# REST APIs - REST Web Services

© luv2code LLC

# You will learn how to …

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


© luv2code LLC

# Practical Results

- Introduction to Spring REST development

- Not an A to Z reference … for that you can see Spring Reference Manual

**https://projects.spring.io/spring-framework/**



© luv2code LLC

# Business Problem

- Build a client app that provides the weather report for a city

- Need to get weather data from an external service


© luv2code LLC

# Application Architecture

© luv2code LLC

# Questions

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

© luv2code LLC

# Answers

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

© luv2code LLC

# Answers

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

© luv2code LLC

# Answers

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

© luv2code LLC

# 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 …


© luv2code LLC

# 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}
```


© luv2code LLC

# Response - Weather Report

- The Weather Service responds with JSON

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

Condensed version


© luv2code LLC

# Multiple Client Apps

**Remember:**
**REST calls can be made over HTTP**
**REST is language independent**

© luv2code LLC

# Currency Converter App

© luv2code LLC

# Movie Tickets App

© luv2code LLC

# Customer Relationship Manager (CRM) App

We will create this code using Spring REST (SERVER)


© luv2code LLC

# 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**



© luv2code LLC

# 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**



© luv2code LLC

# JSON Basics - Syntax

© luv2code LLC

# What is JSON?

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

**JSON is just plain text data**


© luv2code LLC

# Simple JSON Example

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

Name

Value

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



© luv2code LLC

# JSON Values

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

Value

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



© luv2code LLC

# 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



© luv2code LLC

# JSON Arrays

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

Array



© luv2code LLC

# REST HTTP Basics

© luv2code LLC

# REST over HTTP

- Most common use of REST is over HTTP

- Leverage HTTP methods for CRUD operations

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


© luv2code LLC

# HTTP Messages

© luv2code LLC

# HTTP Request Message

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

© luv2code LLC

# HTTP Response Message

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

© luv2code LLC

# HTTP Response - Status Codes

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

401 Authentication Required
404 File Not Found

500 Internal Server Error


© luv2code LLC

# MIME Content Types

- The message format is described by MIME content type
  - **M**ultipurpose **I**nternet **M**ail-**E**xtension
- Basic Syntax: `type/sub-type`
- Examples
  - `text/html`, `text/plain`
  - `application/json`, `application/xml`, ...


© luv2code LLC

# Client Tool

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


© luv2code LLC

# Postman

Free developer plan

**www.getpostman.com**


© luv2code LLC

# Install Postman Now

**www.getpostman.com**

© luv2code LLC

# Postman Demo

© luv2code LLC

# Spring REST Controller

© luv2code LLC

# Spring REST Hello World

© luv2code LLC

# 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


© luv2code LLC

# Testing with REST Client - Postman

Access the REST endpoint at `/test/hello`

The response



© luv2code LLC

# Testing with REST Client - Web Browser

Access the REST endpoint at `/test/hello`

The response



© luv2code LLC

# 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 …


© luv2code LLC

# Spring REST Controller - Dev Process

© luv2code LLC

# Development Process

Step-By-Step

1. Add Maven dependency for Spring Boot Starter Web

2. Create Spring REST Service using @RestController



© luv2code LLC

# Step 1: Add Maven Dependency

File: pom.xml

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

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


© luv2code LLC

# Step 2: Create Spring REST Service

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

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

}
```

Handles HTTP GET requests


© luv2code LLC

# Java JSON Data Binding

© luv2code LLC

# Java JSON Data Binding

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

***Also known as***

**Mapping**

**Serialization / Deserialization**

**Marshalling / Unmarshalling**


© luv2code LLC

# 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**


© luv2code LLC

# Jackson Data Binding

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

```
{
  "id": 14,

  "firstName": "Mario",

  "lastName": "Rossi",

  "active": true
}
```

© luv2code LLC

# JSON to Java POJO

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

```
{
  "id": 14,

  "firstName": "Mario",

  "lastName": "Rossi",

  "active": true
}
```

© luv2code LLC

# 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

```
{
  "id": 14,

  "firstName": "Mario",

  "lastName": "Rossi",

  "active": true
}
```

*Call setId(…)*

*Call setFirstName(…)*

*Name(…)*

**Jackson will do this work**

```
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
}
```



© luv2code LLC

# Java POJO to JSON

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

```
{
  "id": 14,

  "firstName": "Mario",

  "lastName": "Rossi",

  "active": true
}
```

