Course Introduction

Course Development Team

Eric Roby

www.codingwithroby.com

Chád Darby

www.luv2code.com

© luv2code LLC

Course Objective

A lot to cover!

Learn how to build professional Spring Boot REST applications using authentication and databases.

© luv2code LLC

You will learn how to …

- Develop Spring Boot REST applications
- Leverage Hibernate/JPA for database access
- Develop REST API endpoints using Spring Boot
- Apply Spring Security to control application access.
- Implement and leverage your own JWTs you build from scratch.
- Add Roles and Admin status throughout your REST endpoints
- Leverage all Java configuration (no xml) and Maven

© luv2code LLC

Source Code and PDFs

All source code is available for download

All PDFs of slides are available for download

© luv2code LLC

Questions / Help

If you have questions or need help…

Post the question in the classroom discussion forum

© luv2code LLC

Java Development Environment

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: Swagger
- 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

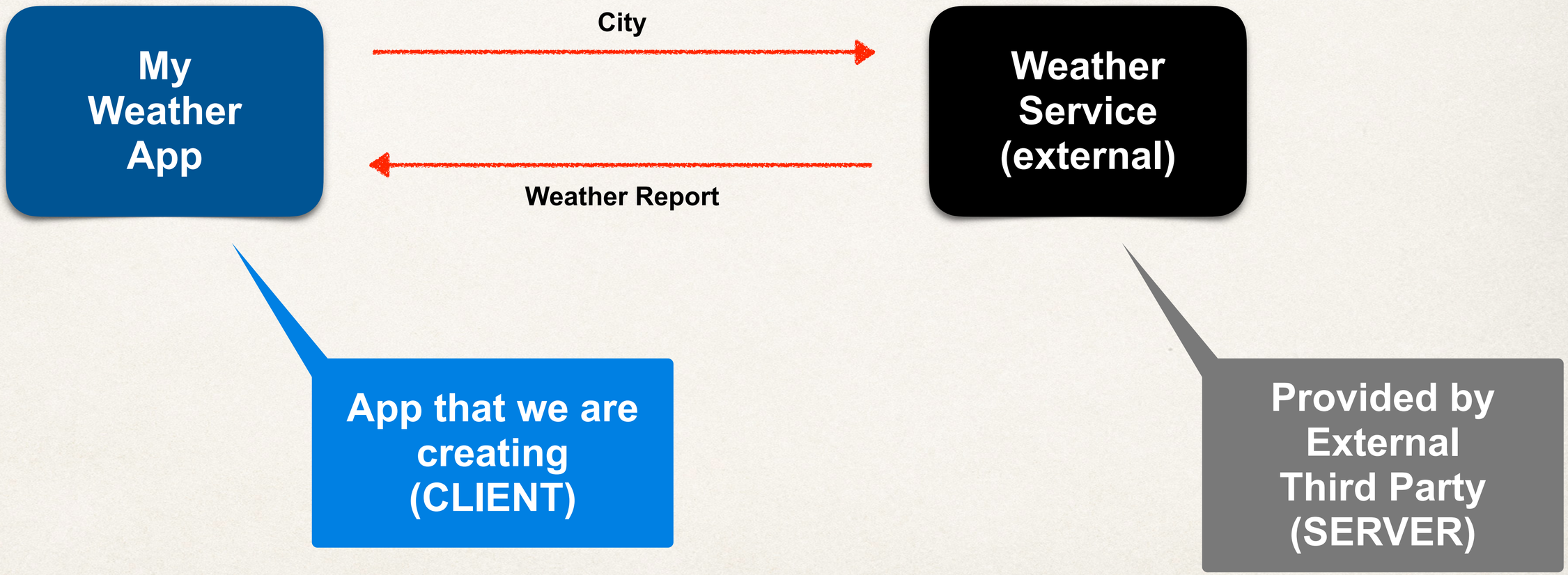
Business Problem

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

- Need to get weather data from an external service

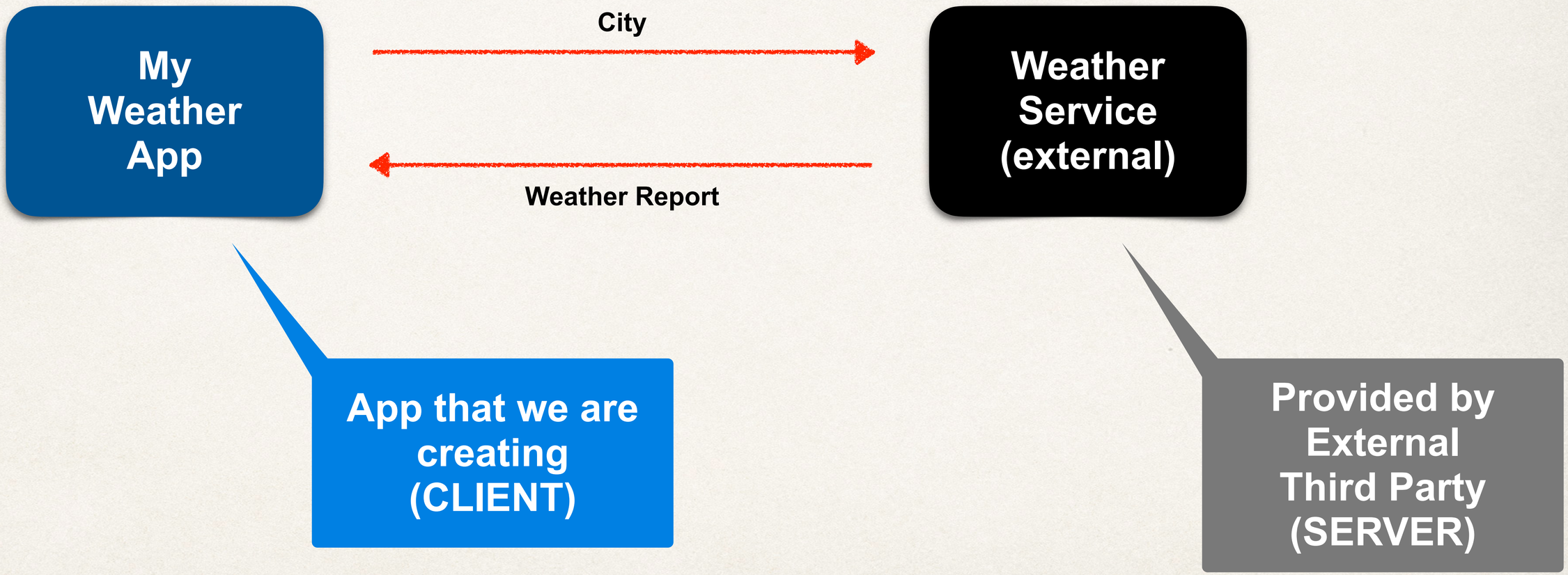
© luv2code LLC

Application Architecture



© luv2code LLC

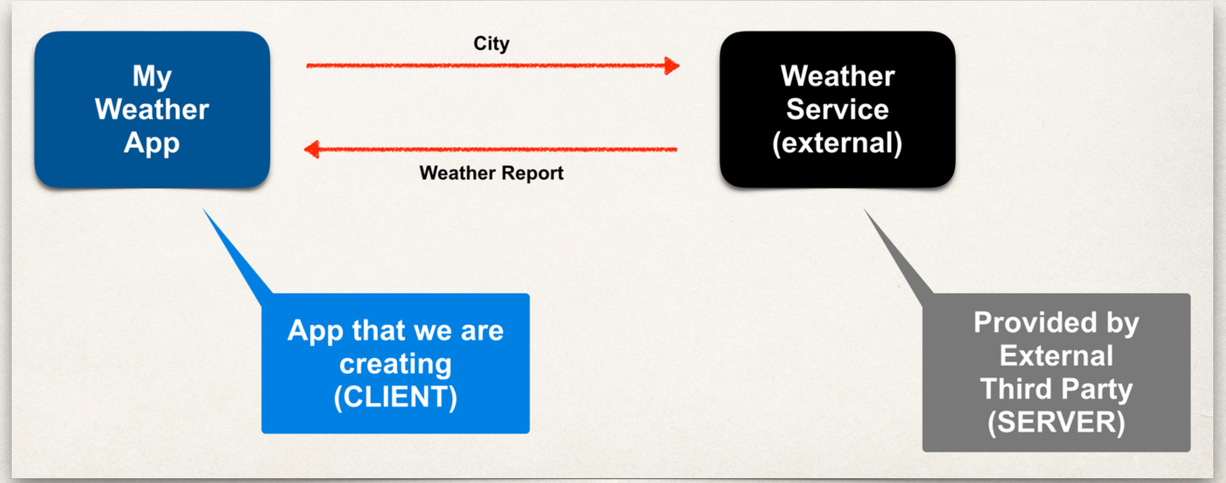
Application Architecture



© luv2code LLC

Questions

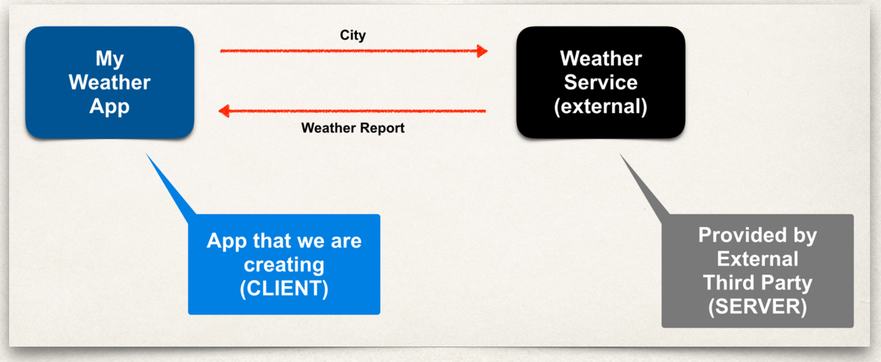
- How will we connect to the Weather Service?
- What programming language do we use?
- What is the data format?



© luv2code LLC

Answers

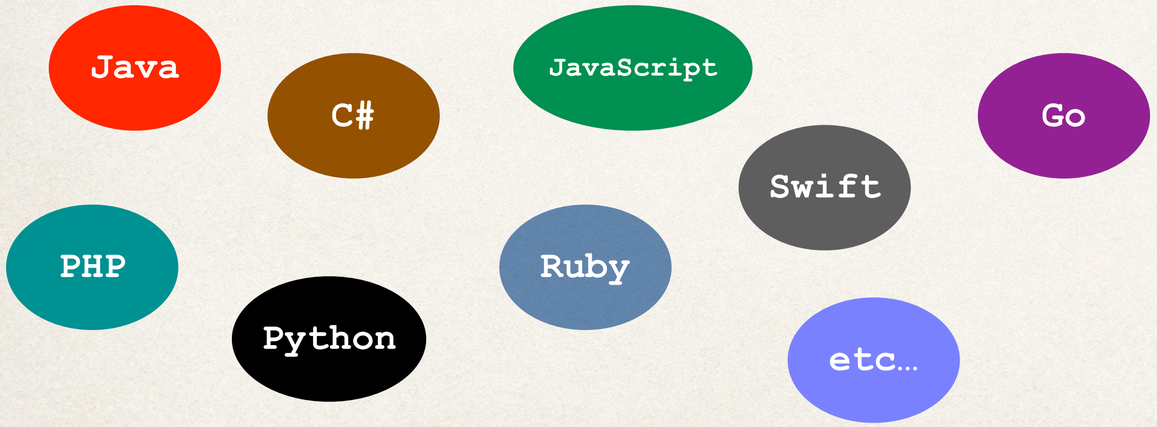
- **How will we connect to the Weather Service?**
  - We can make REST API calls over HTTP
  - REST: REpresentational State Transfer
  - Lightweight approach for communicating between applications

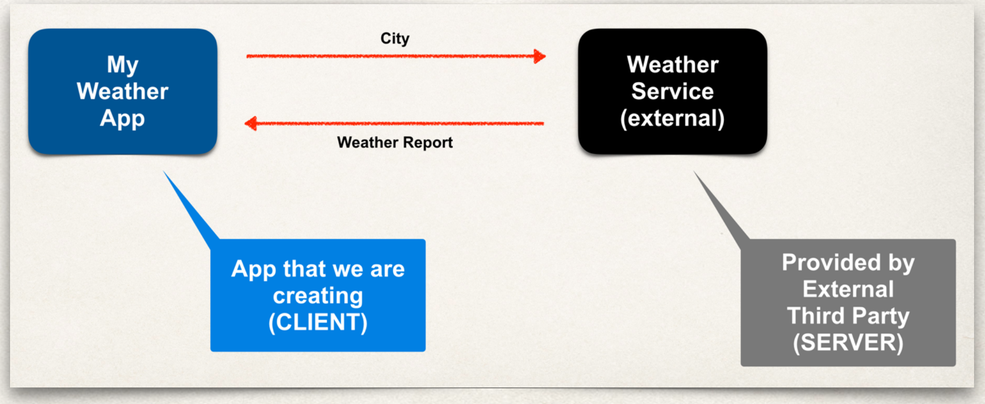


© luv2code LLC

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

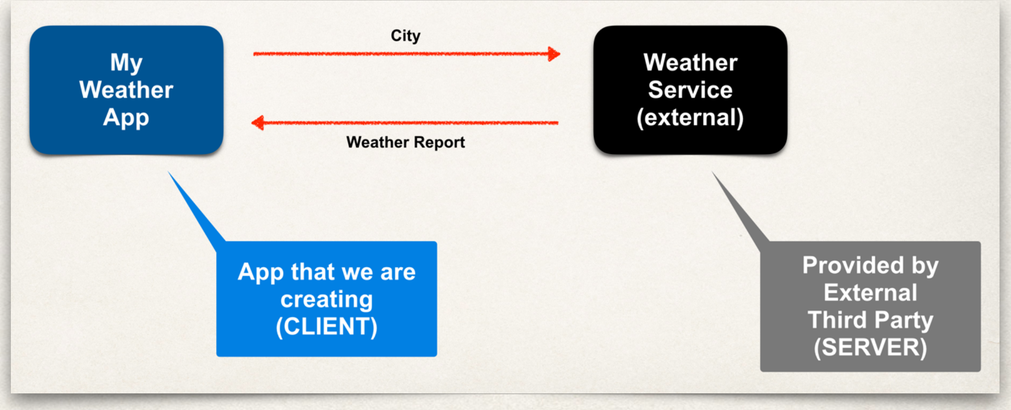




© luv2code LLC

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



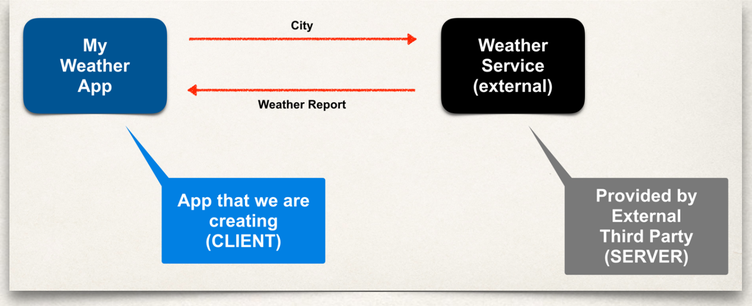
© luv2code LLC

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 latitude and longitude for your desired location

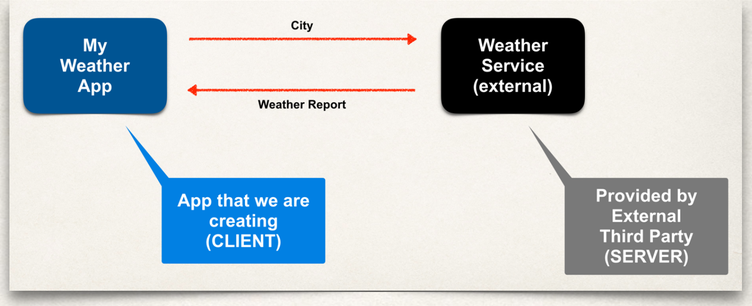
```
api.openweathermap.org/data/<apiVersion>/onecall?lat={theLatitude}&lon={theLongitude}
```

Pass the latitude and longitude of the city

© luv2code LLC

Response - Weather Report

- The Weather Service responds with JSON



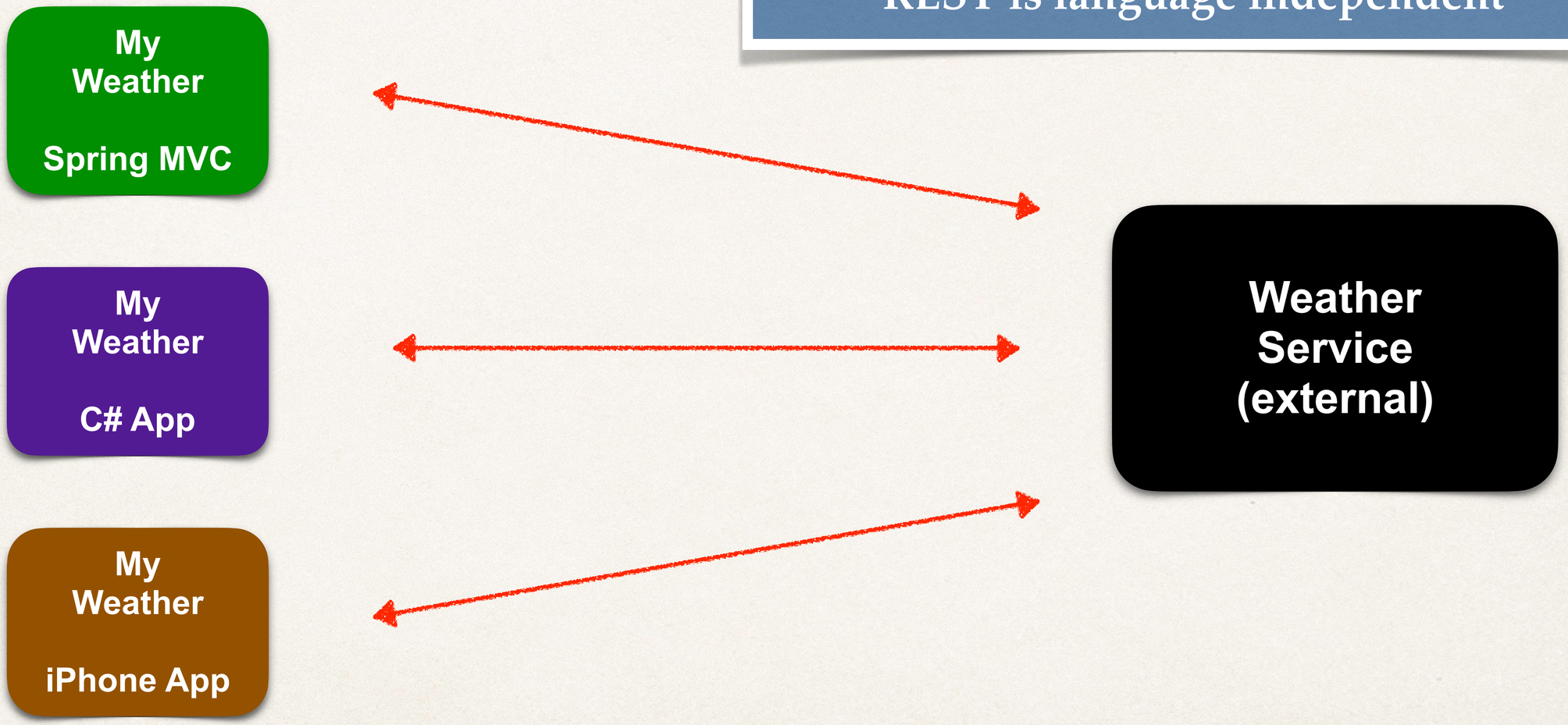

```
{
   …
   "temp": xxx,
   "feels_like": yyy,
   "humidity": zzz,
   …
}
```

Condensed version

© luv2code LLC

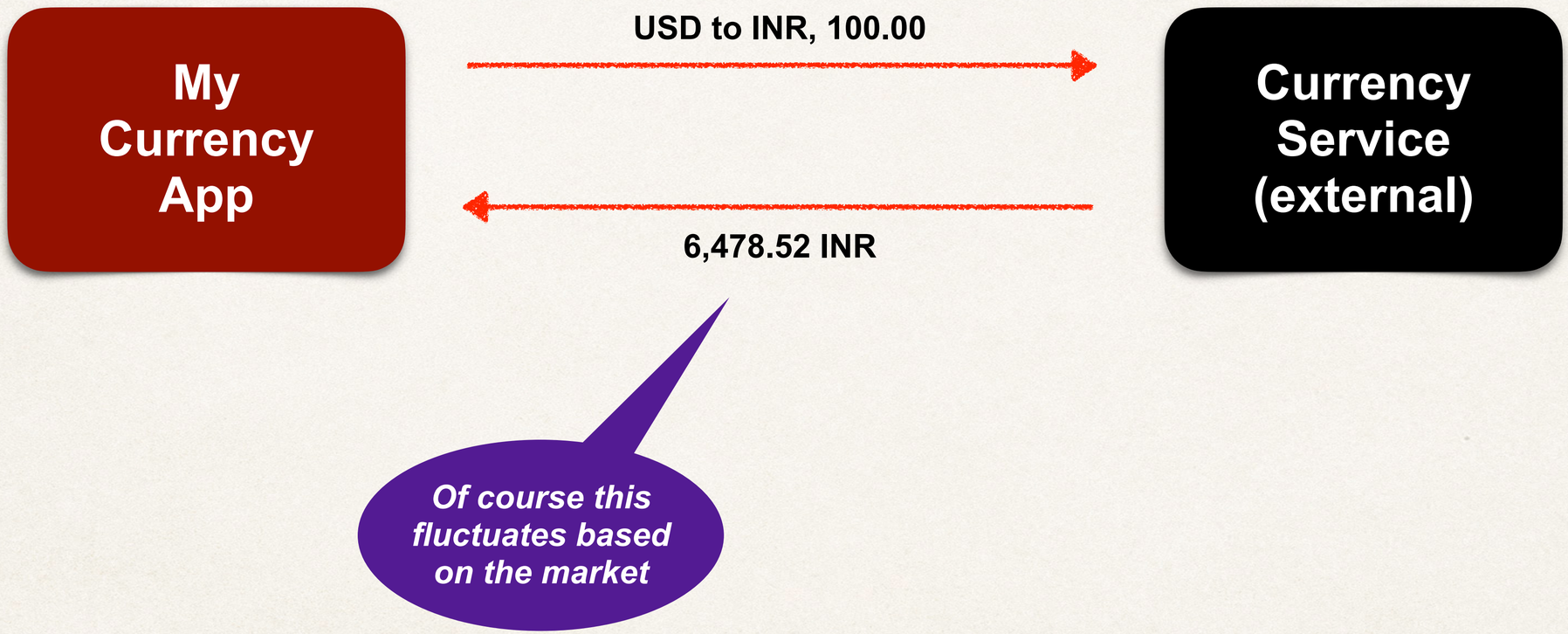
Multiple Client Apps

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



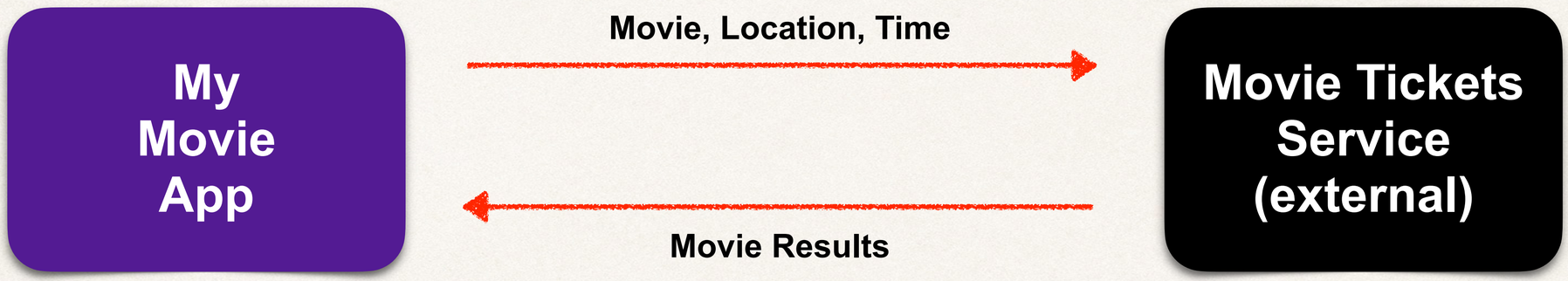
© luv2code LLC

Currency Converter App



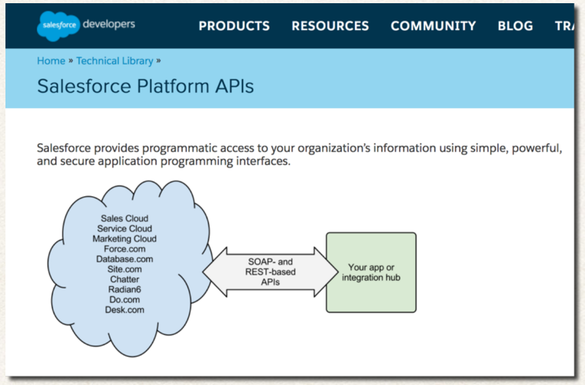
© luv2code LLC

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

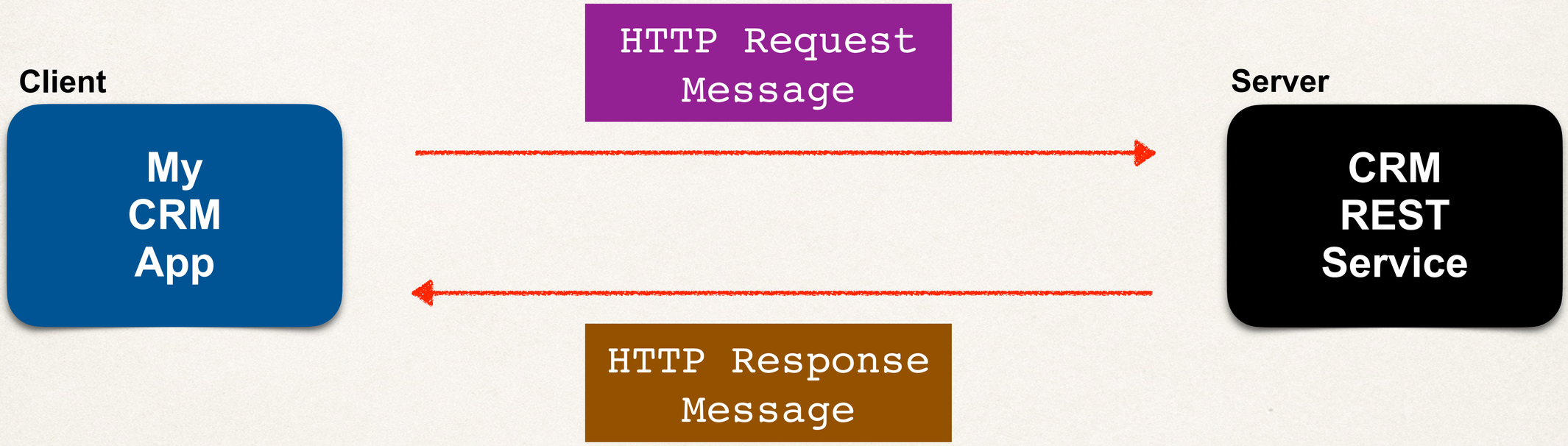
- Most common use of REST is over HTTP

- 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

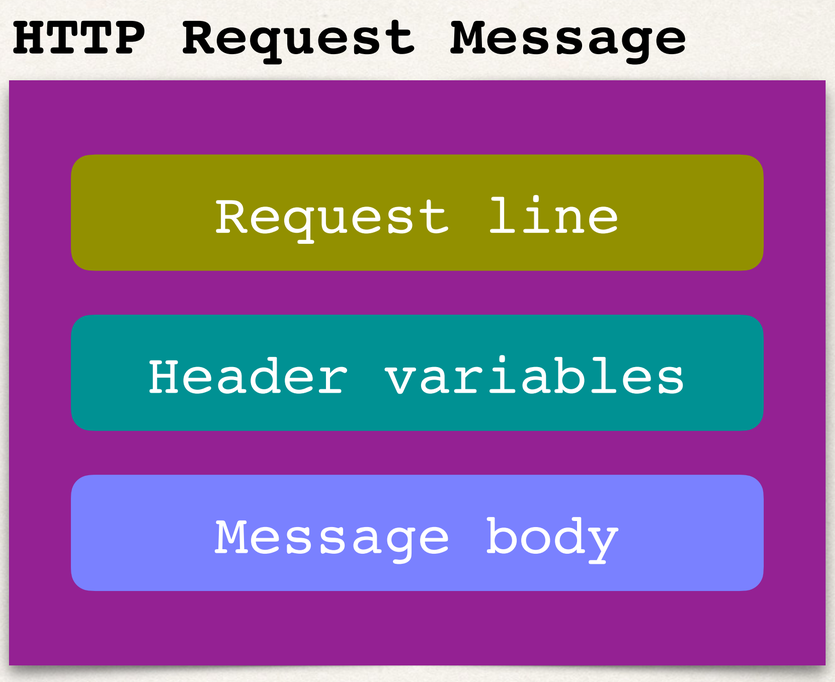
HTTP Messages



© luv2code LLC

HTTP Request Message

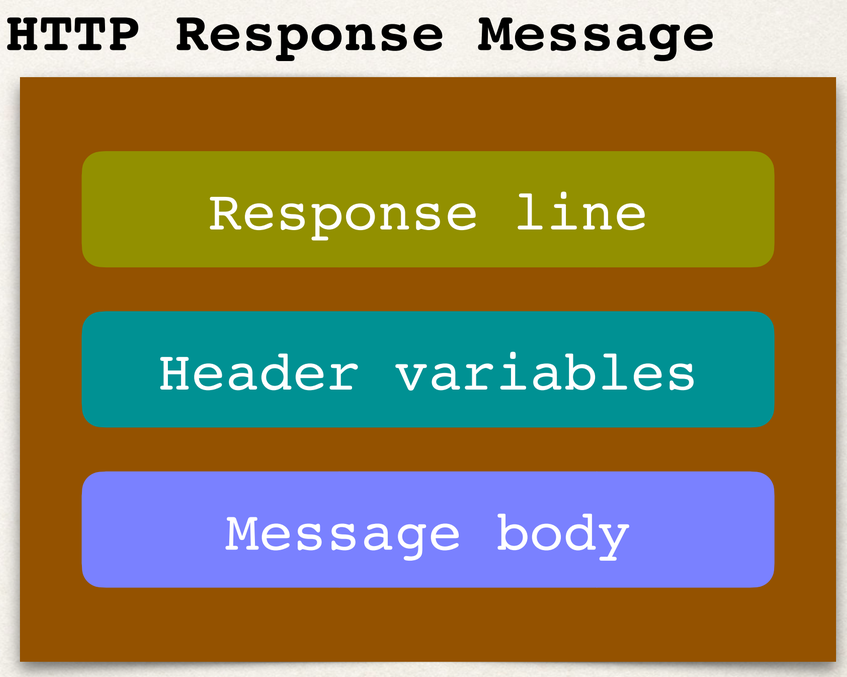
- Request line: the HTTP command
- Header variables: request metadata
- Message body: contents of message



© 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`, ...

