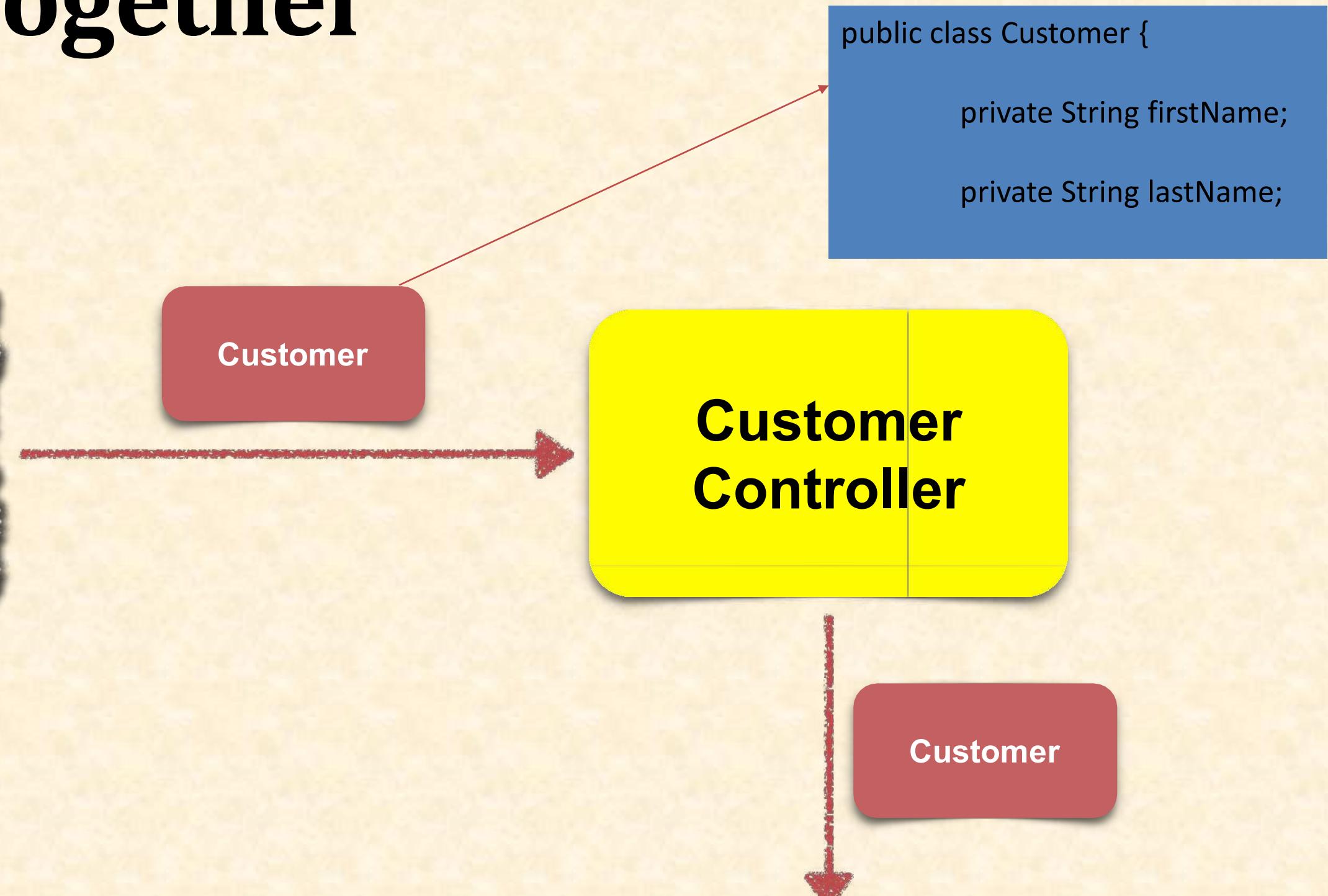
Pulling It All Together

customer-form.html

First name:

Last name:

Submit



```
public class Customer {
```

```
    private String firstName;
```

```
    private String lastName;
```

customer-confirmation.html

The customer is confirmed: John Doe

Development Process

1. Create Customer class and add validation rules

Customer class with Validations (@NotNull and @Size) for required

2. Add Controller code to show HTML form

Customer Controller

3. Develop HTML form and add validation support

customer-form.html

4. Perform validation in the Controller class

BindingResult – to check for errors and @Valid annotation

5. Create confirmation page

customer-confirmation.html

Step 1: Create Customer class and add validation rules

File: Customer.java

```
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;

public class Customer {

    private String firstName;

    @NotNull(message = "is required")
    @Size(min=1, message = "is required")
    private String lastName = "";

    // getter/setter methods ...

}
```

Validation rules

Step 2: Add Controller code to show HTML form

File: CustomerController.java

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.ui.Model;

@Controller
public class CustomerController {

    @GetMapping("/")
    public String showForm(Model theModel) {

        theModel.addAttribute("customer", new Customer());
        return "customer-form";
    }

    ...
}
```

Model allows us to share
information between Controllers
and view pages (Thymeleaf)

name

value

Step 3: Develop HTML form and add validation support

File: customer-form.html

```
<form th:action="@{/processForm}" th:object="${customer}" method="POST">

    First name: <input type="text" th:field="*{firstName}" />
    <br><br>

    Last name (*): <input type="text" th:field="*{lastName}" />
    <!-- Show error message (if present) -->
    <span th:if="#fields.hasErrors('lastName')"
          th:errors="*{lastName}"
          class="error"></span>

    <br><br>

    <input type="submit" value="Submit" />

</form>
```

Where to submit form data

Model attribute name

Property name from Customer class

Property name from Customer class

First name:

Last name (*): is required

Submit

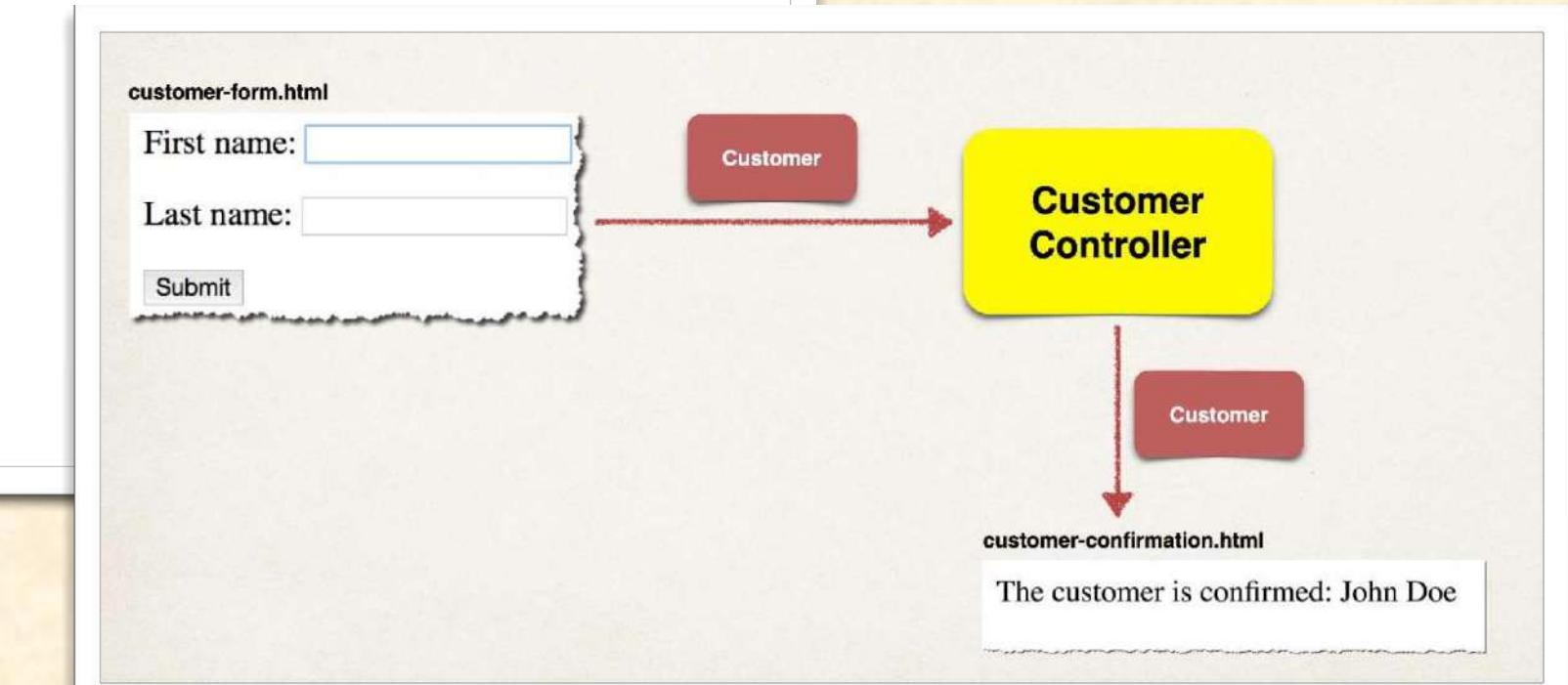
Step 4: Perform validation in Controller class

Tell Spring MVC to perform validation

The results of validation

```
File: CustomerController.java  
...  
  
@PostMapping("/processForm")  
public String processForm(  
    @Valid @ModelAttribute("customer") Customer theCustomer,  
    BindingResult theBindingResult) {  
  
    if (theBindingResult.hasErrors()) {  
        return "customer-form";  
    }  
    else {  
        return "customer-confirmation";  
    }  
}
```

Model attribute name



Step 5: Create confirmation page

File: customer-confirmation.html

```
<!DOCTYPE html>
<html
xmlns:th="http://www.thymeleaf.org">
<body>

The customer is confirmed: <span th:text="${customer.firstName + ' ' + customer.lastName}" />

</body>
</html>
```

Spring MVC Validation

Number Range: @Min and @Max

Validate a Number Range

- Add a new input field on our form for: Free Passes
- User can only enter a range: 0 to 10

Fill out the form. Asterisk () means required.*

First name: Bob

Last name (*): With

Free passes: 5

Submit

Step 1: Add validation rule to Customer class

```
import jakarta.validation.constraints.Min;  
import jakarta.validation.constraints.Max;  
  
public class Customer {  
  
    @Min(value=0, message="must be greater than or equal to zero")  
    @Max(value=10, message="must be less than or equal to 10")  
    private int freePasses;  
  
    // getter/setter methods  
  
}
```

Validate a Postal Code

- Add a new input field on our form for: **Postal Code**
- User can only enter 5 chars / digits
- Apply *Regular Expression*

Fill out the form. Asterisk () means required.*

First name:

Last name (*): I

Free passes: 0

Postal Code:

Step 1: Add validation rule to Customer class

Advanced

```
import jakarta.validation.constraints.Pattern;  
  
public class Customer {  
  
    @Pattern(regexp="^[a-zA-Z0-9]{5}", message="only 5 chars/digits")  
    private String postalCode;  
  