Java
POJO

Student

***Call getXXX methods***

**Jackson will do this work**



© luv2code LLC

# 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**



© luv2code LLC

# Spring REST Service - Students

© luv2code LLC

# Create a New Service

- Return a list of students

GET `/api/students` *Returns a list of students*


© luv2code LLC

# Spring REST Service

We will write this code

REST Client

/api/students

REST Service

```
[
    {
        "firstName": "Poornima",
        "lastName": "Patel"
    },
    {
        "firstName": "Mario",
        "lastName": "Rossi"
    },
    {
        "firstName": "Mary",
        "lastName": "Smith"
    }
]
```

Web Browser
Or
Postman



© luv2code LLC

# 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 …


© luv2code LLC

# 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


© luv2code LLC

# Student POJO (class)

Java
POJO

Student

© luv2code LLC

# Jackson Data Binding

Remember

- Jackson will call appropriate getter / setter method

```
{
  "id": 14,

  "firstName": "Mario",

  "lastName": "Rossi",

  "active": true
}
```

Jackson will do this work

Java POJO

Student

| Student |
| --- |
| firstName : String<br>lastName : String |
| getFirstName() : String<br>setFirstName(...) : void<br>getLastName() : String<br>setLastName(...) : void |



© luv2code LLC

# Spring REST Service

© luv2code LLC

# Behind the scenes

© luv2code LLC

# Development Process

Step-By-Step

1. Create Java POJO class for **Student**

2. Create Spring REST Service using **@RestController**



© luv2code LLC

# 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


© luv2code LLC

# 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


© luv2code LLC

# Spring REST - Path Variables

© luv2code LLC

# 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"


© luv2code LLC

# Spring REST Service

© luv2code LLC

# Behind the scenes

© luv2code LLC

# Development Process

Step-By-Step

1. Add request mapping to Spring REST Service

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



© luv2code LLC

# Step 1: Add Request Mapping

File: StudentRestController.java

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

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

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

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

    // populate theStudents
    ...


    return theStudents.get(studentId);
  }

}
```

Bind the path variable (by default, must match)

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

Jackson will convert Student to JSON


© luv2code LLC

# Spring REST - Exception Handling

© luv2code LLC

# Remember our problem?

- Bad student id of 9999 …

© luv2code LLC

# We really want this …

- Handle the exception and return error as JSON

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


© luv2code LLC

# Spring REST Exception Handling

© luv2code LLC

# Development Process

Step-By-Step

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`



© luv2code LLC

# Step 1: Create custom error response class

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

**You can define any custom fields that you want to track**

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


© luv2code LLC

# Step 1: Create custom error response class

**File: StudentErrorResponse.java**

```
public class StudentErrorResponse {

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

  // constructors

  // getters / setters

}
```

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


© luv2code LLC

# Step 2: Create custom student exception

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


© luv2code LLC

# Step 2: Create custom student exception

File: StudentNotFoundException.java

```
public class StudentNotFoundException extends RuntimeException {

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

}
```

Call super class constructor



© luv2code LLC

# Step 3: Update REST service to throw exception

**File: StudentRestController.java**

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

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

    // check the studentId against list size

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

    return theStudents.get(studentId);

  }
  …
}
```

Could also check results from DB

Throw exception

Happy path


© luv2code LLC

# Spring REST Exception Handling

© luv2code LLC

# 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


© luv2code LLC

# Step 4: Add exception handler method

Exception handler method

Type of the response body

Exception type to handle / catch

```
java

pi")
public class StudentRestController {
  …

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

    StudentErrorResponse error = new StudentErrorResponse();

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

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

Body

Status code

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

© luv2code LLC

# Spring REST Exception Handling

© luv2code LLC

# Spring REST - Global Exception Handling

© luv2code LLC

# Spring REST Exception Handling

© luv2code LLC

# It works, but …

- Exception handler code is only for the specific REST controller
- Can't be reused by other controllers :-(

Large projects
will have
multiple controllers

- We need **global** exception handlers
  - Promotes reuse
  - Centralizes exception handling


© luv2code LLC

# 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**


© luv2code LLC

# Spring REST Exception Handling

© luv2code LLC

# Development Process

Step-By-Step

1. Create new @ControllerAdvice

2. Refactor REST service … remove exception handling code

3. Add exception handling code to @ControllerAdvice



© luv2code LLC

# Step 1: Create new @ControllerAdvice

**File: StudentRestExceptionHandler.java**

```
@ControllerAdvice
public class StudentRestExceptionHandler {

  ...