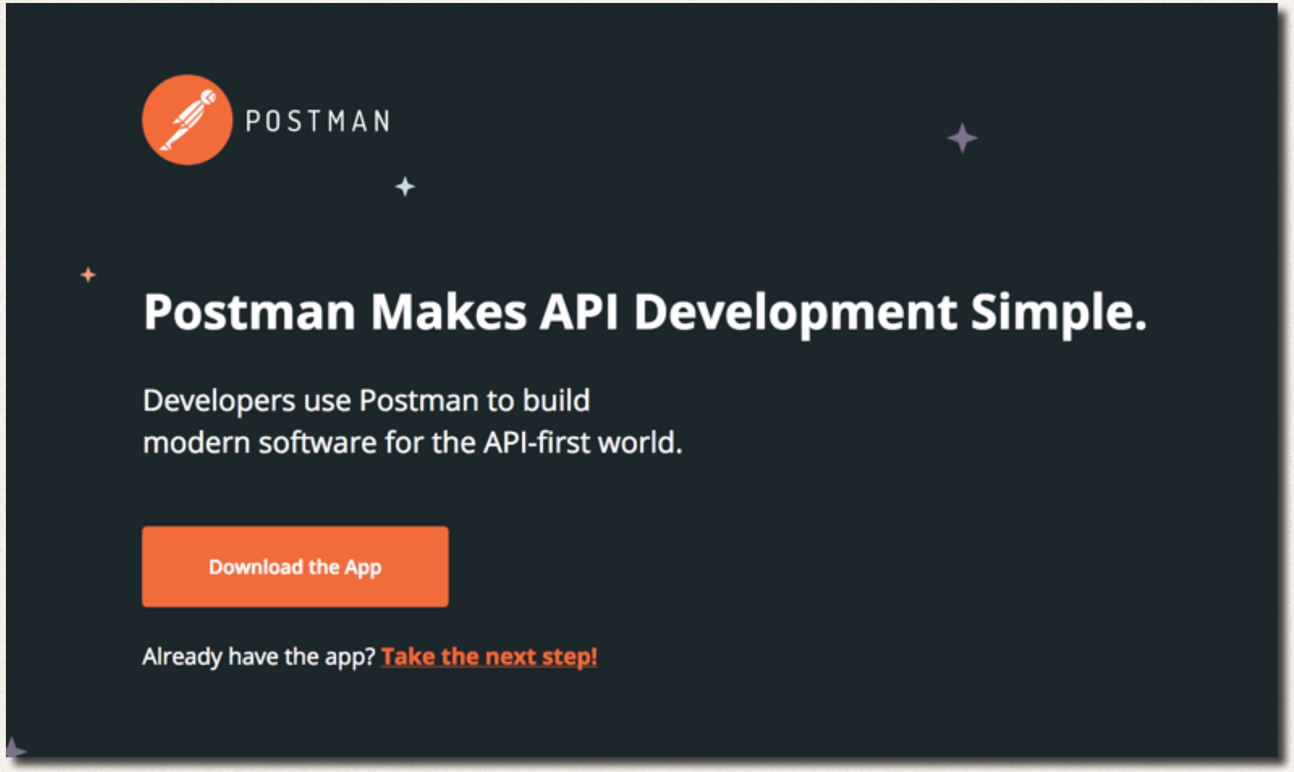
© luv2code LLC

Client Tool

- We need a **client tool**
- Send HTTP requests to the REST Web Service / API
- Plenty of tools available: curl, Postman, etc …
- Can also install our own within the application: Swagger

© luv2code LLC

Postman



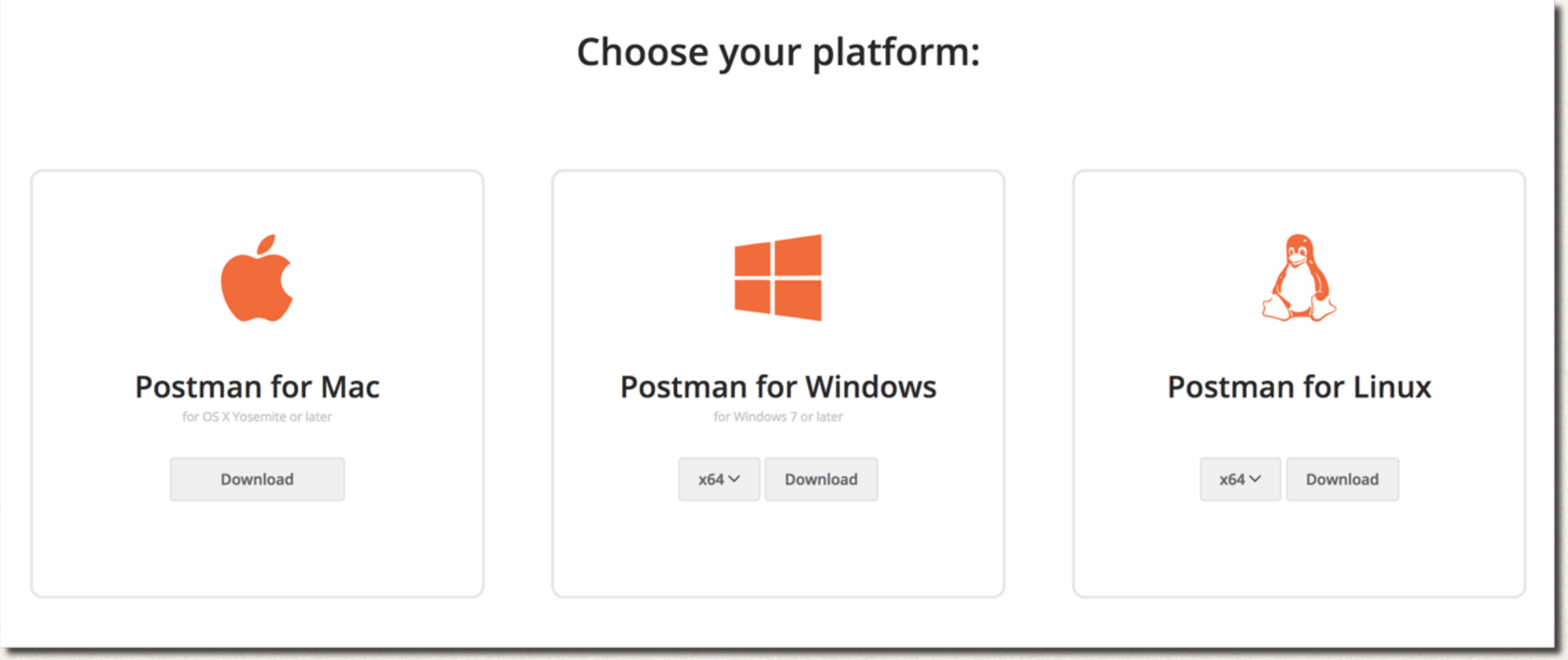
Free developer plan

www.getpostman.com

© luv2code LLC

Install Postman is One Option

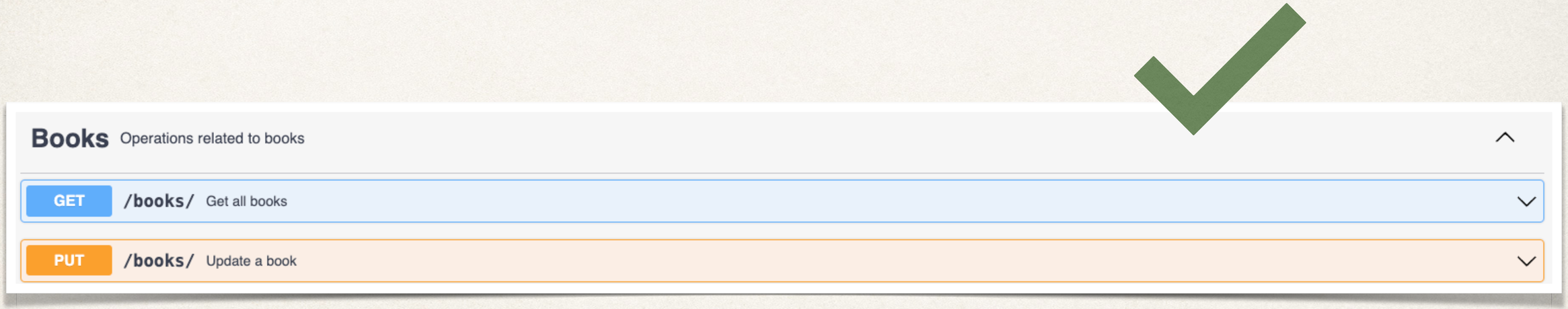
www.getpostman.com



© luv2code LLC

Setup Swagger Option 2

- We will setup Swagger in code
- Send HTTP requests to the REST Web Service / API
- Allows us to be able to run API endpoints from our application



© luv2code LLC

Books Project

What will we be creating?

- Creating and enhancing books to learn the basics of Spring Boot Rest

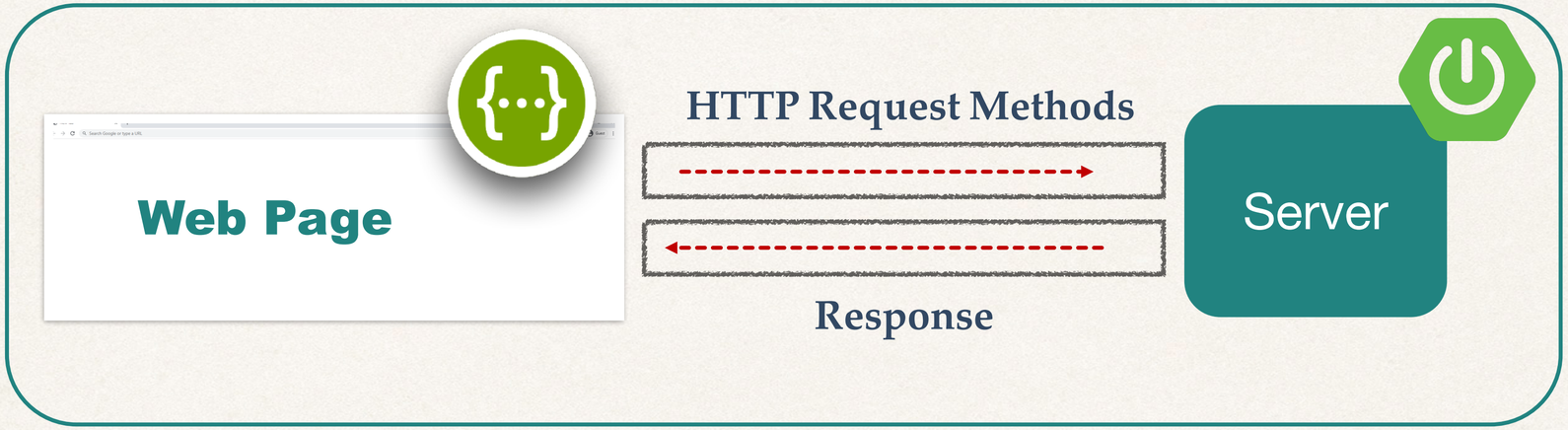
```
public void initializeBooks() {
    books.addAll(List.of(
      new Book("Title One", "Author One", "science"),
      new Book("Title Two", "Author Two", "science"),
      new Book("Title Three", "Author Three", "history"),
      new Book("Title Four", "Author Four", "math"),
      new Book("Title Five", "Author Five", "math"),
      new Book("Title Six", "Author Two", "math")
    ));
}
```

CRUD Operations

Create | Read | Update | Delete

© luv2code LLC

Request and Response



CRUD Operations

- Create
- Read
- Update
- Delete

© luv2code LLC

HTTP Request Methods

CRUD		HTTP Request Methods
Create	→	POST
Read	→	GET
Update	→	PUT
Delete	→	DELETE

© luv2code LLC

Setup our Spring Boot Project

Spring Initializr

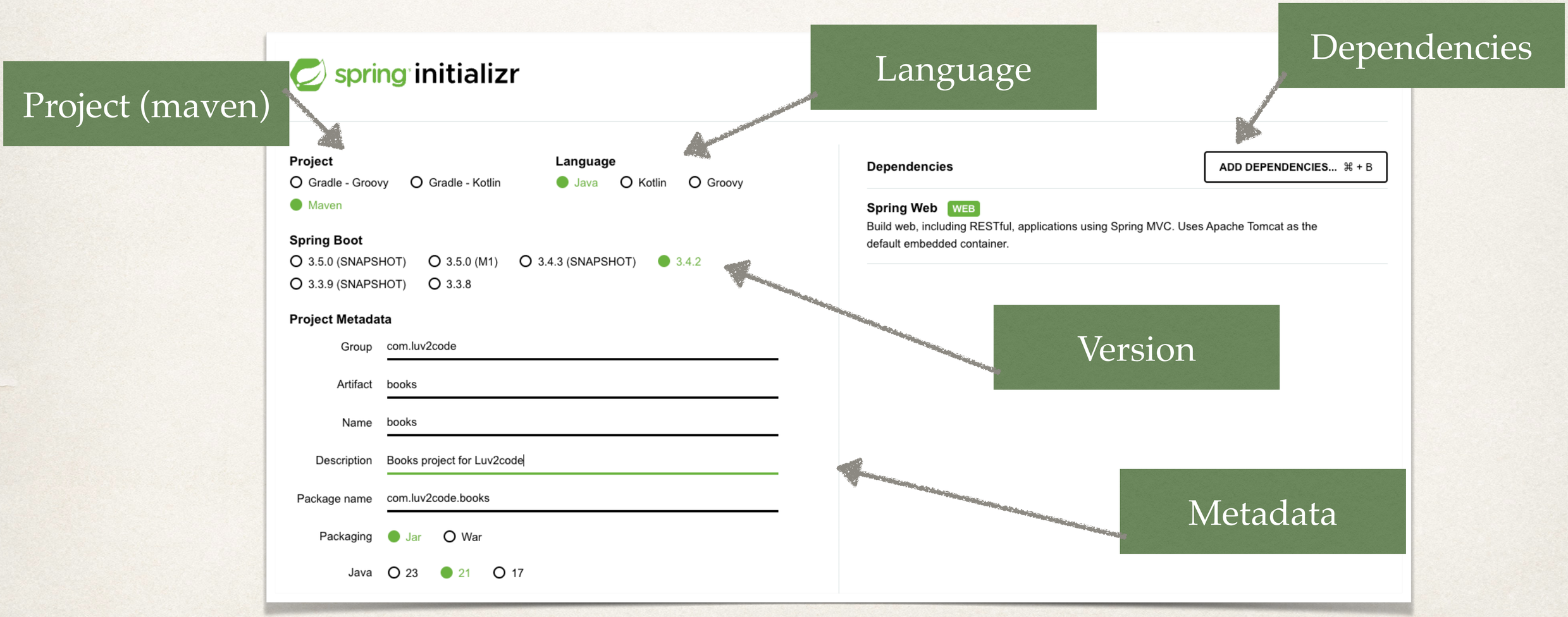
- Here we can set the project configurations for our upcoming Spring project.
- Allows us to choose the project, programming language, Spring Boot version, dependencies and metadata for our application.

start.spring.io

© luv2code LLC

Spring Initializr

- Setup our project using Spring Initializr



© luv2code LLC

Spring Initializr

- Spring Initializr will download a .zip file onto our machine.
- We will need to unzip the .zip file.
- Open the project on IntelliJ.
- Download all dependencies using Maven.
- Sounds fun - Let's go ahead and dive in!

© luv2code LLC

GET HTTP Request Method

Creating a Spring Boot Rest endpoint (updated)

File: BookController.java

```
import org.springframework.web.bind.annotation.*;


@RestController
public class Controller {

@GetMapping("/api-endpoint")
public String firstApi() {
    return "Hello Eric!"
   }
}
```

Lets dive into the first function

© luv2code LLC

Dive in

Read

Get

API Endpoint:

```
@GetMapping("/api-endpoint")
public String firstApi() {
    return "Hello Eric!"
}
```

URL : localhost:8080/api-endpoint

Response:

```
{
    "Hello Eric!"
}
```


© luv2code LLC

Start Spring Boot Rest Application

Read

Get

API Endpoint:

```
@GetMapping("/api-endpoint")
public String firstApi() {
    return "Hello Eric!"
}
```

Run Spring Boot application:

```
ericroby@Erics-MacBook-Pro ~% ./mvnw spring-boot:run
```

URL : localhost:8080

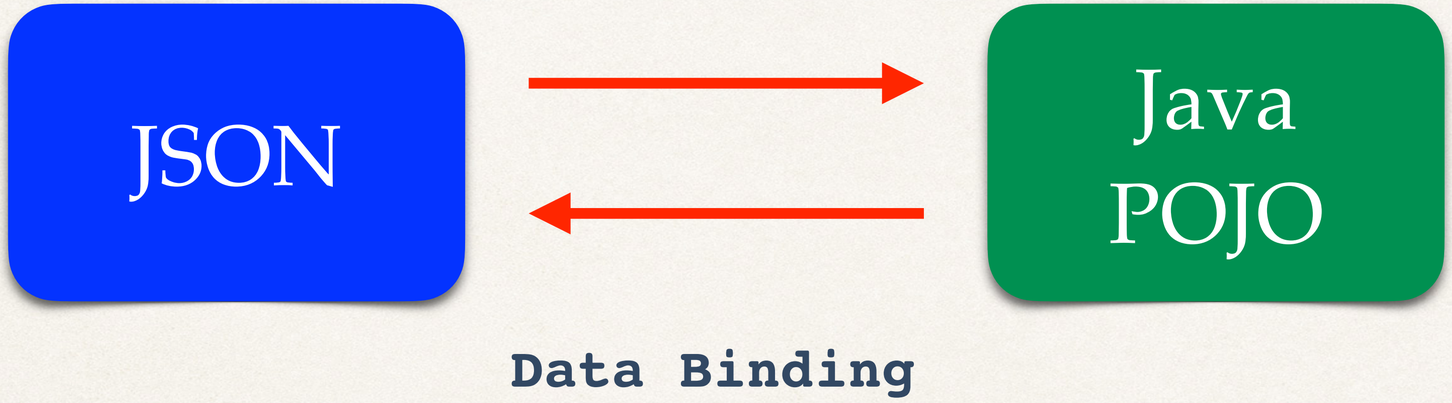
© luv2code LLC

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:

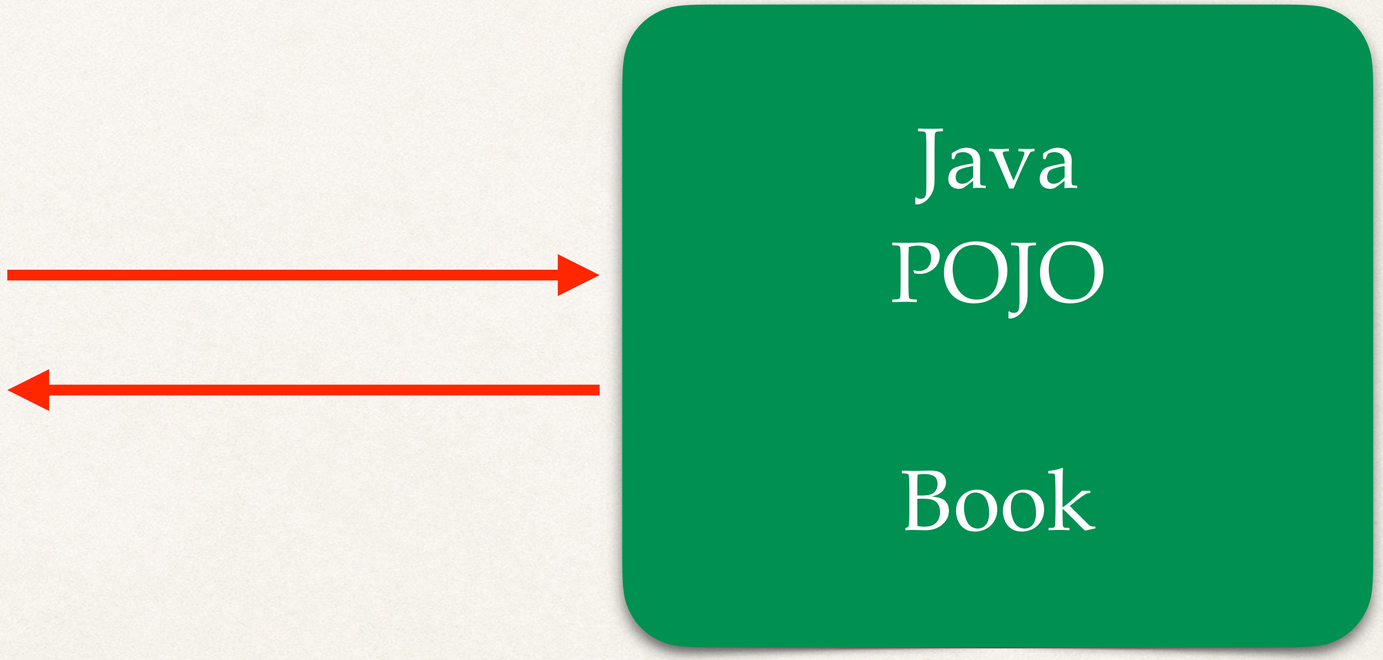
https://github.com/FasterXML/jackson-databind

© luv2code LLC

Jackson Data Binding

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

```
{
  "title": “Title Seven",

  "author": “Author One",

  “category”: ”science"
}
```

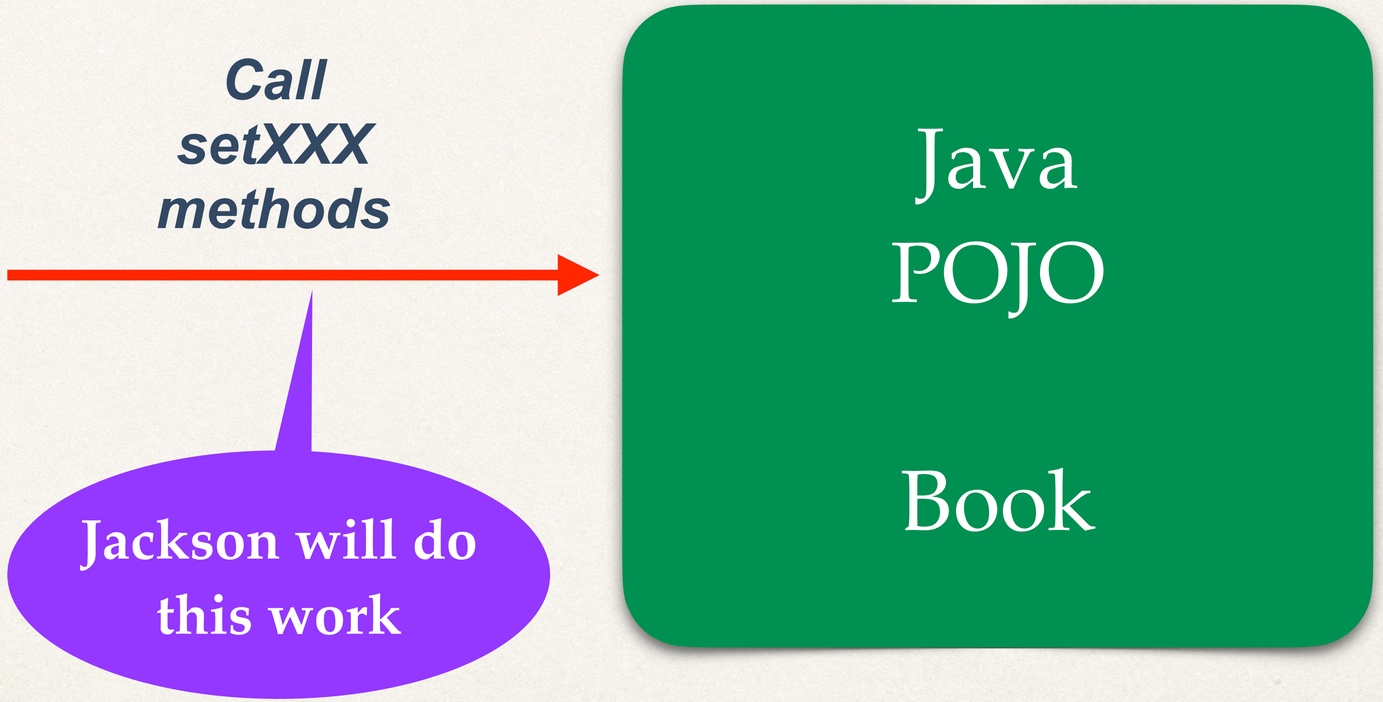


© luv2code LLC

JSON to Java POJO

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

```
{
  "title": “Title Seven",

  "author": “Author One",

  “category”: ”science"
}
```



© 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

```
{
  "title": “Title Seven",

  "author": “Author One",

  “category”: ”science"
}
```

Call setTitle(…)

Call setAuthor(…)

Call setCategory(…)

Jackson will do this work

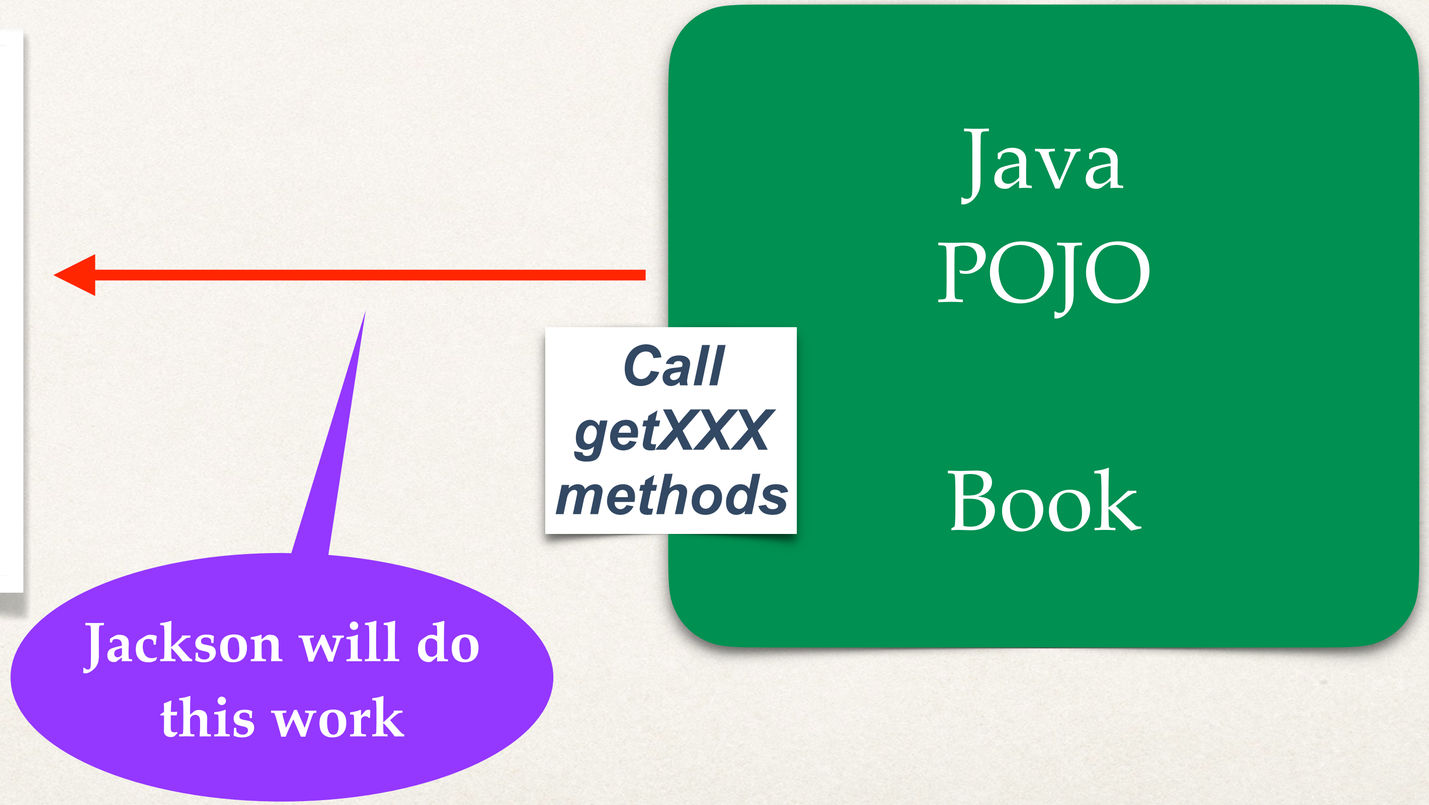
```
public class Book {

    private String title;
    private String author;
    private String category;

    public void setTitle(String title) {
        this.title = title;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public void setCategory(string category) {
        this.category = category;
    }

    // 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

```
{
  "title": "Title Seven",

  "author": "Author One",

  "category": "science"
}
```



© 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 - Books

© luv2code LLC

Create a New Service

- Return a list of books

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

© luv2code LLC

Spring REST Service

We will write this code

REST Client

/api/books

REST Service


```
[
  {
    "title": "Title One",
    "author": "Author One",
    "category": "science"
  },
  {
    "title": "Title Two",
    "author": "Author Two",
    "category": "science"
  },
  {
    "title": "Title Three",
    "author": "Author Three",
    "category": "history"
  },
  {
    "title": "Title Four",
    "author": "Author Four",
    "category": "math"
  },
  {
    "title": "Title Five",
    "author": "Author Five",
    "category": "math"
  },
  {
    "title": "Title Six",
    "author": "Author Two",
    "category": "math"
  }
]
```


Web Browser, Swagger Or Postman

© luv2code LLC

Convert Java POJO to JSON

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

© luv2code LLC

Spring Boot and Jackson Support

Happens automatically behind the scenes

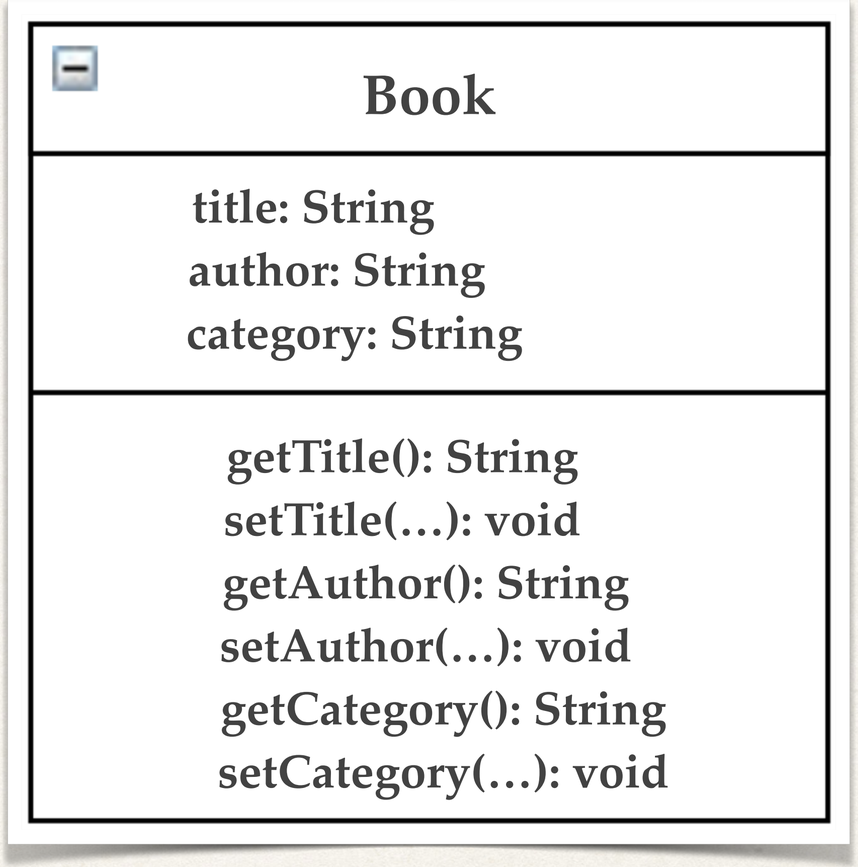
- Spring Boot will automatically handle **Jackson** integration
- JSON data being passed to REST controller is converted to Java POJO
- Java POJO being returned from REST controller is converted to JSON

© luv2code LLC

Book POJO (class)

Java
POJO

Book



© luv2code LLC

Jackson Data Binding

Remember

- Jackson will call appropriate getter / setter method

```
{
  "title": “Title Seven",

  "author": “Author One",

  “category”: ”science"
}
```

Jackson will do this work

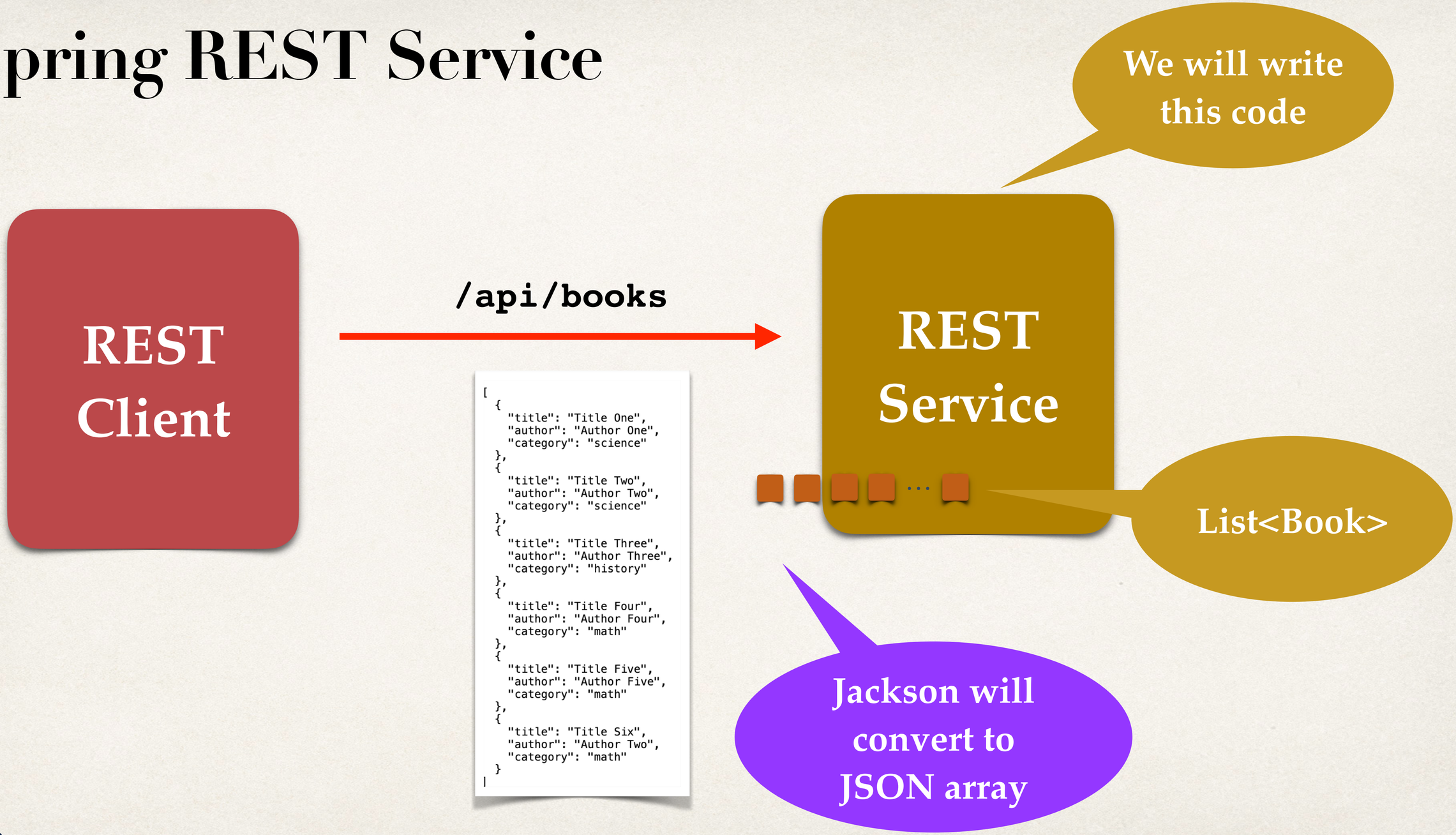
Java POJO

Book

Book
title: String author: String category: String
getTitle(): String setTitle(...): void getAuthor(): String setAuthor(...): void getCategory(): String setCategory(...): void

© luv2code LLC

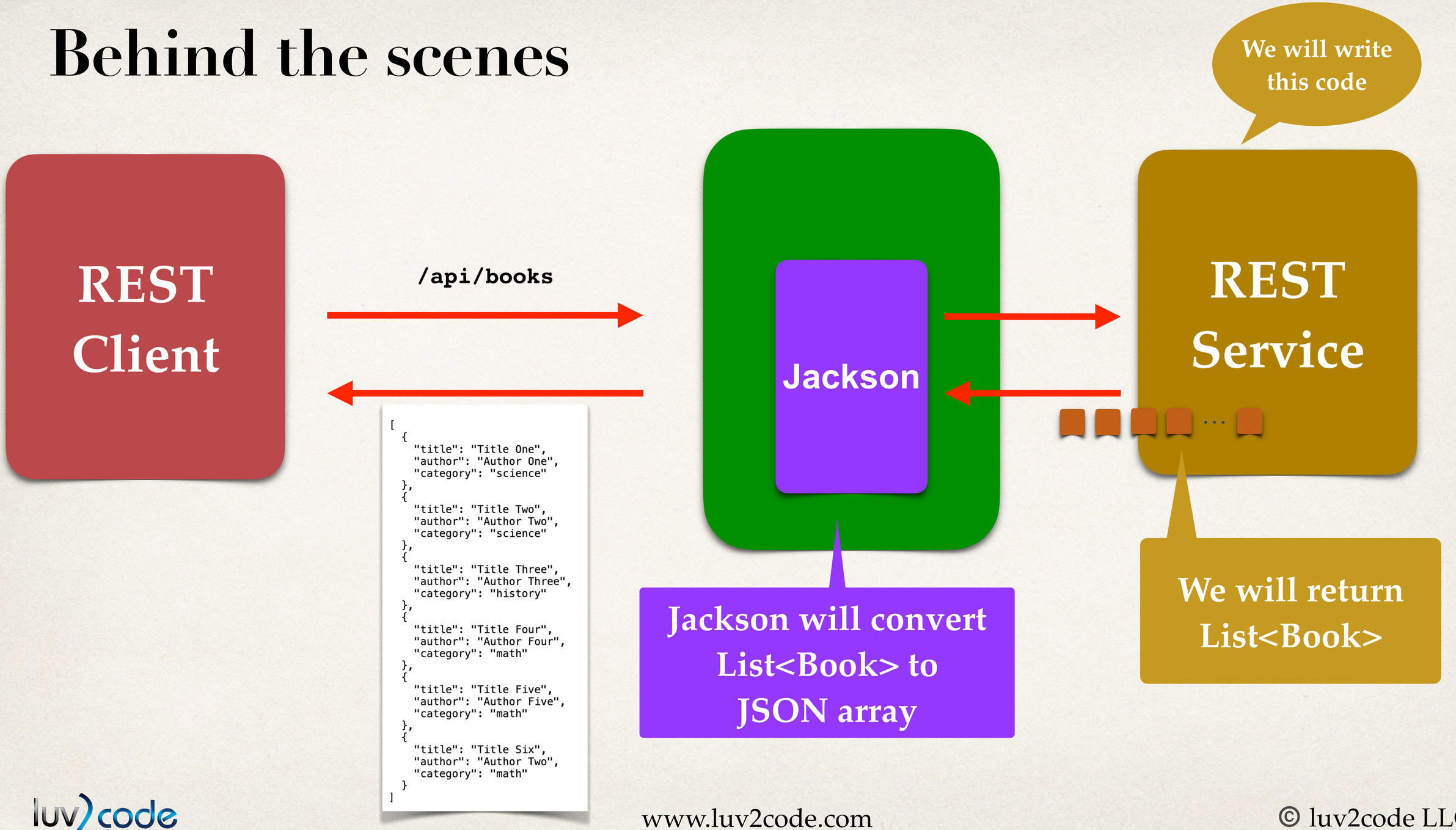
Spring REST Service