    // getter/setter methods  
  
}
```

Spring Boot (REST API, MVC and Microservices)

Spring MVC CRUD - Real Time Project - EMS



Application Requirements

Create a Web Interface for the Employee Directory

Users should be able to

- ✓ Get a list of employees
- ✓ Add a new employee
- ✓ Update an employee
- ✓ Delete an employee

Thymeleaf + Spring MVC CRUD

Real-Time Project - EMS

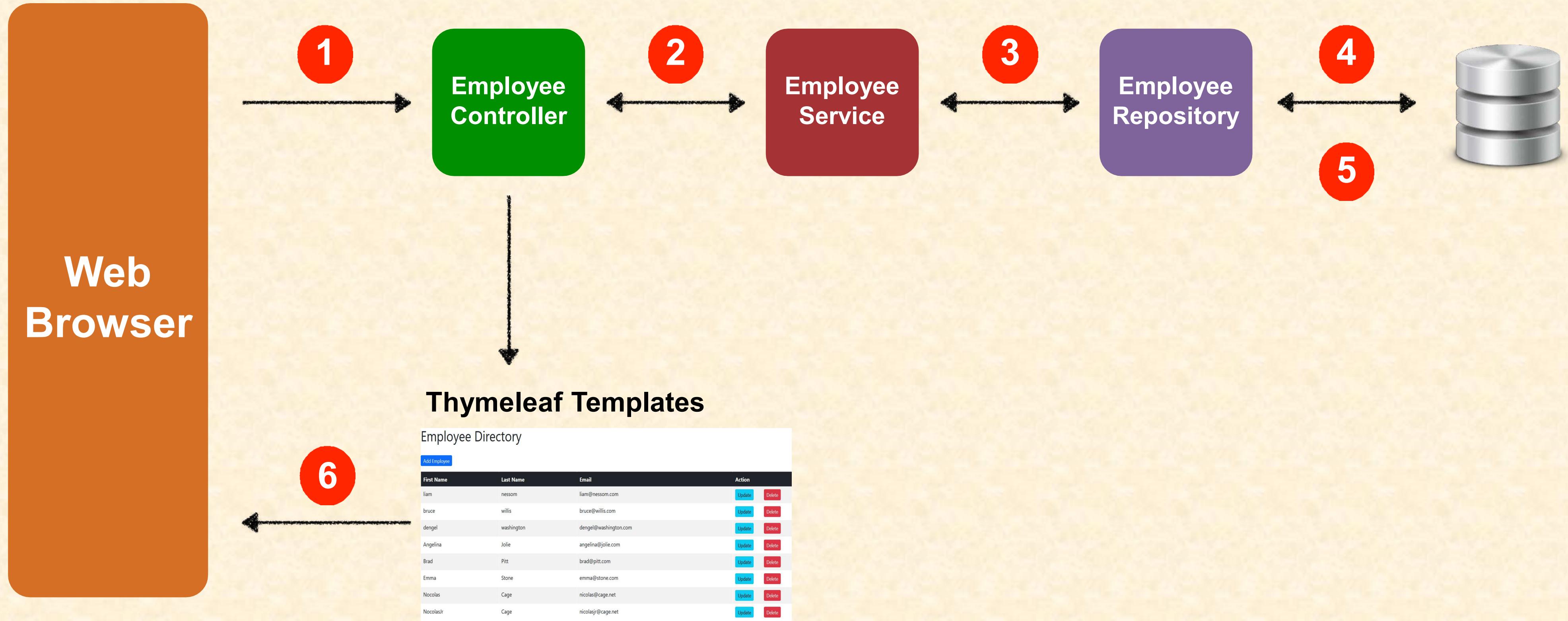
Thymeleaf + Spring MVC CRUD

Employee Directory

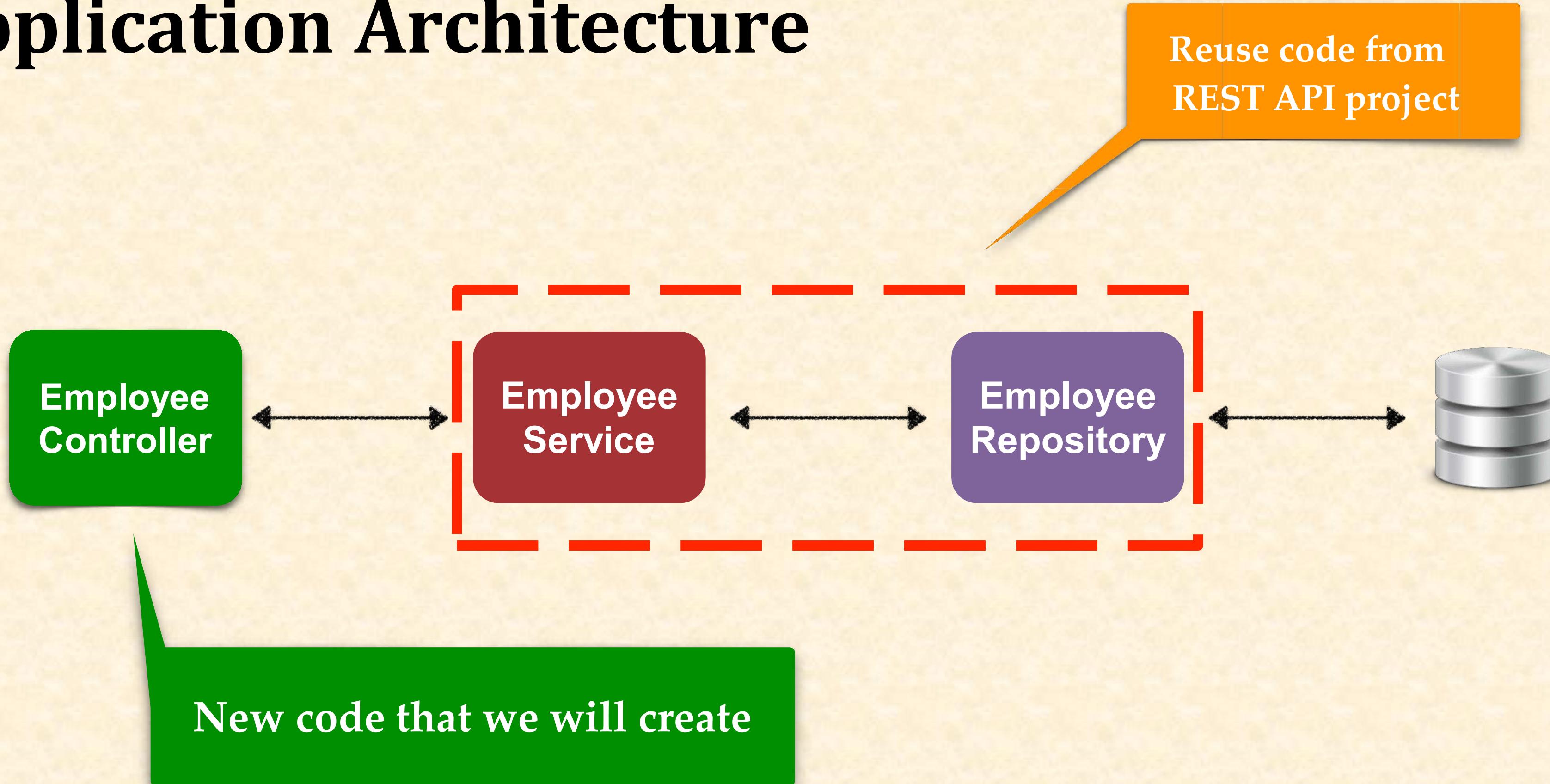
Add Employee

First Name	Last Name	Email	Action
liam	nessom	liam@nessom.com	<button>Update</button> <button>Delete</button>
bruce	willis	bruce@willis.com	<button>Update</button> <button>Delete</button>
dengel	washington	dengel@washington.com	<button>Update</button> <button>Delete</button>
Angelina	Jolie	angelina@jolie.com	<button>Update</button> <button>Delete</button>
Brad	Pitt	brad@pitt.com	<button>Update</button> <button>Delete</button>
Emma	Stone	emma@stone.com	<button>Update</button> <button>Delete</button>
Nocolas	Cage	nicolas@cage.net	<button>Update</button> <button>Delete</button>
NocolasJr	Cage	nicolasjr@cage.net	<button>Update</button> <button>Delete</button>

Big Picture



Application Architecture

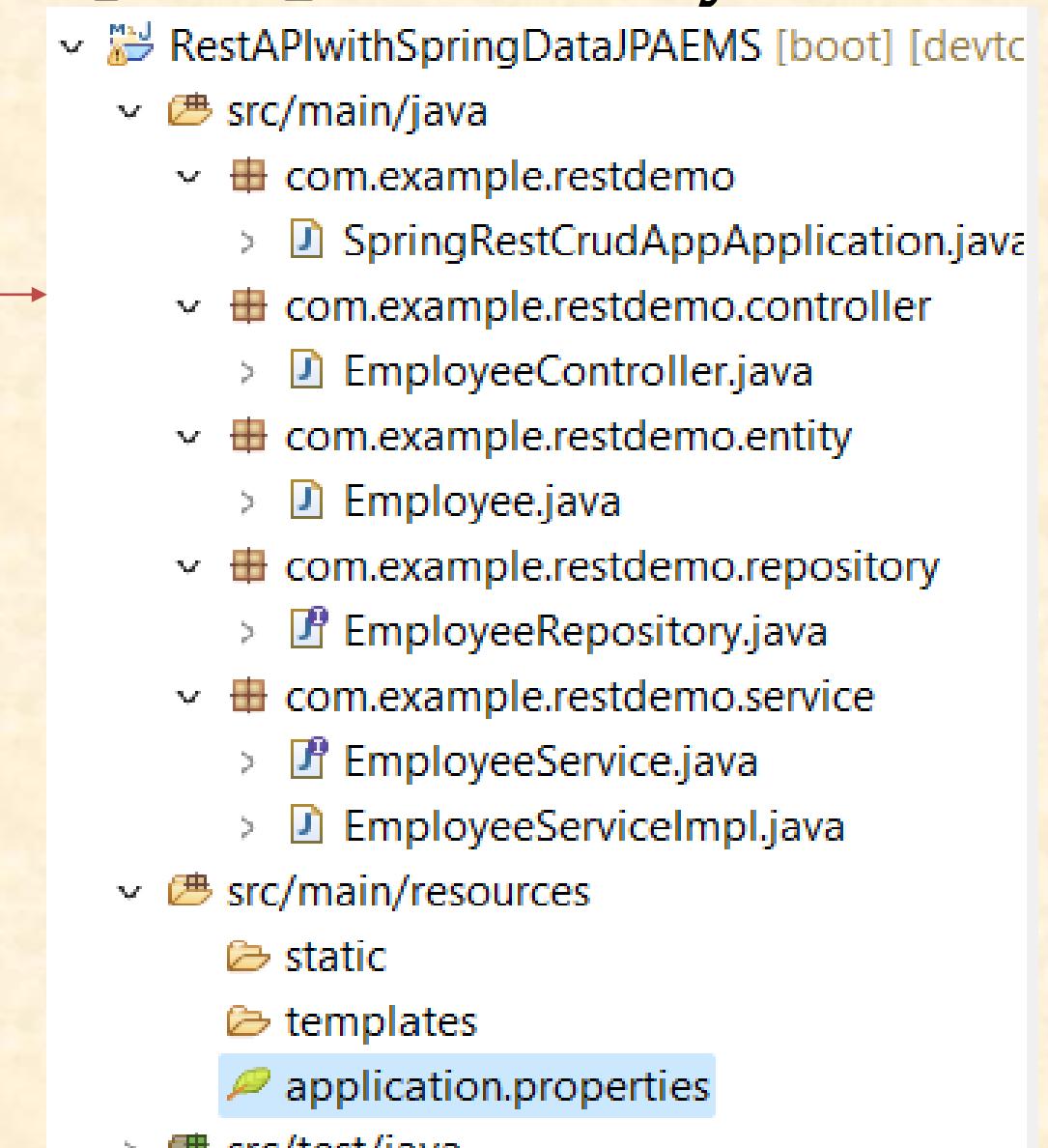


Project Set Up

- We will extend our existing Employee project and add DB integration

- Add **EmployeeService**, **EmployeeRepository** and **Employee** entity

- Available in one of our REST API project
- We created all of this code already from scratch
so we'll just copy/paste it
- Allows us to focus on creating **EmployeeController** and Thymeleaf templates



Development Process - Big Picture

1. Get list of employees

2. Add a new employee

3. Update an existing employee

4. Delete an existing employee

Employee Directory

Add Employee

First Name	Last Name	Email	Action
liam	nessom	liam@nessom.com	<button>Update</button> <button>Delete</button>
bruce	willis	bruce@willis.com	<button>Update</button> <button>Delete</button>
dengel	washington	dengel@washington.com	<button>Update</button> <button>Delete</button>
Angelina	Jolie	angelina@jolie.com	<button>Update</button> <button>Delete</button>
Brad	Pitt	brad@pitt.com	<button>Update</button> <button>Delete</button>
Emma	Stone	emma@stone.com	<button>Update</button> <button>Delete</button>
Nocolas	Cage	nicolas@cage.net	<button>Update</button> <button>Delete</button>
NocolasJr	Cage	nicolasjr@cage.net	<button>Update</button> <button>Delete</button>

Real-Time Project - EMS

Thymeleaf + Spring MVC CRUD

List Employees

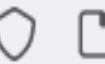
Employee Directory

Add Employee

First Name	Last Name	Email	Action
liam	nessom	liam@nessom.com	<button>Update</button> <button>Delete</button>
bruce	willis	bruce@willis.com	<button>Update</button> <button>Delete</button>
dengel	washington	dengel@washington.com	<button>Update</button> <button>Delete</button>
Angelina	Jolie	angelina@jolie.com	<button>Update</button> <button>Delete</button>
Brad	Pitt	brad@pitt.com	<button>Update</button> <button>Delete</button>
Emma	Stone	emma@stone.com	<button>Update</button> <button>Delete</button>
Nocolas	Cage	nicolas@cage.net	<button>Update</button> <button>Delete</button>
NocolasJr	Cage	nicolasjr@cage.net	<button>Update</button> <button>Delete</button>

Add Employee

← → ⌂



localhost:8080/showFormForAdd

Login Form

First name

Last name

Email

Save

Real-Time Project - EMS

Thymeleaf + Spring MVC CRUD

Update Employee

localhost:8080/showFormForUpdate?employeeId=1

First Name: liam

Last Name: nessom

Email: liam@nessom.com

Save

Login Form

Delete Employee

Email	Action
liam@nessom.com	<button>Update</button> <button>Delete</button>
bruce@willis.com	<button>Update</button> <button>Delete</button>
dengel@washington.com	<button>Update</button> <button>Delete</button>
angela@london.org	<button>Update</button> <button>Delete</button>
brad@paris.org	<button>Update</button> <button>Delete</button>
emily@sydney.org	<button>Update</button> <button>Delete</button>
nicolas@cage.net	<button>Update</button> <button>Delete</button>
nicolasjr@cage.net	<button>Update</button> <button>Delete</button>

localhost:8080

Are you sure you want to delete this employee?

OK Cancel

Spring Boot (REST API, MVC and Microservices)

Spring MVC CRUD – EMS – Get List of Employees

Get List of Employees

List Employees

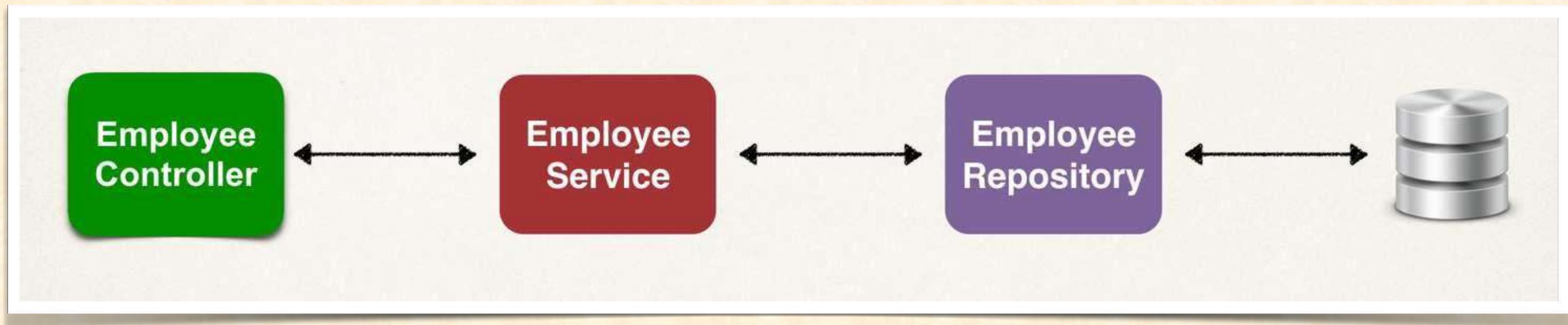
Employee Directory

Add Employee

First Name	Last Name	Email	Action
liam	nessom	liam@nessom.com	<button>Update</button> <button>Delete</button>
bruce	willis	bruce@willis.com	<button>Update</button> <button>Delete</button>
dengel	washington	dengel@washington.com	<button>Update</button> <button>Delete</button>
Angelina	Jolie	angelina@jolie.com	<button>Update</button> <button>Delete</button>
Brad	Pitt	brad@pitt.com	<button>Update</button> <button>Delete</button>
Emma	Stone	emma@stone.com	<button>Update</button> <button>Delete</button>
Nocolas	Cage	nicolas@cage.net	<button>Update</button> <button>Delete</button>
NocolasJr	Cage	nicolasjr@cage.net	<button>Update</button> <button>Delete</button>

Controller method – listEmployees()

```
@GetMapping("/list")
public String listEmployees(Model model) {
    List<Employee> emps = employeeService.findAll();
    model.addAttribute("employees", emps);
    return "list-emps";
}
```





list-emps.html

```
<h1> Employee Directory</h1><br>
<a th:href="@{/showFormForAdd}" class="btn btn-primary btn-sm mb-3">
    Add Employee</a>
<table class="table table-bordered table-striped table-dark">
    <thead class="table-dark">
        <tr><th>First Name</th>
        <th>Last Name</th>
        <th>Email</th>
        <th>Action</th></tr>
    <thead><tbody>
        <tr th:each="emp :${employees}">
            <td th:text="${emp.firstName}"/>
            <td th:text="${emp.lastName}"/>
            <td th:text="${emp.email}"/>
            <td>
                <a th:href="@{/showFormForUpdate(employeeId=${emp.id})}" class="btn btn-info btn-sm">
                    Update
                </a> &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
                <a th:href="@{/delete(employeeId=${emp.id})}" class="btn btn-danger btn-sm"
                    onclick="if (!(confirm('Are you sure you want to delete this employee?'))) return false">
                    Delete
                </a></td>
            </tr></tbody></table>
        </body>
```

Spring Boot (REST API, MVC and Microservices)

Spring MVC CRUD – EMS – Add Employee

Add Employee

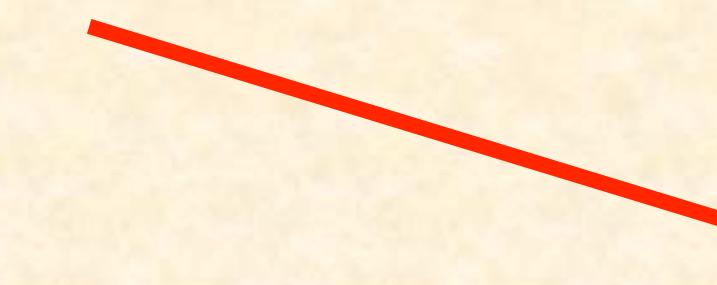
Employee Directory

Add Employee

First Name	Last Name	Email	Action
liam	nessom	liam@nessom.com	<button>Update</button> <button>Delete</button>
bruce	willis	bruce@willis.com	<button>Update</button> <button>Delete</button>
dengel	washington	dengel@washington.com	<button>Update</button> <button>Delete</button>
Angelina	Jolie	angelina@jolie.com	<button>Update</button> <button>Delete</button>
Brad	Pitt	brad@pitt.com	<button>Update</button> <button>Delete</button>
Emma	Stone	emma@stone.com	<button>Update</button> <button>Delete</button>
Nocolas	Cage	nicolas@cage.net	<button>Update</button> <button>Delete</button>
NocolasJr	Cage	nicolasjr@cage.net	<button>Update</button> <button>Delete</button>

Add Employee

1. Add - **Add Employee** button for list-employees.html



Employee Directory

First Name	Last Name	Email	Action
liam	nessom	liam@nessom.com	<button>Update</button> <button>Delete</button>
bruce	willis	bruce@willis.com	<button>Update</button> <button>Delete</button>
dengel	washington	dengel@washington.com	<button>Update</button> <button>Delete</button>
Angelina	Jolie	angelina@jolie.com	<button>Update</button> <button>Delete</button>
Brad	Pitt	brad@pitt.com	<button>Update</button> <button>Delete</button>
Emma	Stone	emma@stone.com	<button>Update</button> <button>Delete</button>
Nocolas	Cage	nicolas@cage.net	<button>Update</button> <button>Delete</button>
NocolasJr	Cage	nicolasjr@cage.net	<button>Update</button> <button>Delete</button>

Add Employee

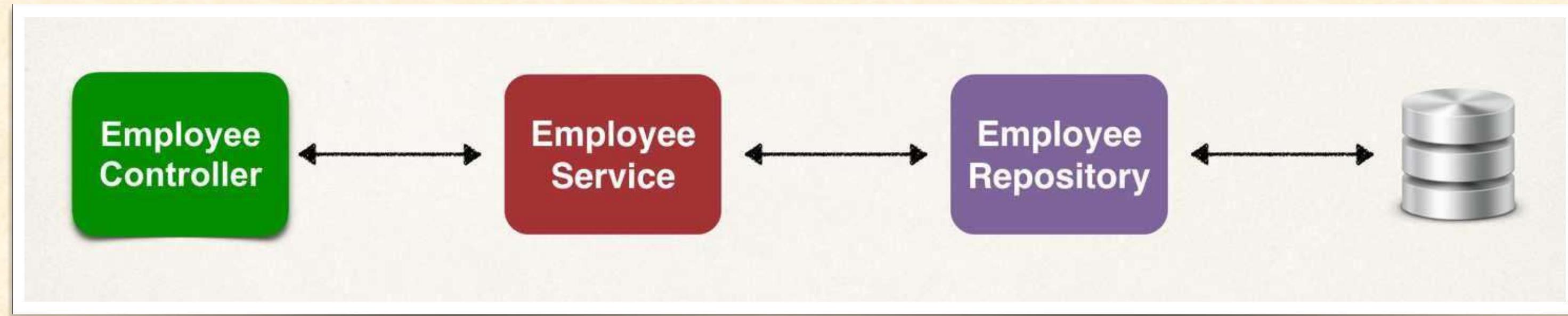
1. New **Add Employee** button for list-employees.html
2. Create HTML form for new employee



ADD or UPDATE Employee

Add Employee

1. New **Add Employee** button for list-employees.html
2. Create HTML form for new employee
3. Process form data to save employee



Step 1: New "Add Employee" button

- Add Employee button will href link to
- request mapping */showFormForAdd*

```
<a th:href="@{/showFormForAdd}"> Add Employee  
</a>
```

Add Employee

@ symbol
Reference context path of your application
(app root)

Step 1: New "Add Employee" button

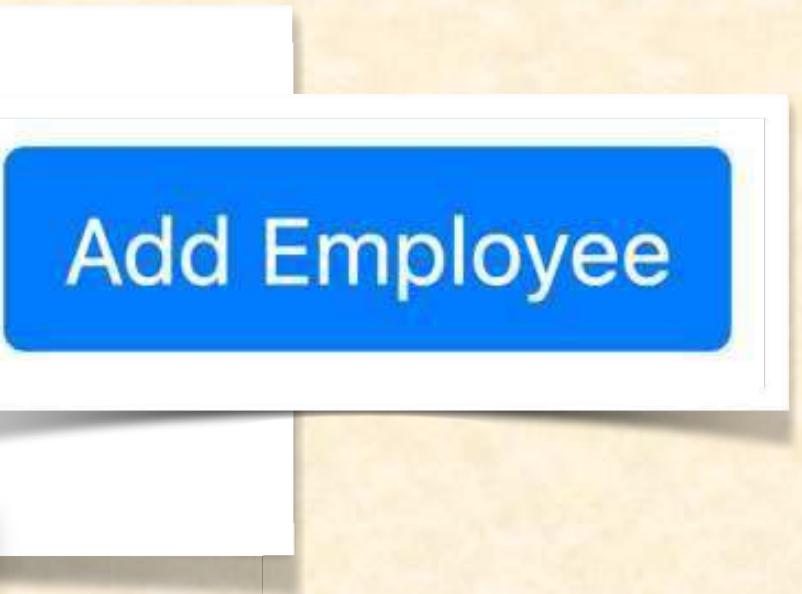
- Add Employee button will href link to
- request mapping */showFormForAdd*