}
```



© luv2code LLC

# Step 2: Refactor - remove exception handling

**File: 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



© luv2code LLC

# Step 3: Add exception handler to @ControllerAdvice

File: StudentRestExceptionHandler.java

Same code as before

```
@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);
  }

}
```


© luv2code LLC

# Spring REST Exception Handling

© luv2code LLC

# Spring REST API Design

© luv2code LLC

# REST API Design

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


© luv2code LLC

# API Design Process

Step-By-Step

1. Review API requirements

2. Identify main resource / entity

3. Use HTTP methods to assign action on resource



© luv2code LLC

# API Requirements

From the Boss

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



© luv2code LLC

# Step 2: Identify main resource / entity

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

```
/api/employees
```


© luv2code LLC

# Step 3: Use HTTP methods to assign action on resource

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

**Full CRUD**



© luv2code LLC

# Employee Real-Time Project

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

© luv2code LLC

# 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



© luv2code LLC

# Employee Real-Time Project

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

© luv2code LLC

# More API Examples

- On the following slides, we'll look at APIs from other real-time projects

- PayPal

- GitHub

- SalesForce


© luv2code LLC

# PayPal

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

PayPal Developer  Docs  APIs  Support

**Create draft invoice**

| | |
|---|---|
| POST | /v1/invoicing/invoices |

**List invoices**

| | |
|---|---|
| GET | /v1/invoicing/invoices |

**Show invoice details**

| | |
|---|---|
| GET | /v1/invoicing/invoices/{invoice_id} |

**Update invoice**

| | |
|---|---|
| PUT | /v1/invoicing/invoices/{invoice_id} |

**Delete draft invoice**

| | |
|---|---|
| DELETE | /v1/invoicing/invoices/{invoice_id} |


© luv2code LLC

# GitHub

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

GitHub Developer

Create a new repository

```
POST /user/repos
```

List your repositories

```
GET /user/repos
```

Delete a repository

```
DELETE /repos/:owner/:repo
```

Get a repository

```
GET /repos/:owner/:repo
```



© luv2code LLC

# SalesForce REST API

- Industries REST API
  - https://sforce.co/2J40ALH

Retrieve All Individuals

```
GET  /services/apexrest/v1/individual/
```

Retrieve One Individual

```
GET  /services/apexrest/v1/individual/{individual_id}
```

Create an individual

```
POST /services/apexrest/clinic01/v1/individual/
```

Update an individual

```
PUT  /services/apexrest/clinic01/v1/individual/
```


© luv2code LLC

# Spring Boot REST API - Real Time Project

© luv2code LLC

# Real-Time Project

## *REST API with Spring Boot* that connects to a database

© luv2code LLC

# API Requirements

From the Boss

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



© luv2code LLC

# Development Process

Step-By-Step

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


© luv2code LLC

# Application Architecture

© luv2code LLC

# Setup Database Table

© luv2code LLC

# employee.sql

1. File: employee.sql

2. Create a new database table: **employee**

3. Load table with sample data

Details on downloading the SQL file provided in next section


© luv2code LLC

# Creating Spring Boot Project

© luv2code LLC

# Development Process

Step-By-Step

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

Our Task



© luv2code LLC

# Create DAO in Spring Boot

© luv2code LLC

# Application Architecture

© luv2code LLC

# 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

Step-By-Step

Let's build a DAO layer for this



© luv2code LLC

# DAO Interface

```
public interface EmployeeDAO {

  List<Employee> findAll();

}
```



© luv2code LLC

# 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



© luv2code LLC

# 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;
}
```


© luv2code LLC

# Development Process

Step-By-Step

1. Update db configs in application.properties
2. Create Employee entity
3. Create DAO interface
4. Create DAO implementation
5. Create REST controller to use DAO


© luv2code LLC

# Define Services with @Service

© luv2code LLC

# Refactor: Add a Service Layer

© luv2code LLC

# Purpose of Service Layer

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

© luv2code LLC

# Integrate Multiple Data Sources

© luv2code LLC

# Most Times - Delegate Calls

© luv2code LLC

# Specialized Annotation for Services

- Spring provides the **@Service** annotation

© luv2code LLC

# Specialized Annotation for Services

- **@Service** applied to Service implementations

- Spring will automatically register the Service implementation
  - thanks to component-scanning