```
[
  {
    "title": "Title One",
    "author": "Author One",
    "category": "science"
  },
  {
    "title": "Title Two",
    "author": "Author Two",
    "category": "science"
  },
  {
    "title": "Title Three",
    "author": "Author Three",
    "category": "history"
  },
  {
    "title": "Title Four",
    "author": "Author Four",
    "category": "math"
  },
  {
    "title": "Title Five",
    "author": "Author Five",
    "category": "math"
  },
  {
    "title": "Title Six",
    "author": "Author Two",
    "category": "math"
  }
]
```


© luv2code LLC

Behind the scenes



© luv2code LLC

Development Process

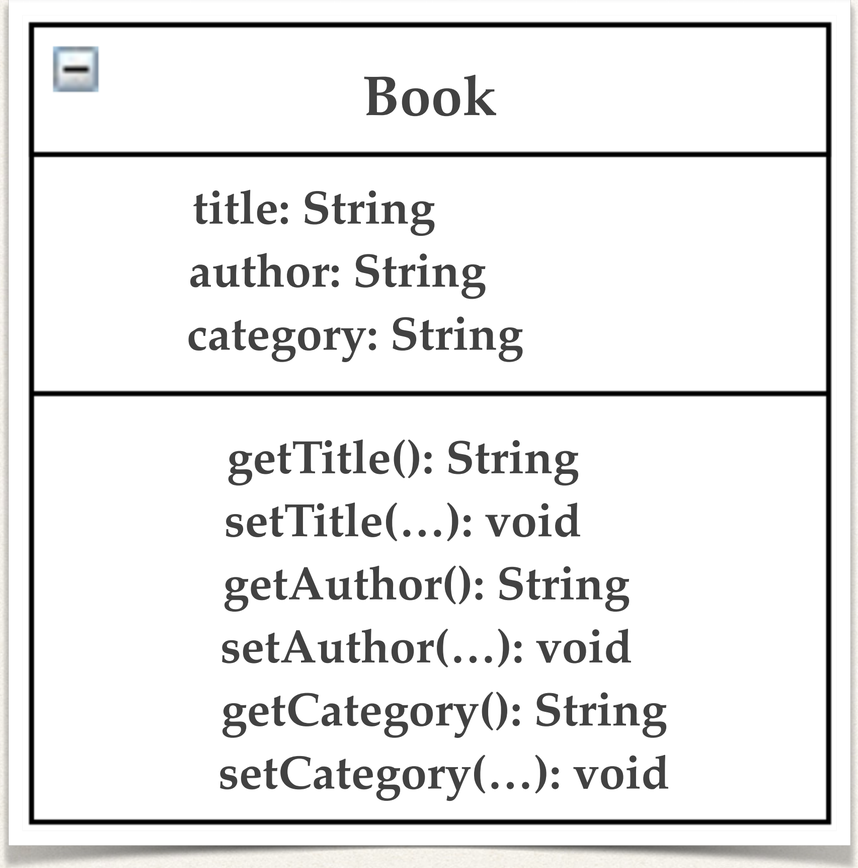
Step-By-Step

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

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

© luv2code LLC

Step 1: Create Java POJO class for Book



File: Book.java

```
public class Book {

    private String title;
    private String author;
    private String category;

    public Book (String title, String author, string category) {
        this.title = title;
        this.author = author;
        this.category = category;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public void setCategory(string category) {
        this.category = category;
    }

    public String getTitle() {
        return title;
    }

    < Other Getters >
}
```

Fields
Constructors
Getter/Setters

© luv2code LLC

Step 2: Create @RestController

File: BookController.java

```
@RestController
public class BookController {

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

  @GetMapping("/api/books")
  public List<Book> getBooks() {
    return books;
  }

}
```

```
public void initializeBooks() {
    books.addAll(List.of(
      new Book("Title One", "Author One", "science"),
      new Book("Title Two", "Author Two", "science"),
      new Book("Title Three", "Author Three", "history"),
      new Book("Title Four", "Author Four", "math"),
      new Book("Title Five", "Author Five", "math"),
      new Book("Title Six", "Author Two", "math")
    ));
}
```

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

Jackson will convert
List<Book> to
JSON array

© luv2code LLC

Swagger Setup

© luv2code LLC

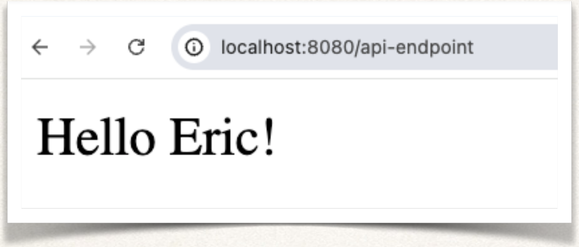
What is Swagger?

- Swagger is a set of tools and specifications for designing, building, documenting and consuming RESTful web services.
- Open-source project for streamlining API development by providing a common framework and API descriptions.
- Swagger UI implementation will allow us to be able to document and call our API endpoints right from our application.
- No third party tools will be needed!
- Win-win solution for implement Swagger within our application.

© luv2code LLC

What is Swagger?

- Right now we go straight to the endpoint:

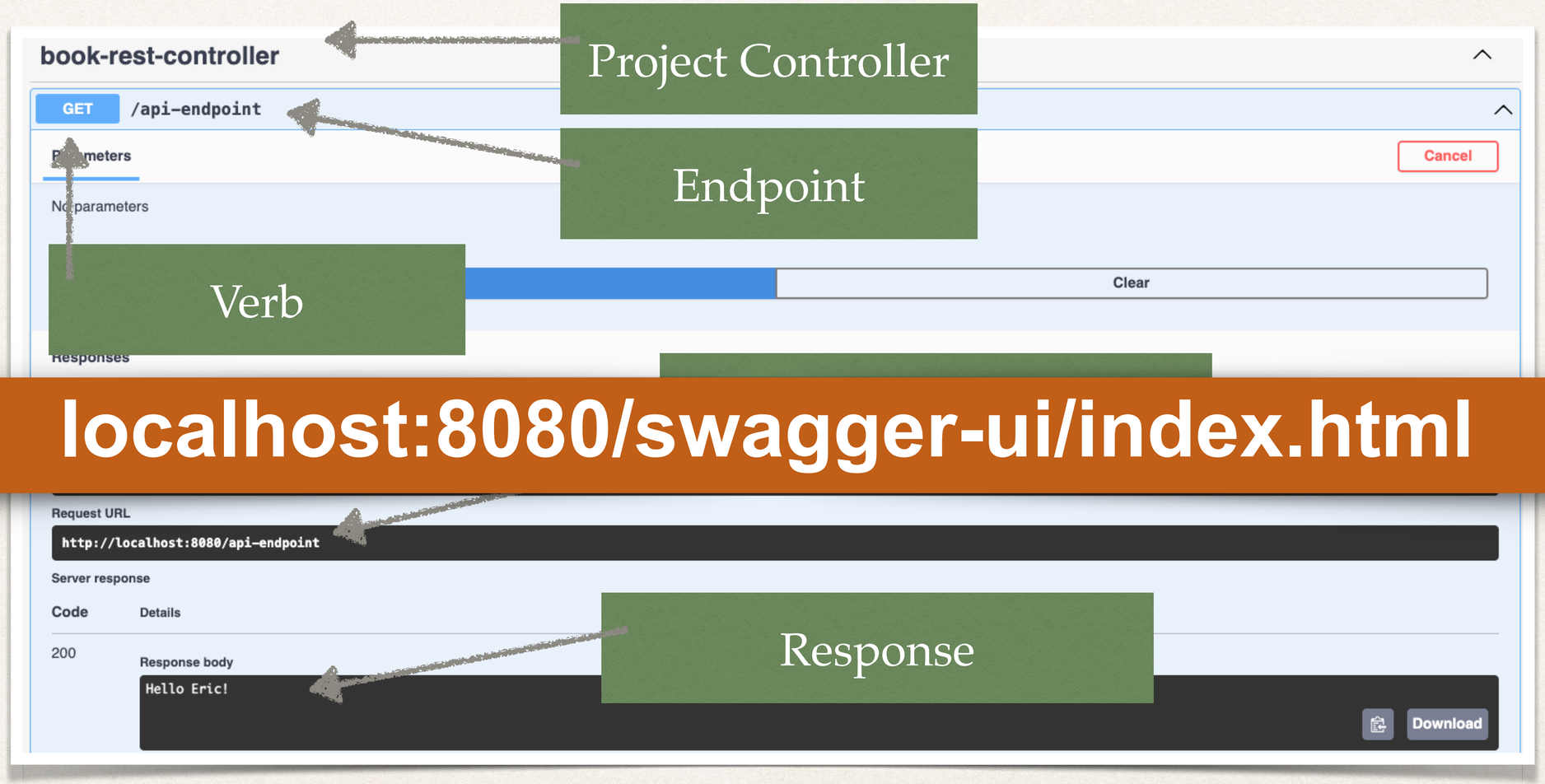


- However, once our application gets more complex, going straight to the endpoint will not be a good solution.
- This is due to security, authentication, HTTP request body/headers and more.
- We will cover the above later on, but for now, we need Swagger.

© luv2code LLC

What is Swagger?

- Instead of going straight to the API endpoint, we can go to Swaggers.



localhost:8080/swagger-ui/index.html

© luv2code LLC

What is Swagger?

- Path is hard to remember:
  - localhost:8080/swagger-ui/index.html
- We can fix this to redirect based off of /docs.
  - New endpoint for Swagger: localhost:8080/docs

Too long to remember

Much better :-)

File: application.properties

```
springdoc.swagger-ui.path=/docs
```


© luv2code LLC

Path Variables

What are Path Variables

- Path Variables (Path Parameters) are request parameters that have been attached to the URL
- Path Variables are usually defined as a way to find information based on location
- Think of a computer file system:
  - You can identify the specific resources based on the file you are in

```
/Users/luv2code/Documents/java/springboot/section1
```


© luv2code LLC

Path Variables

REQUEST:

URL : localhost:8080/api/books

Read

Get

```
@GetMapping("/api/books")
public List<Book> getBooks() {
    return books;
}
```


© luv2code LLC

Path Variables

```
public void initializeBooks() {
    books.addAll(List.of(
      new Book("Title One", "Author One", "science"),
      new Book("Title Two", "Author Two", "science"),
      new Book("Title Three", "Author Three", "history"),
      new Book("Title Four", "Author Four", "math"),
      new Book("Title Five", "Author Five", "math"),
      new Book("Title Six", "Author Two", "math")
    ));
}
```

- Retrieve a single book by index in ArrayList

GET

`/api/books/{id}`

Retrieve a single book

`/api/books/0`

`/api/books/1`

`/api/books/2`

Known as a "path variable"

© luv2code LLC

Path Variables

```
public void initializeBooks() {
    books.addAll(List.of(
      new Book("Title One", "Author One", "science"),
      new Book("Title Two", "Author Two", "science"),
      new Book("Title Three", "Author Three", "history"),
      new Book("Title Four", "Author Four", "math"),
      new Book("Title Five", "Author Five", "math"),
      new Book("Title Six", "Author Two", "math")
    ));
}
```

Read

Get

REQUEST:

URL : localhost:8080/api/books/2

```
@GetMapping("/api/books/{id}")
public Book getBookByBookIndex(@PathVariable int id) {
    return books[2]
}
```

RESPONSE:

```
{
    "title": "Title Three",
    "author": "Author Three",
    "category": "history"
}
```


© luv2code LLC

Path Variables

REQUEST:

URL : localhost:8080/api/books/jump

Read

Get

```
@GetMapping("/api/books/{word}" )
public String getBookByTitle(@PathVariable String word) {
    return word;
}
```

RESPONSE:

```
{
    "jump"
}
```


© luv2code LLC

Path Variables

REQUEST:

URL : localhost:8080/api/books/jump%20ship%20now

Read

Get

```
@GetMapping("/api/books/{sentence}"
public String getBookByTitle(@PathVariable String sentence) {
    return sentence;
}
```

RESPONSE:

```
{
    "jump ship now"
}
```


© luv2code LLC

Path Variables

```
public void initializeBooks() {
    books.addAll(List.of(
      new Book("Title One", "Author One", "science"),
      new Book("Title Two", "Author Two", "science"),
      new Book("Title Three", "Author Three", "history"),
      new Book("Title Four", "Author Four", "math"),
      new Book("Title Five", "Author Five", "math"),
      new Book("Title Six", "Author Two", "math")
    ));
}
```

- Retrieve a single book by title in ArrayList

GET

`/api/books/{title}`

Retrieve a single book

`/api/books/title%20one`

`/api/books/title%20two`

`/api/books/title%20three`

Known as a "path variable"

© luv2code LLC

Path Variables

```
public void initializeBooks() {
    books.addAll(List.of(
      new Book("Title One", "Author One", "science"),
      new Book("Title Two", "Author Two", "science"),
      new Book("Title Three", "Author Three", "history"),
      new Book("Title Four", "Author Four", "math"),
      new Book("Title Five", "Author Five", "math"),
      new Book("Title Six", "Author Two", "math")
    ));
}
```

Read

Get

REQUEST:

URL : localhost:8080/books/title%20four = title four

```
@GetMapping("/api/books/{title}")
public Book getBookByTitle(@PathVariable String title) {
    for (Book book : books) {
        if (book.getTitle().equalsIgnoreCase(title)) {
            return book;
        }
    }
    return null;
}
```

RESPONSE:

```
{
    "title": "Title Four",
    "author": "Author Four",
    "category": "math"
}
```


© luv2code LLC

Path Variables

```
public void initializeBooks() {
    books.addAll(List.of(
      new Book("Title One", "Author One", "science"),
      new Book("Title Two", "Author Two", "science"),
      new Book("Title Three", "Author Three", "history"),
      new Book("Title Four", "Author Four", "math"),
      new Book("Title Five", "Author Five", "math"),
      new Book("Title Six", "Author Two", "math")
    ));
}
```

Read

Get

REQUEST:

URL : localhost:8080/books/title%20four = title four

```
@GetMapping("/api/books/{title}")
public Book getBookByTitle(@PathVariable String title) {
    return book.stream()
    .filter(book -> book.getTitle().equalsIgnoreCase(title))
    .findFirst()
    .orElse(null);
}
```

RESPONSE:

```
{
    "title": "Title Four",
    "author": "Author Four",
    "category": "math"
}
```


© luv2code LLC

Order Matters with Path Parameters (maybe remove?)

REQUEST:

URL : localhost:8080/api/books/1

Read

Get

```
@GetMapping("/api/books/{id}")
public Book getBookByBookLis[illegible]cation(@PathVariable int id) {
    return books[2]
}

@GetMapping("/api/books/{title}")
public String getBookByBookTitle(@PathVariable String title) {
    return title;
}
```

URL : localhost:8080/api/books/title%20four

© luv2code LLC

Query Parameters

What are Query Parameters

- Query Parameters are request parameters that have been attached after a “?”.
- Query Parameters have name=value pairs.
- Used for filtering of data.
- Example:
  - localhost:8080/api/books?category=math

© luv2code LLC

Query Parameters

REQUEST:

URL : localhost:8080/api/books?category=science

```
@GetMapping("/api/books")
public List<Book> getBooks(@RequestParam String category) {
    List<Book> filteredBooks = new ArrayList<>();
    for (Book book : books) {
        if (book.getCategory().equalsIgnoreCase(category)) {
            filteredBooks(book);
        }
    }
    return filteredBooks;
}
```

- By default query parameters are required.

© luv2code LLC

Query Parameters

- We can make query params optional.

```
@GetMapping("/api/books")
public List<Book> getBooks(@RequestParam(required = false) String category) {

    if (category == null) {
        return books;
    }

    List<Book> filteredBooks = new ArrayList<>();
    for (Book book : books) {
        if (book.getCategory().equalsIgnoreCase(category)) {
            filteredBooks(book);
        }
     }
     return filteredBooks;
}
```


© luv2code LLC

Query Parameters

- Return with a filter stream

```
@GetMapping("/api/books")
public List<Book> getBooks(@RequestParam(required = false) String category) {

    if (category == null) {
        return books;
    }

     return books.stream()
        .filter(book -> book.getCategory().equalsIgnoreCase(category))
        .toList();
}
```


© luv2code LLC

When to use Path vs Query Parameters

When to use Path Parameters

- Identify specific resources.
- Location & Identity is critical.
- /users/{userId}
- /products/{productId}
- /orders/{orderId}
- /articles/{articleId}/comments

Specific resources

© luv2code LLC

When to use Query Parameters

- Identify specific resources.
- Location & Identity is critical.
- /movies?genre=Comedy
- /products?priceMin=10
- /products?priceMin=10&priceMax=100
- /books?category=Science

Filtered resources

© luv2code LLC

Path + Query Parameters

REQUEST:

URL : localhost:8080/api/books/1?category=math

```
@GetMapping("/books/{libraryId}")
public List<Book> getBooksByLibraryAndCategory(@PathVariable int libraryId,
                                               @RequestParam String category) {
    <fetch books from library>
}
```


© luv2code LLC

POST HTTP Request Method

What is the POST Request Method

- Used to create data
- POST can have a body that has additional information that GET does not have
- Example of Body:

```
{“title”:”Title Seven”, “author”:”Author Two”, “category”: “math”}
```


© luv2code LLC

POST Request Method

```
public void initializeBooks() {
    books.addAll(List.of(
      new Book("Title One", "Author One", "science"),
      new Book("Title Two", "Author Two", "science"),
      new Book("Title Three", "Author Three", "history"),
      new Book("Title Four", "Author Four", "math"),
      new Book("Title Five", "Author Five", "math"),
      new Book("Title Six", "Author Two", "math")
    ));
}
```

REQUEST:

URL : localhost:8080/api/books

Create

Post

```
@PostMapping("/api/books")
public void createBook(@RequestBody Book newBook) {
    for (Book book : books) {
        if (book.getTitle().equalsIgnoreCase(newBook.getTitle())) {
            return;
        }
     }
    books.add(newBook);
```

```
{"title":"Title Seven", "author":"Author Two", "category": "math"}
```


© luv2code LLC

POST Request Method

```java
public void initializeBooks() {
   books.addAll(List.of(
     new Book("Title One", "Author One", "science"),
     new Book("Title Two", "Author Two", "science"),
     new Book("Title Three", "Author Three", "history"),
     new Book("Title Four", "Author Four", "math"),
     new Book("Title Five", "Author Five", "math"),
     new Book("Title Six", "Author Two", "math")
   ));
}
```

REQUEST:

URL : localhost:8080/api/books

Create

Post

```java
@PostMapping("/api/books")
public void createBook(@RequestBody Book newBook) {
   for (Book book : books) {
       if (book.getTitle().equalsIgnoreCase(newBook.getTitle())) {
           return;
       }
    }
   books.add(newBook);
```

```
{"title":"Title Seven", "author":"Author Two", "category": "math"}
```


© luv2code LLC

POST Request Method

```
public void initializeBooks() {
    books.addAll(List.of(
      new Book("Title One", "Author One", "science"),
      new Book("Title Two", "Author Two", "science"),
      new Book("Title Three", "Author Three", "history"),
      new Book("Title Four", "Author Four", "math"),
      new Book("Title Five", "Author Five", "math"),
      new Book("Title Six", "Author Two", "math")
    ));
}
```

REQUEST:

URL : localhost:8080/api/books

```
@PostMapping("/api/books")
public void createBook(@RequestBody Book newBook) {
    boolean isNewBook = books
      .stream()
      .noneMatch(book -> book.getTitle().equalsIgnoreCase(newBook.getTitle()));

    if (isNewBook) {
        books.add(newBook);
    }
```

```
{"title":"Title Seven", "author":"Author Two", "category": "math"}
```


© luv2code LLC

PUT HTTP Request Method

What is the PUT Request Method

- Used to update data
- PUT can have a body that has additional information (like POST) that GET does not have
- Example of Body:

```
{“title”:”Title Six”, “author”:”Author Two”, “category”: “science”}
```


© luv2code LLC

PUT Request Method

Update

Put

REQUEST:

URL : localhost:8080/api/books/{title}

```
@PutMapping("/api/books/{title}")
public void updateBook(@PathVariable String title, @RequestBody Book updatedBook)
{
    for (int i = 0; i < books.size(); i++) {
        if (book.get(i).getTitle().equalsIgnoreCase(title)) {
            books.set(i, updatedBook);
            return;
        }
    }
}
```


© luv2code LLC

DELETE HTTP Request Method

What is the DELETE Request Method

- Used to delete data

Delete

Delete

REQUEST:

URL : localhost:8080/api/books/{title}

```
@DeleteMapping("/api/books/{title}")
public void deleteBook(@PathVariable String title) {
    books.removeIf(book -> book.getTitle().equalsIgnoreCase(title));
}
```


© luv2code LLC

@RequestMapping

What is @RequestMapping

- So far, we have to add /api/books into each of our endpoints.

```
@GetMapping("/api/books")
@GetMapping("/api/books/{title}")
@PutMapping("/api/books/{title}")
@PostMapping("/api/books")
@DeleteMapping("/api/books/{title}")
```

Each endpoint we need to type in /api/books

© luv2code LLC

What is @RequestMapping

- Add @RequestMapping to a class to auto add init endpoint values.

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

   @GetMapping
   @GetMapping("/{title}")
   @PutMapping("/{title}")
   @PostMapping
   @DeleteMapping("/{title}"
}
```

Keep original code

/api/books is now automatically implemented on each endpoint within this controller

© luv2code LLC

Books Project 2

Project 2

- Project two will still be focused on creating Book API Endpoints
- Continued Education includes:
  - GET, POST, PUT, DELETE Request Methods
- New Information will include:
  - Data Validation, Exception Handling, Status Codes, Swagger Configuration, Java Request Objects

© luv2code LLC

Creating a new books project

- We will update our class called Book
- We will be using these books throughout this project:

```
public class Book {
    private long id;
    private String title;
    private String author;
    private String category;
    private int rating;

   public Book(long id, String title, String author, String category, <…> ) {
        this.id = id;
        this.title = title;
         this.author = author;
         this.category = category;
        <…>
   }

< Getters and Setters >
```


© luv2code LLC

HTTP Request Methods

CRUD		HTTP Request Methods
Create	→	POST
Read	→	GET
Update	→	PUT
Delete	→	DELETE

© luv2code LLC

Enhance Book Objects

Enhancing our Book object

- We will enhance our class called Book
- We will be using these books throughout this project:

```
public class Book {
    private long id;
    private String title;
    private String author;
    private String category;
    private int rating;

   public Book(long id, String title, String author, String category, <…> ) {
        this.id = id;
        this.title = title;
         this.author = author;
         this.category = category;
        <…>
   }

< Getters and Setters >
```

Unique identifier

New fields

© luv2code LLC

New fields

- Creating and enhancing books to learn the basics of Spring Boot Rest

```
public void initializeBooks() {
    books.addAll(List.of(
     new Book("Title One", "Author One", "science"),
      new Book("Title Two", "Author Two", "science"),
      new Book("Title Three", "Author Three", "history"),
      new Book("Title Four", "Author Four", "math"),
      new Book("Title Five", "Author Five", "math"),
      new Book("Title Six", "Author Two", "math")
    ));
}
```

```
public void initializeBooks() {
    books.addAll(List.of(
      add(new Book(1, "Java Master", "Eric R.", "Comp Science", 5));
}}
```

New fields

© luv2code LLC

API Endpoints ID

Path Variables

Read

Get

REQUEST:

URL : localhost:8080/api/books/title%20four

```
@GetMapping("/{title}")
public Book getBookByBookTitle(@PathVariable String title) {
    return book.stream()
      .filter(book -> book.getTitle().equalsIgnoreCase(title))
      .findFirst()
      .orElse(null);
}
```


© luv2code LLC

Path Parameters

Read

Get

REQUEST:

URL : localhost:8080/api/books/2

URL : localhost:8080/api/books/title%20four

- Endpoints match.
- We need to change to not "confuse" Spring Boot.

URL : localhost:8080/api/books/{dynamic}

URL : localhost:8080/api/books/2

URL : localhost:8080/api/books/title%20four

© luv2code LLC

Path Variables

Read

Get

REQUEST:

URL : localhost:8080/api/books/2

```
@GetMapping("/{id}")
public Book getBookById(@PathVariable long id) {
    return book.stream()
    .filter(book -> book.getId() == id)
    .findFirst()
    .orElse(null);
}
```

RESPONSE:

```
{
    "id": 2,
    "title": "Title Two",
    "author": "Author Two",
    "category": "science",
    "rating": 3
}
```


© luv2code LLC

Path Variables

Update

Put

REQUEST:

URL : localhost:8080/api/books/2

```
@PutMapping("/{id}")
public void updateBook(@PathVariable long id, @RequestBody Book updatedBook) {
    for (int i = 0; i < books.size(); i++) {
        if (book.get(i).getId() == id) {
            books.set(i, updatedBook);
            return;
        }
    }
}
```


© luv2code LLC

Path Variables

Delete

Delete

REQUEST:

URL : localhost:8080/api/books/2

```
@DeleteMapping("/{id}")
public void deleteBook(@PathVariable long id) {
    books.removeIf(book -> book.getId() == id);
}
```


© luv2code LLC

BookRequest Object

What are Request Objects

- Request objects are designed specifically to capture and validate incoming requests from HTTP.
- Only contain data necessary for the request.
- Only used for sending data to server, unlike entity objects which typically have more information and has a tighter connection to business logic.
- Improves security.
- Flexible by decoupling entity models

© luv2code LLC

Book Entity

```
public class Book {
    private long id;
    private String title;
    private String author;
    private String category;
    private int rating;

   public Book(String title, String author, String category, <…> ) {
        this.title = title;
         this.author = author;
         this.category = description;
        <…>
   }

< Getters and Setters >
```

- Includes an id ..
- Id is not needed for POST Requests.

© luv2code LLC

Book Entity

Create

Post

```
{
  ~~"id": 8~~

  "title": "Title Seven",

  "author": "Author One",

  "category": "science",

  "rating": 5
}
```

- Pass in id.
- We want to not include id.
- Have app create id for us.
- All we have to do:
  - Title
  - Author
  - Category
  - Rating

© luv2code LLC

BookRequest Object

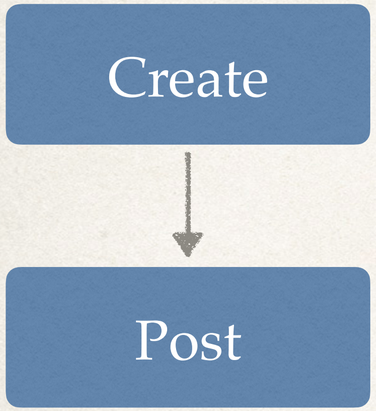
```
public class BookRequest {
    private String title;
    private String author;
    private String category;
    private int rating;

   public BookRequest(String title, String author, String category, <…> ) {
        this.title = title;
         this.author = author;
         this.category = description;
        <…>
   }

< Getters and Setters >
```

- Dedicated Book Object for sending a request to server

© luv2code LLC

POST Request Method

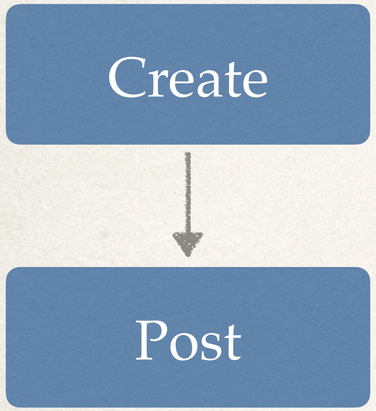
REQUEST:
URL : localhost:8080/api/books




```
@PostMapping()
public void createBook(@RequestBody BookRequest bookRequest) {
    long id;
    if (books.isEmpty()) {
       id = 1;
    } else {
       id = books.get(books.size() - 1).getId() + 1;
    }

    Book book = new Book (
       id, bookRequest.getTitle(), bookRequest.getAuthor,
       bookRequest.getCategory, bookRequest.getRating()
    );

    books.add(book);
```


© luv2code LLC

POST Request Method




```
@PostMapping()
public void createBook(@RequestBody BookRequest bookRequest) {
    long id;
    if (books.isEmpty()) {
       id = 1;
    } else {
        id = books.get(books.size() - 1).getId() + 1;
    }
    Book book = convertToBook(id, bookRequest);

    books.add(book);
```

```
private Book convertToBook(long id, BookRequest bookRequest) {

    return new Book(id, bookRequest.getTitle(), bookRequest.getAuthor(),
                    bookRequest.getCategory(), bookRequest.getRating());
    }
```


© luv2code LLC

POST Request Method

```
@PostMapping
public void createBook(@RequestBody BookRequest bookRequest) {

    long id = books.isEmpty() ? 1 : books.get(books.size() - 1).getId() + 1;

    Book newBook = convertToBook(id, bookRequest);

    books.add(newBook);
  }
```

```
private Book convertToBook(long id, BookRequest bookRequest) {

    return new Book(id, bookRequest.getTitle(), bookRequest.getAuthor(),
                    bookRequest.getCategory(), bookRequest.getRating());
    }
```


© luv2code LLC

Put Request Update

Updates to Put Request

- Consumes Book Entity.
  - We want to change to BookRequest.
- Have it use our convertToBook functionality where we can pass in the:
  - id from the path variable
  - BookRequest object.
- Update current book.

© luv2code LLC

PUT Request Method

Update

Put

REQUEST:

URL : localhost:8080/api/books/2

BookRequest

```
@PutMapping("/{id}")
public void updateBook(@PathVariable long id, @RequestBody BookRequest bookRequest) {
    for (int i = 0; I < books.size(); i++) {
        if (book.get(i).getId() == id) {
            Book updatedBook = convertToBook(id, bookRequest)
            books.set(i, updatedBook);
            return;
        }
    }
}
```


© luv2code LLC

Data Validation

What is Data Validation

- Process of ensuring that data is accurate, complete and compliant with rules and formats.
- Check specific data (incoming requests) to validate they are within specific constraints.
- Validation is crucial in maintaining data integrity, preventing errors and ensuring data is correct.

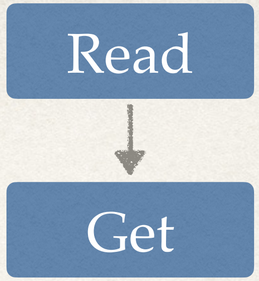
© luv2code LLC

Main Types of Data Validation

- **Presence Validation**: Ensures that required fields are not null or empty.
- **Range Validation**: Ensures that numerical values fall within a specified range.
- **Format Validation**: Checks if data follows appropriate formats (email, phone number, etc.)

© luv2code LLC

Path Variables Validation



- Problem:
  - Pass any data at an id
  - Entire function runs until no matches are found.

URL : localhost:8080/api/books/2

```
@GetMapping("/{id}")
public Book getBookById(@PathVariable long id) {

    return book.stream()
        .filter(book -> book.getId() == id)
        .findFirst()
        .orElse(null);
}
```

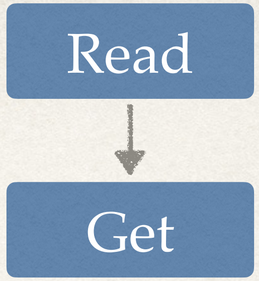
0

-10

-1001

© luv2code LLC

Path Variables Validation



- Implement data validation on id

```
@GetMapping("/{id}")
public Book getBookById(@PathVariable long id) {

    if (id < 1) {
        return null;
    }

    return book.stream()
        .filter(book -> book.getId() == id)
        .findFirst()
        .orElse(null);
}
```

Guard Clause

© luv2code LLC

Path Variables Validation

- Use validation package from Jakarta

Read

Get

```
@GetMapping("/{id}")
public Book getBookById(@PathVariable @Min(value = 1) long id) {

    return book.stream()
        .filter(book -> book.getId() == id)
        .findFirst()
        .orElse(null);
}
```

Data Validation or throws an error

Also works for query parameters

© luv2code LLC

Popular Validations

- jakarta.validation.contraints
  - @NotEmpty
    - Ensures that the field is not null or empty. It can be applied to String, Collection, Map, or Array types. It checks that the field has at least one element or character.
  - @Size
    - Defines a constraint on the size of a field. It can be applied to String, Collection, Map, or Array types. The annotation allows you to specify the minimum and maximum size (or length) allowed.

© luv2code LLC

Popular Validations

- jakarta.validation.contraints
  - @Min
    - Ensures that the num value of the field is at least the specified minimum value. It can be applied to numeric types like int, long, float, double, etc.
  - @Max
    - Ensures that the numeric value of the field does not exceed the specified maximum value. It can be applied to numeric types like int, long, float, double, etc

© luv2code LLC

Data validation on Request Objects

- Data validation can be added on Request Objects not just Path and Query Parameters.
- BookRequest object has 4 fields: Title, Author, Category, Rating.
- We want to add data validation on the BookRequest to ensure data quality before processing.