```
<a th:href="@{/showFormForAdd}"  
    class="btn btn-primary btn-sm mb-3">  
    Add Employee  
</a>
```

Apply Bootstrap styles

Docs on Bootstrap styles: www.getbootstrap.com

Button
Button Primary
Button Small
Margin Bottom, 3 pixels



Step 1: New "Add Employee" button

- Add Employee button will href link to
 - request mapping */showFormForAdd*

```
<a th:href="@{/showFormForAdd}"  
class="btn btn-primary btn-sm mb-3">  
Add Employee  
</a>
```



TODO:
Add controller request mapping for
/showFormForAdd

Showing Form

In our Spring Controller

- Before you show the form, you must add a *model attribute*
- This is an object that will hold form data for the *data binding*

Controller code to show form

```
@Controller  
public class EmployeeController {  
  
    @GetMapping("/showFormForAdd")  
    public String showFormForAdd(Model theModel) {  
  
        // create model attribute to bind form data  
        Employee theEmployee = new Employee();  
  
        theModel.addAttribute("employee", theEmployee);  
  
        return "employee-form";  
    }  
    ...  
}
```

src/main/resources/templates/employee-form.html

Our Thymeleaf template will access this data for binding form data



Thymeleaf and Spring MVC Data Binding

- Thymeleaf has special expressions for binding Spring MVC form data
- Automatically setting / retrieving data from a Java object

Thymeleaf Expressions

- Thymeleaf expressions can help you build the HTML form

Expression	Description
<code>th:action</code>	Location to send form data
<code>th:object</code>	Reference to model attribute
<code>th:field</code>	Bind input field to a property on model attribute
<code>more ...</code>	

Step 2: Create HTML form for new employee

Empty place holder
Thymeleaf will handle real work

Real work
Send form data to
`/employees/save`

```
<form action="#" th:action="@{/save}"  
      th:object="${employee}" method="POST">
```

```
</form>
```

Our model attribute

```
theModel.addAttribute("employee", theEmployee);
```

Step 2: Create HTML form for new employee

ADD or UPDATE Employee

Step 2: Create HTML form for new employee

```
<form action="#" th:action="@{/save}"  
      th:object="${employee}" method="POST">  
  
<input type="text" th:field="*{firstName}" placeholder="First name">  
  
<input type="text" th:field="*{lastName}" placeholder="Last name">  
  
<input type="text" th:field="*{email}" placeholder="Email">  
  
<button type="submit">Save</button>  
  
</form>
```

*{...}
Selects property on referenced
th:object

ADD or UPDATE Employee

First name
Last name
Email
Save

Step 2: Create HTML form for new employee

```
<form action="#" th:action="@{/save}"  
      th:object="${employee}" method="POST">  
  
  <input type="text" th:field="*{firstName}" placeholder="First name">  
  
  <input type="text" th:field="*{lastName}" placeholder="Last name">  
  
  <input type="text" th:field="*{email}" placeholder="Email">  
  
<button type="submit">Save</button>  
</form>
```

1

When form is **loaded**,
will call:

employee.getFirstName()

...

employee.getLastName

2

When form is **submitted**,
will call:

employee.setFirstName(...)

...

employee.setLastName(...)

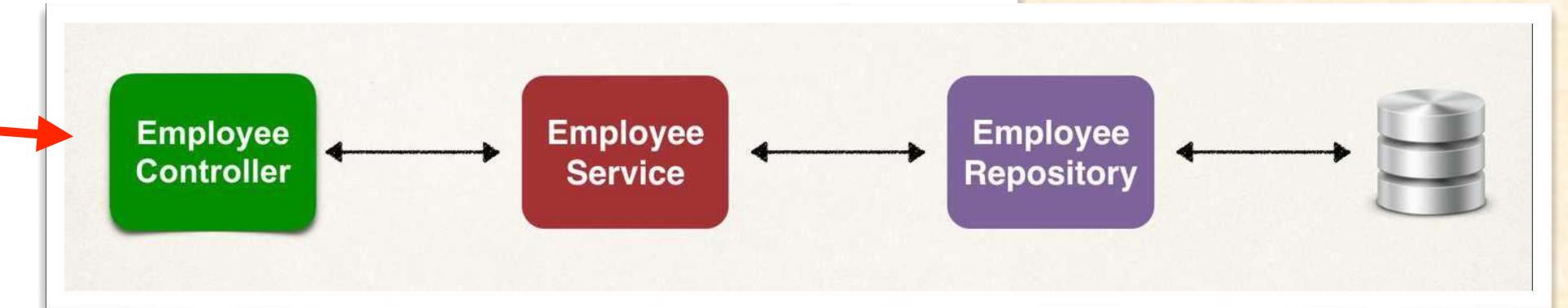
Step 3: Process form data to save employee

```
Since only one constructor  
@Controller  
  
public class EmployeeController {  
  
    private EmployeeService employeeService;  
  
    public EmployeeController(EmployeeService theEmployeeService) {  
        employeeService = theEmployeeService;  
    }  
  
    @PostMapping("/save")  
    public String saveEmployee(@ModelAttribute("emp":  
    {  
        // save the employee  
        employeeService.save(theEmployee);  
  
        // use a redirect to prevent duplicate submissions  
        return "redirect:/list";  
    }  
    ...  
}
```

Constructor injection

Step 3: Process form data to save employee

```
@Controller  
#@RequestMapping("/employees")  
public class EmployeeController {  
  
    private EmployeeService employeeService;  
  
    public EmployeeController(EmployeeService theEmployeeService) {  
        employeeService = theEmployeeService;  
    }  
  
    @PostMapping("/save")  
    public String saveEmployee(@ModelAttribute("employee") Employee theEmployee) {  
  
        // save the employee  
        employeeService.save(theEmployee);  
  
        // use a redirect to prevent duplicate submissions  
        return "redirect:/list";  
    }  
    ...  
}
```



Step 3: Process form data to save employee

```
@Controller
public class EmployeeController{

    private EmployeeService employeeService;

    public EmployeeController(EmployeeService theEmployeeService) {
        employeeService = theEmployeeService;
    }

    @PostMapping("/save")
    public String saveEmployee(@ModelAttribute("employee") Employee theEmployee) {

        // save the employee
        employeeService.save(theEmployee);

        // use a redirect to prevent duplicate submissions
        return "redirect:/list";
    }
    ...
}
```

Spring Boot (REST API, MVC and Microservices)

Spring MVC CRUD – EMS – Update Employee

Update Employee

Employee Directory

Add Employee

First Name	Last Name	Email	Action
liam	nessom	liam@nessom.com	<button>Update</button> <button>Delete</button>
bruce	willis	bruce@willis.com	<button>Update</button> <button>Delete</button>
dengel	washington	dengel@washington.com	<button>Update</button> <button>Delete</button>
Angelina	Jolie	angelina@jolie.com	<button>Update</button> <button>Delete</button>
Brad	Pitt	brad@pitt.com	<button>Update</button> <button>Delete</button>
Emma	Stone	emma@stone.com	<button>Update</button> <button>Delete</button>
Nocolas	Cage	nicolas@cage.net	<button>Update</button> <button>Delete</button>
NocolasJr	Cage	nicolasjr@cage.net	<button>Update</button> <button>Delete</button>

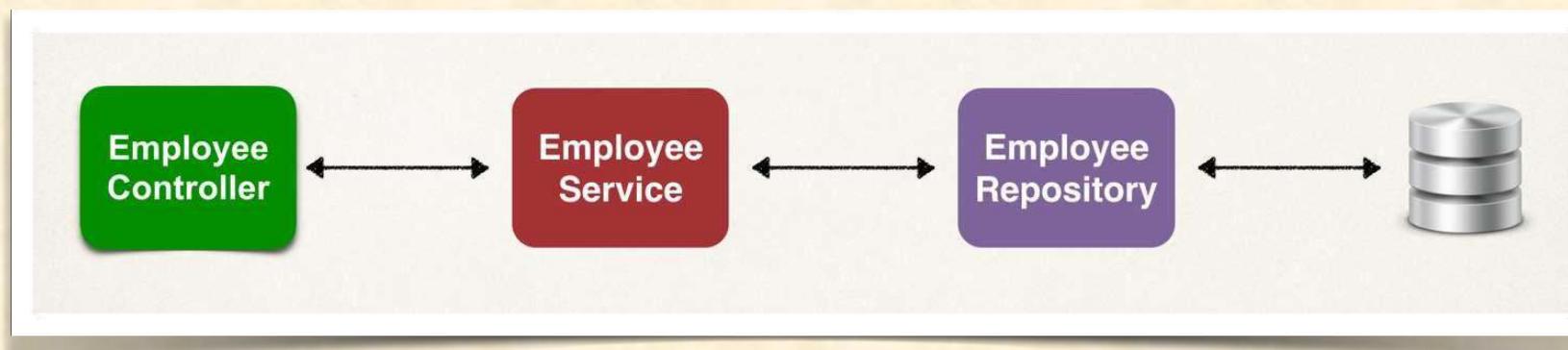
Update Employee

1. "Update" button

2. Pre-populate the form

3. Process form data

Employee Directory			
First Name Last Name Email			Action
liam	nessom	liam@nessom.com	<button>Update</button> <button>Delete</button>
bruce	willis	bruce@willis.com	<button>Update</button> <button>Delete</button>
dengel	washington	dengel@washington.com	<button>Update</button> <button>Delete</button>
Angelina	Jolie	angelina@jolie.com	<button>Update</button> <button>Delete</button>
Brad	Pitt	brad@pitt.com	<button>Update</button> <button>Delete</button>
Emma	Stone	emma@stone.com	<button>Update</button> <button>Delete</button>
Nocolas	Cage	nicolas@cage.net	<button>Update</button> <button>Delete</button>
NocolasJr	Cage	nicolasjr@cage.net	<button>Update</button> <button>Delete</button>



Step 1: "Update" Button

Employee Directory

Add Employee

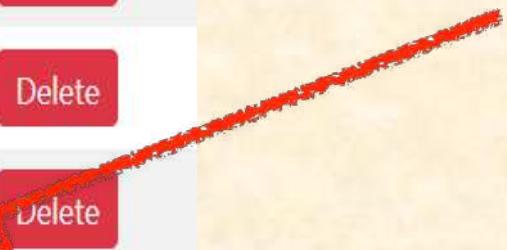
First Name	Last Name	Email	Action
liam	nessom	liam@nessom.com	Update Delete
bruce	willis	bruce@willis.com	Update Delete
dengel	washington	dengel@washington.com	Update Delete
Angelina	Jolie	angelina@jolie.com	Update Delete
Brad	Pitt	brad@pitt.com	Update Delete
Emma	Stone	emma@stone.com	Update Delete
Nocolas	Cage	nicolas@cage.net	Update Delete
NocolasJr	Cage	nicolasjr@cage.net	Update Delete

Each row has an **Update** link

- current employee id embedded in link

When **clicked**

- will load the employee from database
- prepopulate the form



Step 1: "Update" button

- Update button includes employee id

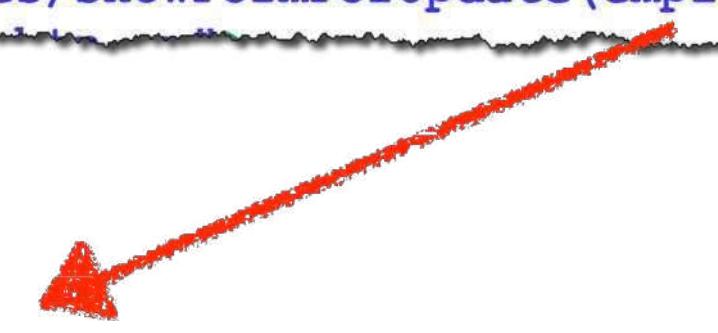
```
<tr th:each="tempEmployee : ${employees}">  
...  
<td>  
  
    <a th:href="@{/showFormForUpdate(employeeId=${tempEmployee.id})}"  
        class="btn btn-info btn-sm">  
        Update  
    </a>  
</td>  
</tr>
```

Appends to URL

?employeeId=xxx

Step 2: Pre-populate Form

```
@Controller  
#@RequestMapping("/employees") public class EmployeeController {  
    ...  
  
    @GetMapping("/showFormForUpdate")  
    public String showFormForUpdate(@RequestParam("employeeId") int theId,  
                                    Model theModel) {  
  
        // get the employee from the service  
        Employee theEmployee = employeeService.findById(theId);  
  
        // set employee as a model attribute to pre-populate the form  
        theModel.addAttribute("employee", theEmployee);  
  
        // send over to our form  
        return "/employee-form";  
    }  
}
```



Step 2: Pre-populate Form

1

When form is **loaded**,
will call:

employee.getFirstName()

...

employee.getLastName

```
<form action="#" th:action="@{/save}"
      th:object="${employee}" method="POST">

    <!-- Add hidden form field to handle update -->
    <input type="hidden" th:field="*{id}" />

    <input type="text" th:field="*{firstName}"
           class="form-control mb-4 w-25" placeholder="First name">

    <input type="text" th:field="*{lastName}"
           class="form-control mb-4 w-25" placeholder="Last name">

    <input type="text" th:field="*{email}"
           class="form-control mb-4 w-25" placeholder="Email">

    <button type="submit" class="btn btn-info col-2">Save</button>

</form>
```

This is how form is
pre-populated
Thanks to calls to getters

Step 2: Pre-populate Form

```
<form action="#" th:action="@{/employees/save}"
      th:object="${employee}" method="POST">
    <!-- Add hidden form field to handle update -->
    <input type="hidden" th:field="*{id}" />

    <input type="text" th:field="*{firstName}"
           class="form-control mb-4 w-25" placeholder="First name">

    <input type="text" th:field="*{lastName}"
           class="form-control mb-4 w-25" placeholder="Last name">

    <input type="text" th:field="*{email}"
           class="form-control mb-4 w-25" placeholder="Email">

    <button type="submit" class="btn btn-info col-2">Save</button>

</form>
```

Hidden form field
required for updates

Step 2: Pre-populate Form

```
<form action="#" th:action="@{/employees/save}"
      th:object="${employee}" method="POST">

    <!-- Add hidden form field to handle update -->
    <input type="hidden" th:field="*{id}" />

    <input type="text" th:field="*{firstName}"
           class="form-control mb-4 w-25" placeholder="First name">

    <input type="text" th:field="*{lastName}"
           class="form-control mb-4 w-25" placeholder="Last name">

    <input type="text" th:field="*{email}"
           class="form-control mb-4 w-25" placeholder="Email">

    <button type="submit" class="btn btn-info col-2">Save</button>

</form>
```

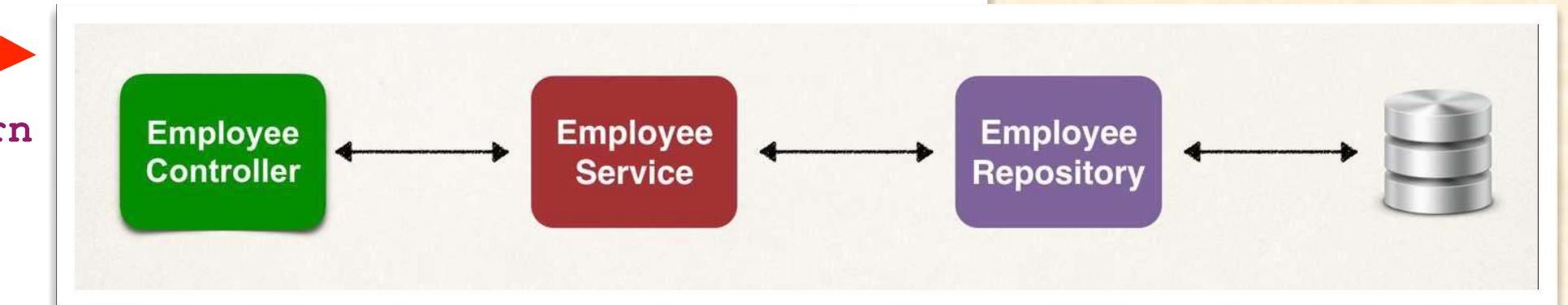
This binds to the model attribute

Tells our app
which employee to update

Step 3: Process form data to save employee

- No need for new code ... we can reuse our existing code
- Works the same for add or update

```
@Controller  
#@RequestMapping("/employees") public  
class EmployeeController {  
  
    ...  
  