© luv2code LLC

# Employee Service

Step-By-Step

1. Define Service interface

2. Define Service implementation

   - Inject the EmployeeDAO

© luv2code LLC

# Step 1: Define Service interface

```
public interface EmployeeService {

   List<Employee> findAll();

}
```



© luv2code LLC

# 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();
  }
}
```


© luv2code LLC

# DAO: Find, Add, Update and Delete

© luv2code LLC

# 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

© luv2code LLC

# Development Process

Step-By-Step

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

**DAO methods**

© luv2code LLC

# DAO: Get a single employee

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

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

  // return employee
  return theEmployee;
}
```





© luv2code LLC

# DAO: Add or Update employee

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

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

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

  // return dbEmployee
  return dbEmployee;
}
```

if id == 0
then save/insert
else update

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


© luv2code LLC

# 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);
}
```


© luv2code LLC

# DAO: Find, Add, Update and Delete

© luv2code LLC

# Development Process

Step-By-Step

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

**DAO methods**

© luv2code LLC

# Service: Find, Add, Update and Delete

© luv2code LLC

# Development Process

Step-By-Step

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

**Service methods**

© luv2code LLC

# Rest Controller Methods - Find and Add

© luv2code LLC

# Development Process

Step-By-Step

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

**Rest Controller methods**

© luv2code LLC

# Read a Single Employee

REST Client

GET /api/employees/{employeeId}

```
{
    "id": 1,
    "firstName": "David",
    "lastName": "Adams",
    "email": "david@luv2code.com"
}
```

Employee REST Controller


© luv2code LLC

# Create a New Employee

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

REST Client

POST /api/employees

```
{
    "firstName": "Juan",
    "lastName": "Perez",
    "email": "juan.perez@luv2code.com"
}
```

Employee REST Controller

```
{
    "id": 7,
    "firstName": "Juan",
    "lastName": "Perez",
    "email": "juan.perez@luv2code.com"
}
```

Response contains new id / primary key


© luv2code LLC

# Sending JSON to Spring REST Controllers

- When sending JSON data to Spring REST Controllers

- For controller to process JSON data, need to set a HTTP request header
  - `Content-type: application/json`

- Need to configure REST client to send the correct HTTP request header


© luv2code LLC

# Postman - Sending JSON in Request Body

- Must set HTTP request  header in Postman

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


© luv2code LLC

# Rest Controller Methods - Update and Delete

© luv2code LLC

# Development Process

Step-By-Step

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

**Rest Controller methods**

© luv2code LLC

# Update Employee

ID of employee to update
With updated info

REST Client

PUT /api/employees

```
{
    "id" : 1,
    "firstName" : "Daniel",
    "lastName" : "Vega",
    "email" : "daniel.vega@luv2code.com"
}
```

```
{
    "id" : 1,
    "firstName" : "Daniel",
    "lastName" : "Vega",
    "email" : "daniel.vega@luv2code.com"
}
```

Employee REST Controller

Response contains updated info (echoed)



© luv2code LLC

# Delete Employee

© luv2code LLC

# Spring Data JPA in Spring Boot

© luv2code LLC

# Application Architecture

© luv2code LLC

# 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???

```java
public interface EmployeeDAO {

    public List<Employee> findAll();

    public Employee findById(int theId);

    public void save(Employee theEmployee);

    public void deleteById(int theId);

}
```

```java
@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;
    }
```



© luv2code LLC

# Creating DAO

- You may have noticed a pattern with creating DAOs

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

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

  // return data
  return theData;
}
```

Most of the code is the same

Only difference is the entity type and primary key

Entity type

Primary key


© luv2code LLC

# My Wish

- I 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***



© luv2code LLC

# My Wish Diagram

© luv2code LLC

# Spring Data JPA - Solution

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

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

- Create a DAO and just plug in your entity type and primary key

- Spring will give you a CRUD implementation for FREE …. like MAGIC!!
  - Helps to minimize boiler-plate DAO code … yaaay!!!

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

© luv2code LLC

# JpaRepository

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

© luv2code LLC

# Development Process

Step-By-Step

1. Extend **`JpaRepository`** interface

2. Use your Repository in your app

**No need for implementation class**



© luv2code LLC

# Step 1: Extend JpaRepository interface

Entity type

Primary key

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

  // that's it ... no need to write any code LOL!

}
```