© luv2code LLC

How to implement validation?

- jakarta.validation.contraints
  - @Size
  - @Min
  - @Max

```
@Size(min=1, max=30)
private String title;
```

```
@Size(min=1, max=40)
private String author;
```

```
@Size(min=1, max=30)
private String category;
```

```
@Min(value=1)
@Max(value=5)
private int rating;
```


© luv2code LLC

Update BookRequest Entity

- Request that comes from client for validation

```
public class BookRequest {

    @Size(min=1, max=30)
    private String title;

    @Size(min=1, max=40)
    private String author;

    @Size(min=1, max=30)
    private String category;

    @Min(value=1)
    @Max(value=5)
    private int rating;

< Constructor, Getters & Setters >
```


© luv2code LLC

BookRequest Validation

- Add @Valid to create the validation on the RequestBody.

```
@PostMapping
public Book createBook(@Valid @RequestBody BookRequest bookRequest) {

    long id = books.isEmpty() ? 1 : books.get(books.size() - 1).getId() + 1;

    Book newBook = convertToBook(id, bookRequest);

    books.add(newBook);
}
```


© luv2code LLC

(Not complete) OpenAPI Specs

© luv2code LLC

@ResponseStatus

© luv2code LLC

What is @ResponseStatus

- Used to specify the HTTP status code that should be returned by a handler method or an exception class.
- Typically applied to methods in a controller or directly on an exception class.
- Customizing HTTP Status Codes:
  - Customize the HTTP response status for certain conditions, like returning 404 Not Found when a resource is not found or 201 Created when a new resource is successfully created.

© luv2code LLC

What is @ResponseStatus

```
@ResponseStatus(HttpStatus.NOT_FOUND)
@GetMapping("/resource-not-found")
public void resourceNotFound() {
    // Logic that returns a resource not found
}
```

```
@ResponseStatus(HttpStatus.CREATED)
@PostMapping
public void createBook(@RequestBody BookRequest bookRequest) {

    long id = books.isEmpty() ? 1 : books.get(books.size() - 1).getId() + 1;

    Book newBook = convertToBook(id, bookRequest);

    books.add(newBook);
}
```


© luv2code LLC

Swagger Clean Up

Swagger Clean Up

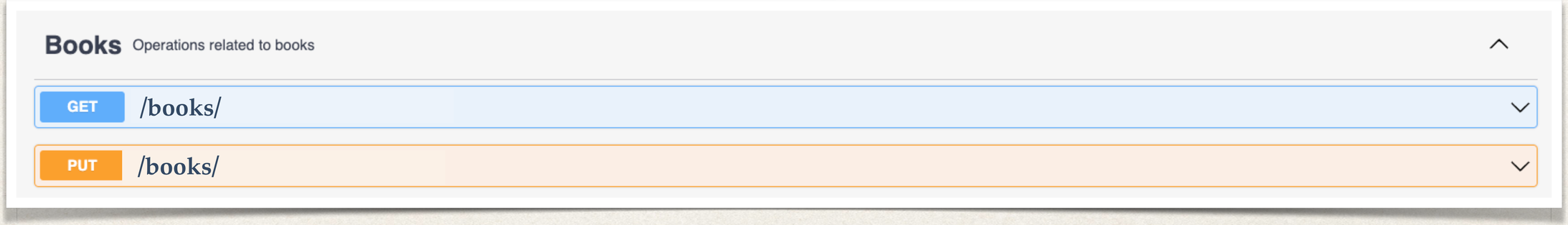
- Adding more logic to our Swagger UI
  - @Tag
    - Ability to see why the group of endpoints are used for.
  - @Operation
    - Ability to see why an endpoint exists.
  - @Parameter
    - Ability to see why a parameter must be included

© luv2code LLC

Swagger @Tag

- Adding more logic to our Swagger UI
  - @Tag
    - Ability to see why the group of endpoints are used for.

```
@Tag(name="Books", description="Operations related to books")
```

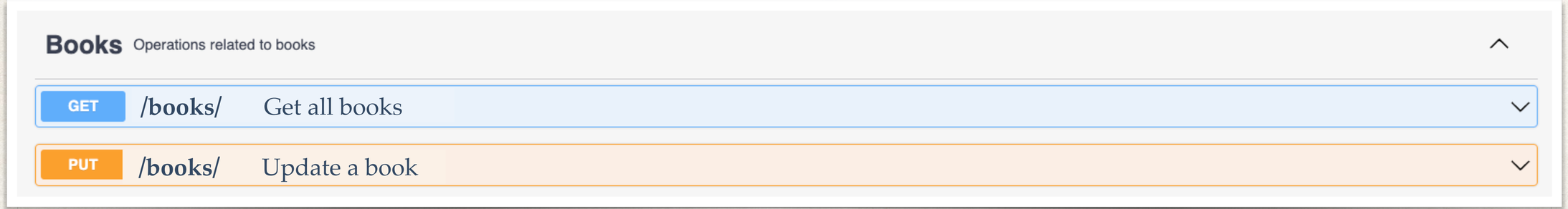


© luv2code LLC

Swagger @Operation

- Adding more logic to our Swagger UI
  - @Operation
    - Ability to see why an endpoint exists.

```
@Operation(summary="Get all books", description="Retrieve list of all available books")
```

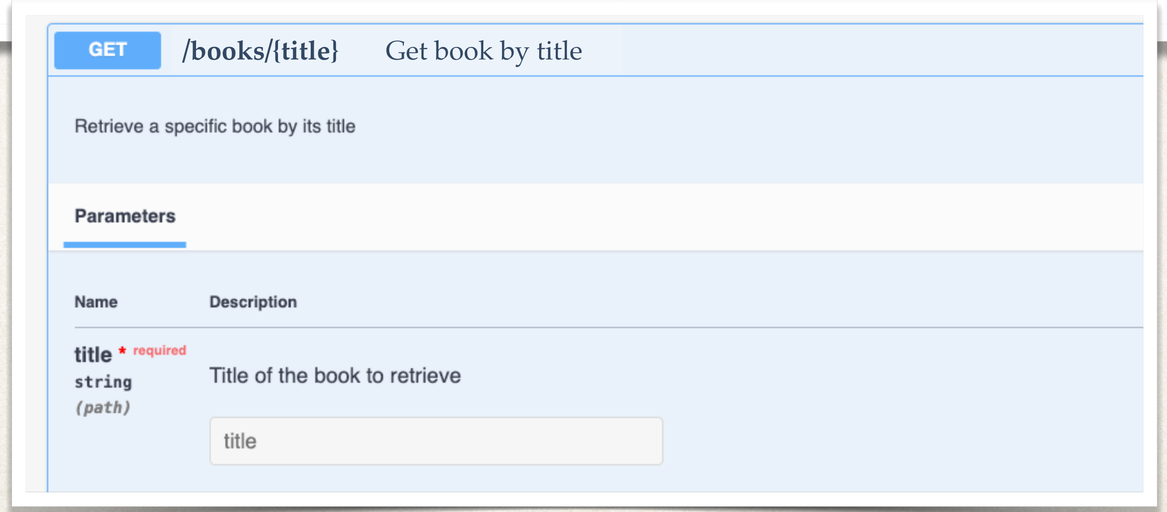


© luv2code LLC

Swagger @Parameter

- Adding more logic to our Swagger UI
  - @Parameter
    - Ability to see why a parameter must be included

```
@Parameter(description="Title of the book to retrieve")
```



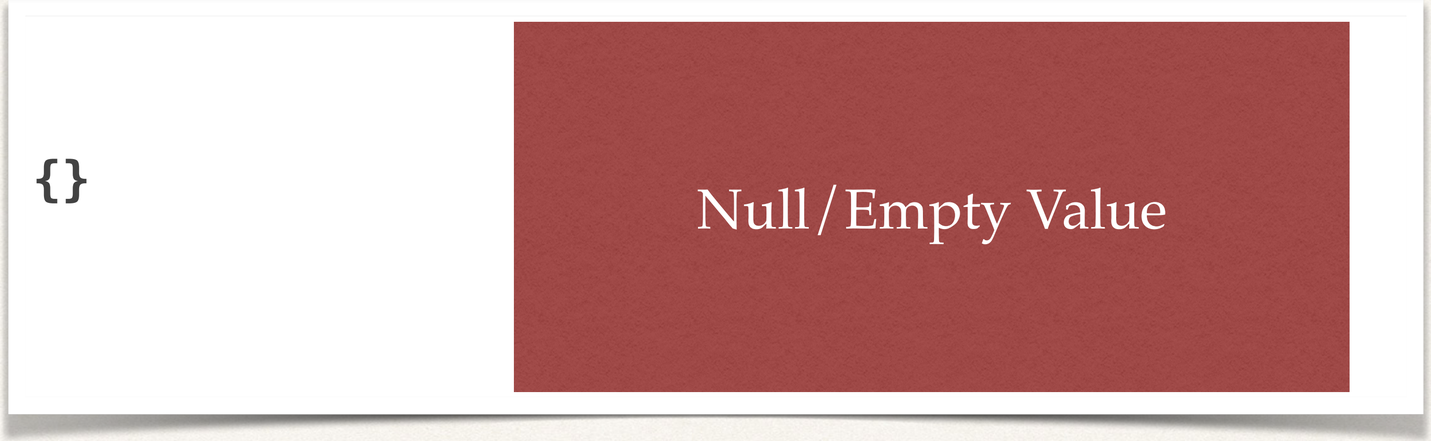
© luv2code LLC

Spring REST - Exception Handling

© luv2code LLC

What happens with a bad value?

- Bad book id of 9999 …



- This is not what we want to happen.
- We want to return something useful by throwing a Book Not Found Exception.

© luv2code LLC

We really want this …

- Handle the exception and return error as JSON

```
{
    “status”: 404,
    “message”: “Book 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

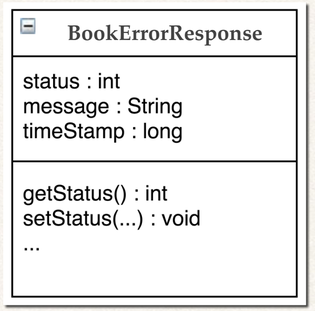
3. Update REST service to throw exception if book not found

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

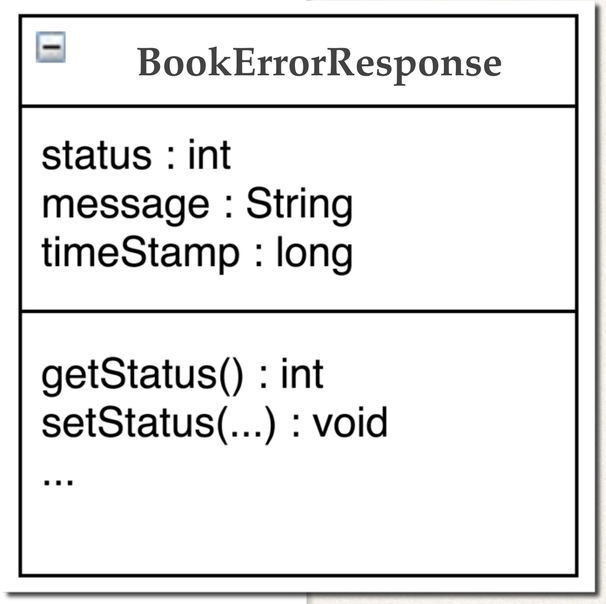
```
{
    "status": 404,
    "message": "Book id not found - 9999",
    "timestamp": "1526149650271"
}
```


© luv2code LLC

Step 1: Create custom error response class

File: BookErrorResponse.java

```
public class BookErrorResponse {

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

  // constructors

  // getters / setters

}
```




```
{
    "status": 404,
    "message": "Book id not found - 9999",
    "timestamp": "1526149650271"
}
```


© luv2code LLC

Step 2: Create custom book exception

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

© luv2code LLC

Step 2: Create custom book exception

File: BookNotFoundException.java

```
public class BookNotFoundException extends RuntimeException {

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

}
```

Call super class constructor

© luv2code LLC

Step 3: Update REST service to throw exception

File: BookController.java

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

  @GetMapping("/id}")
  public Book getBookById(@PathVariable long id) {

    // check the id against list size

    return books.stream()
                .filter(book -> book.getId() == id)
                .findFirst()
                .orElseThrow(() -> new BookNotFound("Book id not found - " + bookId));

  }
  …
}
```

Happy path

Could also check results from DB

Throw exception

© 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

File: Book

Exception handler method

Type of the response body

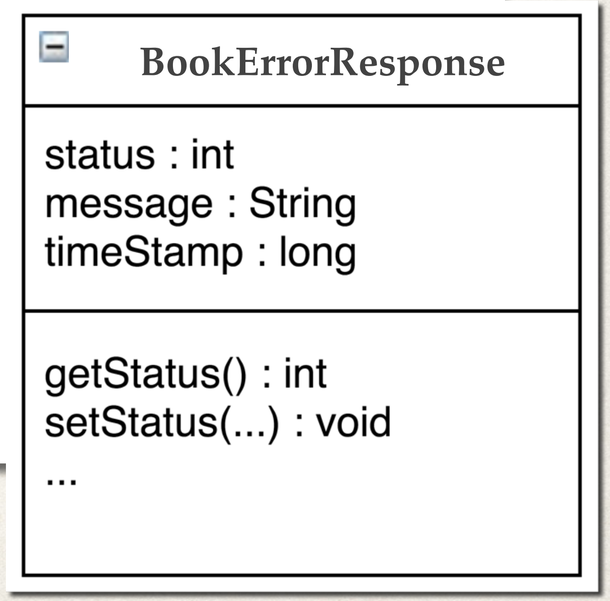
Exception type to handle / catch

```
@RestCon
@RequestMapping("/api/books")
public class BookRestController {
  …

  @ExceptionHandler
  public ResponseEntity<BookErrorResponse> handleException(BookNotFoundException exc) {

    BookErrorResponse error = new BookErrorResponse();

    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": "Book id not found - 9999",
   "timestamp": "1526149650271"
}
```



© luv2code LLC

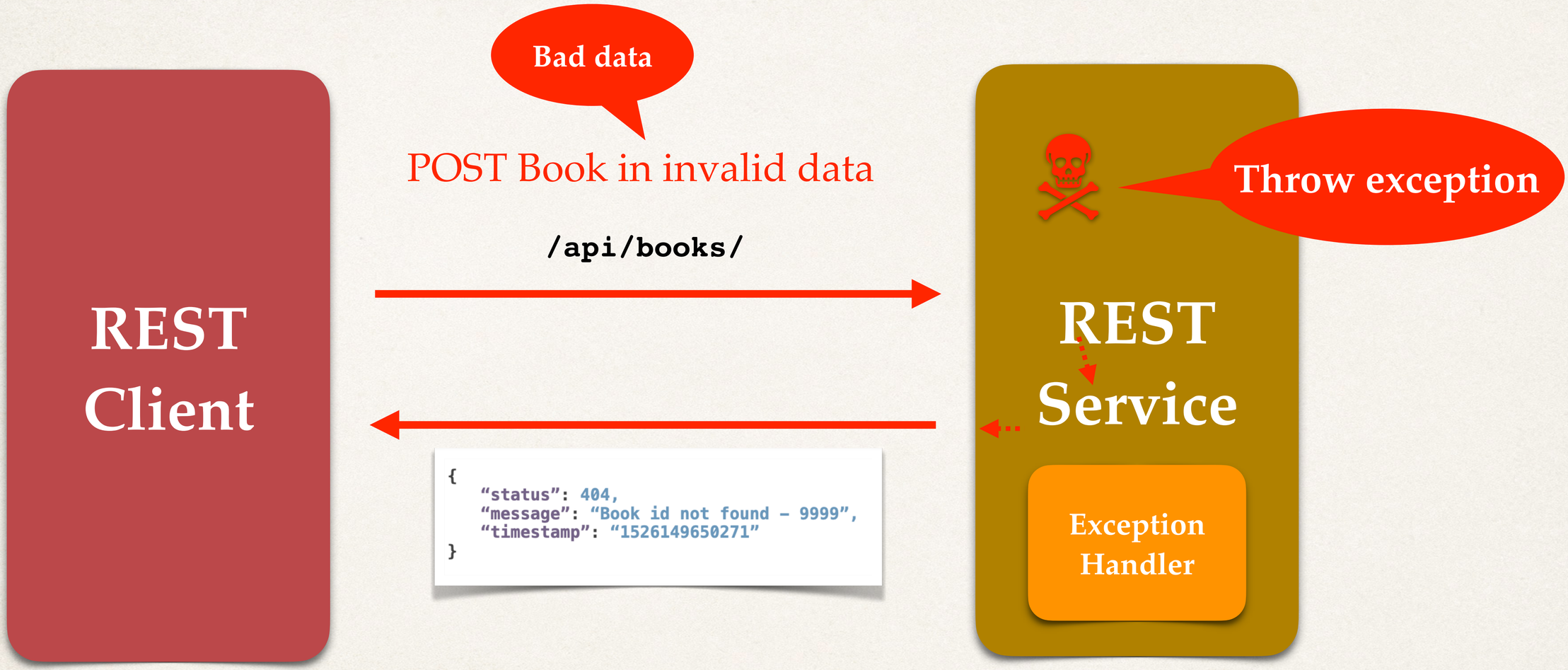
Spring REST Exception Handling

© 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

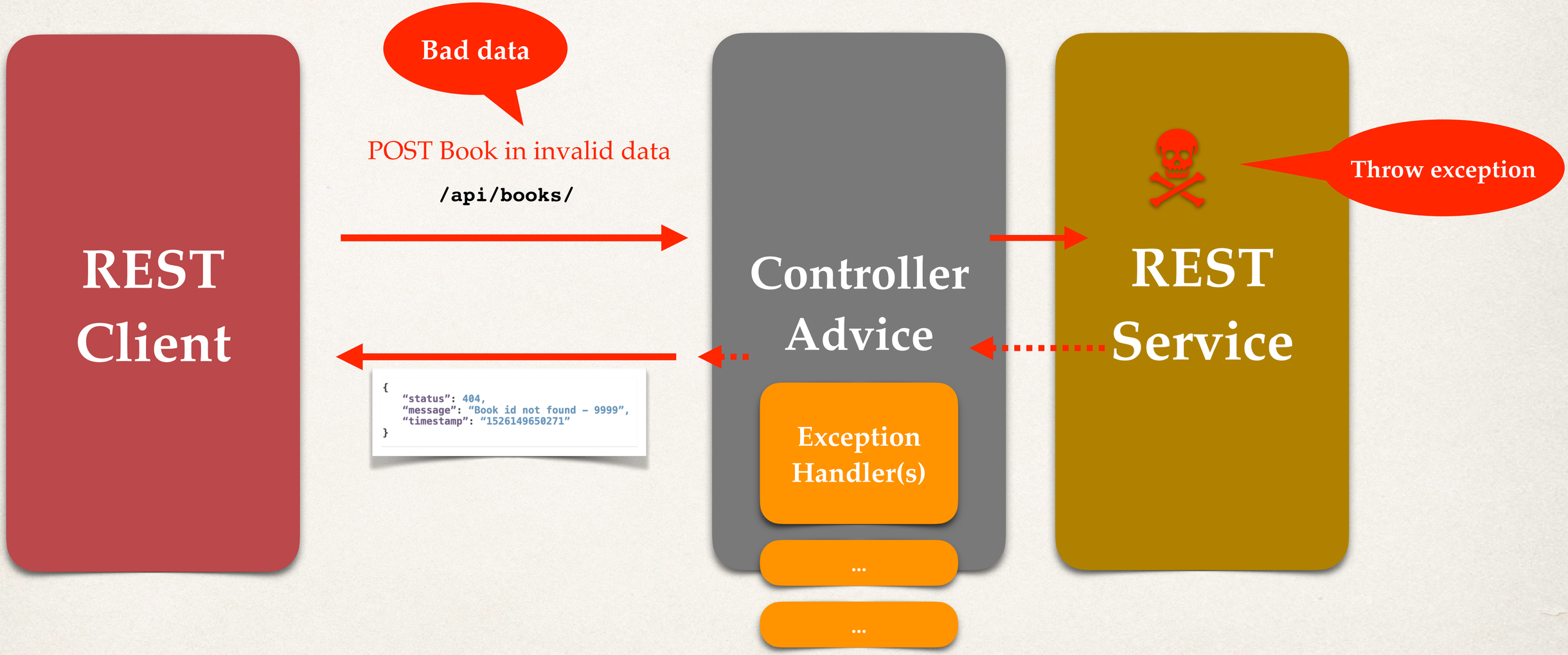
Spring @ControllerAdvice

- `@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: BookRestExceptionHandler.java

```
@ControllerAdvice
public class BookRestExceptionHandler {

  ...

}
```


© luv2code LLC

Step 2: Refactor - remove exception handling

File: BookRestController.java

```
@RestController
@RequestMapping("/api/books")
public class BookRestController {
  …

  @ExceptionHandler
  public ResponseEntity<BookErrorResponse> handleException(BookNotFoundException exc) {

    BookErrorResponse error = new BookErrorResponse();

    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

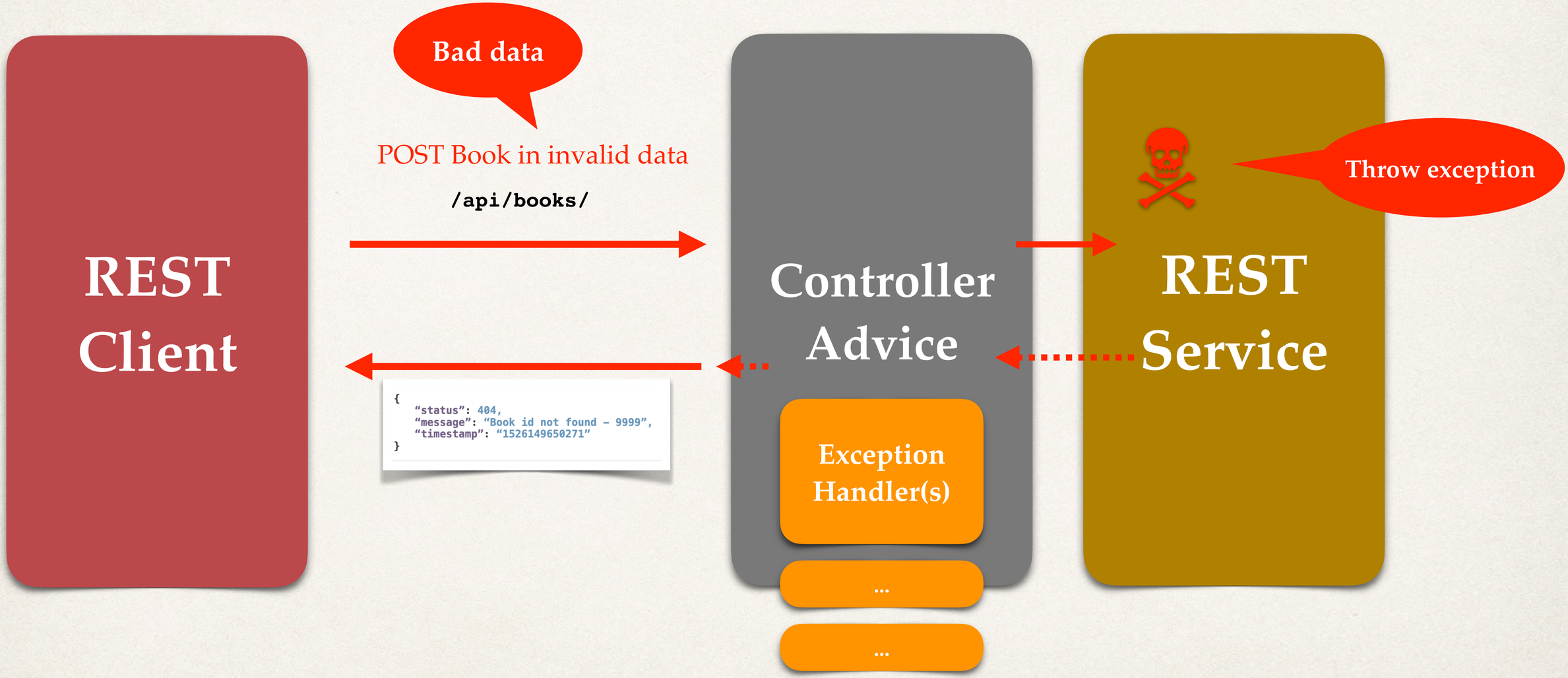
File: BookRestExceptionHandler.java

Same code as before

```
@ControllerAdvice
public class BookRestExceptionHandler {

  @ExceptionHandler
  public ResponseEntity<BookErrorResponse> handleException(BookNotFoundException exc) {

    BookErrorResponse error = new BookErrorResponse();

    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

Project 3 Overview

© luv2code LLC

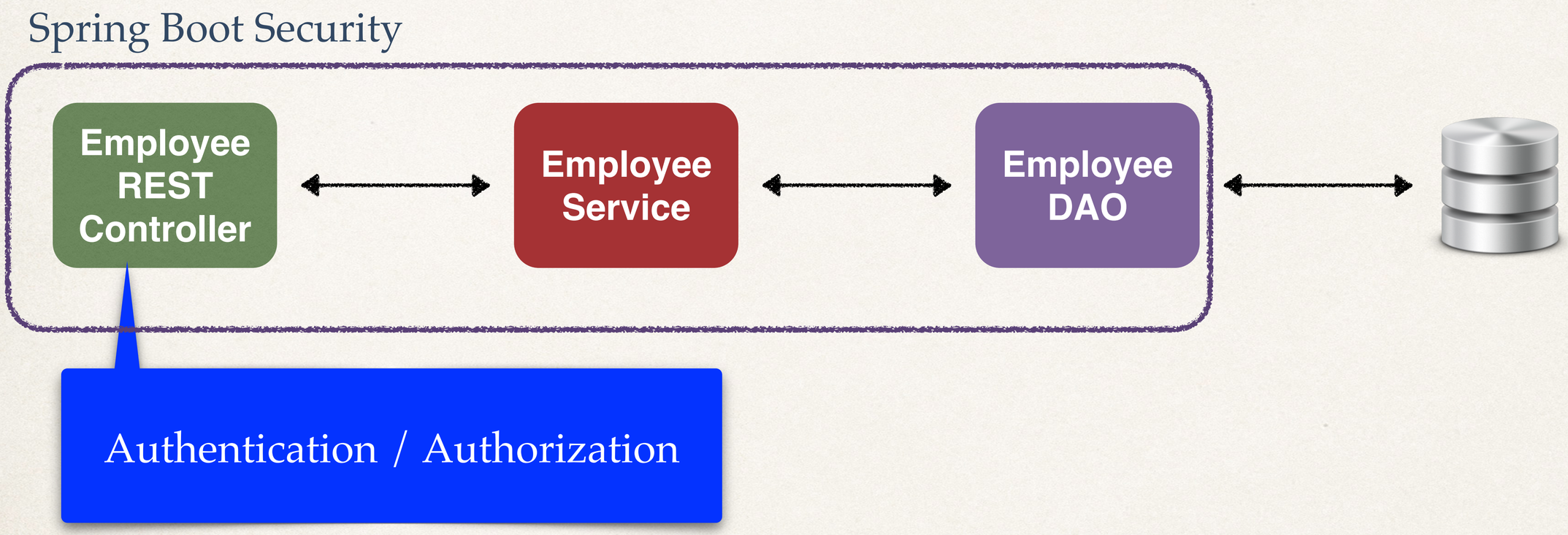
Project 3

- Project three we will be switching our focus to Employees instead of Books
- New Information will include:
  - SQL Databases
  - Authentication
  - Authorization
  - Hashing Passwords

Continued learning from other projects

© luv2code LLC

Application Architecture



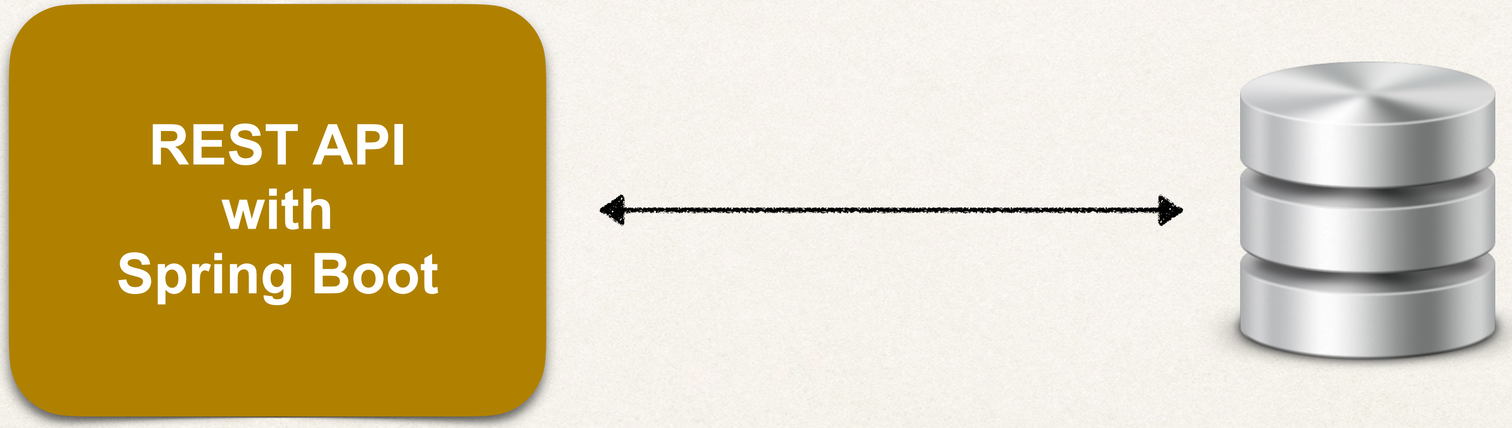
© luv2code LLC

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

REST API

HTTP Method		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/{employeeId}	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee

© luv2code LLC

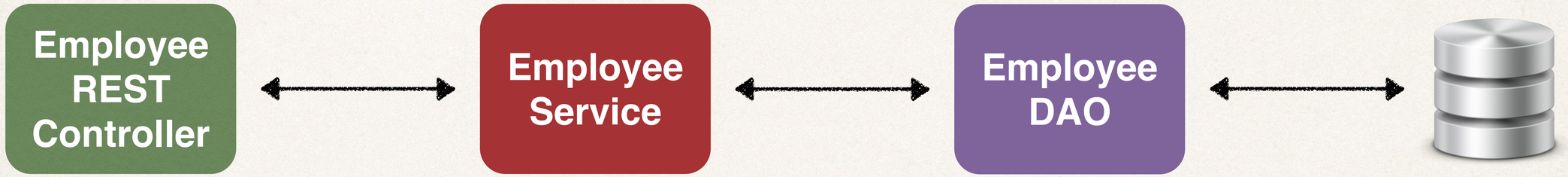
Development Process

Step-By-Step

1. Create Spring Boot project using Spring Initializer
2. Set up Database Dev Environment
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

Creating Spring Boot Project

© luv2code LLC

Spring Initializr

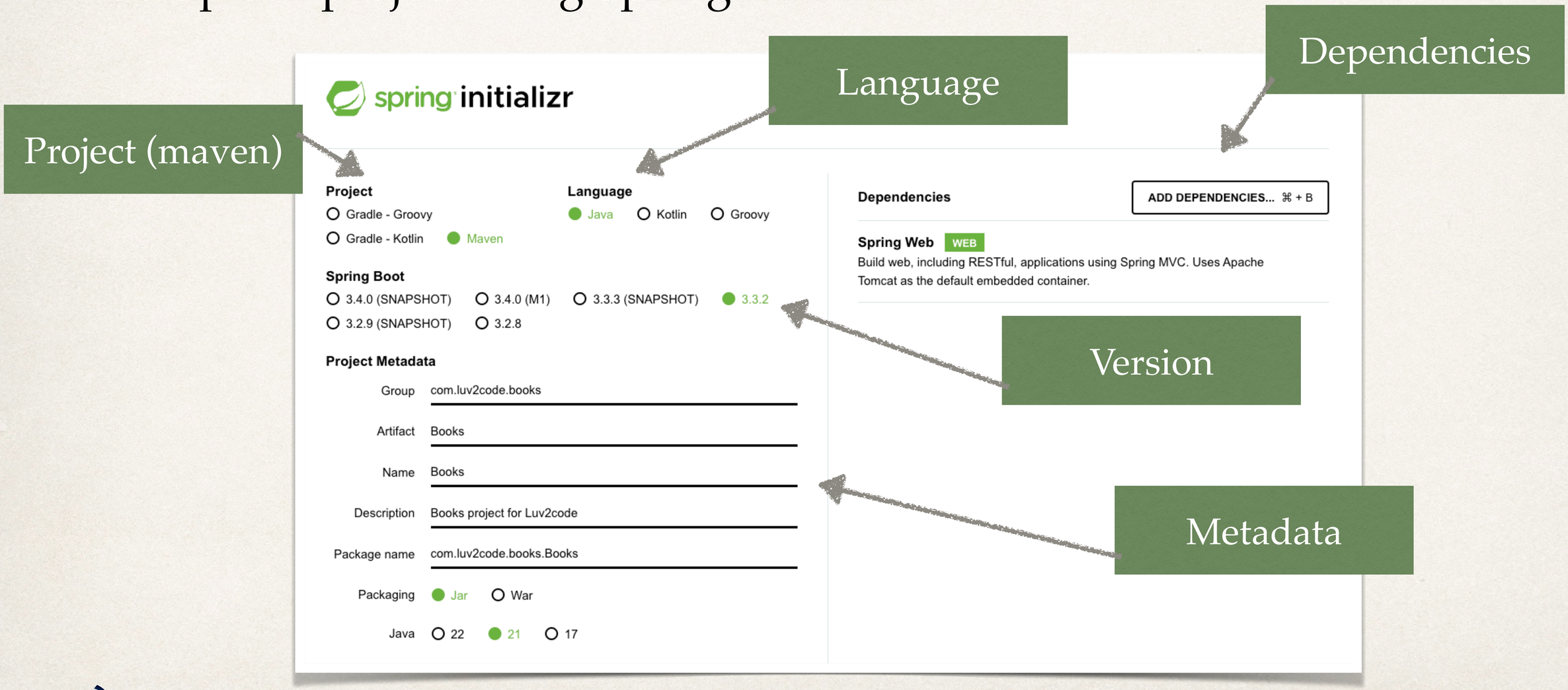
- Here we can set the project configurations for our upcoming Spring project.
- Allows us to choose the project, programming language, Spring Boot version, dependencies and metadata for our application.

start.spring.io

© luv2code LLC

Spring Initializr

- Setup our project using Spring Initializr



© luv2code LLC

Spring Initializr

- Spring Initializr will download a .zip file onto our machine.
- We will need to unzip the .zip file.
- Open the project on IntelliJ.
- Download all dependencies using Maven.
- Sounds fun - Let's go ahead and dive in!