    @PostMapping("/save")  
    public String saveEmployee(@ModelAttribute("employee") Employee theEmployee) {  
  
        // save the employee  
        employeeService.save(theEmployee);  
  
        // use a redirect to prevent duplicate submissions return  
        "redirect:/list";  
    }  
    ...  
}
```



Spring Boot (REST API, MVC and Microservices)

Spring MVC CRUD – EMS – Delete Employee

Delete Employee

Employee Directory

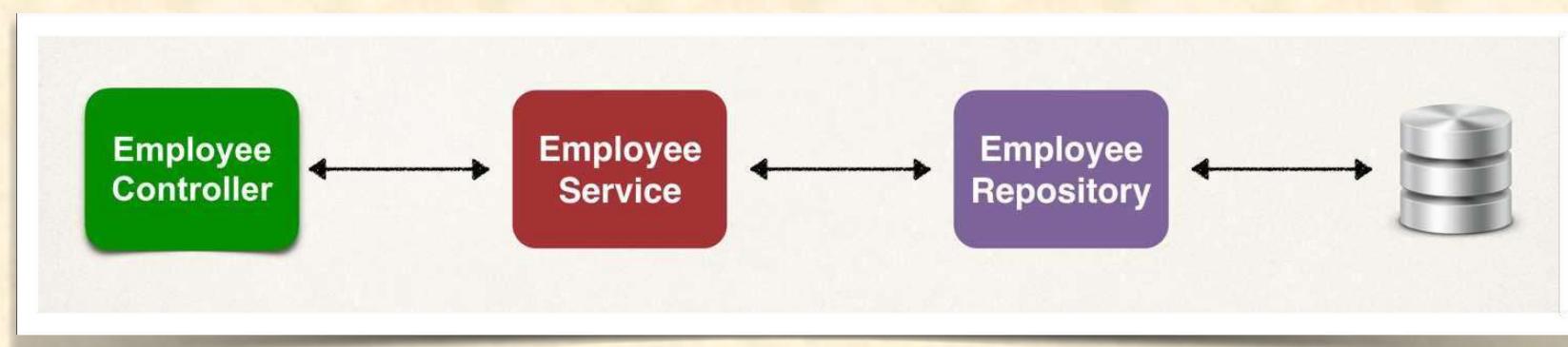
[Add Employee](#)

First Name	Last Name	Email	Action
liam	nessom	liam@nessom.com	Update Delete
bruce	willis	bruce@willis.com	Update Delete
dengel	washington	dengel@washington.com	Update Delete
Angelina	Jolie	angelina@jolie.com	Update Delete
Brad	Pitt	brad@pitt.com	Update Delete
Emma	Stone	emma@stone.com	Update Delete
Nocolas	Cage	nicolas@cage.net	Update Delete
NocolasJr	Cage	nicolasjr@cage.net	Update Delete

Delete Employee

1. Add “Delete” button/link on page

2. Add controller code for “Delete”



Employee Directory

Add Employee			
First Name	Last Name	Email	Action
liam	nessom	liam@nessom.com	<button>Update</button> <button>Delete</button>
bruce	willis	bruce@willis.com	<button>Update</button> <button>Delete</button>
dengel	washington	dengel@washington.com	<button>Update</button> <button>Delete</button>
Angelina	Jolie	angelina@jolie.com	<button>Update</button> <button>Delete</button>
Brad	Pitt	brad@pitt.com	<button>Update</button> <button>Delete</button>
Emma	Stone	emma@stone.com	<button>Update</button> <button>Delete</button>
Nocolas	Cage	nicolas@cage.net	<button>Update</button> <button>Delete</button>
NocolasJr	Cage	nicolasjr@cage.net	<button>Update</button> <button>Delete</button>

Step 1: "Delete" button

Employee Directory

Add Employee

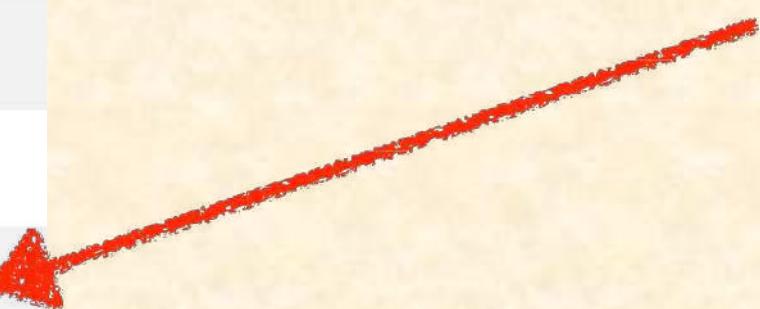
First Name	Last Name	Email	Action
liam	nessom	liam@nessom.com	Update Delete
bruce	willis	bruce@willis.com	Update Delete
dengel	washington	dengel@washington.com	Update Delete
Angelina	Jolie	angelina@jolie.com	Update Delete
Brad	Pitt	brad@pitt.com	Update Delete
Emma	Stone	emma@stone.com	Update Delete
Nocolas	Cage	nicolas@cage.net	Update Delete
NocolasJr	Cage	nicolasjr@cage.net	Update Delete

Each row has a **Delete** button/link

- current employee id embedded in link

When **clicked**

- prompt user
- will delete the employee from database



Step 1: "Delete" button

- Delete button includes employee id

```
<tr th:each="tempEmployee : ${employees}">
...
<td>

    <a th:href="@{/delete (employeeId=${tempEmployee.id})}"
       class="btn btn-danger btn-sm"
       onclick="if (!confirm('Are you sure you want to delete this employee?')) return false">
        Delete
    </a>

</td>
</tr>
```

Appends to URL

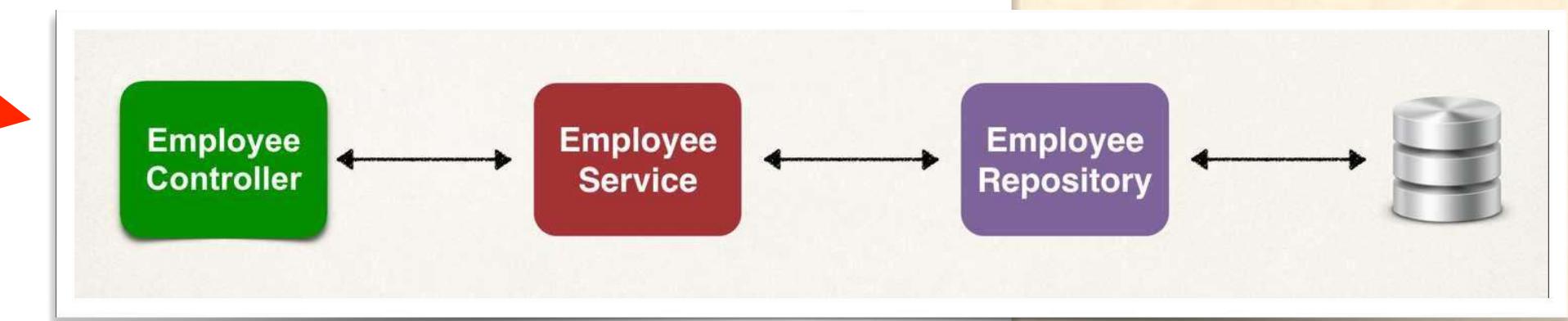
?employeeId=xxx

JavaScript to prompt user before deleting

Step 2: Add controller code for delete

```
@Controller  
#@RequestMapping("/employees")  
public class EmployeeController {  
    ...  
  
    @GetMapping("/delete")  
    public String delete(@RequestParam("employeeId") int theId) {  
  
        // delete the employee  
        employeeService.deleteById(theId);  
  
        // redirect to /employees/list  
        return "redirect:/list";  
    }  
    ...  
}
```

```
<a th:href="@{/employees/delete(employeeId=${tempEmployee.id})}"
```



Spring Boot (REST API, MVC and Microservices)

Introduction to Microservices

The image features a yellow background with a white rectangular overlay containing the Spring Boot and Java logos. The Spring Boot logo consists of a green hexagon with a white power button symbol, followed by the words "spring boot" in green lowercase letters. To the left of this is the Java logo, which includes a blue flame icon above the word "Java". To the right of the logo area is a white lightbulb with radiating lines, symbolizing ideas or innovation. At the bottom right, there is a white rounded rectangle containing the name "SADHU SREENIVAS".

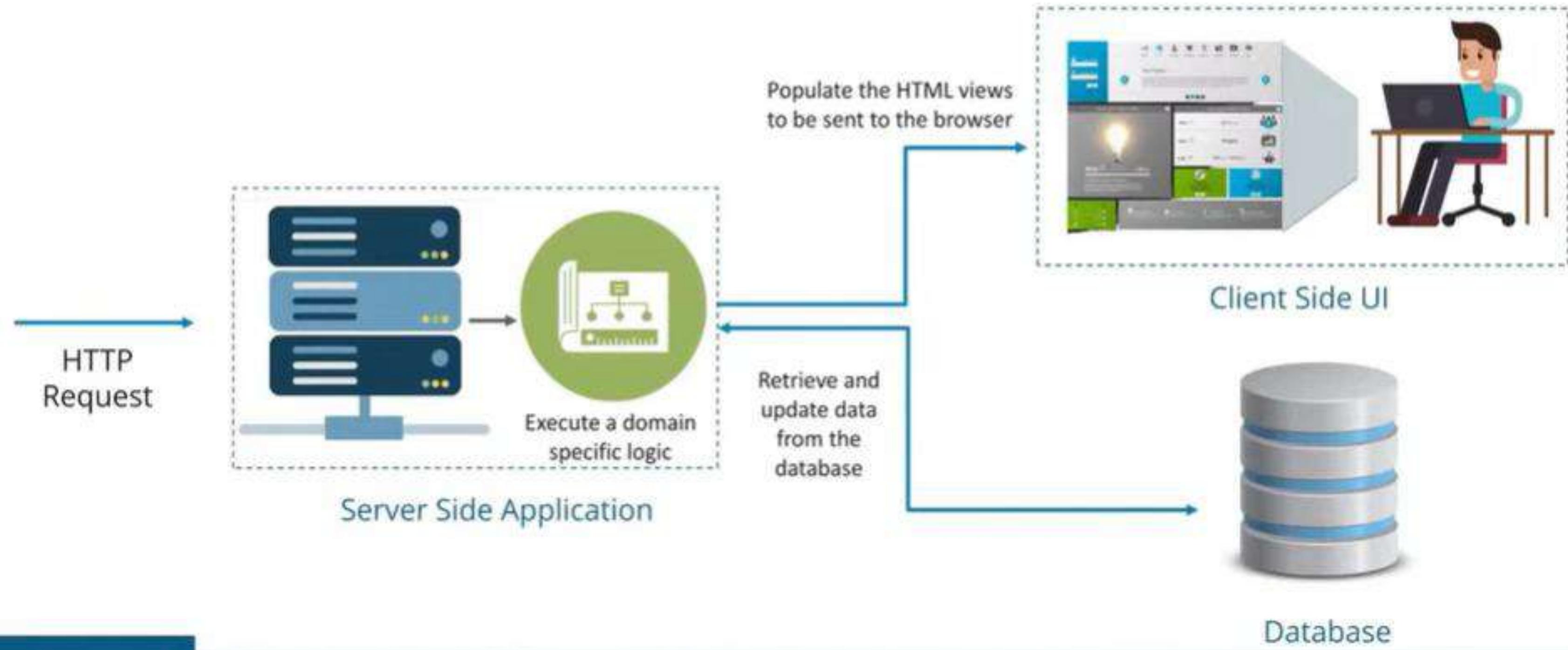
SPRING BOOT

Java

SADHU SREENIVAS

Before Microservices - Monolithic Architecture

Monolithic Architecture is like a big container wherein all the software components of an application are assembled together and tightly packaged



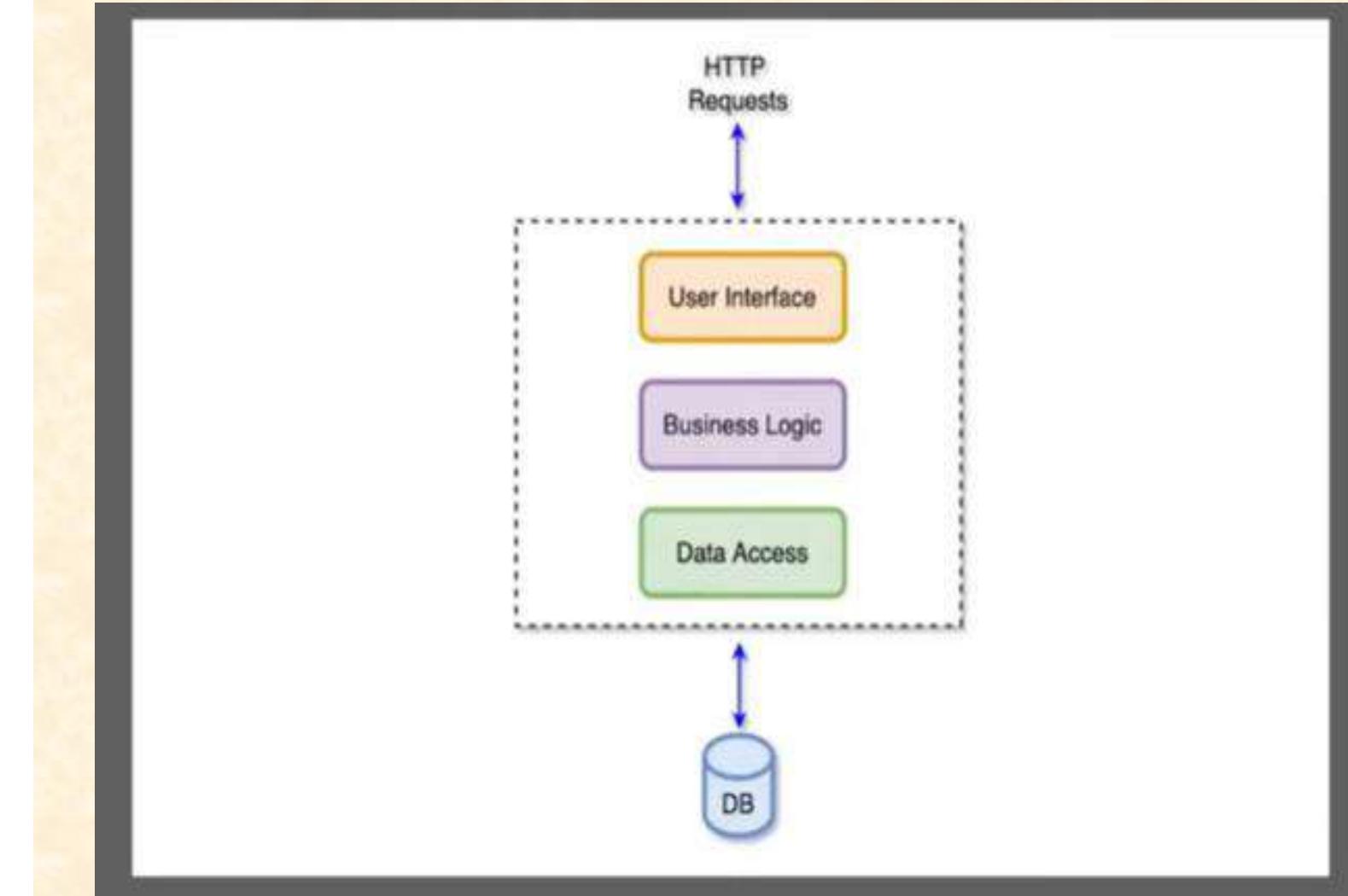
Monolithic Architecture -Before Microservices

Monolith

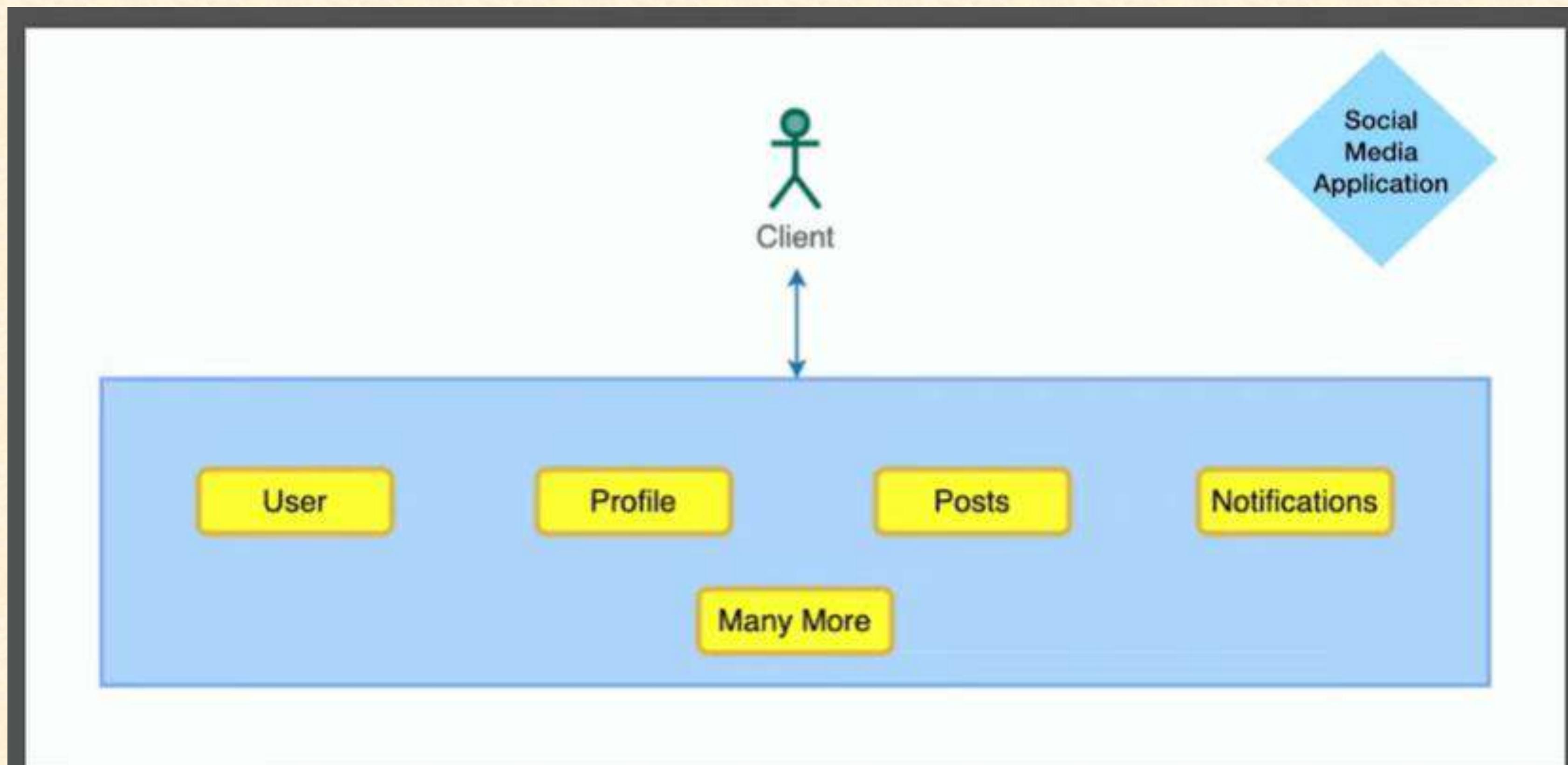
- ▶ All components are part of a **single unit**
- ▶ Everything is **developed, deployed and scaled as 1 unit**
- ▶ App must be written with **1 tech stack**
- ▶ Teams need to be careful to not affect each other's work
- ▶ **1 single artifact**, so you must redeploy the entire application on each update