No need for implementation class

Get these methods for free

Entity: Employee

Primary key: Integer

findAll()
findById(…)
save(…)
deleteById(…)
*… others …*



© luv2code LLC

# JpaRepository Docs

- Full list of methods available … see JavaDoc for **`JpaRepository`**

**www.luv2code.com/jpa-repository-javadoc**



© luv2code LLC

# 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


© luv2code LLC

# Minimized Boilerplate Code

## Before Spring Data JPA

```
public interface EmployeeDAO {

    public List<Employee> findAll();

    public Emplo          yId(int theId);

    public void           ee theEmployee

    public void d         theId)

}
```

```
@Repository
public class Employee

    private EntityManag

    @Autowired
    public EmployeeDAOJpa                    tyManager) {
        entityManager = th
    }

    @Override
    public List<Empl

        // create
        TypedQue                                    loyee.class);

        //
        L                    = theQ

    }

    @Ov
    publ                dById(int theId) {

        // get          ee
        Employe          eEmployee =
                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**



© luv2code LLC

# Advanced Features

- Advanced features available for
  - Extending and adding custom queries with JPQL
  - Query Domain Specific Language (Query DSL)
  - Defining custom methods (low-level coding)

**www.luv2code.com/spring-data-jpa-defining-custom-queries**


© luv2code LLC

# Spring Data REST in Spring Boot

© luv2code LLC

# Spring Data JPA

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

© luv2code LLC

Hmmm …

Can this apply to REST APIs?



© luv2code LLC

# The Problem

- We saw how to create a REST API for **`Employee`**

© luv2code LLC

# My 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***



© luv2code LLC

# Spring Data REST - Solution

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

**https://spring.io/projects/spring-data-rest**

- Leverages your existing **`JpaRepository`**

- Spring will give you a REST CRUD implementation for FREE …. like MAGIC!!
  - Helps to minimize boiler-plate REST code!!!
  - No new coding required!!!

© luv2code LLC

# 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> {

}
```


© luv2code LLC

# 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 later video*

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

}
```

`/employees`


© luv2code LLC

# Development Process

Step-By-Step

1. Add Spring Data REST to your Maven POM file

**That's it!!!**

**Absolutely NO CODING required**



© luv2code LLC

# 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



© luv2code LLC

# In A Nutshell

For Spring Data REST, you only need 3 items

We already have these two

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

Only item that is new


© luv2code LLC

# Application Architecture

© luv2code LLC

# Minimized Boilerplate Code

## Before Spring Data REST

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

    private EmployeeService employeeService;

    @Autowired
    public EmployeeRestController(EmployeeSer
        employeeService = theEmployeeService
    }

    // expose "/employees" and return list of
    @GetMapping("/employees")
    public List<Employee> findAll() {
        return employeeService.findAll();
    }

    // add mapping for GET /e

    @GetMapping("/employees/{
    public Employee getEmploy

        Employee theEmployee

        if (theEmployee == nu
            throw new Runtime
        }

        return theEmployee;
    }
```

```
public inter
    public Li
    public
    pu
}
```

```
mp
Employ
owired
blic EmployeeSer
    employeeRepository = theEmployeeRepository;
}

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

d(int theId) {
result = employeeRepository.findById(theId);
= null;
t()) {
esult.get();

nd the employee
meException("Did not find employee id - " + theId);
```

**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



© luv2code LLC

# 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**


© luv2code LLC

# 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@luv2code.com",
    "_links": {
        "self": {
            "href": "http://localhost:8080/employees/3"
        },
        "employee": {
            "href": "http://localhost:8080/employees/3"
        }
    }
}
```

Employee data

Response meta-data
Links to data


© luv2code LLC

# HATEOAS

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

Get list of employees

Response

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

JSON Array of employees

Response meta-data
Information about the page

*More on configuring page sizes later*


© luv2code LLC

# 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**


© luv2code LLC

# Advanced Features

- Spring Data REST advanced features
  - Pagination, sorting and searching
  - Extending and adding custom queries with JPQL
  - Query Domain Specific Language (Query DSL)

**https://spring.io/projects/spring-data-rest**



© luv2code LLC

# Spring Data REST
# Configuration, Pagination and Sorting

© luv2code LLC

# 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


© luv2code LLC

# 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 |
| … | … |


© luv2code LLC

# 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`**


© luv2code LLC

# Solution

- Specify plural name / path with an annotation

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

}
```

**http://localhost:8080/members**


© luv2code LLC

# 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


© luv2code LLC

# Spring Data REST Configuration

- Following properties available: application.properties

| Name | Description |
|---|---|
| `spring.data.rest.base-path` | Base path used to expose repository resources |
| `spring.data.rest.default-page-size` | Default size of pages |
| `spring.data.rest.max-page-size` | Maximum size of pages |
| … | … |

**More properties available**

**www.luv2code.com/spring-boot-props**

`spring.data.rest.*`



© luv2code LLC

# 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



© luv2code LLC

# 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
```


© luv2code LLC