© luv2code LLC

Development Process

Step-By-Step

1. Create Spring Boot project using Spring Initializer
2. Set up Database Dev Environment
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

H2 Database Overview

© luv2code LLC

What is a H2 database?

- H2 is a lightweight, open-source RDBMS written in Java.
- Most used as a memory or file database.
- Memory - On RAM, usually used for testing.
- File - Persistent database where it stores data on a file within our project directory.
- We will be using a file database configuration for this project.

© luv2code LLC

How do we use a H2 database?

- We already have H2 installed within our POM file:

```
<dependency>
 <groupId>com.h2database</groupId>
 <artifactId>h2</artifactId>
 <scope>runtime</scope>
</dependency>
```

- This is all the installation we need to properly use a H2 database!

© luv2code LLC

How do we use a H2 database?

- We then need to add the configurations in our application.properties file.

```
# Configs
spring.datasource.url=jdbc:h2:file:./data/employeedb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# Console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.jpa.hibernate.ddl-auto=update
```

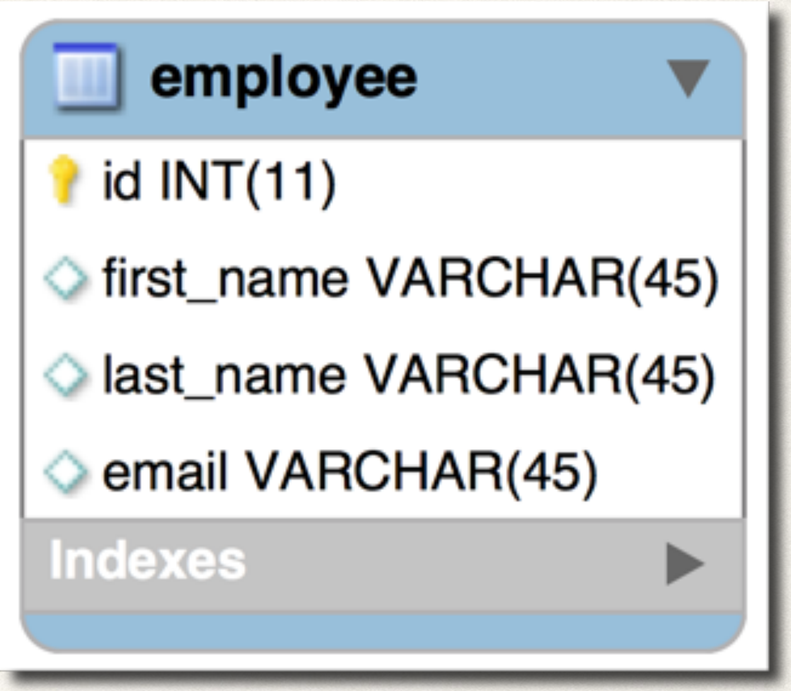

© luv2code LLC

employee.sql

1. File: employee SQL table.

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

3. Load table with sample data.



SQL Scripts available in resources

© luv2code LLC

Create Employee Entity

- Create the Employee entity that matches the database table

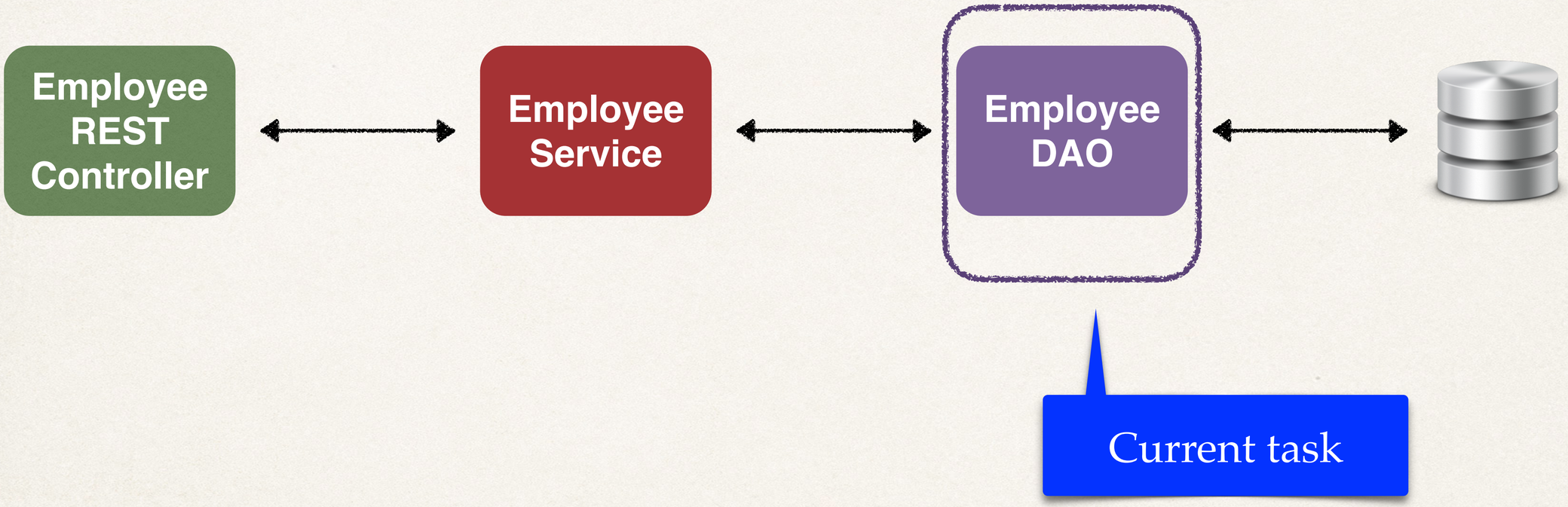
```
public class Employee {

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

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

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

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

< Constructor, Getters & Setters >
```


© luv2code LLC

Create DAO in Spring Boot

© luv2code LLC

Application Architecture



© luv2code LLC

Development Process

1. Create Spring Boot project using Spring Initializer
2. Set up Database Dev Environment
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

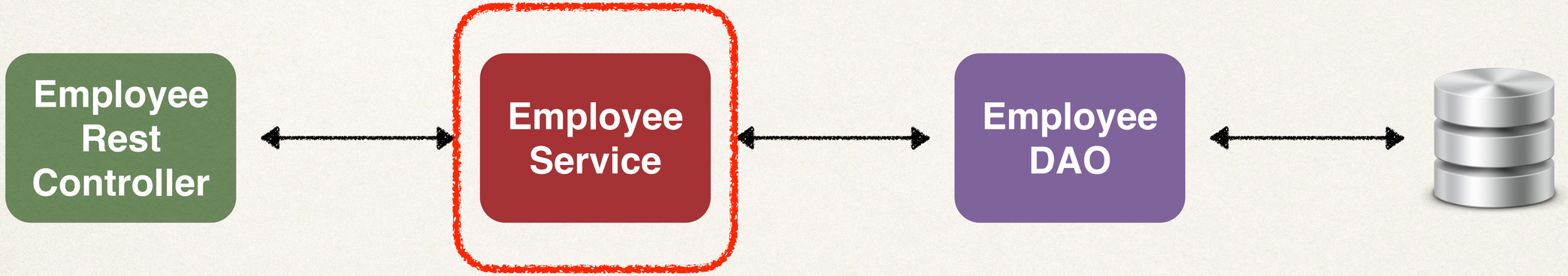
1. Update db configs in application.properties
2. Create Employee entity
3. Create DAO interface
4. Create DAO implementation
5. Create REST controller to use DAO

© luv2code LLC

Define Services with @Service

© luv2code LLC

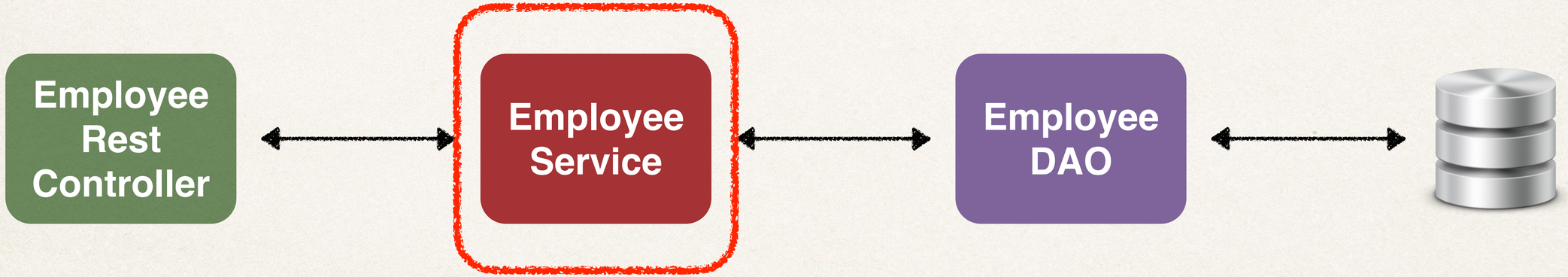
Refactor: Add a Service Layer



© luv2code LLC

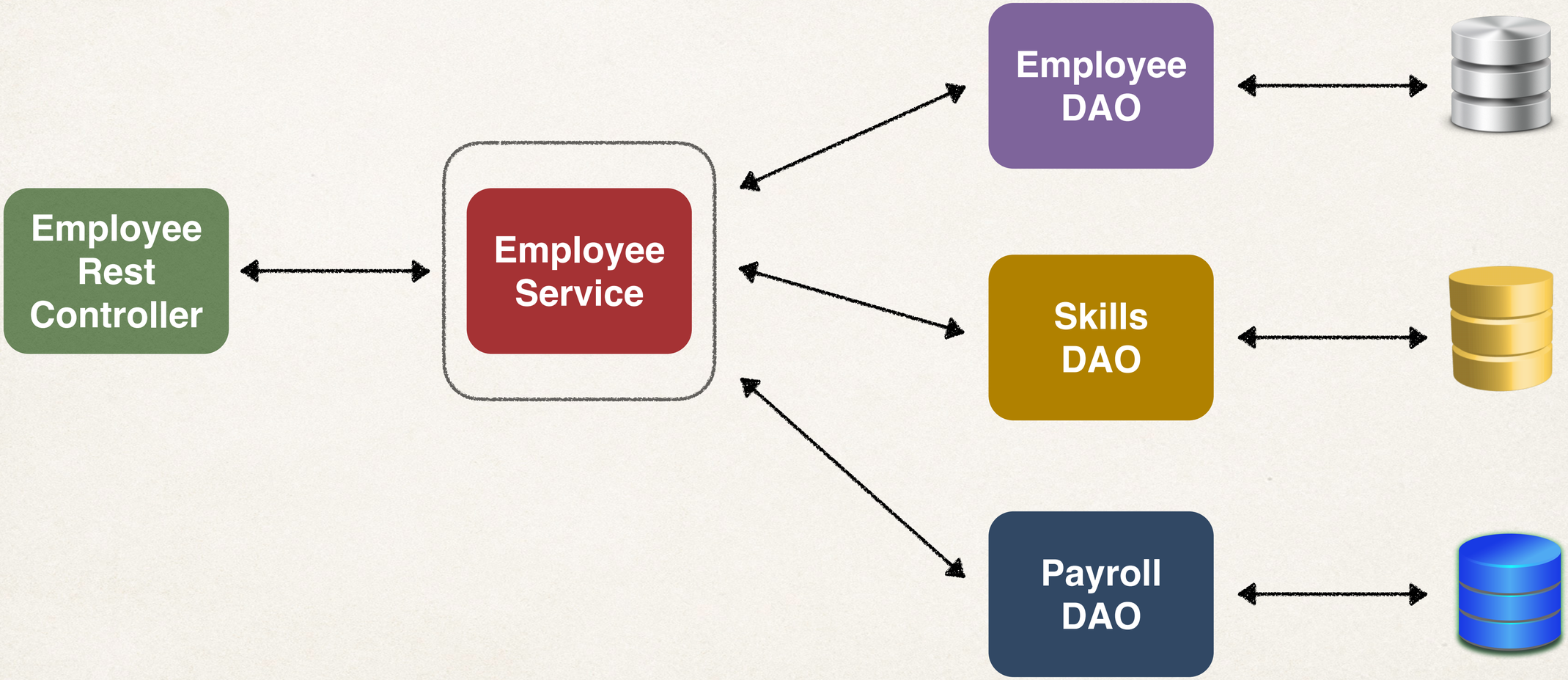
Purpose of Service Layer

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



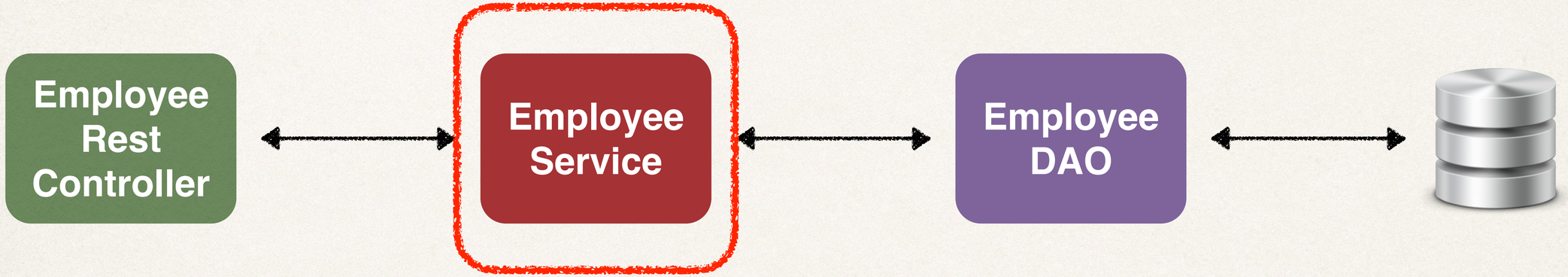
© luv2code LLC

Integrate Multiple Data Sources



© luv2code LLC

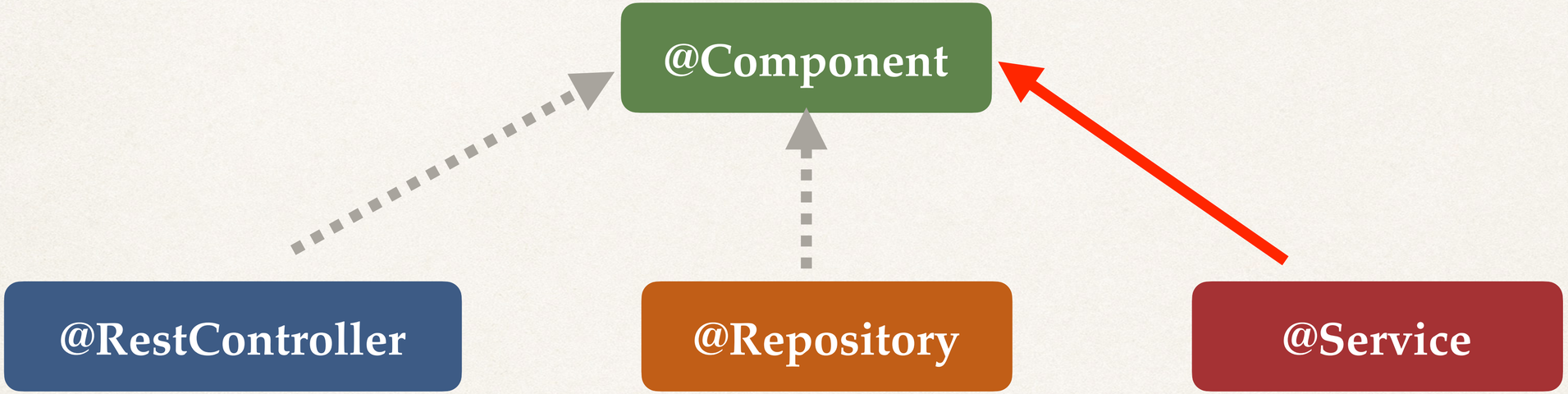
Most Times - Delegate Calls



© 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

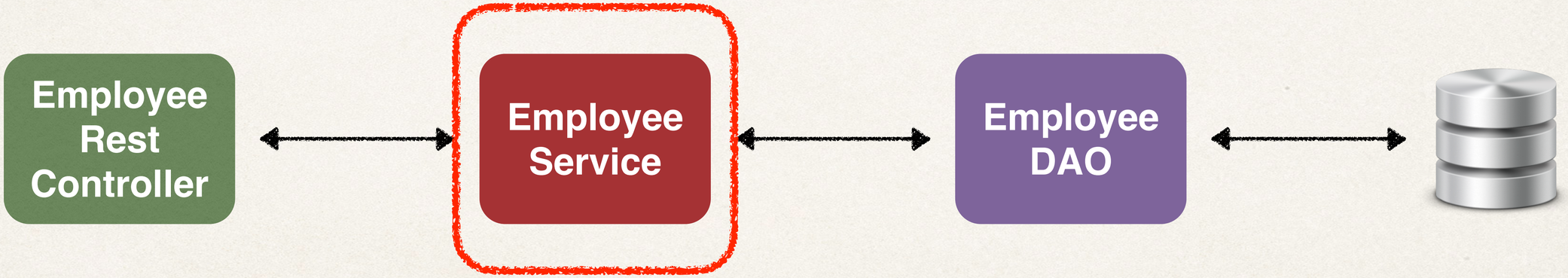
Employee Service

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();
  }
}
```

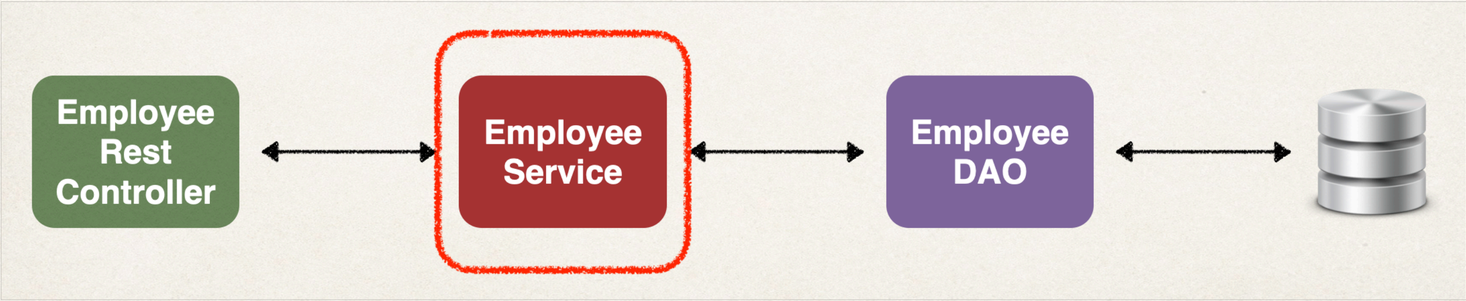

© luv2code LLC

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



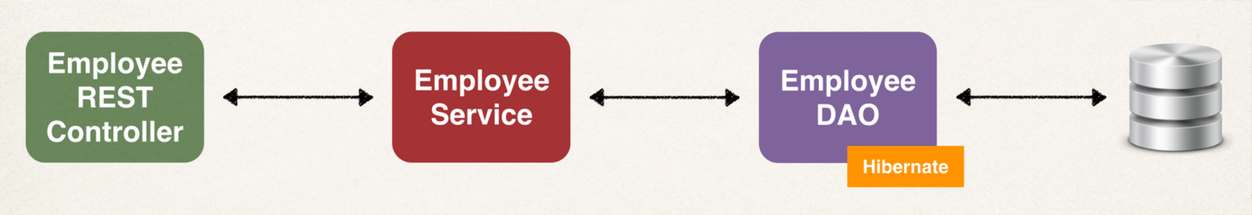
© luv2code LLC

Development Process

Step-By-Step

1. Create Spring Boot project using Spring Initializer
2. Set up Database Dev Environment
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

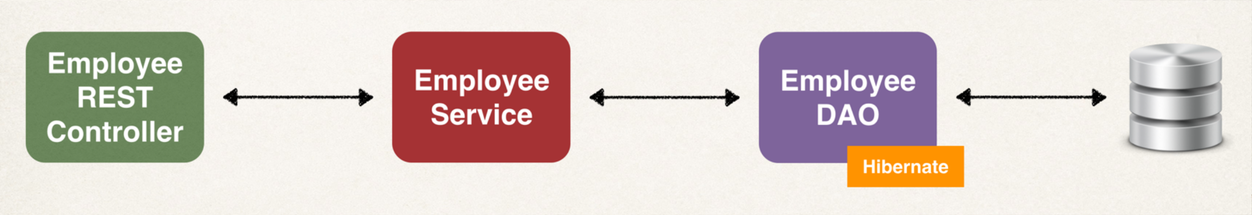
© luv2code LLC

Development Process

Step-By-Step

1. Create Spring Boot project using Spring Initializer
2. Set up Database Dev Environment
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

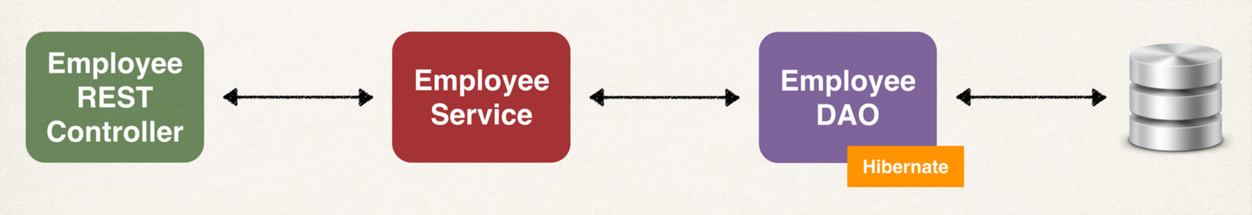
© luv2code LLC

Development Process

Step-By-Step

1. Create Spring Boot project using Spring Initializer
2. Set up Database Dev Environment
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

Create EmployeeRequest Entity

- Request that comes from client for validation

```
public class EmployeeRequest {

    @NotBlank
    @Size(min=2, max=50)
    private String firstName;

    @NotBlank
    @Size(min=2, max=50)
    private String lastName;

    @NotBlank
    @Email
    private String email;

< Constructor, Getters & Setters >
```

We will start with no data validation and then add in future lecture.

© luv2code LLC

EmployeeRequest Conversion

- Convert EmployeeRequest to Employee in Service Layer.

```
public Employee save(EmployeeRequest employeeRequest) {

    Employee employee = convertToEmployee(0, employeeRequest);
    return employeeRepository.save(employee);
}
```

- convertToEmployee logic:

```
public Employee convertToEmployee(long id, EmployeeRequest employeeRequest) {

    return new Employee(id, employeeRequest.getFirstName(),
    employeeRequest.getLastName …);
}
```


© luv2code LLC

Swagger Implementation

© luv2code LLC

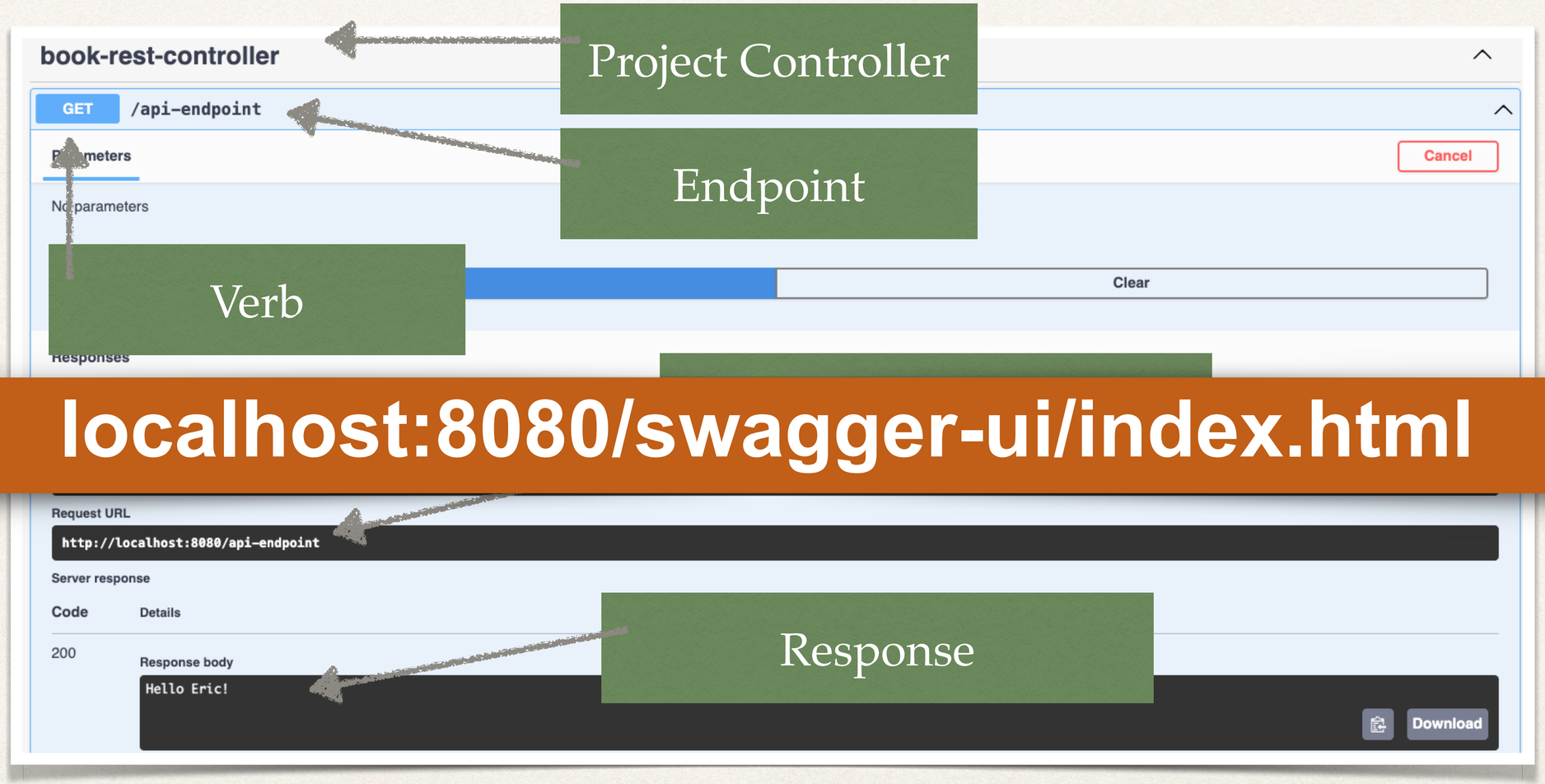
What is Swagger?

- Swagger is a set of tools and specifications for designing, building, documenting and consuming RESTful web services.
- Open-source project for streamlining API development by providing a common framework and API descriptions.
- Swagger UI implementation will allow us to be able to document and call our API endpoints right from our application.
- No third party tools will be needed!
- Win-win solution for implement Swagger within our application.

© luv2code LLC

What is Swagger?

- Instead of going straight to the API endpoint, we can go to Swaggers.



localhost:8080/swagger-ui/index.html

© luv2code LLC

What is Swagger?

- Path is hard to remember:
  - localhost:8080/swagger-ui/index.html
- We can fix this to redirect based off of /docs.
  - New endpoint for Swagger: localhost:8080/docs

Too long to remember

Much better :-)

File: application.properties

```
springdoc.swagger-ui.path=/docs
```


© luv2code LLC

Create EmployeeRequest Entity

- Request that comes from client for validation

```
public class EmployeeRequest {

    @NotBlank
    @Size(min=2, max=50)
    private String firstName;

    @NotBlank
    @Size(min=2, max=50)
    private String lastName;

    @NotBlank
    @Email
    private String email;

< Constructor, Getters & Setters >
```

Add data validation to our EmployeeRequest object.

© luv2code LLC

Rest Controller Methods - Find

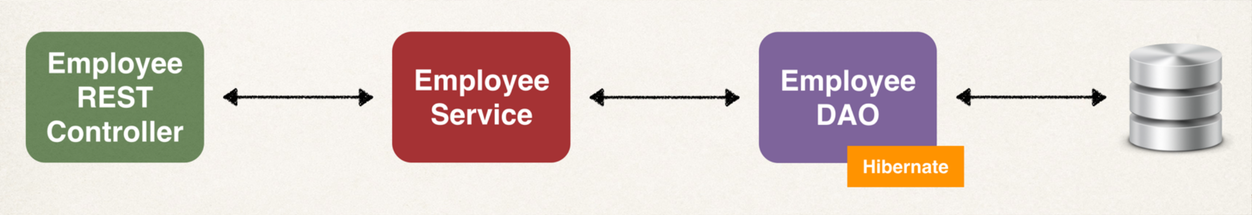
© luv2code LLC

Development Process

Step-By-Step

1. Create Spring Boot project using Spring Initializer
2. Set up Database Dev Environment
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

Real-Time Project

Checkpoint

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

© 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

Rest Controller Methods - Create

© luv2code LLC

Real-Time Project

Checkpoint

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

© luv2code LLC

EmployeeRequest Validation

- Add @Valid to create the validation on the RequestBody.

```
public Employee addEmployee(@Valid @RequestBody EmployeeRequest theEmployee) {

    Employee dbEmployee = employeeService.save(theEmployee)
    return dbEmployee;
}
```

- Convert EmployeeRequest into Employee in Service layer.

```
@ResponseStatus(HttpStatus.CREATED)
@PostMapping()
public Employee addEmployee(@Valid @RequestBody EmployeeRequest theEmployee)
{}
```


© luv2code LLC

Create a New Employee

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

REST Client

POST /api/employees

EmployeeRequest

```
{
    "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

Employee

© 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
- Swagger handles this for us!

© luv2code LLC

Rest Controller Methods - Update and Delete

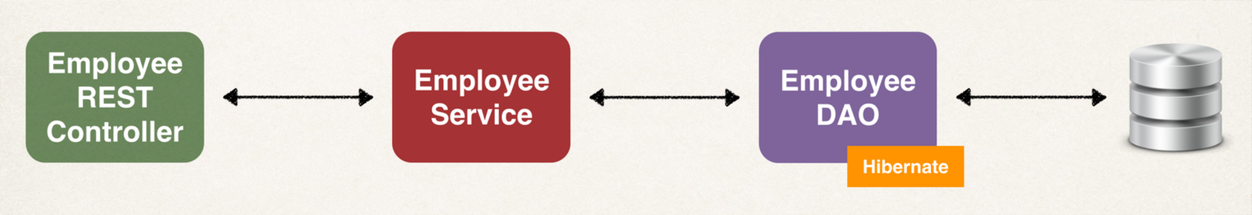
© luv2code LLC

Development Process

Step-By-Step

1. Create Spring Boot project using Spring Initializer
2. Set up Database Dev Environment
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

Real-Time Project

Checkpoint

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

© luv2code LLC

Update Employee

REST Client

PUT /api/employees/{employeeId}

ID of employee to update
With updated info

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

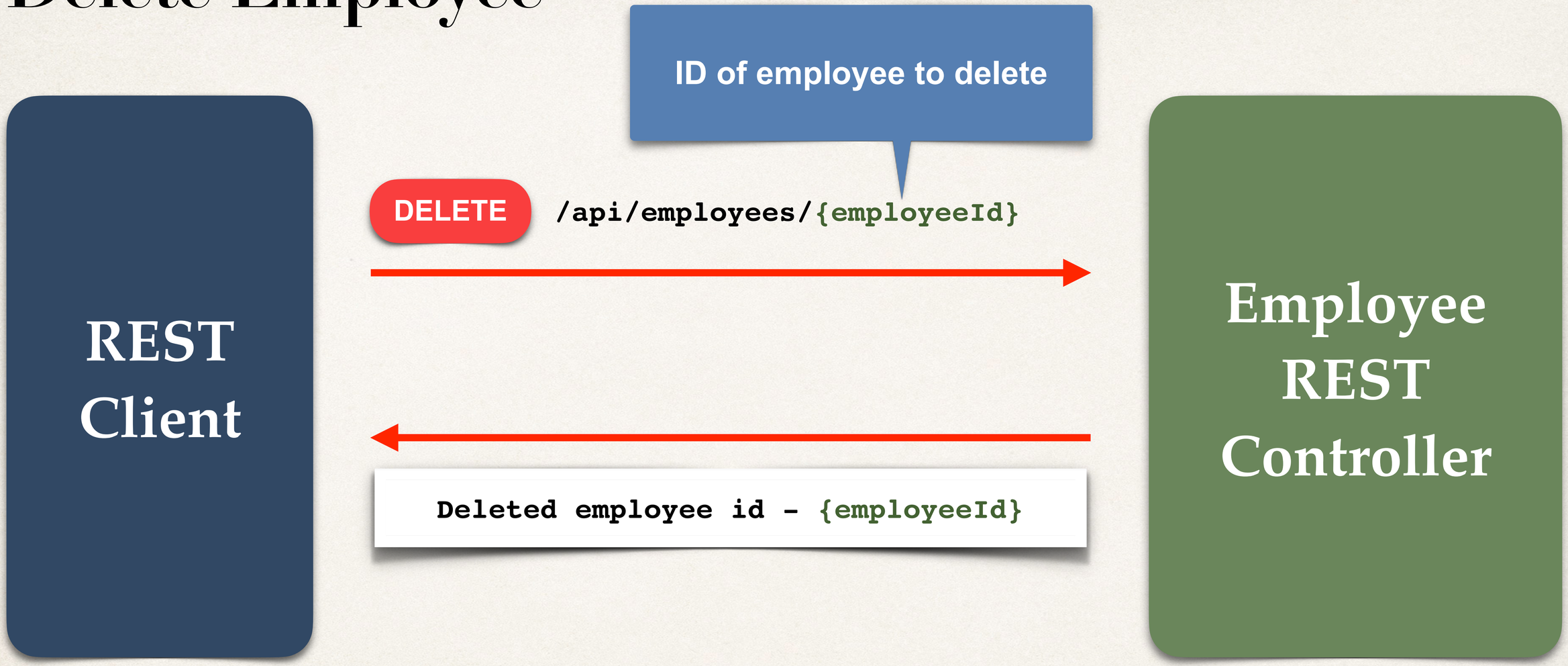
Employee REST Controller

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

Response contains updated info (echoed)

© luv2code LLC

Delete Employee

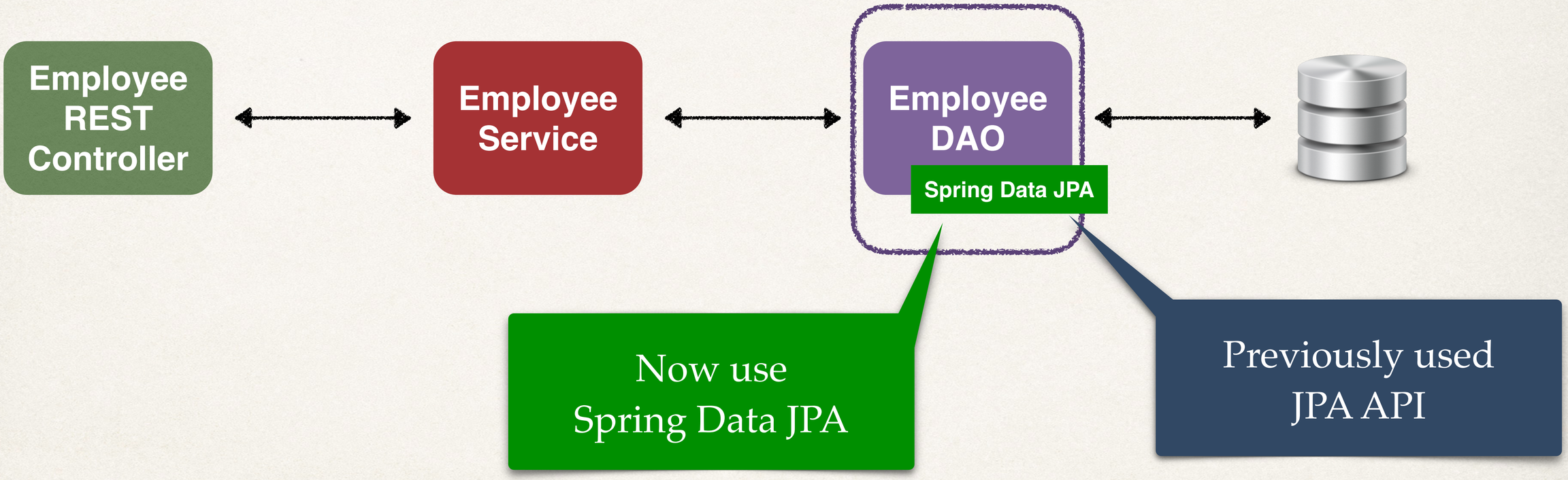


© luv2code LLC

Spring Data JPA in Spring Boot

© luv2code LLC

Application Architecture



© luv2code LLC

The Problem

- We saw how to create a DAO for **`Employee`**

```java
public interface EmployeeDAO {

    public List<Employee> findAll();

    public Employee findById(int theId);

    public void save(Employee theEmployee);

    public void deleteById(int theId);

}
```

- What if we need to create a DAO for another entity?
  - **`Customer`**, **`Student`**, **`Product`**, **`Book`** …

```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;
    }
```

- Do we have to repeat all of the same code again???