```
graph TD; subgraph MONOLITHIC [MONOLITHIC]; direction LR; SA[shopping-cart] --- PA[payment]; end; subgraph TA [ ]; direction TB; UAU[user-auth]; end; subgraph PCU [ ]; direction TB; PCU[product-catalog]; end; subgraph SCU [ ]; direction TB; SCU[shopping-cart]; end; subgraph PAU [ ]; direction TB; PAU[payment]; end; TA --> UAU; TA --> PCU; TA --> SCU; TA --> PAU; SCU --> SCU_UA[ ]; PAU --> PAU_UA[ ];
```

Monolithic Architecture - Example 1



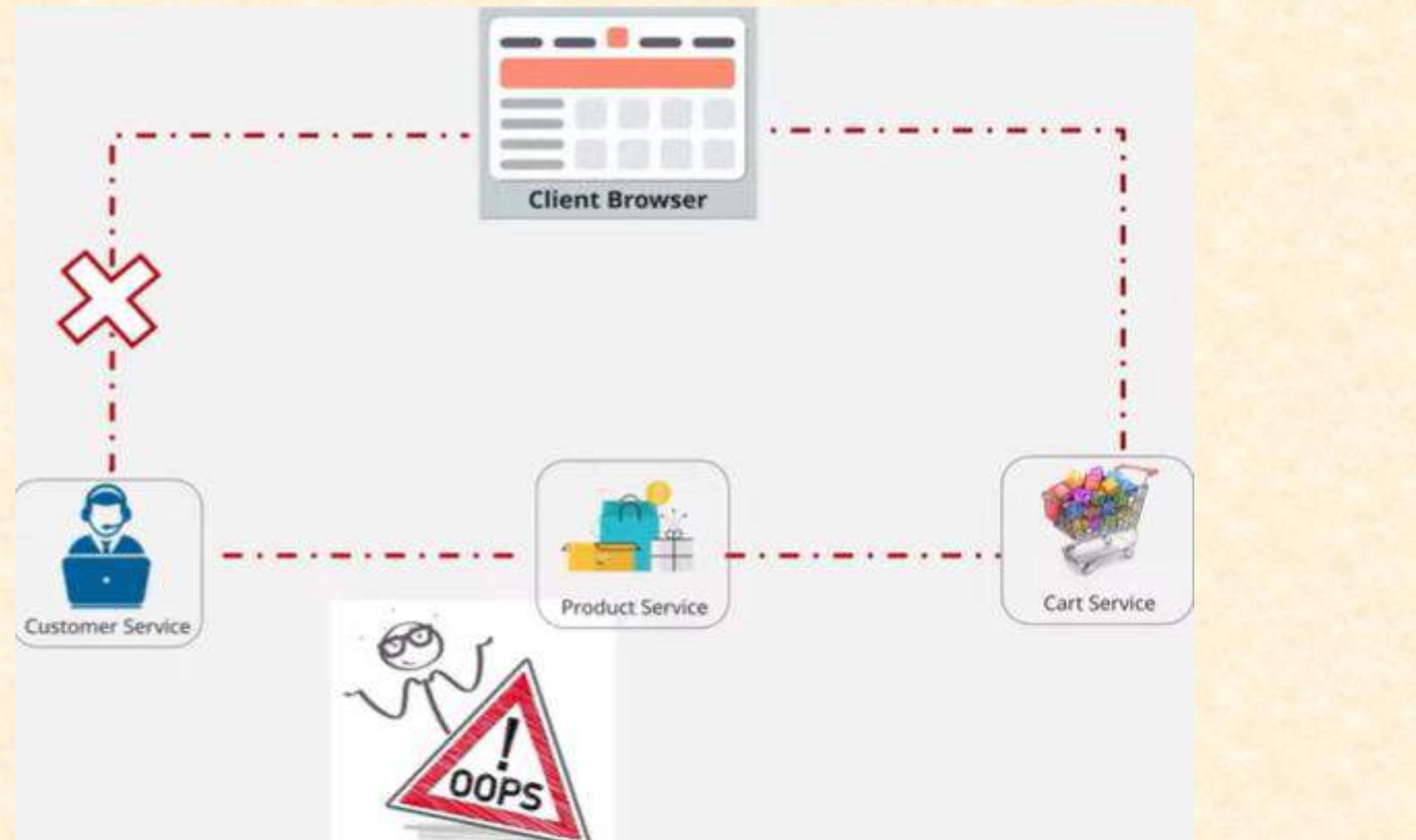
Monolithic Architecture - Example 2



Monolithic Architecture - Challenges

★ Large & Complex applications ★ Slow Development ★ Blocks continuous Development

★ Unscalable



★ Unreliable



★ Inflexible

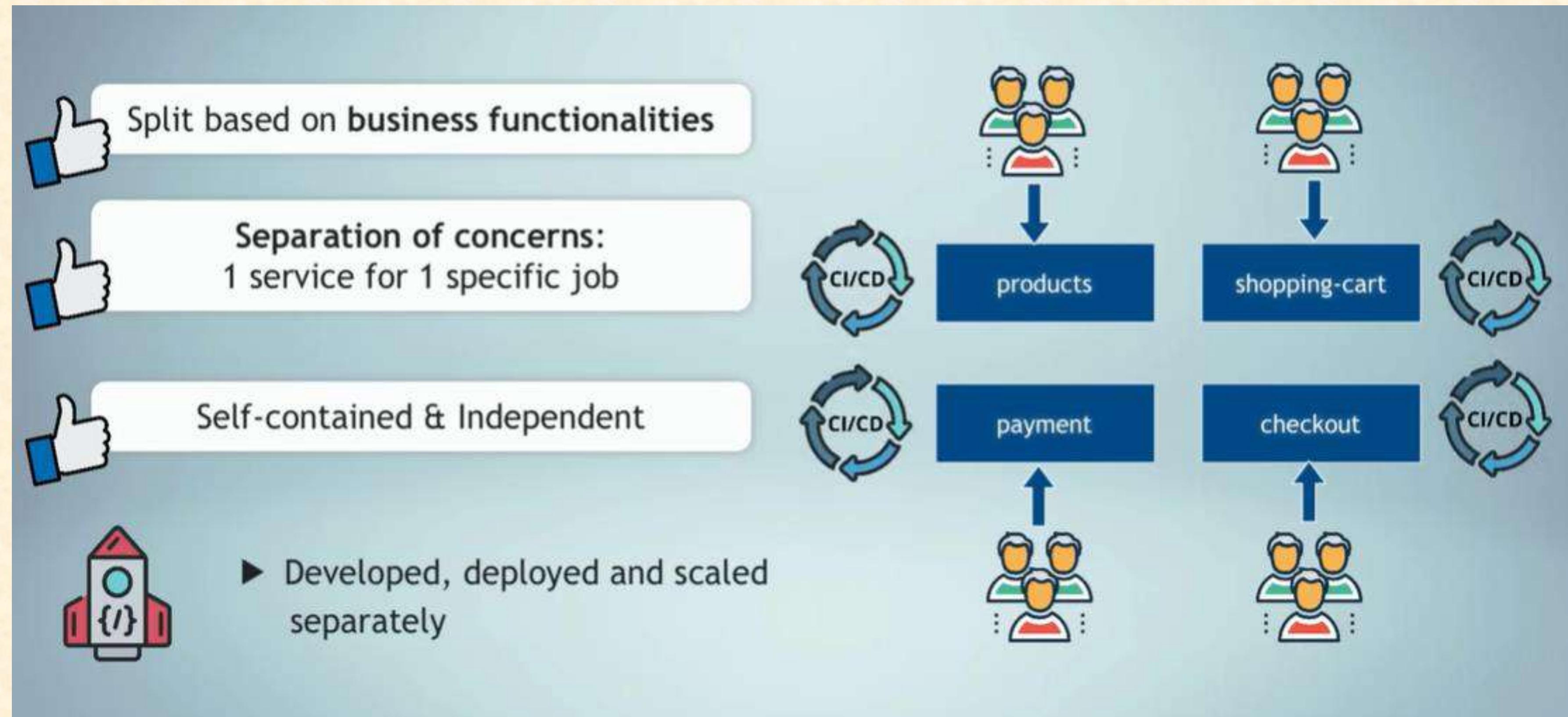
What are Microservices?

Microservices, aka *Microservice Architecture*, is an **architectural style** that structures an application as a **collection of small autonomous services**, modelled around a **Business Domain**

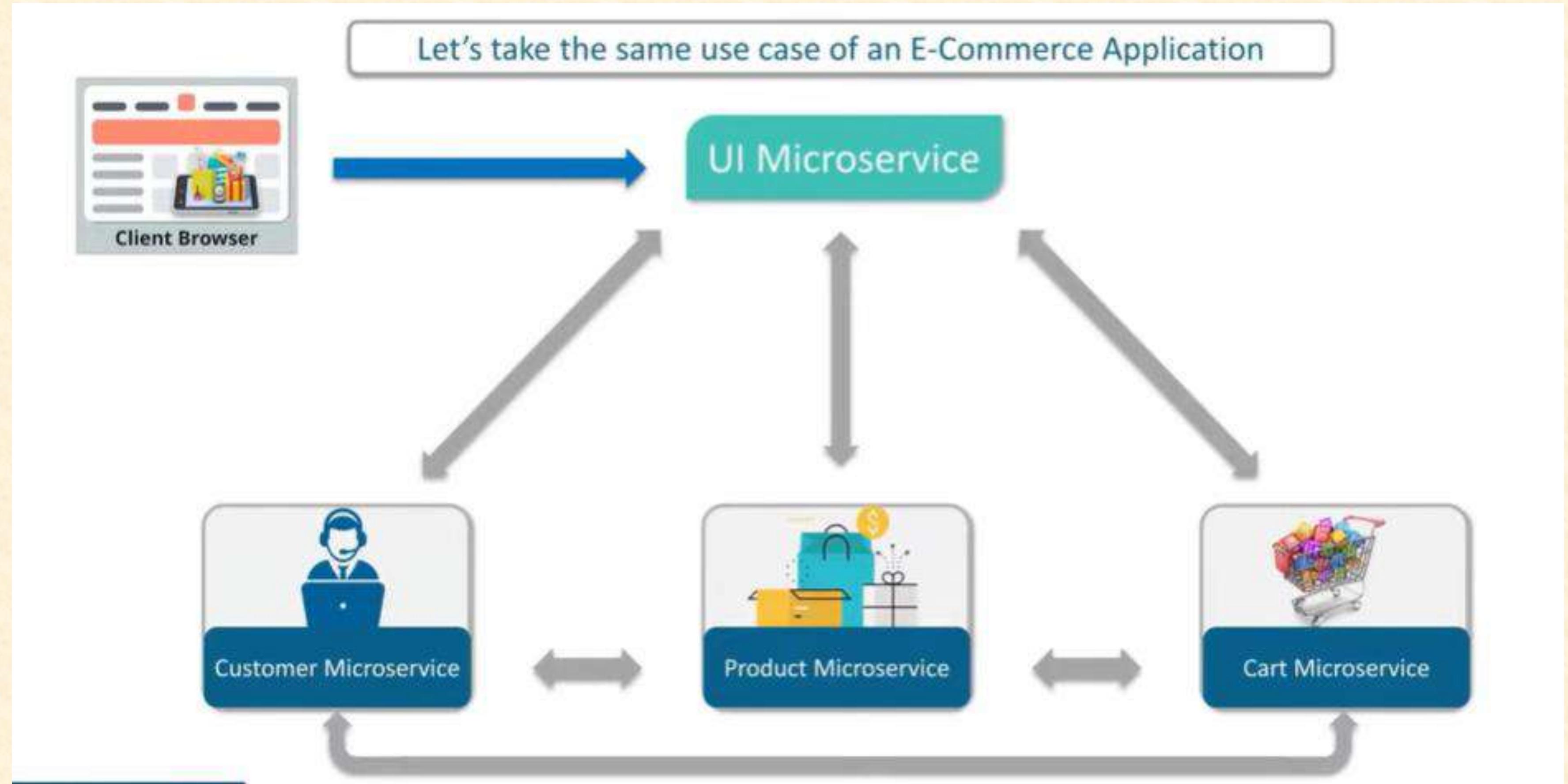


In Microservice Architecture, each service is **self-contained** and implements a **single Business capability**

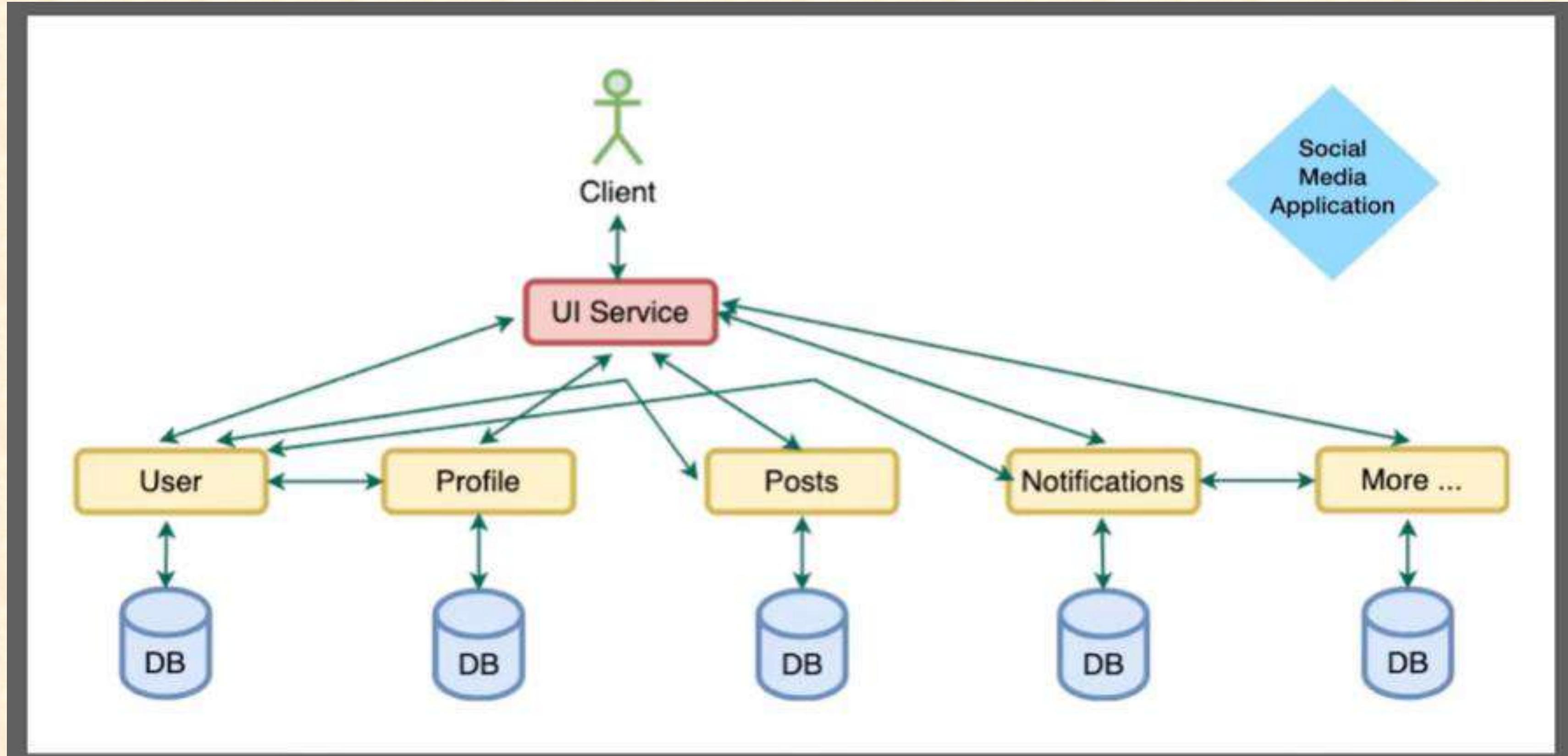
What are Microservices?



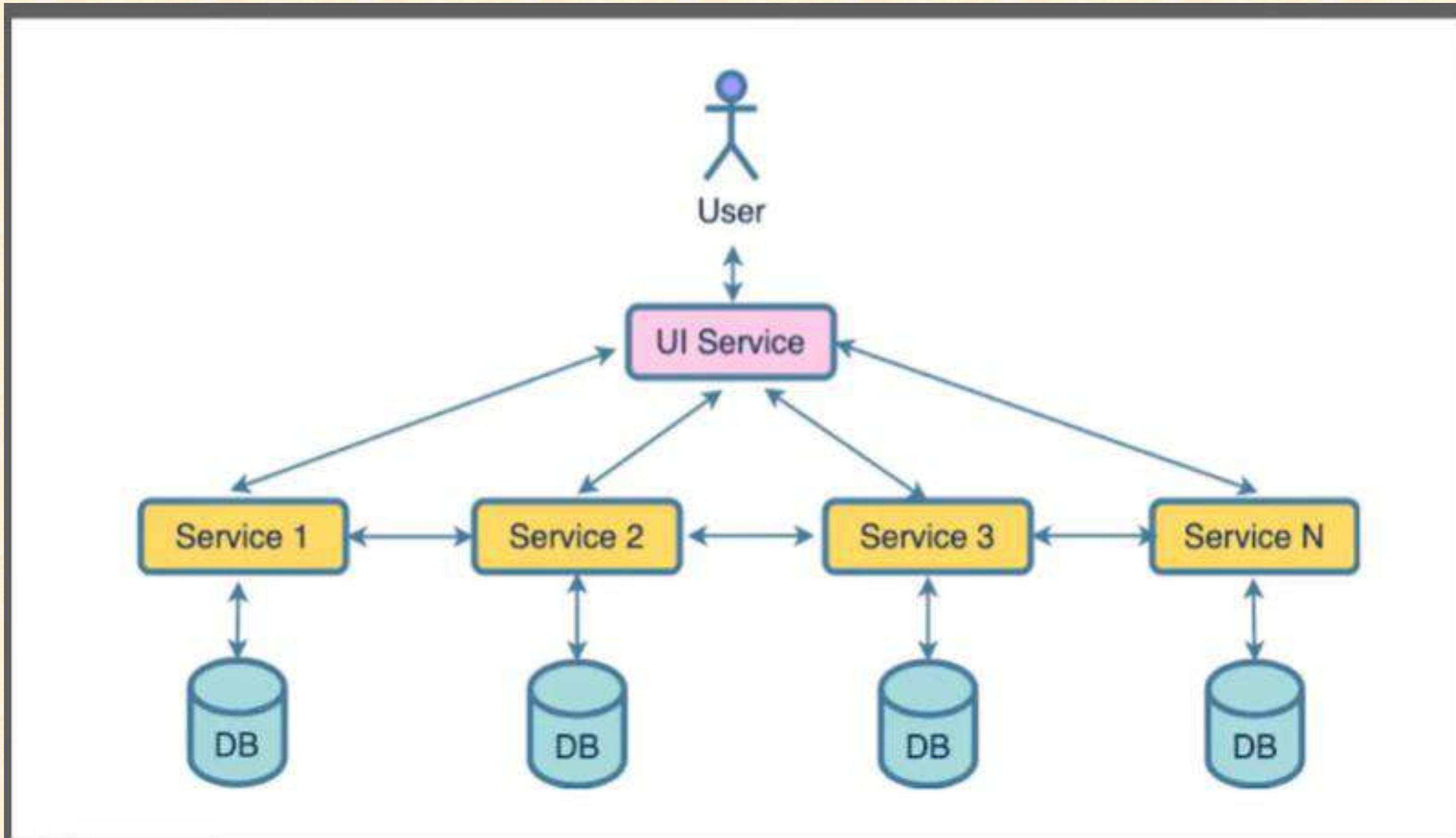
Microservices - Example