© 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:

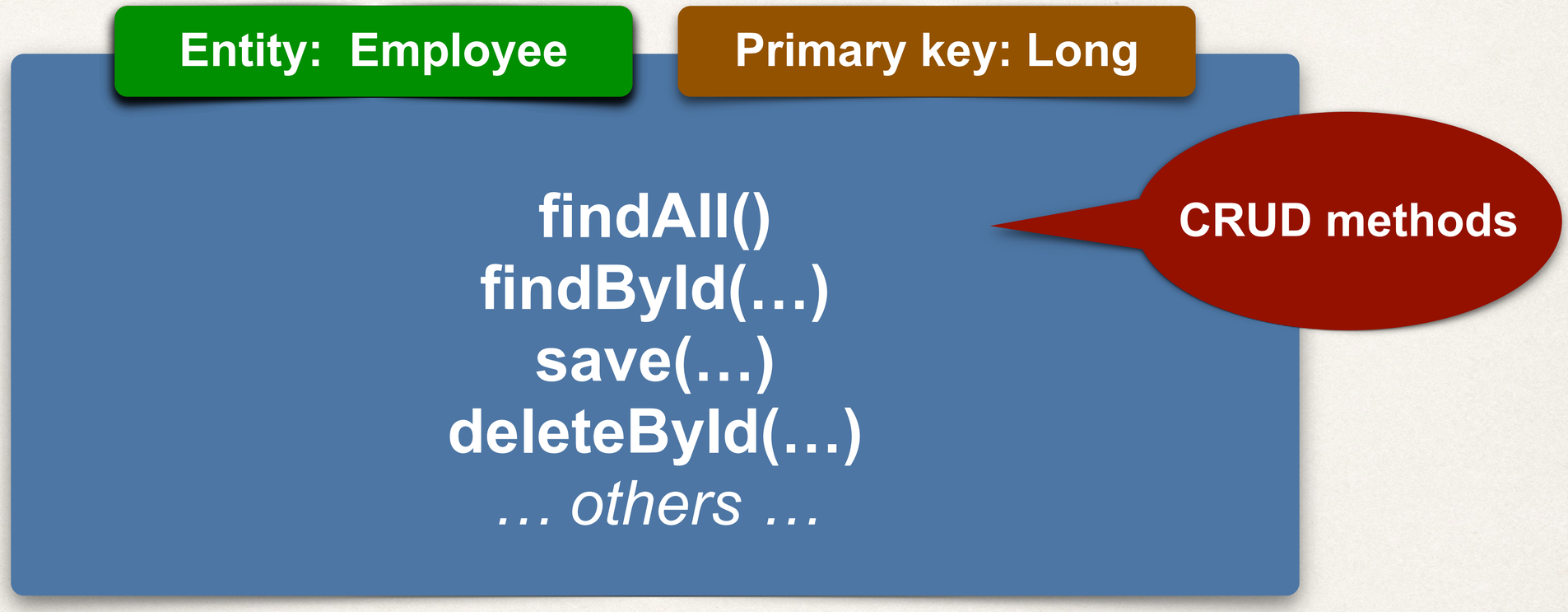
Create a DAO for me

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

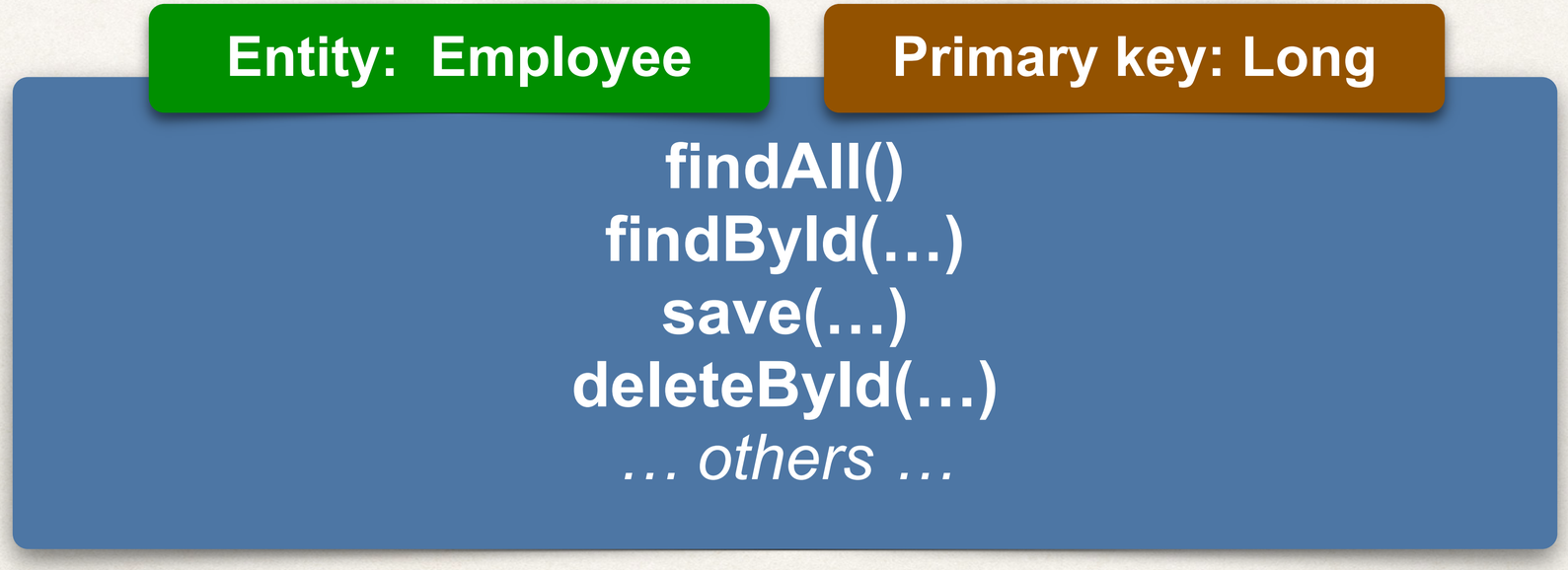
- Spring will give you a CRUD implementation for FREE …. like MAGIC!!
  - Helps to minimize boiler-plate DAO code … yaaay!!!

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, Long> {

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

}
```

No need for implementation class

Get these methods for free

Entity: Employee

Primary key: Long

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

© luv2code LLC

JpaRepository Docs

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

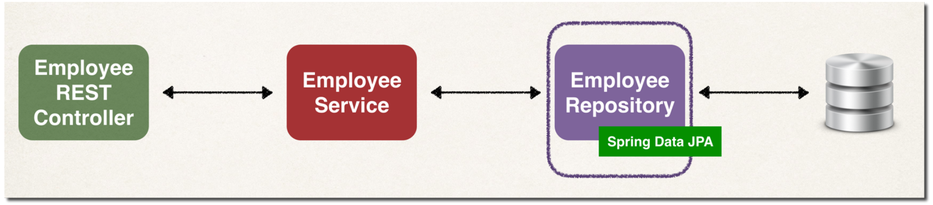
www.luv2code.com/jpa-repository-javadoc

© 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          yManager) {
        entityManager = the
    }

    @Override
    public List<Emplo

        // create
        TypedQue                   loyee.class);

        //          suil
        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 Boot REST API Security Overview

You will learn how to …

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

Practical Results

- Cover the most common Spring Security tasks that you will need on daily projects
- Not an A to Z reference … for that you can see **Spring Security Reference Manual**

http://www.luv2code.com/spring-security-reference-manual

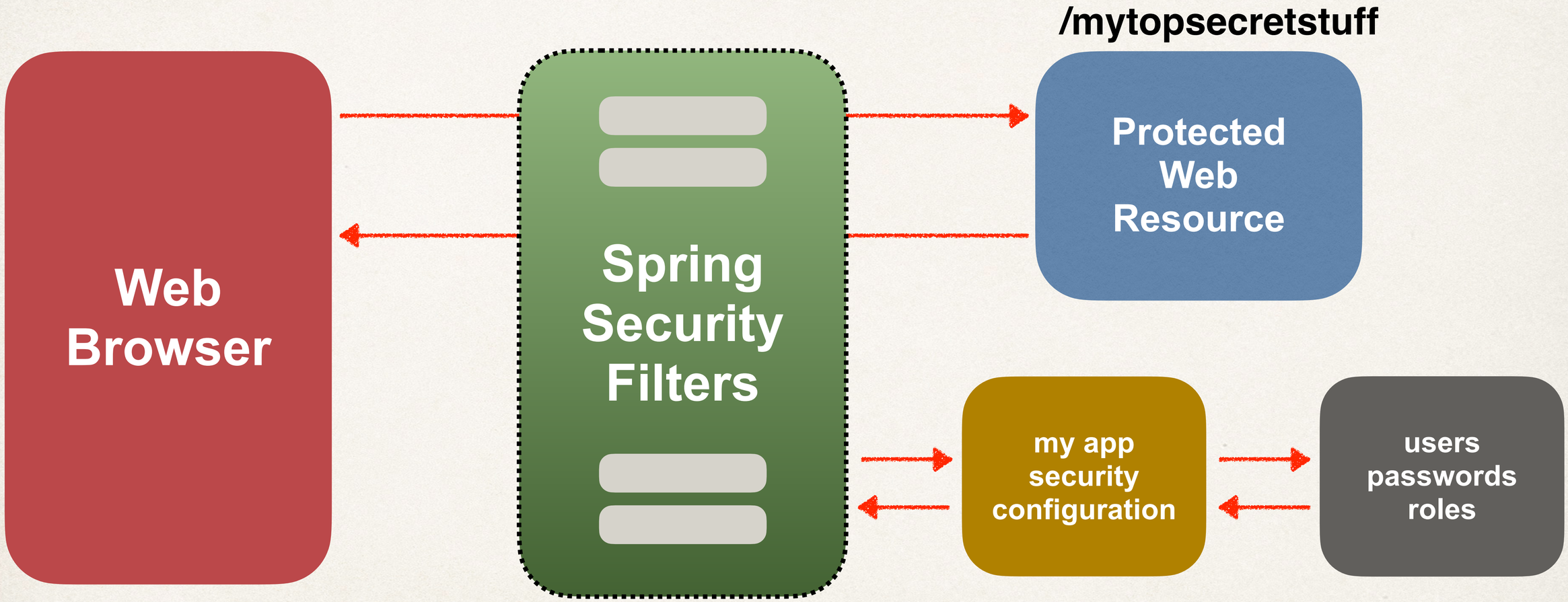
Spring Security Model

- Spring Security defines a framework for security
- Implemented using Servlet filters in the background
- Two methods of securing an app: declarative and programmatic

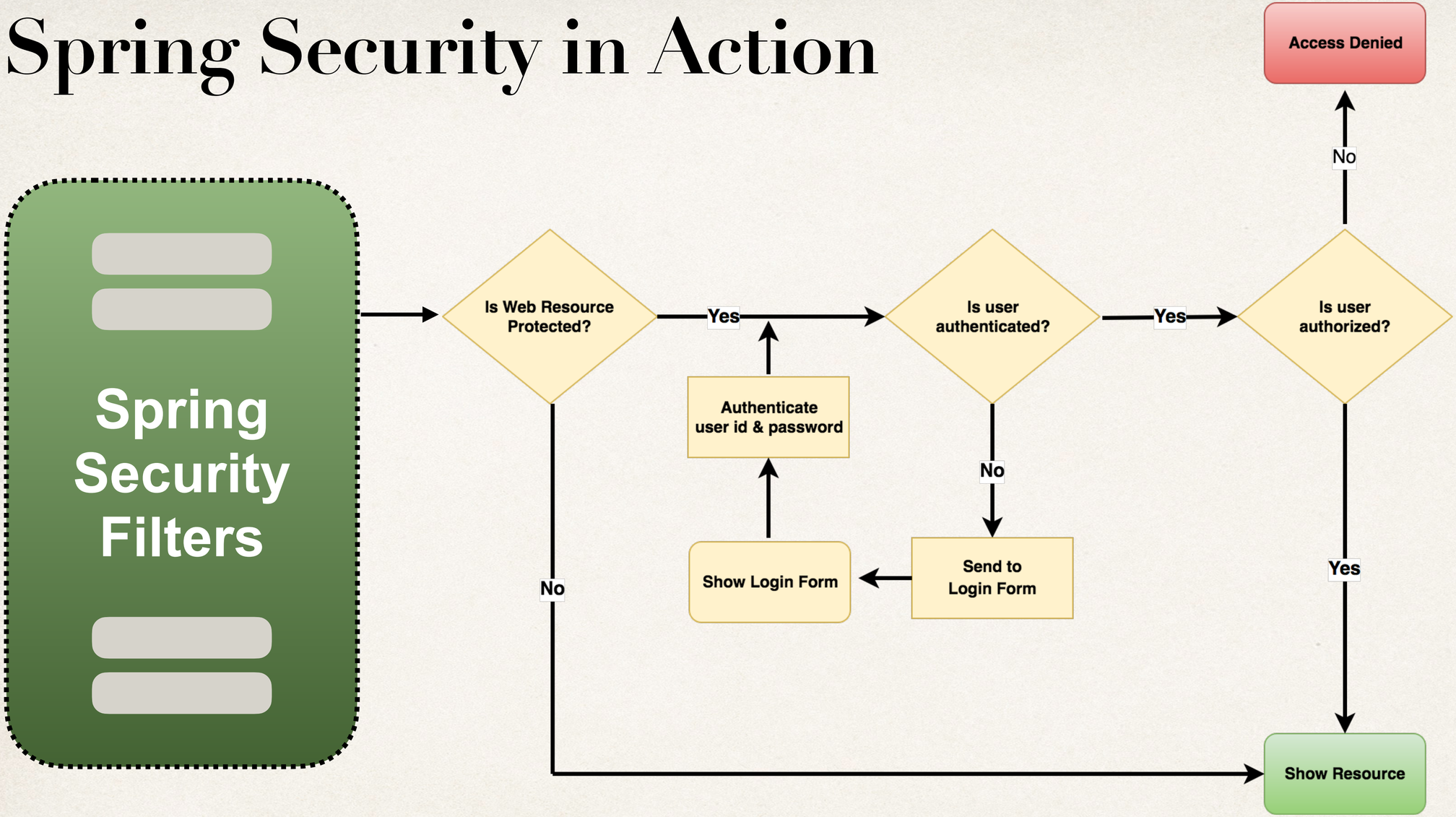
Spring Security with Servlet Filters

- 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

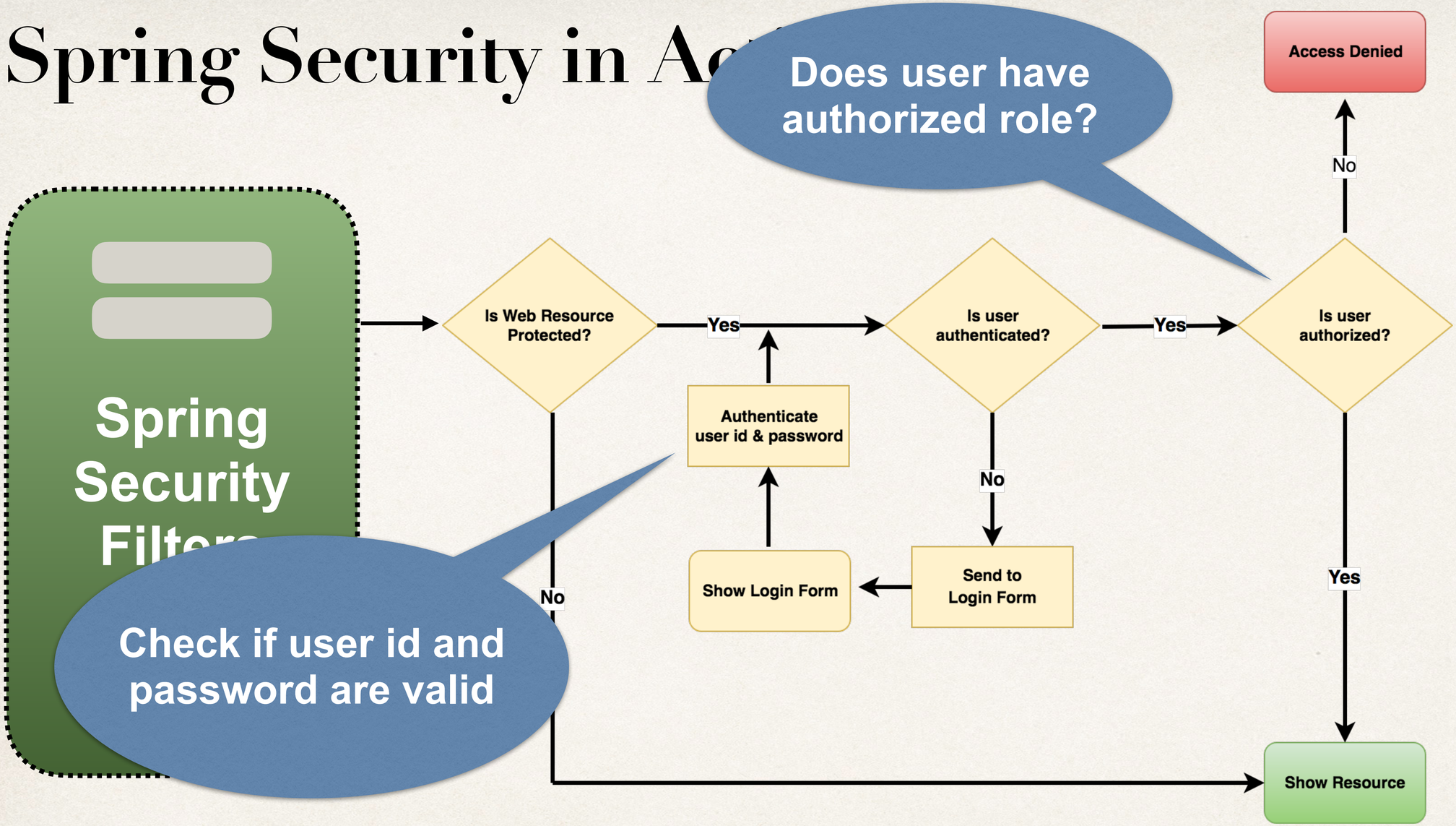
Spring Security Overview



Spring Security in Action



Spring Security in Ac



Security Concepts

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

Declarative Security

- Define application's security constraints in configuration
  - All Java config: @Configuration
- Provides separation of concerns between application code and security

Programmatic Security

- Spring Security provides an API for custom application coding
- Provides greater customization for specific app requirements

Enabling Spring Security

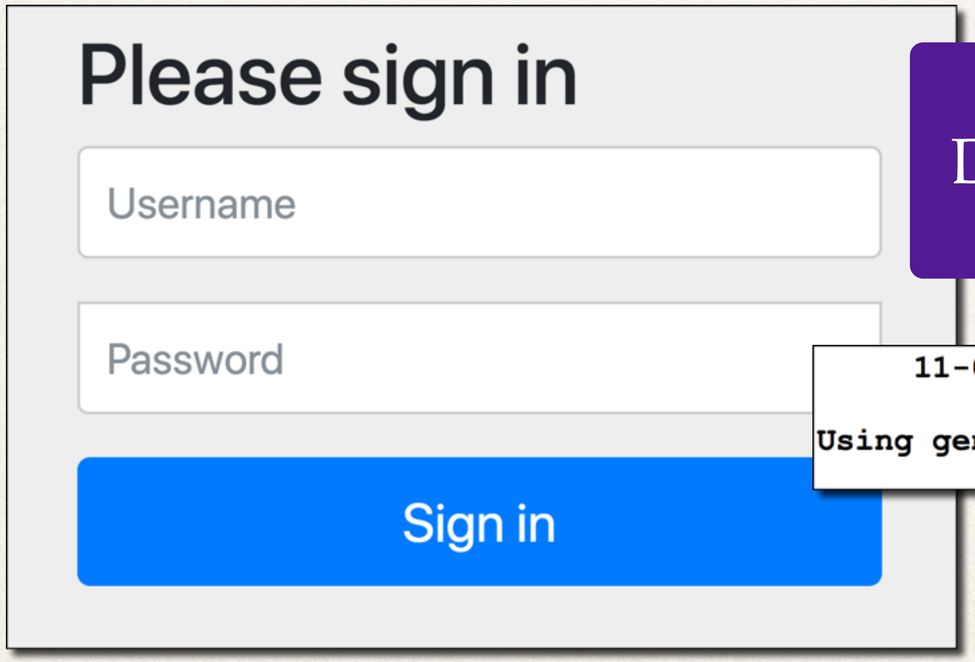
1. Edit **`pom.xml`** and add **`spring-boot-starter-security`**

```
<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



Default user name: **`user`**

Check console logs for password

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

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=test123
```

Authentication and Authorization

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

users
passwords
roles

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

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

Step-By-Step

1. Create Spring Security Configuration (@Configuration)

2. Add users, passwords and roles

Step 1: Create Spring Security Configuration

File: 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

File: DemoSecurityConfig.java

```java
@Configuration
public class DemoSecurityConfig {

    @Bean
    public InMemoryUserDetailsManager userDetailsManager() {

        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 in later videos

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

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

The ** syntax:
match on all sub-paths

Authorize Requests for MANAGER role

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

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

Authorize Requests for ADMIN role

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

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

Pull It Together

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

```
@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

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

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

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

Add SwaggerConfig

Pain Point for Basic Authenticaion

- No easy way to handle basic authentication
- Browser stores data so logging off is difficult for testing.
- The browser basic authentication is not part of Swagger.
- Make Basic Auth part of Swagger for calling and testing our endpoints.

Disable Basic Authentication

```
http.httpBasic(httpBasicCustomizer -> httpBasicCustomizer.disable());
```

Method configuring basic authentication

Disables the basic authetication

Return an Error when Auth Fails

```
@Bean
public AuthenticationEntryPoint authenticationEntryPoint() {

    // Sets the HTTP Status Code
    response.setStatus(HttpStatus.UNAUTHORIZED.value());

    // Sets the response type to JSON
    response.setContentType("application/json");

    // Prevents browser authentication popup
    response.setHeader("WWW-Authenticate", "");

    // Writes the error message
    response.getWriter().write("{\"error\": \"Unauthorized access\"}");
}
```

```
http.httpBasic(httpBasicCustomizer -> httpBasicCustomizer.disable());

http.csrf(csrf -> csrf.disable());

http.exceptionHandling(exceptionHandling -> exceptionHandling
         .authenticationEntryPoint(authenticationEntryPoints()));
```

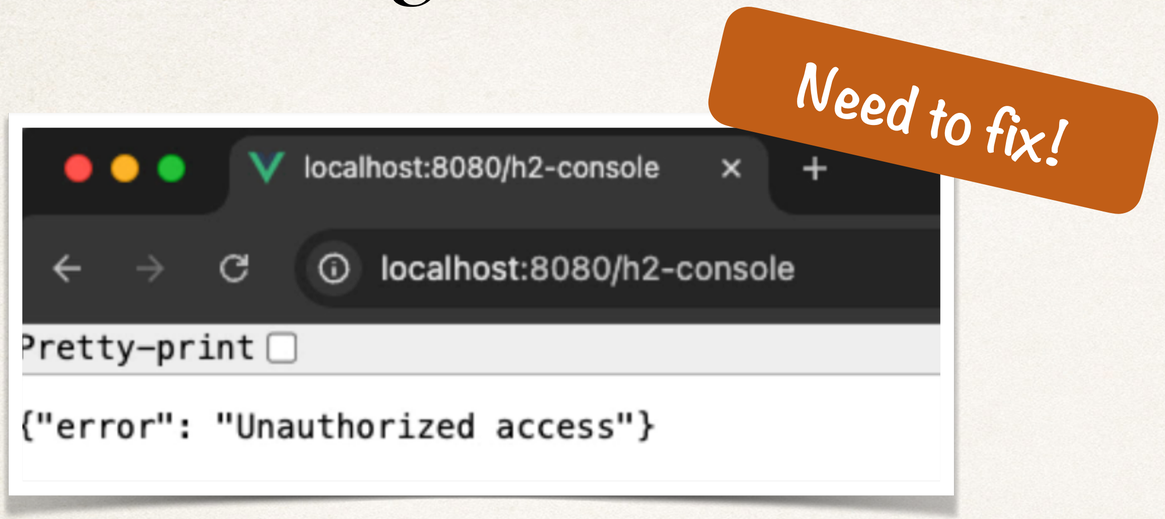
SwaggerConfig File

```
@Configuration
@OpenAPIDefinition(info = @Info(title="My API", version="v1"),
   security = @SecurityRequirement(name="basicAuth"))
@SecurityScheme(name="basicAuth", type= SecuritySchemeType.HTTP, scheme =
"basic")
public class SwaggerConfig {
 // This is all you need
}
```

- After this basic auth will be configured for Swagger.
- And disabled for the browser.

H2 Config Security

H2 Config issue



- After adding our requestMatchers we broke our H2 path!
- Luckily, it is a simple fix :-)

Pull It Together

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

   http.authorizeHttpRequests(configurer ->
                 configurer
                        .requestMatchers(HttpMethod.GET, “/h2-console/**“).permitAll()
                        .requestMatchers(HttpMethod.POST, “/h2-console/**“).permitAll()
                        .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"));

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

   http.exceptionHandling(exceptionHandling -> exceptionHandling
            .authenticationEntryPoint(authenticationEntryPoints()));

   http.headers(headers -> headers.frameOptions(frameOptions -> frameOptions.disable()));

   return http.build();
}
```

H2 console uses frames

Spring Security
User Accounts Stored in H2 Database

Database Access

- So far, our user accounts were hard coded in Java source code

- We want to add database access

Advanced

Recall Our User Roles

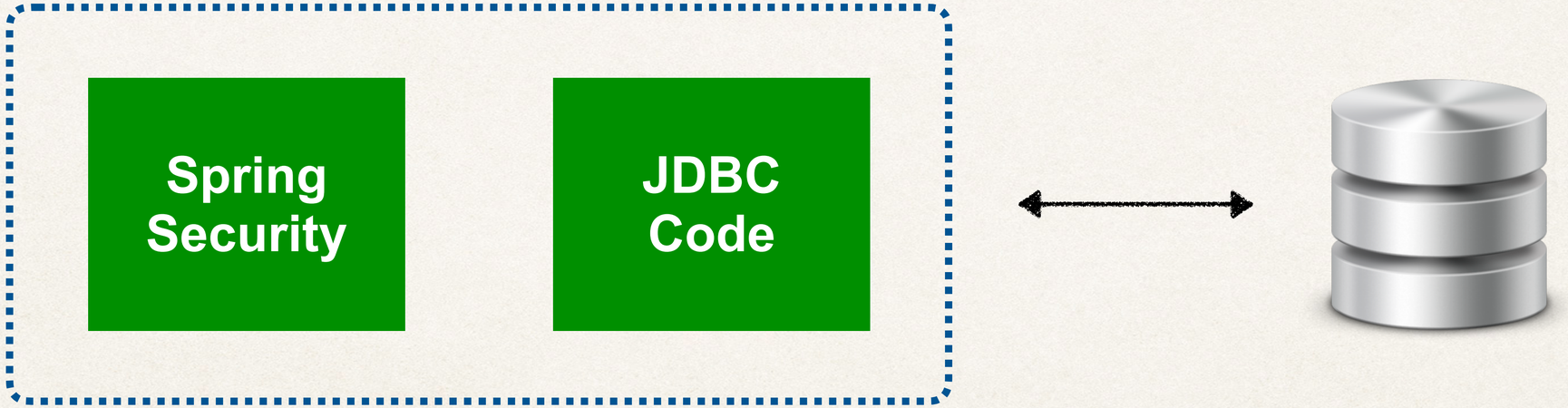
User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

Database Support in Spring Security

Out-of-the-box

- Spring Security can read user account info from database

- By default, you have to follow Spring Security's predefined table schemas



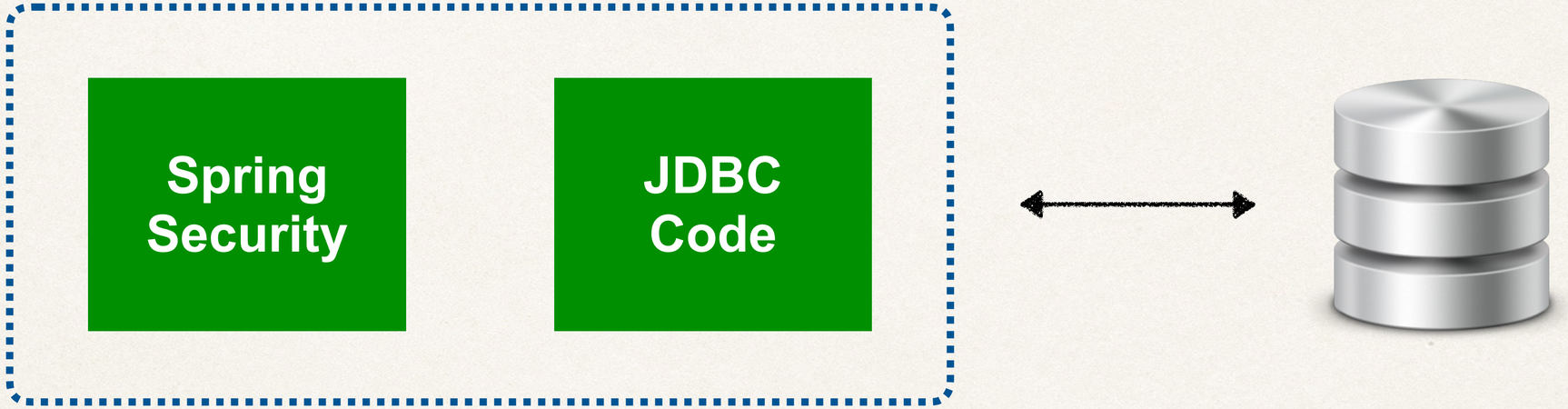
Customize Database Access with Spring Security

- Can also customize the table schemas
- Useful if you have custom tables specific to your project / custom
- You will be responsible for developing the code to access the data
  - JDBC, JPA/Hibernate etc …

Database Support in Spring Security

Out-of-the-box

- Follow Spring Security's predefined table schemas



Development Process

Step-By-Step

1. Develop SQL Script to set up database tables

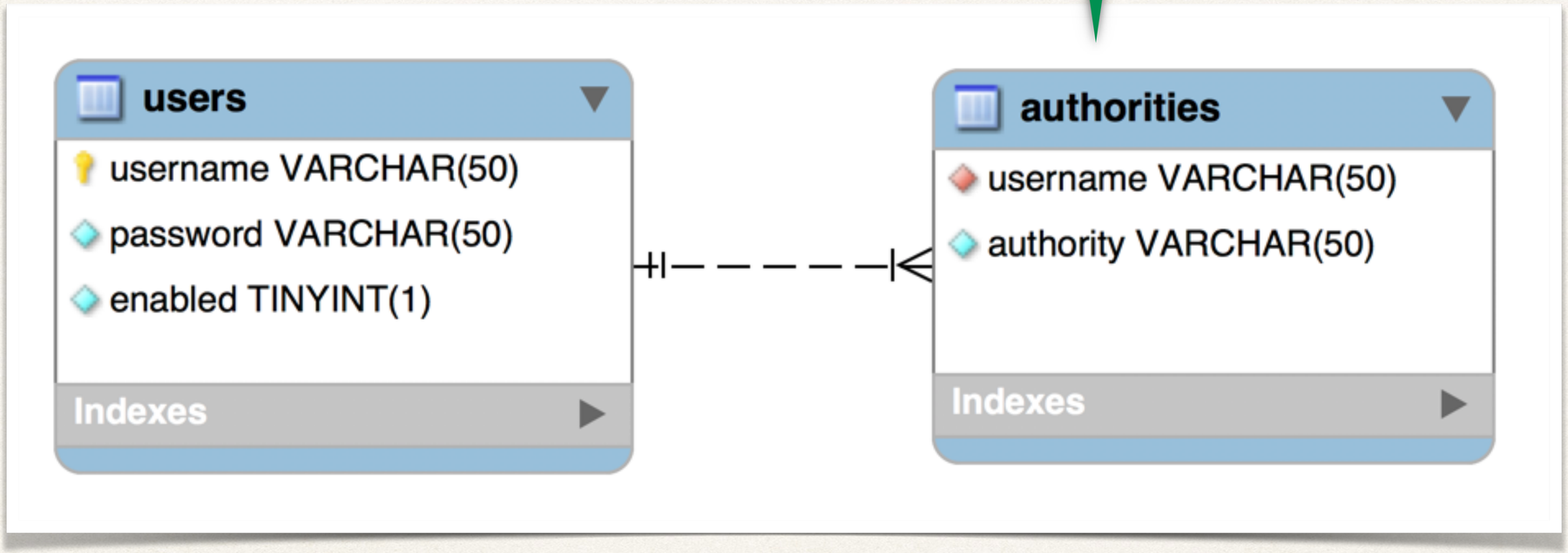
2. Add database support to Maven POM file

3. Create JDBC properties file

4. Update Spring Security Configuration to use JDBC

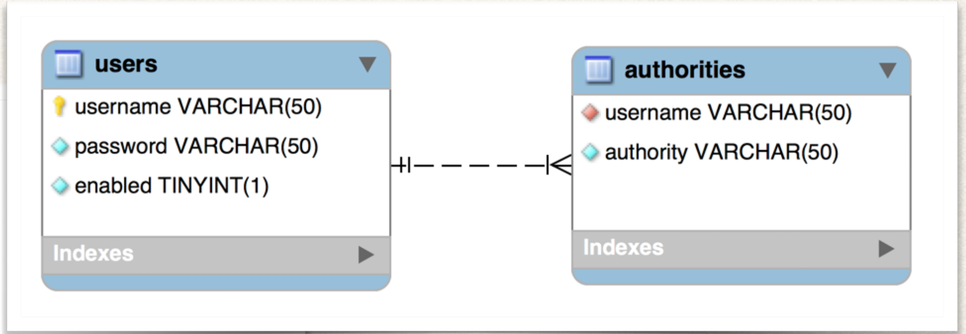
Default Spring Security Database Schema

"authorities" same as "roles"



Step 1: Develop SQL Script to setup database tables

```sql
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

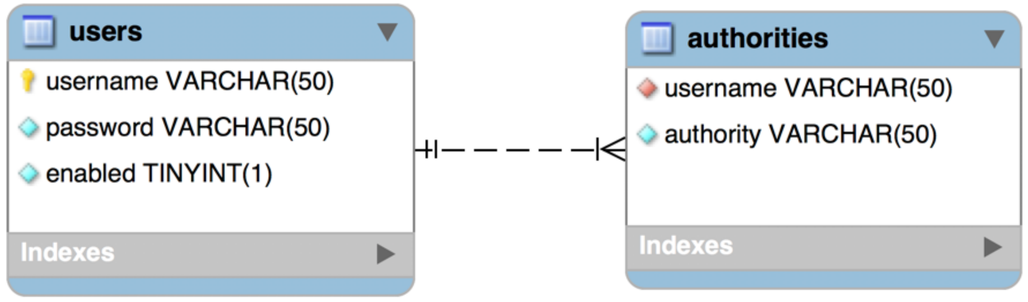
The encoding algorithm id

The password

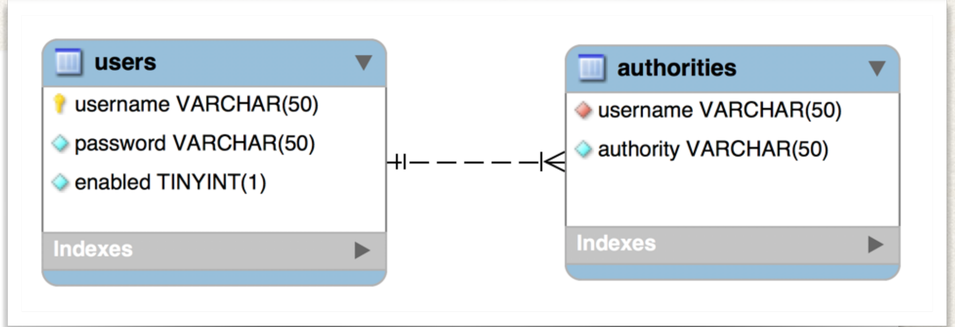
```
INSERT INTO `users`
VALUES
('john','{noop}test123',1),
('mary','{noop}test123',1),
('susan','{noop}test123',1);
```

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

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



Step 1: Develop SQL Script to setup database tables




```sql
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;
```

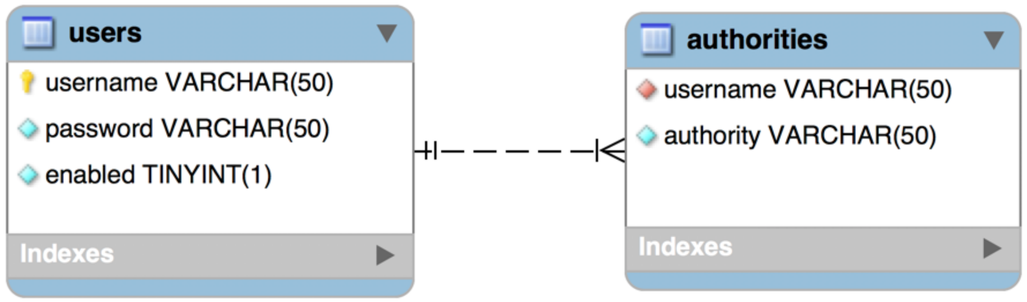
Step 1: Develop SQL Script to setup database tables

"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: Update Spring Security to use JDBC

```
@Configuration
public class SecurityConfig {

    @Bean
    public UserDetailsManager userDetailsManager(DataSource dataSource) {

        return new JdbcUserDetailsManager(dataSource);
    }

    …
}
```

Inject data source
Auto-configured by Spring Boot

No longer
hard-coding users :-)

Tell Spring Security to use
JDBC authentication
with our data source

Spring Security
Password Encryption

© luv2code LLC

Password Storage

- So far, our user passwords are stored in plaintext … yikes!

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

- Ok for getting started … but not for production / real-time project :-(

© luv2code LLC

Password Storage - Best Practice

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

© luv2code LLC

Spring Security Team 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

© luv2code LLC

Bcrypt Additional Information

- Why you should use bcrypt to hash passwords

  www.luv2code.com/why-bcrypt

- Detailed bcrypt algorithm analysis

  www.luv2code.com/bcrypt-wiki-page

- Password hashing - Best Practices

  www.luv2code.com/password-hashing-best-practices

© luv2code LLC

How to Get a Bcrypt password

You have a plaintext password and you want to encrypt using bcrypt

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

© luv2code LLC

How to Get a Bcrypt password - Website

- Visit: **`www.luv2code.com/generate-bcrypt-password`**
- Enter your plaintext password
- The website will generate a bcrypt password for you

© luv2code LLC

DEMO

© luv2code LLC

Development Process

Step-By-Step

1. Run SQL Script that contains encrypted passwords

- Modify DDL for password field, length should be 68

THAT'S IT … no need to change Java source code :-)

© luv2code LLC

Spring Security Password Storage

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

```
{bcrypt}encodedPassword
```

Password column must be at least 68 chars wide

{bcrypt} - 8 chars
***encodedPassword* - 60 chars**

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

© luv2code LLC

Modify DDL 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**

© luv2code LLC

Step 1: Develop SQL Script to setup database tables

The encoding algorithm id

The encrypted password: fun123

```
INSERT INTO `users`
VALUES
('john','{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q',1),

...
```

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

© luv2code LLC

Step 1: Develop SQL Script to setup database tables

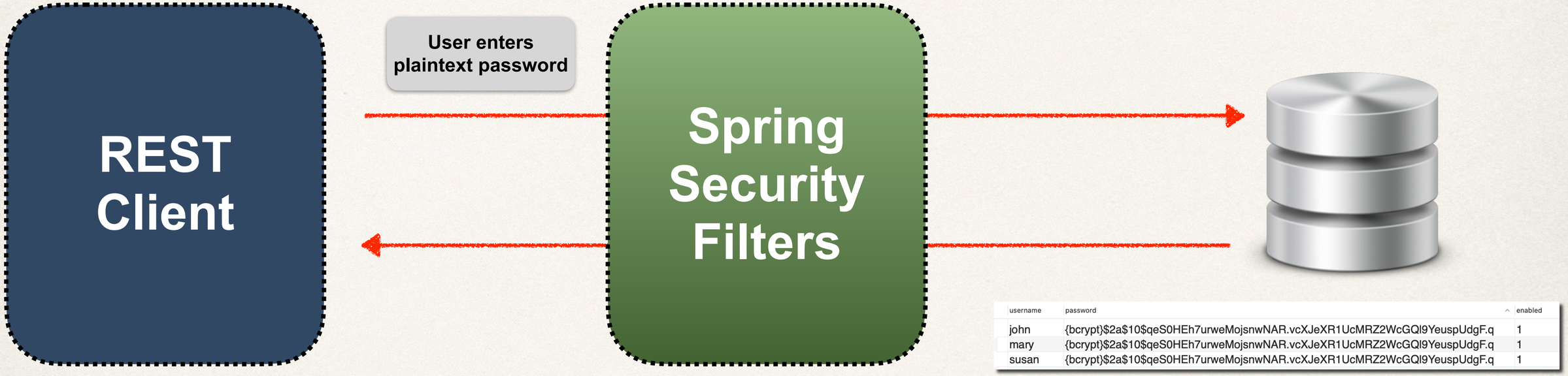
The encoding algorithm id

The encrypted password: fun123

```
INSERT INTO `users`
VALUES
(`john','{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q',1),

('mary','{bcrypt}$2a$04$eFytJDGtjbThXa80FyOOBuFdK2IwjyWefYkMpiBEFlpBwDH.5PM0K',1),

('susan','{bcrypt}$2a$04$eFytJDGtjbThXa80FyOOBuFdK2IwjyWefYkMpiBEFlpBwDH.5PM0K',1);
```

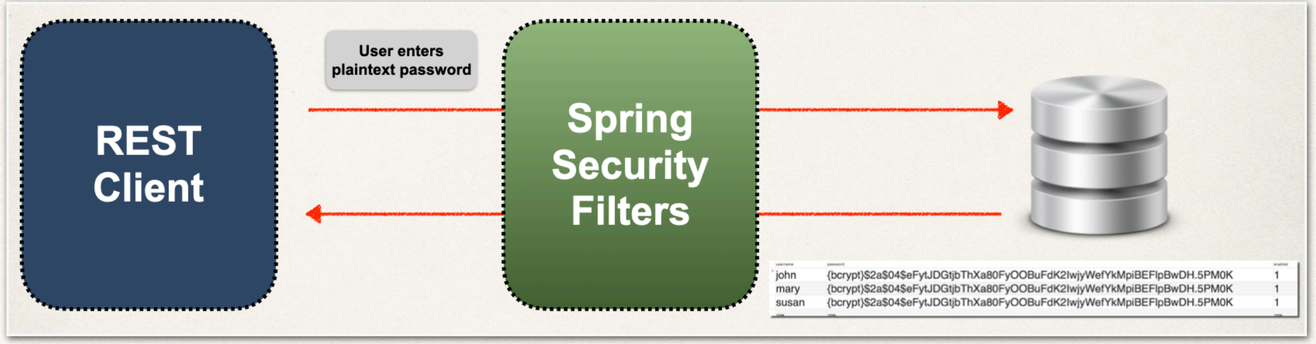

© luv2code LLC

Spring Security Login Process



© luv2code LLC

Spring Security Login Process



1. Retrieve password from db for the user
2. Read the encoding algorithm id (bcrypt etc)
3. For case of bcrypt, encrypt plaintext password from login form (using salt from db password)
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

Note:
The password from db is NEVER decrypted

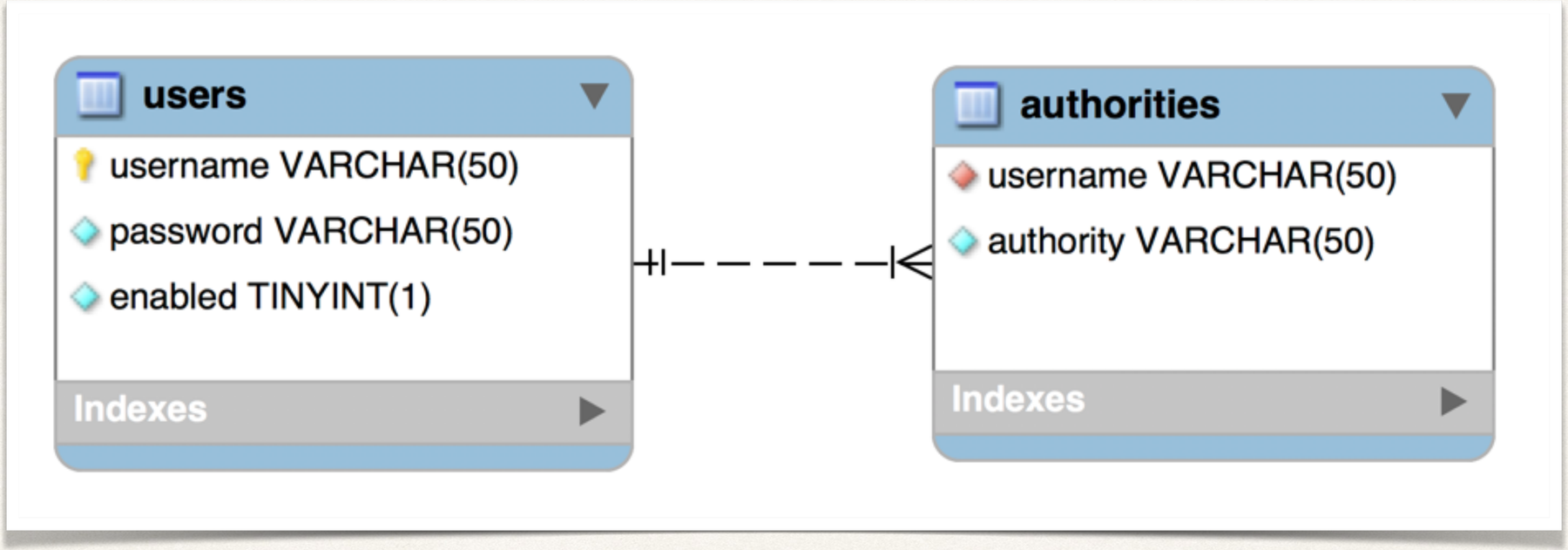
Because bcrypt is a one-way encryption algorithm

© luv2code LLC

Spring Security
Custom Tables

© luv2code LLC

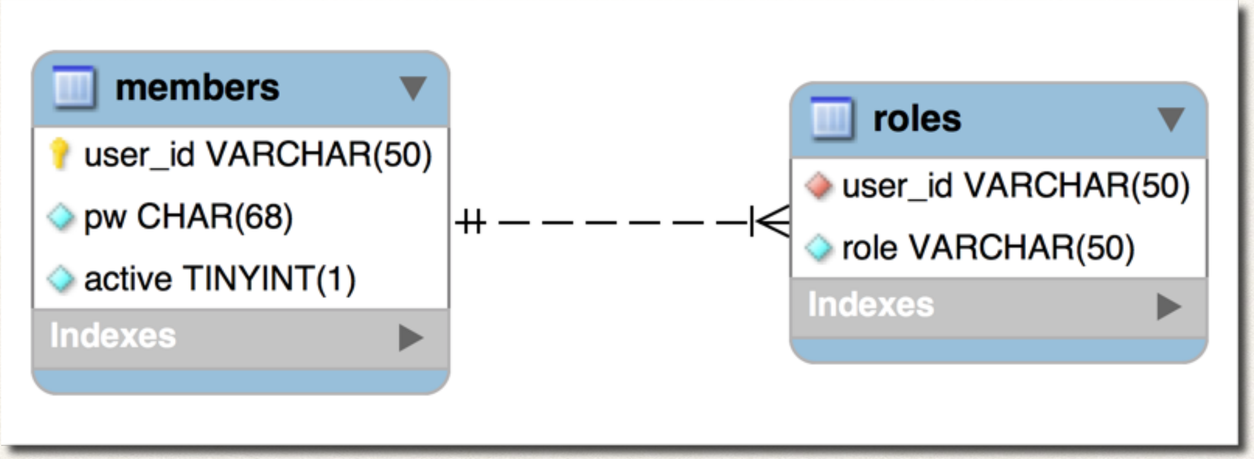
Default Spring Security Database Schema



© luv2code LLC

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

© luv2code LLC

For Security Schema Customization

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

© luv2code LLC

Development Process

Step-By-Step

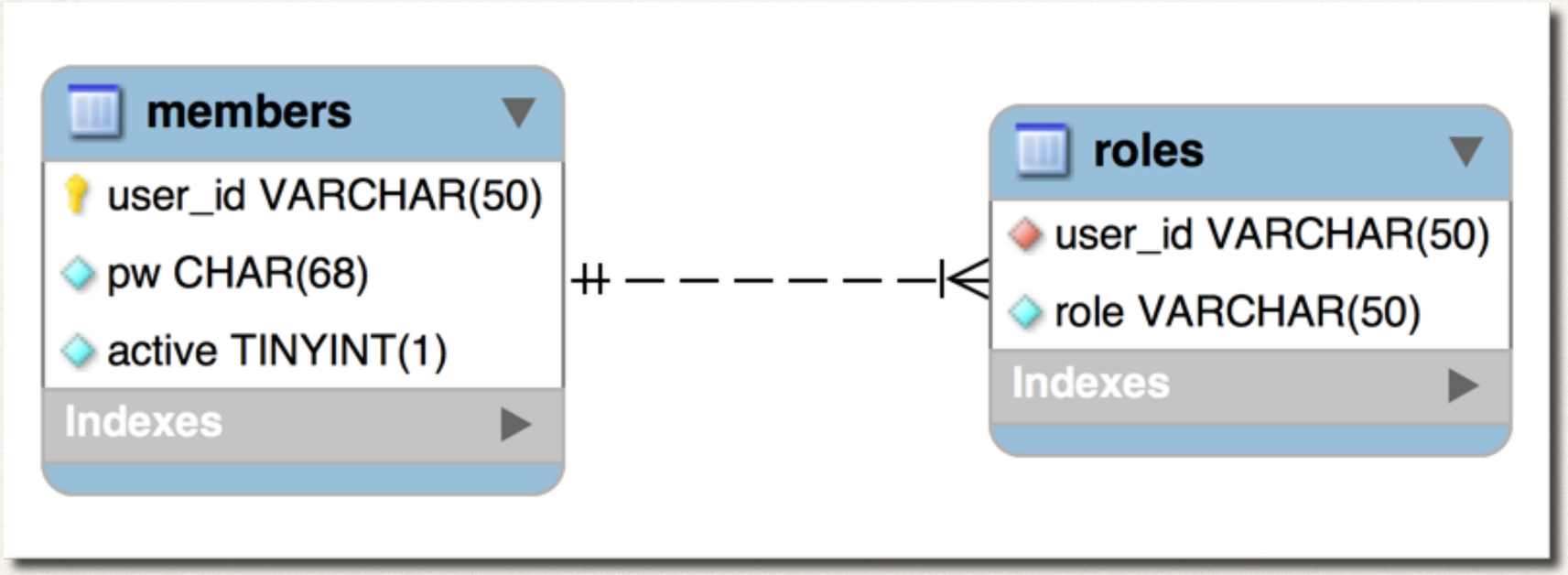
1. Create our custom tables with SQL

2. Update Spring Security Configuration

- Provide query to find user by user name
- Provide query to find authorities / roles by user name

© luv2code LLC

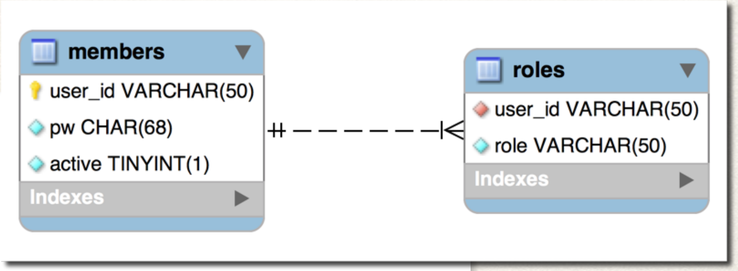
Step 1: Create our custom tables with SQL



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

© luv2code LLC

Step 2: Update Spring Security Configuration

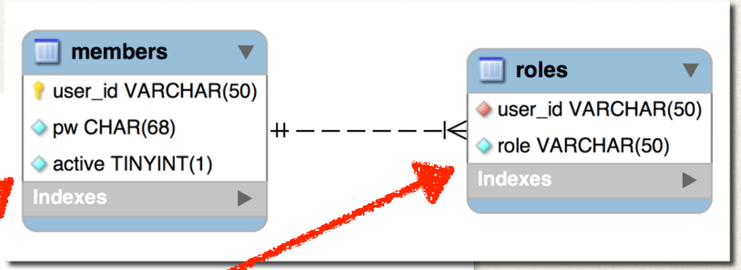



```
@Configuration
public class DemoSecurityConfig {

    @Bean
    public UserDetailsManager 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;
    }

    ...

}
```


© luv2code LLC

Step 2: Update Spring Security Configuration

```
@Configuration
public class DemoSecurityConfig {

    @Bean
    public UserDetailsManager 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;
    }

    ...

}
```



How to find users

How to find roles

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

© luv2code LLC

Project 4

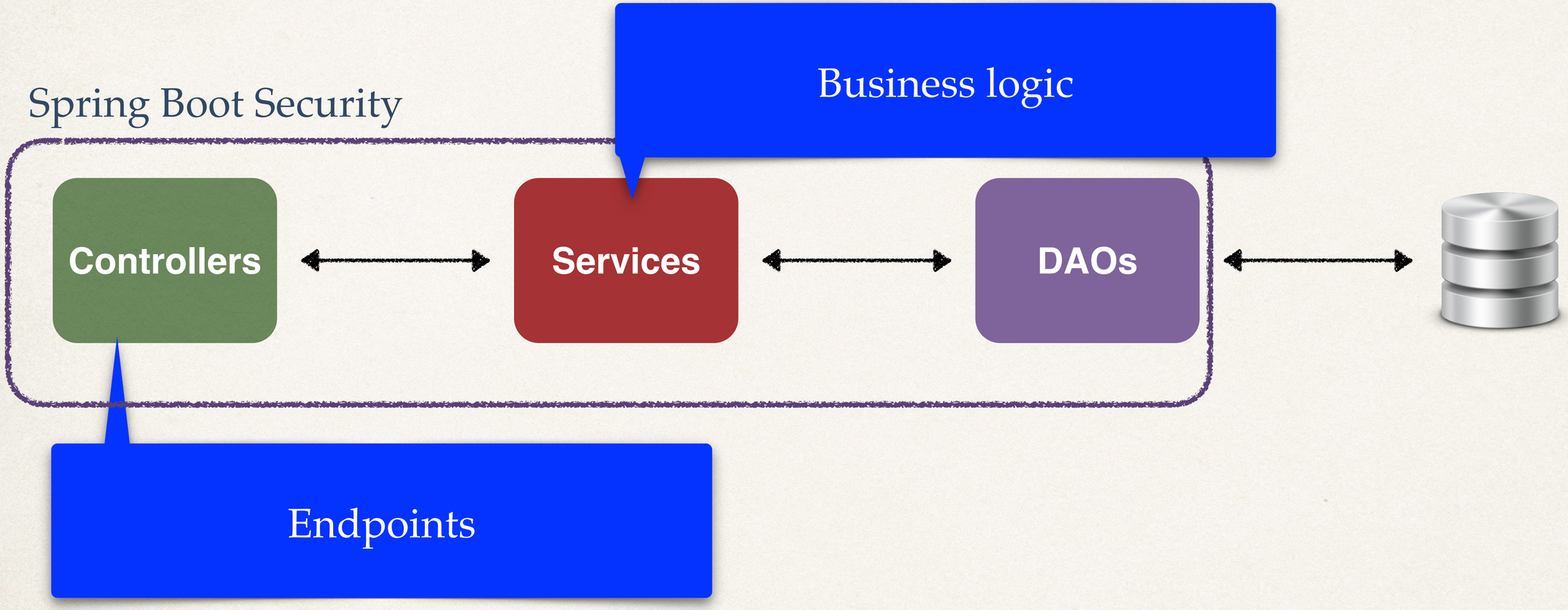
Project 4

- Project four we will be switching our focus to a deep Employee/Admin Todo project.
- New Information will include:
  - Professional (MySQL) Databases
  - Authentication/Authorization using JWTs
  - Admin roles for global permissions
  - User registration.

Continued learning from other projects

© luv2code LLC

Application Architecture



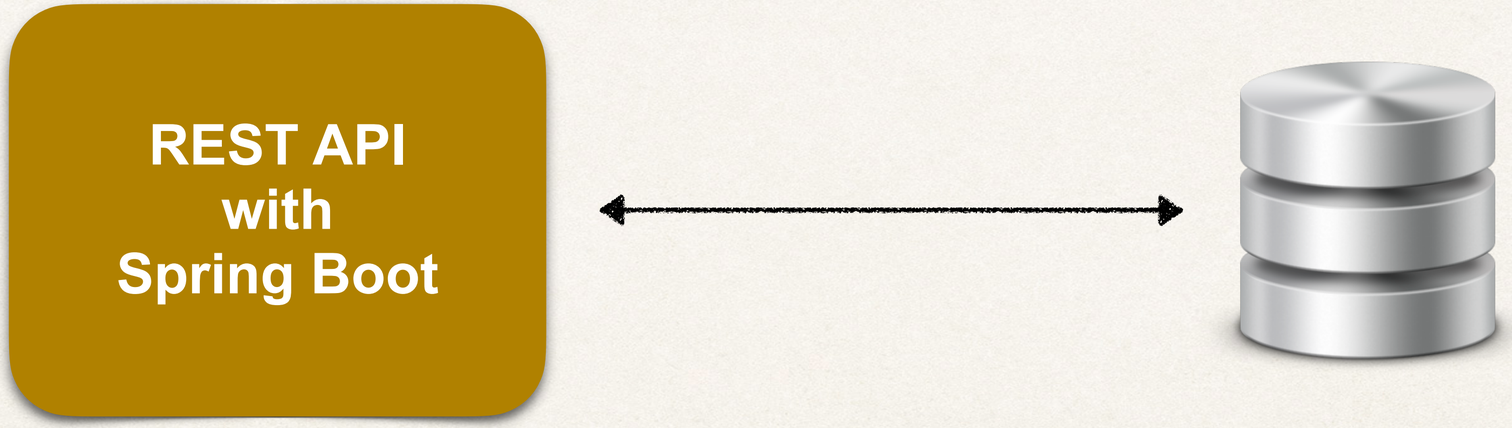
© luv2code LLC

Project Overview

© luv2code LLC

Real-Time Project

REST API with Spring Boot
that connects to a professional database



© luv2code LLC

Requirements

From the Boss

Create multiple REST APIs for the Todo Directory:

1. Users who can request their own information.
2. Todos that can be fetched, created, updated and deleted by their user.
3. Admins who can get all users, promote users to admin and delete non-admin users.
4. Authentication for registration & login using JWTs.

All using a MySQL database

© luv2code LLC

Step 1: API Requirements

step-by-step

Create entities & MySQL database.

Create the necessary requirements to get started:

- Create Todo Entity in code.
- Create User Entity in code.
- Create Authority Entity in code.
- Add the mappings between each entity.
- Setup MySQL.

© luv2code LLC

Step 2: API Requirements

step-by-step

Create REST APIs for the Authentication & Authorization

- Install JWT dependencies.
- Setup JWTs within our project.
- Create and add security configurations to project
- Add Authentication Service functionality.
- Add Authentication Controller functionality.

© luv2code LLC

REST API - Authentication

HTTP Method		CRUD Action
POST	/api/auth/register	Create a new employee
POST	/api/auth/login	Login a user and return a JWT

© luv2code LLC

Step 3: API Requirements

step-by-step

Create REST APIs for the Users

- Add User Request & Response DTOs
- Add User Service functionality.
- Add User Controller functionality.

© luv2code LLC

REST API - Users

HTTP Method		CRUD Action
GET	/api/users/info	Read info about user
PUT	/api/users/password	Update password for user
DELETE	/api/users	Delete user

© luv2code LLC

Step 4: API Requirements

step-by-step

Create REST APIs for the Todos

- Add Todo Request & Response DTOs
- Add Todo Service functionality.
- Add Todo Controller functionality.

© luv2code LLC

REST API - Todos

HTTP Method		CRUD Action
POST	/api/todos	Create a new todo
GET	/api/todos	Read all todos
PUT	/api/todos/{id}	Update a todo
DELETE	/api/todos/{id}	Delete a todo

© luv2code LLC

Step 5: API Requirements

step-by-step

Create REST APIs for the Admin roles.

- Add Admin roles to security configurations.
- Add Admin Service functionality.
- Add Admin Controller functionality.

© luv2code LLC

REST API - Admin

HTTP Method		CRUD Action
GET	/api/admin	Read a list of users
PUT	/api/admin/{id}/role	Update user to admin role
DELETE	/api/admin/{id}	Delete a non admin user

© luv2code LLC

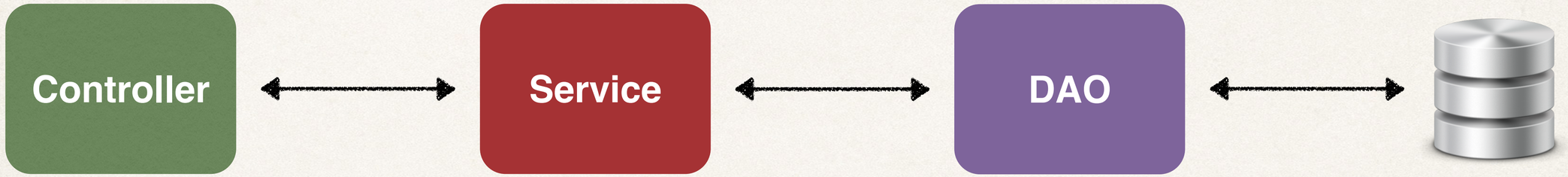
Development Process

Step-By-Step

1. Create Spring Boot project using Spring Initializer .
2. Create entities.
3. Implement creating & reading of JWTs.
4. Add authentication & authorization to product.
5. Add users to product.
6. Add Todo to product.
7. Add admin roles to product.

© luv2code LLC

Application Architecture



© luv2code LLC

Create Project - Spring Initizer

Development Process

Step-By-Step

1. Create Spring Boot project using Spring Initializer .
2. Create entities.
3. Implement creating & reading of JWTs.
4. Add authentication & authorization to product.
5. Add users to product.
6. Add Todo to product.
7. Add admin roles to product.

© luv2code LLC

Spring Initializr

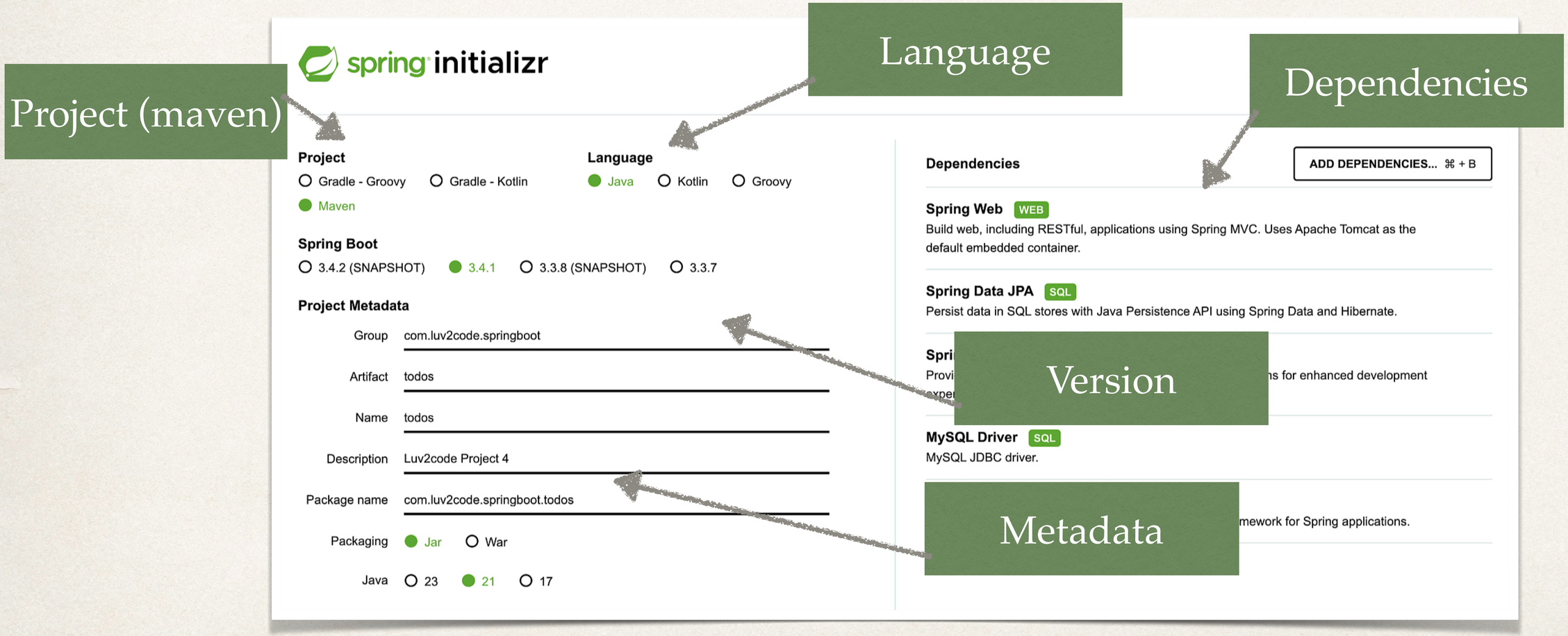
- Here we can set the project configurations for our upcoming Spring project.
- Allows us to choose the project, programming language, Spring Boot version, dependencies and metadata for our application.

start.spring.io

© luv2code LLC

Spring Initializr

- Setup our project using Spring Initializr



© luv2code LLC

Spring Initializr

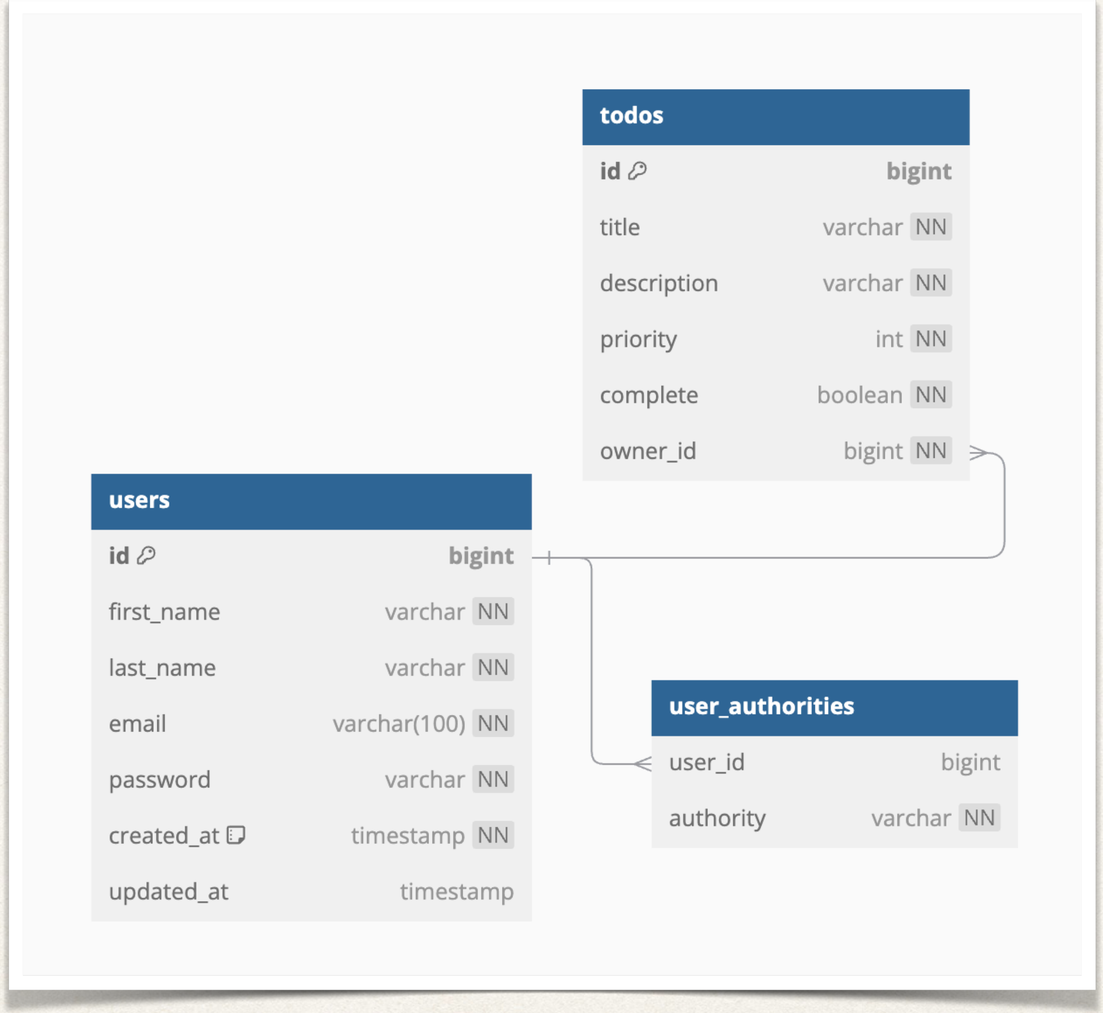
- Spring Initializr will download a .zip file onto our machine.
- We will need to unzip the .zip file.
- Open the project on IntelliJ.
- Download all dependencies using Maven.
- Sounds fun - Let's go ahead and dive in!

© luv2code LLC

Create Entities

What entities do we need?

- Todo Entity
- User Entity
- Authority Entity



© luv2code LLC

Todo Entity

File: Todo.java