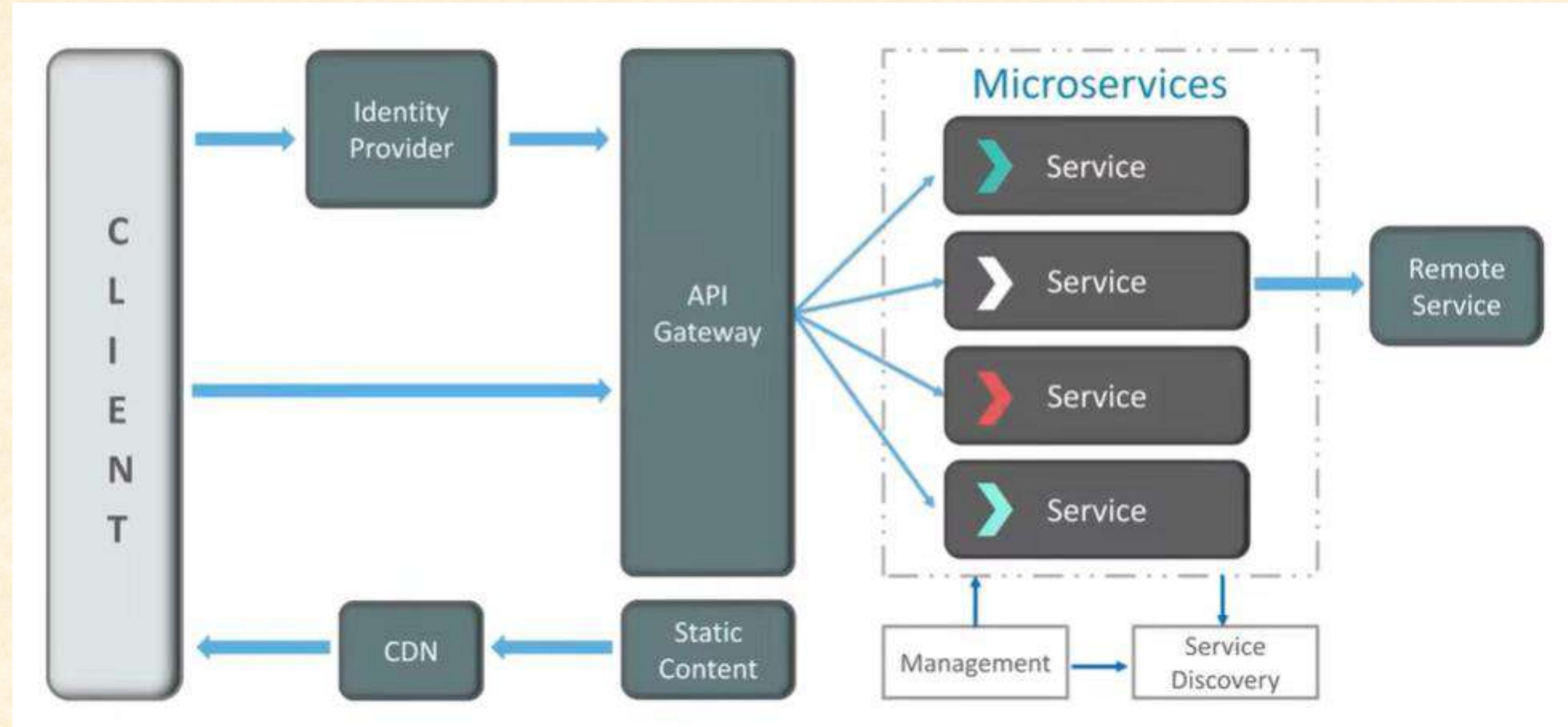
Microservices - Example



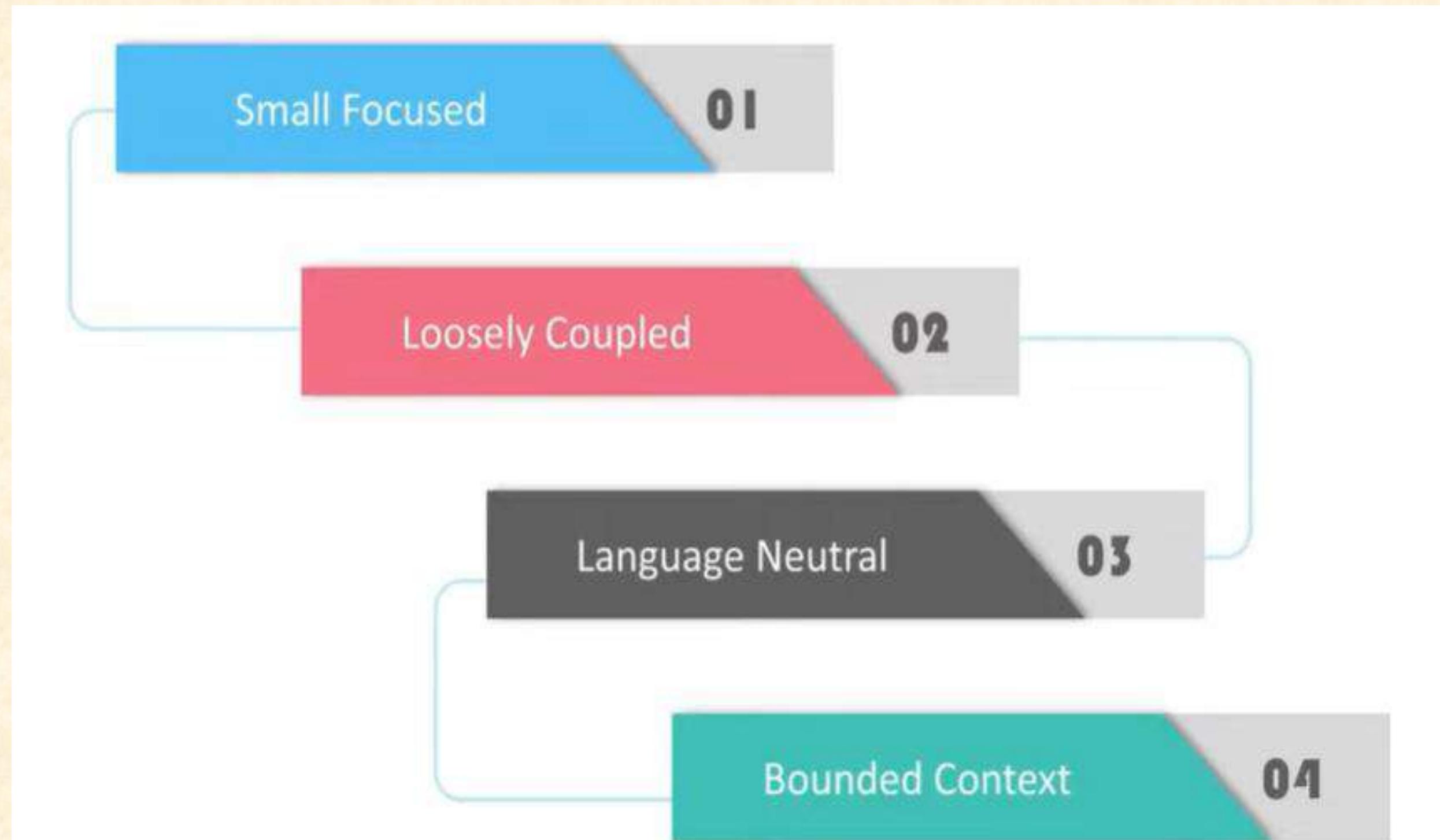
Microservices - Example



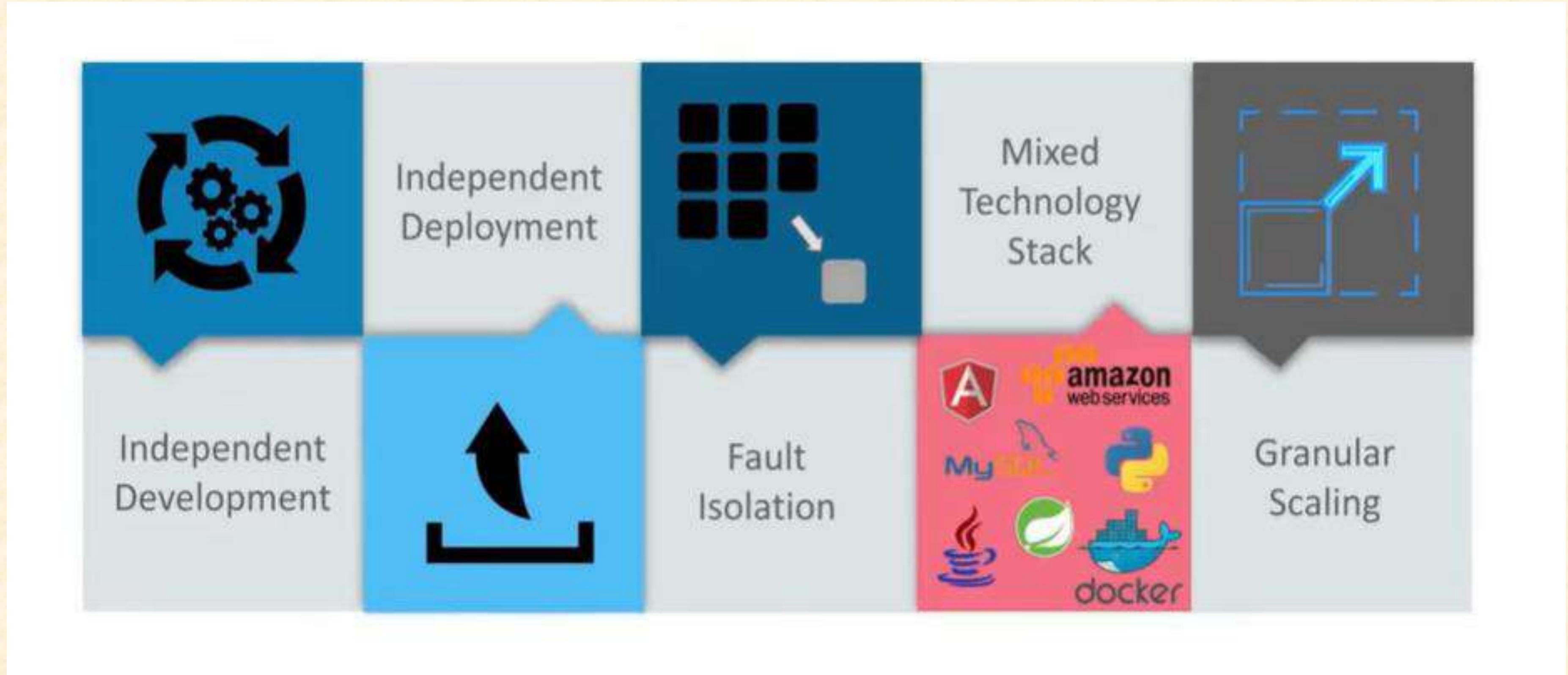
Microservices - Architecture



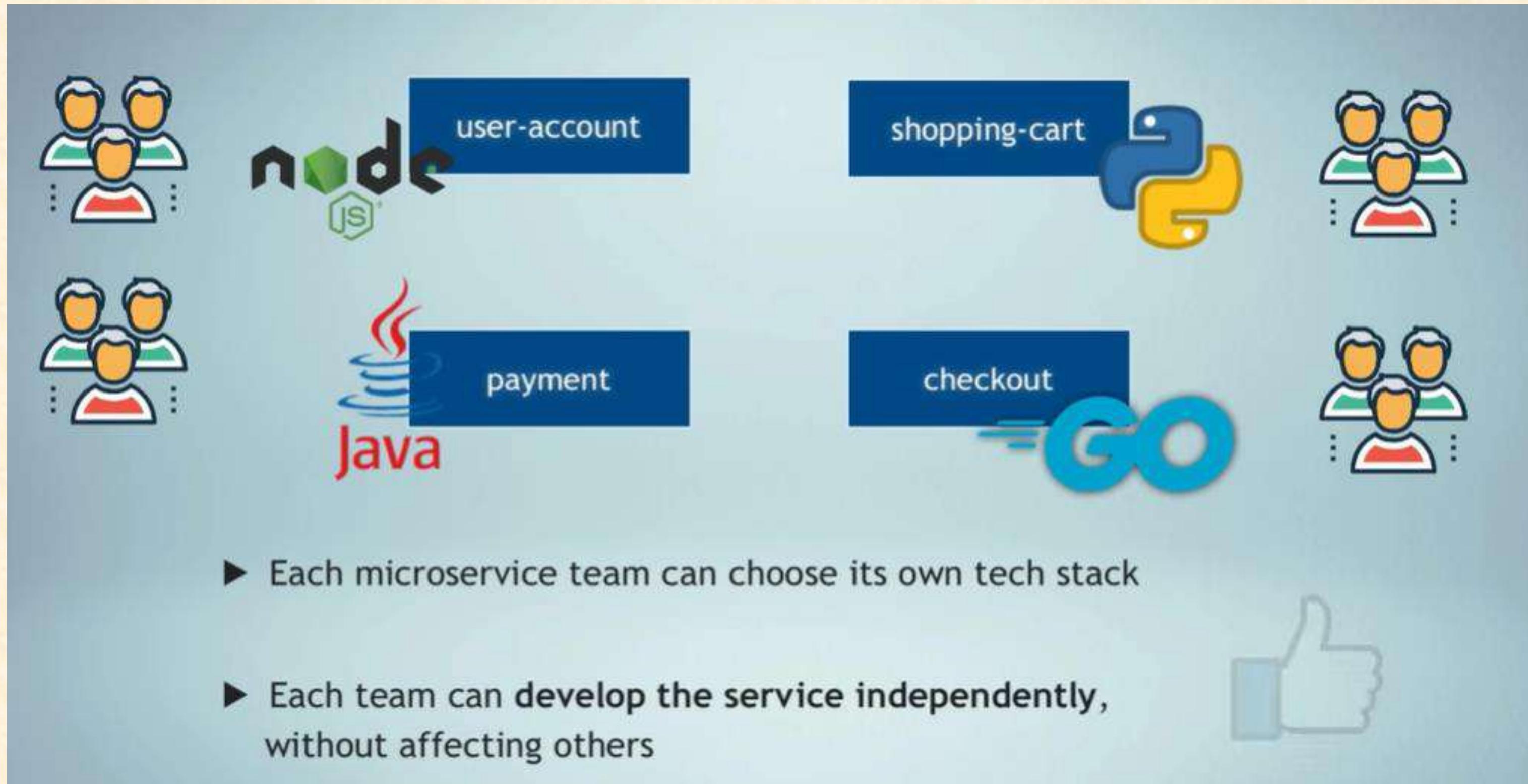
Microservices - Features



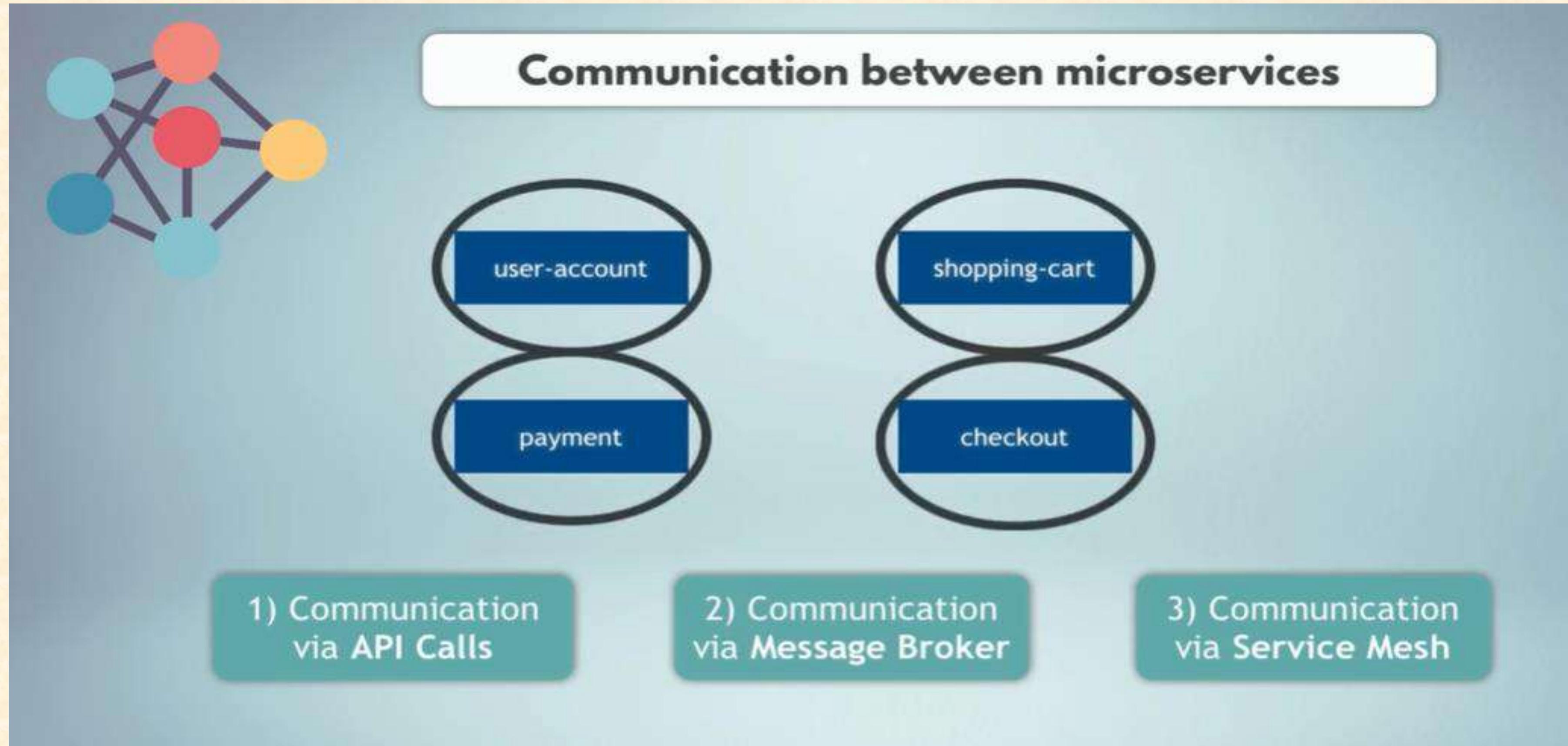
Microservices - Advantages



Microservices - Advantages



Microservices - Communication



Microservices – Users



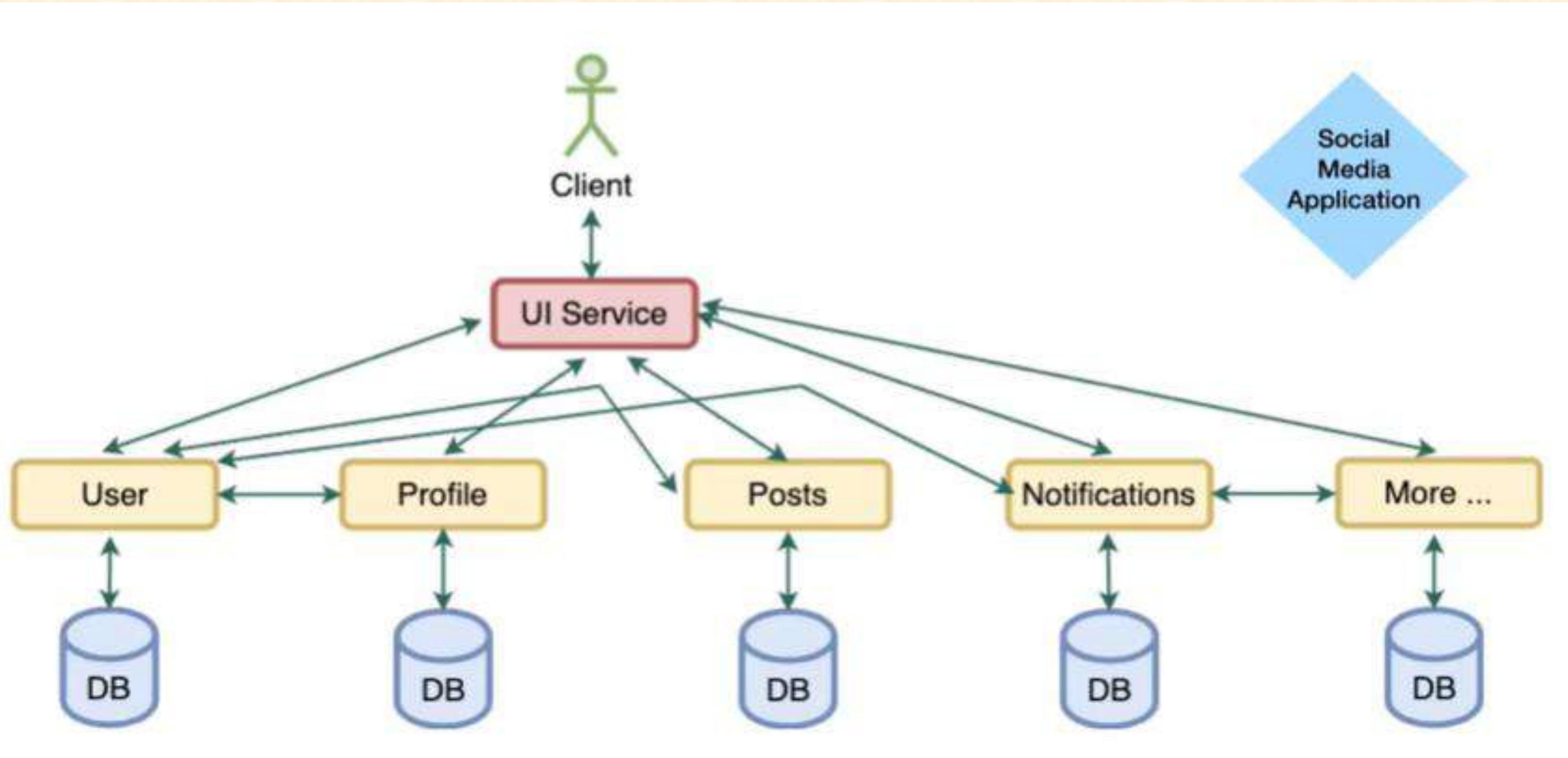
and many more...

Spring Boot (REST API, MVC and Microservices)

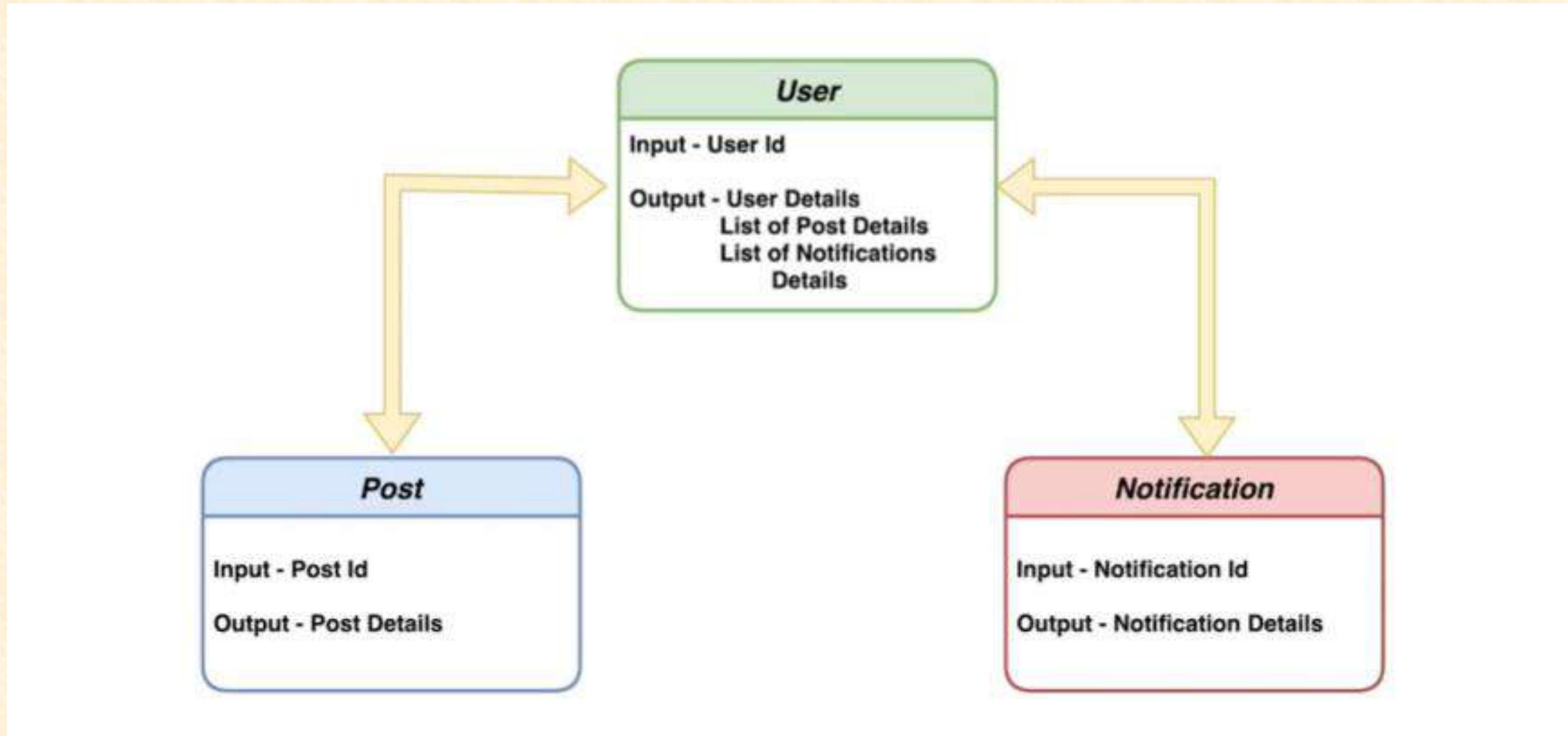
Basic Microservices App with Rest Template

Part - I

Microservices – Basic App Development



Microservices – Designing Services



Microservices – Development Process

- ★ Create 3 Spring Boot Projects
- ★ Build User API Service
- ★ Build Post API Service
- ★ Build Notification API Service
- ★ User Service calls to Post and Notification Service - Using
RestTemplate

Microservices - The 3 Services

The image shows a code editor with three tabs representing different microservices:

- User Service:** The application name is set to `UserService` and the port is `8080`. The code editor shows the `application.properties` file with the following content:

```
spring.application.name=UserService
server.port = 8080
```
- Post Service:** The application name is set to `PostService` and the port is `8081`. The code editor shows the `application.properties` file with the following content:

```
spring.application.name=PostService
server.port = 8081
```
- Notification Service:** The application name is set to `NotificationService` and the port is `8082`. The code editor shows the `application.properties` file with the following content:

```
spring.application.name=NotificationService
server.port = 8082
```

Each service has its own directory structure with Java source files (e.g., `Notifications.java`, `Posts.java`, `User.java`, `UserController.java`, `UserServiceApplication.java`) and configuration files (`application.properties`).

Microservices - Model and Controllers classes

```
public class User {  
    private int userId;  
    private String userName;  
}
```

```
public class Posts {  
    private String postId;  
    private String description;  
}
```

```
@RestController  
@RequestMapping("/users")  
public class UserController {...}
```

```
@RestController  
@RequestMapping("/posts")  
public class PostController { ... }
```

```
public class Notifications {  
    private String notificationId;  
    private String description;  
}
```

```
@RestController  
@RequestMapping("/notifications")  
public class NotificationController { ... }
```

Microservices - Up and Running Services

localhost:8080/users/1

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

```
userId: 1
userName: "ABC"
userPhoneNumber: 12345
posts:
  postId: "1"
  description: "Post1 Descriptions as follows"
notifications:
  notificationId: "1"
  description: "Notofication1 description as afollows.."
```

localhost:8081/posts/1

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

```
postId: "1"
description: "Post1 Descriptions as follows"

localhost:8082/notifications/1
```

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

```
notificationId: "1"
description: "Notofication1 description as afollows.."
```

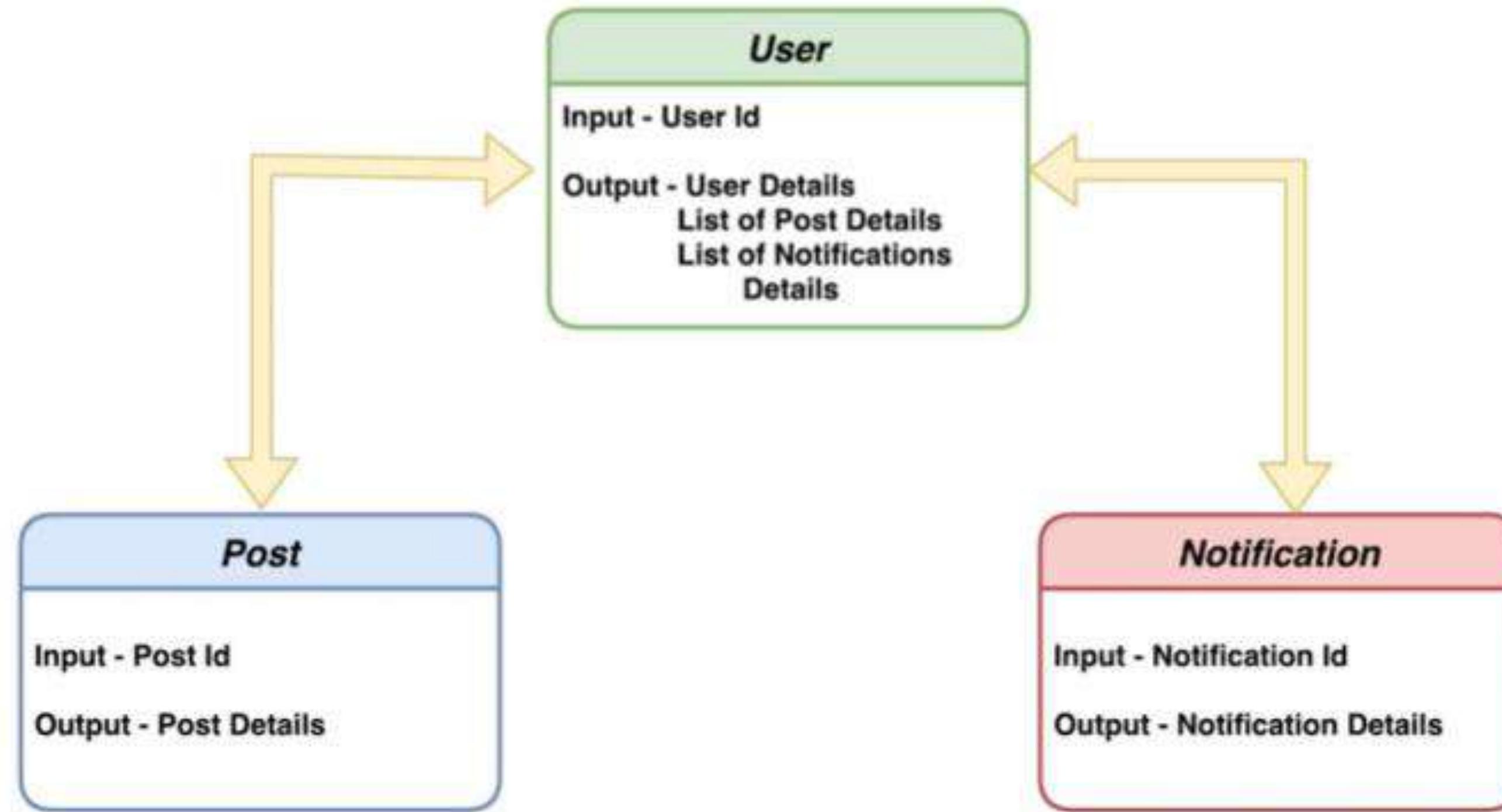
DEMO

Spring Boot (REST API, MVC and Microservices)

Basic Microservices App with Rest Template

Part - II

Calling Services with RestTemplate



RestTemplate

```
@Bean  
RestTemplate getRestTemplate() {  
    return new RestTemplate();  
}
```

```
Posts posts = restTemplate.getForObject("http://localhost:8081/posts/1", Posts.class);
```

```
Notifications notifications =  
restTemplate.getForObject("http://localhost:8082/notifications/1", Notifications.class);
```

Microservices - User Service Calling other Services

localhost:8080/users/1

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

```
userId: 1
userName: "ABC"
userPhoneNumber: 12345
posts:
  postId: "1"
  description: "Post1 Descriptions as follows"
notifications:
  notificationId: "1"
  description: "Notofication1 description as afollows.."
```

localhost:8081/posts/1

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

```
postId: "1"
description: "Post1 Descriptions as follows"
```

localhost:8082/notifications/1

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

```
notificationId: "1"
description: "Notofication1 description as afollows.."
```

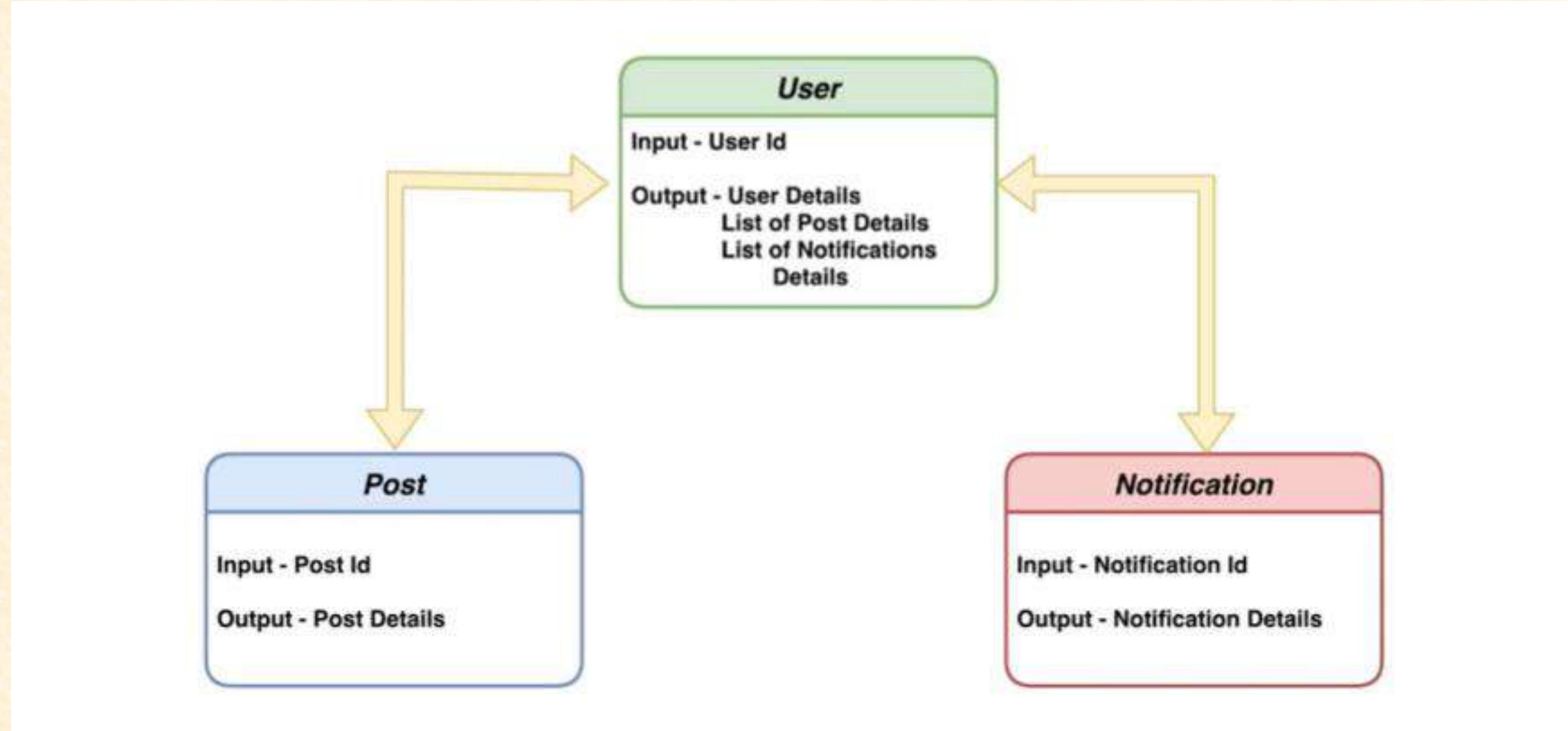
Spring Boot (REST API, MVC and Microservices)

Service Registration & Discovery – Netflix Eureka Server

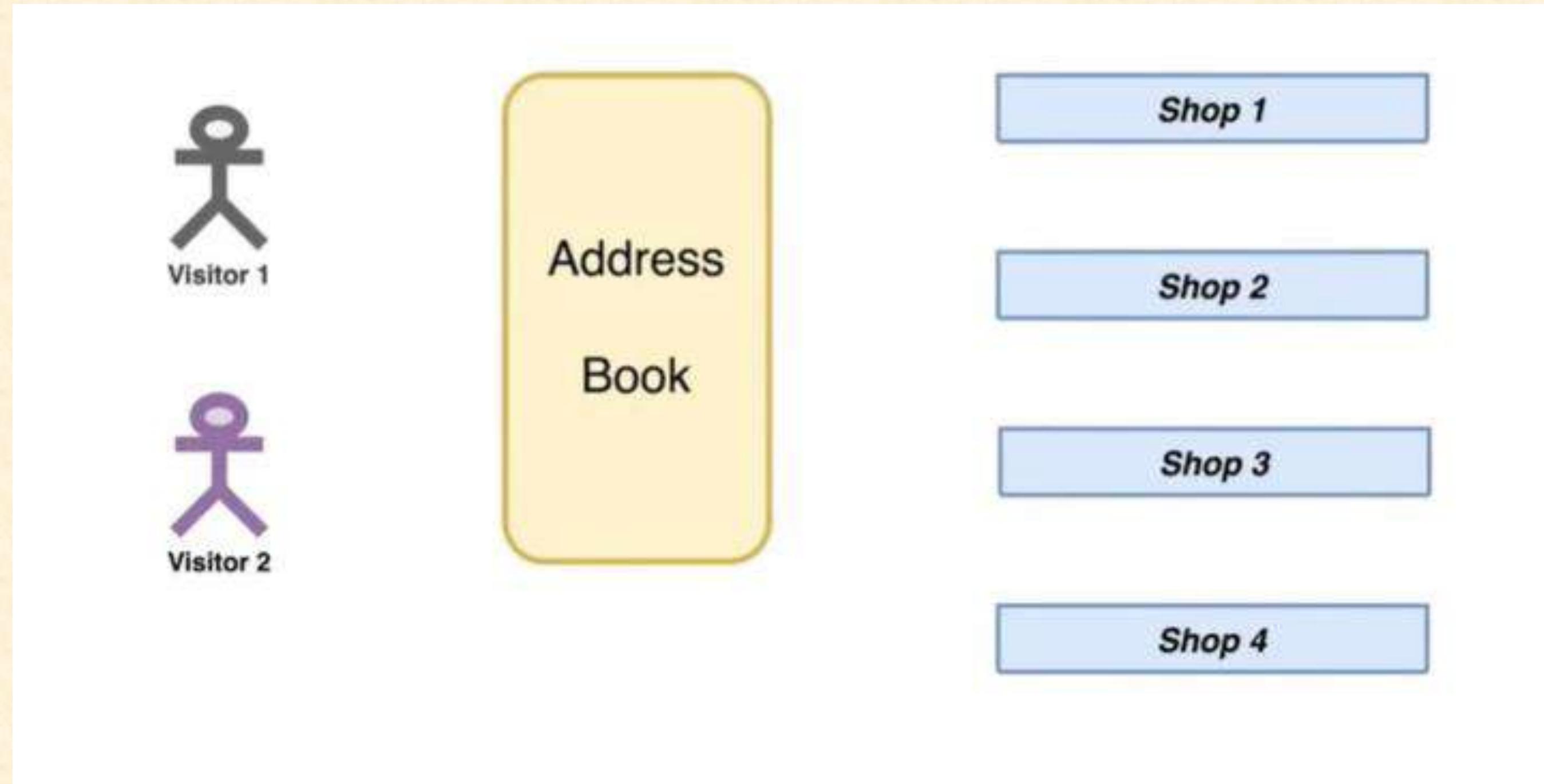
Microservices - Registration & Discovery

- ★ What is Service Registration and Discovery
- ★ What is Eureka Server and Client
- ★ How to create Eureka Server
- ★ How to create Eureka Client
- ★ Enable Eureka at User, Post and Notification Services
- ★ Create and enable multiple instances of Post and Notification Services with registration to Eureka Server
- ★ Test User, Post and Notification Services communication

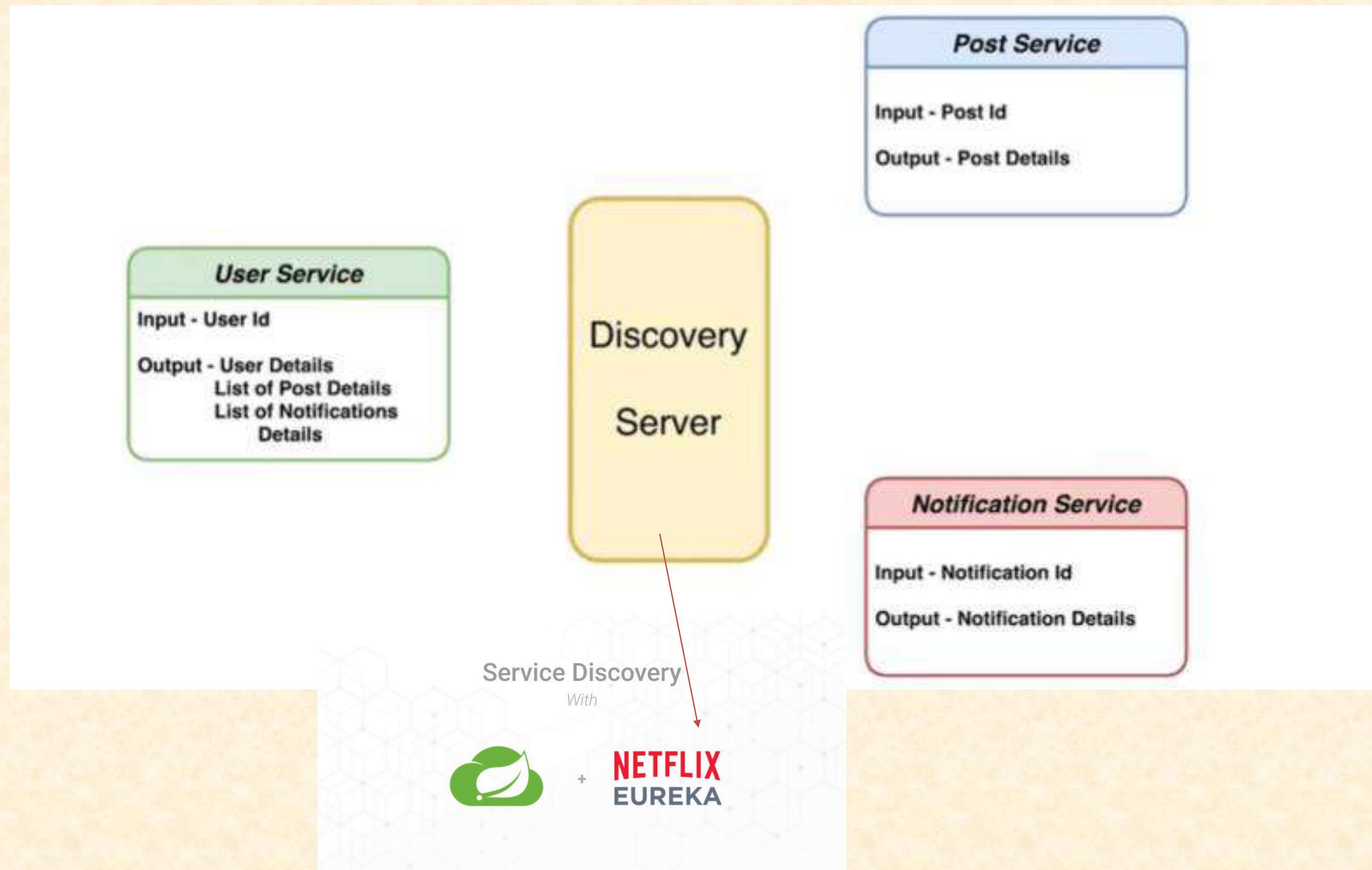
Microservices – Recall the Services



Microservices - Registration & Discovery



Microservices - Registration & Discovery



Registration & Discovery – Netflix Eureka

- Eureka Server is a service registry and discovery server provided by Netflix
- It is used for service registration, discovery, and load balancing in a microservices architecture
- Eureka Server maintains a registry of available services and their instances, allowing other services and clients to dynamically discover and communicate with them
- Eureka Server helps in managing service instances, monitoring health checks, and facilitating service-to-service communication within a distributed system.



Microservices - Registration & Discovery

Dependency – Spring Cloud Eureka Server:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

To suppress Warnings:

```
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Microservices - Registration & Discovery

```
@SpringBootApplication  
@EnableEurekaServer  
public class EurekaServerDemoApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(EurekaServerDemoApplication.class, args);  
    }  
  
}
```

Registration & Discovery – Netflix Eureka

The screenshot shows the Netflix Eureka dashboard at `localhost:9999`. The top navigation bar includes links for `HOME` and `LAST 1000 SINCE STARTUP`.

System Status

Environment	test	Current time	2024-03-31T22:13:10 +0530
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	188mb

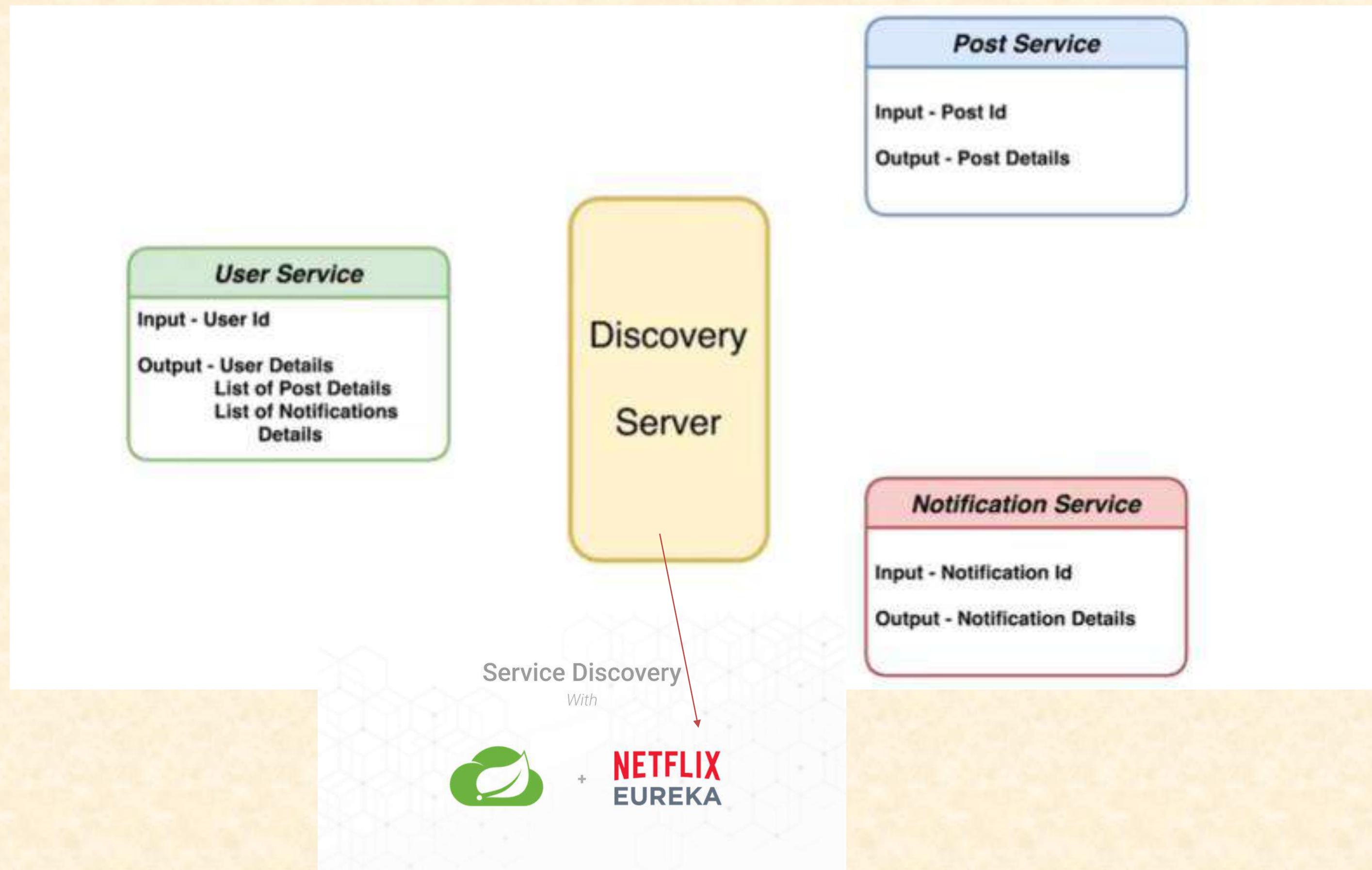
Spring Boot (REST API, MVC and Microservices)

Service Registration & Discovery – Netflix Eureka Server

Microservices - Registration & Discovery

- ★ What is Service Registration and Discovery
- ★ What is Eureka Server and Client
- ★ How to create Eureka Server
- ★ How to create Eureka Client
- ★ Enable Eureka at User, Post and Notification Services
- ★ Create and enable multiple instances of Post and Notification Services with registration to Eureka Server
- ★ Test User, Post and Notification Services communication

Microservices - Registration & Discovery



Microservices - Registration & Discovery

Dependency – Spring Cloud Eureka Client:

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

<spring-cloud.version>2023.0.1</spring-cloud.version>
```

```
<dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>${spring-cloud.version}</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
</dependencies>
</dependencyManagement>
```

Microservices - Registration & Discovery

Application.properties for each service:

```
spring.application.name=p-service  
server.port = 8081  
  
eureka.client.serviceUrl.defaultZone = http://localhost:8999/eureka/
```

Annotate each service:

```
@EnableDiscoveryClient or  
  
@EnableEurekaClient
```

Microservices - Registration & Discovery

User Controller :

```
Posts posts = restTemplate.getForObject("http://p-service/posts/1", Posts.class);
user1.setPosts(posts);
```

```
Notifications notifications = restTemplate.getForObject("http://n-service/notifications/2", Notifications.class);
user1.setNotifications(notifications);
```

Microservices - Registration & Discovery

DEMO

Thank You!