```
@Table(name = "todos")
@Entity
public class Todo {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(nullable = false)
    private long id;

    @Column(nullable = false)
    private String title;

    @Column(nullable = false)
    private String description;

    @Column(nullable = false)
    private int priority;

    @Column(nullable = false)
    private boolean complete;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "owner_id", nullable = false)
    private User owner;
…
```

File: Todo.java (continued)

```
// Contructors
public Todo() {}

public Todo(String title, String description,
         int priority, boolean complete, User owner) {

     this.title = title;
     this.description = description;
     this.priority = priority;
     this.complete = complete;
     this.owner = owner;
    }

<<getters & setters for each item>>
}
```

todos	
id	bigint
title	varchar NN
description	varchar NN
priority	int NN
complete	boolean NN
owner_id	bigint NN

© luv2code LLC

User Entity

File: User.java

```
@Table(name = "users")
@Entity
public class User implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(nullable = false)
    private long id;

    @Column(nullable = false)
    private String firstName;

    @Column(nullable = false)
    private String lastName;

    @Column(unique = true, length = 100, nullable = false)
    private String email;

    @Column(nullable = false)
    private String password;
…
```

File: User.java (continued)

```
    @CreationTimestamp
    @Column(updatable = false, name = "created_at")
    private Date createdAt;

    @UpdateTimestamp
    @Column(name = "updated_at")
    private Date updatedAt;

    @ElementCollection(fetch = FetchType.EAGER)
    @CollectionTable(name = "user_authorities",
              joinColumns = @JoinColumn(name = "user_id"))
    private List<Authority> authorities;

    @OneToMany(mappedBy = "owner",
           casade = CascadeType.ALL, orphanRemoval = true)
    private List<Todo> todos;

    public User() {}
}
```

users	
id	bigint
first_name	varchar NN
last_name	varchar NN
email	varchar(100) NN
password	varchar NN
created_at	timestamp NN
updated_at	timestamp

© luv2code LLC

Authority Entity

File: Authority.java

```java
@Embeddable
public class Authority implements GrantedAuthority {

    private String authority;

    public Authority() {}

    public Authority(String authority) {

    this.authority = authority;
  }

    @Override
    public String getAuthority() {
        return authority;
  }
}
```

- @Embeddable indicates this class can be embedded as a component inside other entities.
- Implements Granted Authority which is a core interface in Spring Security, representing permissions granted to an authentication object.

File: User.java

```java
@ElementCollection(fetch = FetchType.EAGER)
@CollectionTable(name = "user_authorities",
          joinColumns = @JoinColumn(name = "user_id"))
private List<Authority> authorities;
```

user_authorities	
user_id	bigint
authority	varchar NN

© luv2code LLC

Setup MySQL Database

MySQL Setup Options:

- 1. You can download MySQL onto your machine and install from there.
- 2. Use Docker to host a container running MySQL for us.
- The second option is a really fast implementation and we can delete/create the MySQL database as often as we like.
- So, we will be proceeding with a Docker setup.

© luv2code LLC

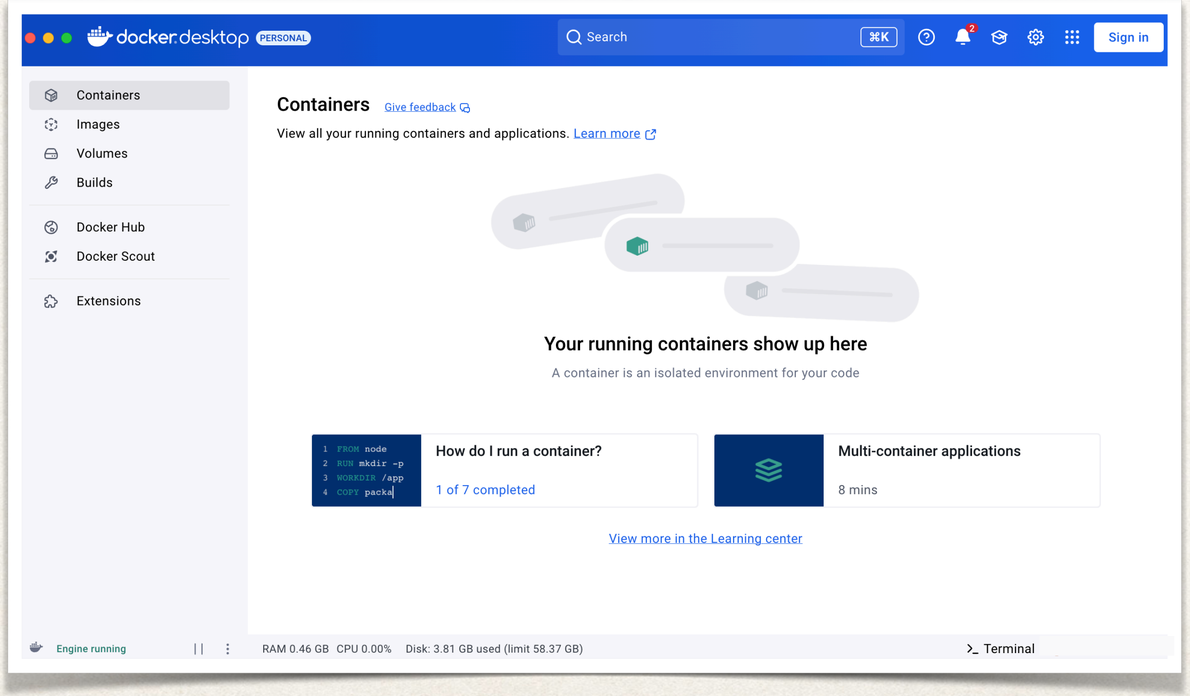
What is Docker?

- Docker is a containerization platform that allows you to package applications into something called containers.
- These containers ensure that our applications run consistently across enviornments.
- We will be only working with two commands:
  - Containers: Isolated environments (where MySQL will run locally).
  - Images: Blueprints for the containers (how to create MySQL)

© luv2code LLC

Easiest way to get Docker

- Go to docker.com
- Download Docker Desktop. ~2GB download.
  - Compatiable with Windows, Mac and Linux.



Once Docker is installed it is one command to get MySQL running

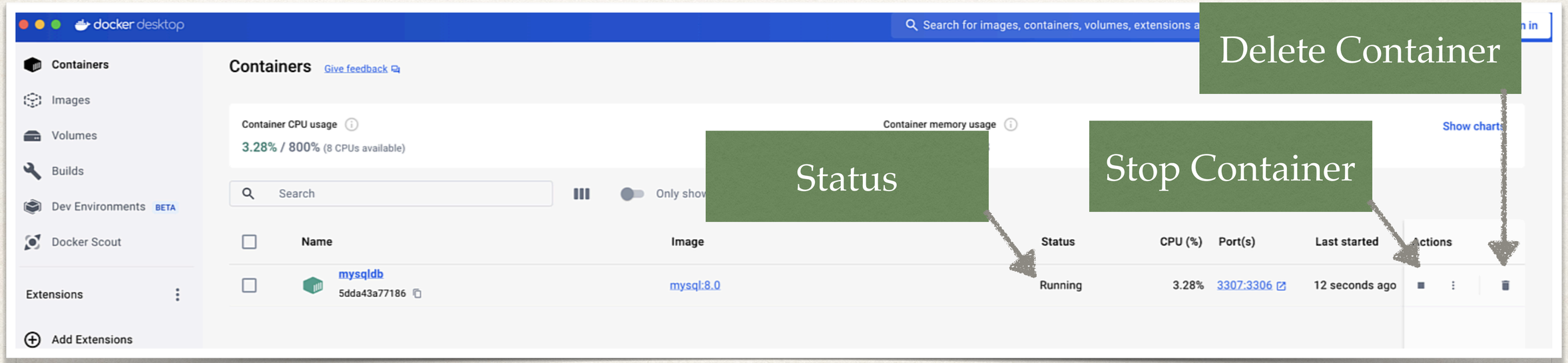
© luv2code LLC

MySQL Setup Options:

- Command to spin up a MySQL Docker command:

```
docker run -d -e MYSQL_ROOT_PASSWORD=secret -e MYSQL_DATABASE=tododb –name mysqldb
-p 3307:3306 mysql:8.0
```

- MySQL now running within Docker Desktop



© luv2code LLC

Add MySQL Configurations

- We need to add properties to connect to MySQL database:

File: application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3307/tododb?serverTimezone=UTC&allowPublicKeyRetrieval=true&useSSL=false
spring.datasource.username=root
spring.datasource.password=secret

spring.jpa.hibernate.ddl-auto=update
spring.jpa.open-in-view=false
```

- spring.datasource.url = configures MySQL database connection to our local database instance
- spring.datasource.username = sets our database username
- spring.datasource.password = sets our database password
- spring.jpa.hibernate.ddl.auto = controls how the database schema is handled. Update means we will automatically update the table based on entities but won't drop any tables.
- spring.jpa.open-in-view = closes JPA EntityManager when transactions end.

© luv2code LLC

DBeaver

- GUI to see our MySQL data easily.
- Supports many other databases as well.
- Community edition is FREE and open source.
- Over 40k+ stars on GitHub

© luv2code LLC

Swagger Implementation

© luv2code LLC

Install OpenAPI for Swagger

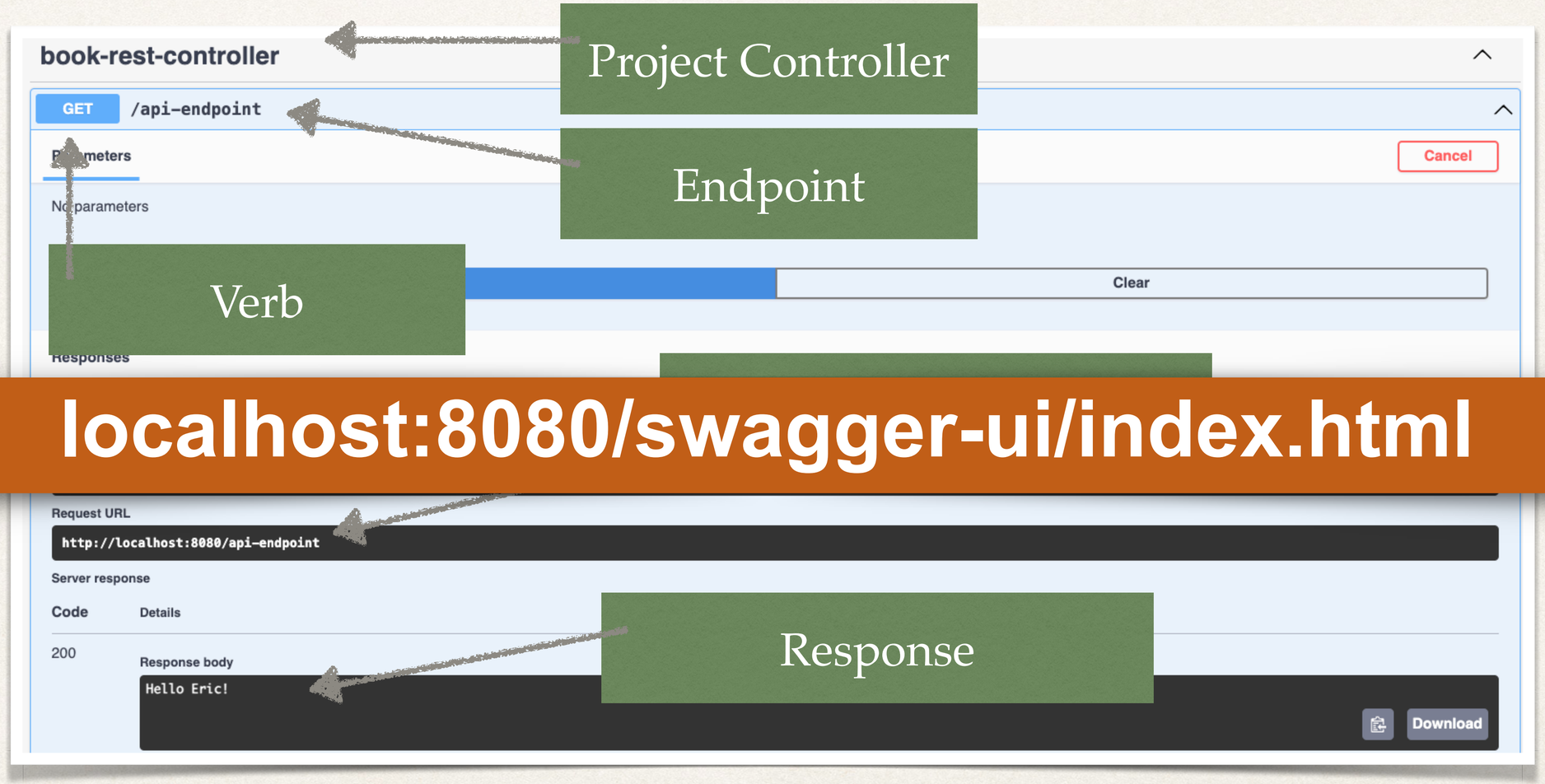
```
<dependency>
     <groupId>org.springdoc</groupId>
     <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
     <version>2.8.3</version>
</dependency

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

Use version compatiable with your Spring Boot

© luv2code LLC

What is Swagger?

- Instead of going straight to the API endpoint, we can go to Swaggers.



localhost:8080/swagger-ui/index.html

© luv2code LLC

What is Swagger?

- Path is hard to remember:
  - localhost:8080/swagger-ui/index.html
- We can fix this to redirect based off of /docs.
  - New endpoint for Swagger: localhost:8080/docs

Too long to remember

Much better :-)

File: application.properties

```
springdoc.swagger-ui.path=/docs
```


© luv2code LLC

What are JWTs?

© luv2code LLC

What is a JSON Web Token?

- JSON Web Token is a self-contained way to securely transmit data and information between two parties using a JSON Object.
- JSON Web Tokens can be trusted because each JWT can be digitally signed, which in return allows the server to know if the JWT has been changed at all
- JWT should be used when dealing with authorization
- JWT is a great way for information to be exchanged between the server and client

© luv2code LLC

What is a JSON Web Token?

- A JSON Web Token is created of three separate parts separated by dots (.) which include:
  - Header (a)
  - Payload (b)
  - Signature (c)

JWT Structure

```
aaaaaaaa.bbbbbbbb.cccccccc
```


© luv2code LLC

JWT Header

- A JWT header usually consist of two parts:
  - (alg) The algorithm for signing
  - (typ) The specific type of token

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- The JWT header is then encoded using Base64 to create the first part of the JWT (a)

JWT Structure

```
aaaaaaaa.bbbbbbbb.cccccccc
```


© luv2code LLC

JWT Payload

- A JWT Payload consists of the data. The Payloads data contains claims, data about the request or user
- The JWT Payload is then encoded using Base64 to create the second part of the JWT (b)

JWT Structure

```
aaaaaaaa.bbbbbbbb.cccccccc
```

```
{
  "sub": "1234567890",
  "name": "Eric Roby",
  "given_name": "Eric",
  "family_name": "Roby",
  "email": "codingwithroby@gmail.com"
  "admin": true
}
```


© luv2code LLC

JWT Signature

- A JWT Signature is created by using the algorithm in the header to hash out the encoded header, encoded payload with a secret.
- The secret can be anything, but is saved somewhere on the server that the client does not have access to
- The signature is the third and final part of a JWT (c)

JWT Structure

```
aaaaaaaa.bbbbbbbb.cccccccc
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```


© luv2code LLC

JWT Example

JWT Header

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

JWT Payload

```
{
  "sub": "1234567890",
  "name": "Eric Roby",
  "given_name": "Eric",
  "family_name": "Roby",
  "email": "codingwithroby@gmail.com"
  "admin": true
}
```

JWT Signature

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  learnonline)
```

JSON Web Token


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkVyaWMgUm9ieSIsImdpdmVuX25hbW
UiOiJFcmljIiwiZmFtaWx5X25hbWUiOiJSb2J5IiwiZW1haWwiOiJjb2Rpbmd3aXRocm9ieUBnbWFpbC5jb20iLCJhZG1pbiI6dHJ1Z
X0.i8yqzOz-nWr5hwXqRYP18W9igUPoKMZiBZW315tK5g8
```


© luv2code LLC

Setup JWTs

© luv2code LLC

Development Process

Step-By-Step

1. Install JJWT into our Java Spring Boot Project
2. Create JWT Service.
    1. Token Parsing
    2. Token Validation
    3. Token Generation
3. JWT Security Authentication Filter
    1. Intercept every HTTP request.
    2. If endpoint require authentication, validate token.

© luv2code LLC

Install JJWT for JSON Web Tokens

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
</dependency>
```

Versions are for the time of the recording.

© luv2code LLC

JJWT Configurations

- We need to add properties for our JWTs

File: application.properties

```
spring.jwt.secret=370fc2ad32927ff329d2807b020a92cbde3d349b44c19304d3c8e8b107365042
spring.jwt.expiration=900000
```

- spring.jwt.secret = Secret to sign and verify JWTs. Used for integrity and authenticity of the tokens.

- spring.jwt.expiration = Expiration of the token after creation in ms. 900,000ms = 15 minutes.

© luv2code LLC

Development Process

Step-By-Step

1. Install JJWT into our Java Spring Boot Project
2. Create JWT Service.
    1. Token Parsing
    2. Token Validation
    3. Token Generation
3. JWT Security Authentication Filter
    1. Intercept every HTTP request.
    2. If endpoint require authentication, validate token.

© luv2code LLC

We need to create a JWT Service

- Token Generation
  - Creates JWTs for authenticated users.
  - Signs token using HS256 algorithm
- Token Validation
  - Verifies if a token is valid or expired.
  - Check if token belongs to correct user.
- Token Parsing
  - Extracts username, claims and expiration date.

© luv2code LLC

JWT Interface

File: JwtService.java

```
public interface JwtService {
   String extractUsername(String token);
   Boolean isTokenValid(String token, UserDetails userDetails);
   String generateToken(Map<String, Object> claims, UserDetails userDetails);
}
```

2. Create JWT Service.
   1. Token Parsing
   2. Token Validation
   3. Token Generation

© luv2code LLC

JWT Service

File: JwtServiceImpl.java

```
@Service
public class JwtServiceImpl implements JwtService {

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

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

    @Override
    public String generateToken(Map<String, Object> claims, UserDetails userDetails) {
        return Jwts.builder()
                .setClaims(claims)
                .setSubject(userDetails.getUsername())
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + JWT_EXPIRATION))
                .signWith(getSigningKey(), SignatureAlgorithm.HS256)
                .compact();
    }

    private Key getSigningKey() {
        byte[] keyBytes = Decoders.BASE64.decode(SECRET_KEY);
        return Keys.hmacShaKeyFor(keyBytes);
    }
}
```

2. Create JWT Service.
   1. Token Parsing
   2. Token Validation
   3. Token Generation ✓

© luv2code LLC

JWT Service

File: JwtServiceImpl.java

```
@Service
public class JwtServiceImpl implements JwtService {

<other code…>

    @Override
    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    private <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }

    private Claims extractAllClaims(String token) {
        return Jwts.parserBuilder()
                .setSigningKey(getSigningKey())
                .build()
                .parseClaimsJws(token)
                .getBody();
    }

}
```

2. Create JWT Service.
   1. Token Parsing ✓
   2. Token Validation
   3. Token Generation ✓

© luv2code LLC

JWT Service

File: JwtServiceImpl.java

2. Create JWT Service.
   1. Token Parsing ✓
   2. Token Validation ✓
   3. Token Generation ✓

```
@Service
public class JwtServiceImpl implements JwtService {

<other code…>

    @Override
    public boolean isTokenValid(String token, UserDetails userDetails) {
        final String username = extractUsername(token);
        return (username.equals(userDetails.getUsername())) && !isTokenExpired(token);
    }

    private boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    private Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }

}
```


© luv2code LLC

JWT Authentication Filter

© luv2code LLC

Development Process

Step-By-Step

1. Install JJWT into our Java Spring Boot Project
2. Create JWT Service.
    1. Token Parsing
    2. Token Validation
    3. Token Generation
3. JWT Security Authentication Filter
    1. Intercept every HTTP request.
    2. If endpoint require authentication, validate token.

© luv2code LLC

Deeper dive in JWT Auth Filter

- Essentially a security checkpoint for our application.
- Every request gets checked, if the endpoint requires authentication, we valid the JWT attached.
  - Extract their credentials
  - Verify the token is legitimate and hasn't expired.

© luv2code LLC

JWT Authentication filter

File: JwtAuthenticationFilter.java

```
@Component
public class JwtAuthenticationFilter implements OncePerRequestFilter {

    private final JwtService jwtService;

    private final UserDetailsService userDetailsService;

    public JwtAuthenticationFilter(JwtService jwtService, @Lazy UserDetailsService userDetailsService) {
        this.jwtService = jwtService;
        this.userDetailsService = userDetailsService;
       }

…Continues on next slide…

}
```


© luv2code LLC

JWT Authentication filter

File: JwtAuthenticationFilter.java

```
    @Override
    protected void doFilterInternal(@NonNull HttpServletRequest request, @NonNull HttpServletResponse response,
       @NonNull FilterChainfilterChain) throws ServletException, IOException {
        final String authHeader = request.getHeader("Authorization");
        final String jwt;
        final String userEmail;

        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }

        jwt = authHeader.substring(7);
        userEmail = jwtService.extractUsername(jwt);

        if (userEmail != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails = this.userDetailsService.loadUserByUsername(userEmail);

            if (jwtService.isTokenValid(jwt, userDetails)) {
                UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(
                        userDetails, null, userDetails.getAuthorities());

                authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }
        filterChain.doFilter(request, response);
    }
}
```


© luv2code LLC

Security Filter & User Repository

© luv2code LLC

User Repository

- Spring Data JPA repository interface for managing User entities.

File: UserRepository.java

```
@Repository
public interface UserRepository extends CrudRepository<User, Long> {
    Optional<User> findByEmail(String email);
}
```

- @Repository marks the interface a Spring Repository making it responsible for data access operations.
- CrudRepository provides basic CRUD operations. User is the entity and Long is the primary key.
- The function automatically creates a query to find a user by their email.

© luv2code LLC

Security Config

- Setup our Spring Boot Security configurations.

File: SecurityConfig.java

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private final UserRepository userRepository;
    private final JwtAuthenticationFilter jwtAuthenticationFilter;

    public SecurityConfig(UserRepository userRepository,
              JwtAuthenticationFilter jwtAuthenticationFilter) {
        this.userRepository = userRepository;
        this.jwtAuthenticationFilter = jwtAuthenticationFilter;
}
```

…Continues on next slide…

© luv2code LLC

Security Config

File: SecurityConfig.java

```
@Bean
UserDetailsService userDetailsService() {
    return username -> userRepository.findByEmail(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not found"));
}

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

@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {
    return config.getAuthenticationManager();
}

@Bean
public AuthenticationEntryPoint authenticationEntryPoint() {
    return (request, response, ex) -> {
        response.setStatus(HttpStatus.UNAUTHORIZED.value());
        response.setContentType("application/json");
        response.setHeader("WWW-Authenticate", "");
        response.getWriter().write("{\"error\": \"Unauthorized access\"}");
    };
}
```

{bcrypt}<hash>

© luv2code LLC

Security Config

File: SecurityConfig.java

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests(configurer ->
            configurer
                    .requestMatchers("/swagger-ui/**", "/v3/api-docs/**",
                            "/swagger-resources/**", "/webjars/**", "/docs").permitAll());

    http.csrf(csrf -> csrf.disable());

    http.exceptionHandling(exceptionHandling ->
            exceptionHandling
                    .authenticationEntryPoint(authenticationEntryPoint()));

    http.sessionManagement(session ->
            session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

    http.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```


© luv2code LLC

Register User

© luv2code LLC

Development Process

Step-By-Step

1. Create RegisterRequest with data validation
2. Create AuthenticationService interface (register)
    1. Create AuthenticationServiceImpl (register logic)
3. Authentication Controller (register)

© luv2code LLC

Register Request

File: RegisterRequest.java

```
public class RegisterRequest {

    @NotEmpty(message = "First name is mandatory")
    @Size(min = 3, max = 30, message = "First name must be at least 3 characters long")
    private String firstName;

    @NotEmpty(message = "Last name is mandatory")
    @Size(min = 3, max = 30, message = "Last name must be at least 3 characters long")
    private String lastName;

    @NotEmpty(message = "Email is mandatory")
    @Email(message = "Invalid email format")
    private String email;

    @NotEmpty(message = "Password is mandatory")
    @Size(min = 5, max = 30, message = "Password must be at least 5 characters long")
    private String password;

    public RegisterRequest(String firstName, String lastName, String email, String password) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.password = password;
    }

<…getters & setters…>
```


© luv2code LLC

Development Process

Step-By-Step

1. Create RegisterRequest with data validation
2. Create AuthenticationService interface (register)
    1. Create AuthenticationServiceImpl (register logic)
3. Authentication Controller (register)

© luv2code LLC

AuthenticationService

File: AuthenticationService.java

```
public interface AuthenticationService {
    void register(RegisterRequest input) throws Exception;
}
```

- Only one function for now that we need to implement in our AuthenticationServiceImpl class.

© luv2code LLC

AuthenticationServiceImpl

File: AuthenticationServiceImpl.java

```java
@Service
public class AuthenticationServiceImpl implements
                                AuthenticationService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    public AuthenticationServiceImpl(UserRepository userRepository,
              PasswordEncoder passwordEncoder) {
        this.userRepository = userRepository;
        this.passwordEncoder = passwordEncoder;
    }

    @Override
    @Transactional
    public void register(RegisterRequest input) throws Exception {
        if (isEmailTaken(input.getEmail())) {
            throw new Exception("Email already taken");
        }
        User user = buildNewUser(input);
        userRepository.save(user);
    }


    private boolean isEmailTaken(String email) {
        return userRepository.findByEmail(email).isPresent();
    }
```

```java
private User buildNewUser(RegisterRequest input) {
   User user = new User();
   user.setId(0);
   user.setFirstName(input.getFirstName());
   user.setLastName(input.getLastName());
   user.setEmail(input.getEmail());
   user.setPassword(passwordEncoder.encode(
            input.getPassword()
       ));
   user.setAuthorities(initialAuthority());
   return user;
}

private List<Authority> initialAuthority() {
   boolean isFirstUser = userRepository.count() == 0;
   List<Authority> authorities = new ArrayList<>();
   authorities.add(new Authority("ROLE_EMPLOYEE"));
   if (isFirstUser) {
        authorities.add(new Authority("ROLE_ADMIN"));
    }
    return authorities;
 }
}
```


© luv2code LLC

Development Process

Step-By-Step

1. Create RegisterRequest with data validation
2. Create AuthenticationService interface (register)
   1. Create AuthenticationServiceImpl (register logic)
3. Authentication Controller (register)

© luv2code LLC

AuthenticationController

File: AuthenticationController.java

```java
@RestController
@RequestMapping("/api/auth")
@Tag(name = "Authentication REST API Endpoints", description = "Operations related to register & login")
public class AuthenticationController {

    private final AuthenticationService authenticationService;

    public AuthenticationController(AuthenticationService authenticationService) {
        this.authenticationService = authenticationService;
    }

    @Operation(summary = "Register a user", description = "Create new user in database")
    @ResponseStatus(HttpStatus.CREATED)
    @PostMapping("/register")
    public void register(@Valid @RequestBody RegisterRequest registerRequest) throws Exception {
        authenticationService.register(registerRequest);
    }
```


© luv2code LLC

Security Config

File: SecurityConfig.java

Add this path to be permitted

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests(configurer ->
            configurer
                    .requestMatchers("/api/auth/**","/swagger-ui/**", "/v3/api-docs/**",
                            "/swagger-resources/**", "/webjars/**", "/docs").permitAll());

    http.csrf(csrf -> csrf.disable());

    http.exceptionHandling(exceptionHandling ->
            exceptionHandling
                    .authenticationEntryPoint(authenticationEntryPoint()));

    http.sessionManagement(session ->
            session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

    http.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```


© luv2code LLC

Authenticate User

© luv2code LLC

Development Process

Step-By-Step

1. Create AuthenticateRequest with data validation and AuthenticateResponse

2. Add login to AuthenticationService interface

   1. Add login to AuthenticationServiceImpl

3. Add Login to Authentication Controller

© luv2code LLC

Authenticate Request

File: AuthenticationRequest.java

```
public class AuthenticationRequest {

    @NotEmpty(message = "Email is mandatory")
    @Email(message = "Invalid email format")
    private String email;

    @NotEmpty(message = "Password is mandatory")
    @Size(min = 5, max = 30, message = "Password must be at least 5 characters long")
    private String password;

    public String getEmail() {
        return email;
    }

    public String getPassword() {
        return password;
    }
}
```


© luv2code LLC

Authenticate Response

File: AuthenticationResponse.java

```
public class AuthenticationResponse {

    private String token;

    public AuthenticationResponse(String token) {
        this.token = token;
    }

    public void setToken(String token) {
        this.token = token;
    }
}
```


© luv2code LLC

Development Process

Step-By-Step

1. Create AuthenticateRequest with data validation and AuthenticateResponse
2. Add login to AuthenticationService interface
   1. Add login to AuthenticationServiceImpl
3. Add Login to Authentication Controller

© luv2code LLC

AuthenticationService

File: AuthenticationService.java

```
public interface AuthenticationService {
    void register(RegisterRequest input) throws Exception;
    AuthenticationResponse login(AuthenticationRequest request);
}
```

- Only one new function for now that we need to implement in our AuthenticationServiceImpl class.

© luv2code LLC

AuthenticationServiceImpl

File: AuthenticationServiceImpl.java

```
@Override
@Transactional(readOnly = true)
public AuthenticationResponse login(AuthenticationRequest request) {

    authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(request.getEmail(), request.getPassword())
    );

    User user = userRepository.findByEmail(request.getEmail())
            .orElseThrow(() -> new IllegalArgumentException("Invalid email or password"));

    String jwtToken = jwtService.generateToken(new HashMap<>(), user);

    return new AuthenticationResponse(jwtToken);
}
```


© luv2code LLC

Development Process

Step-By-Step

1. Create AuthenticateRequest with data validation and AuthenticateResponse
2. Add login to AuthenticationService interface
   1. Add login to AuthenticationServiceImpl
3. Add Login to Authentication Controller

© luv2code LLC

AuthenticationController

File: AuthenticationController.java

```
@RestController
@RequestMapping("/api/auth")
@Tag(name = "Authentication REST API Endpoints", description = "Operations related to register & login")
public class AuthenticationController {

    private final AuthenticationService authenticationService;

    public AuthenticationController(AuthenticationService authenticationService) {
        this.authenticationService = authenticationService;
    }

    @Operation(summary = "Login a user", description = "Submit email & password to authenticate user.")
    @ResponseStatus(HttpStatus.OK)
    @PostMapping("/login")
    public AuthenticationResponse login(@Valid @RequestBody AuthenticationRequest authRequest) throws Exception {
        authenticationService.login(authRequest);
    }
}
```


© luv2code LLC

Get User Info & Swagger Header

© luv2code LLC

Development Process

Step-By-Step

1. Create UserService interface
   1. Create UserServiceImpl (business logic)
2. User Service Controller (getting the user info)
3. Swagger Configurations for headers
4. Create UserResponse

© luv2code LLC

UserService

File: UserService.java

```
public interface UserService {
    User getUser();
}
```

- Only one function for now that we need to implement in our UserServiceImpl class.

© luv2code LLC

UserServiceImpl

File: UserServiceImpl.java

```
@Service
public class UserServiceImpl implements UserService {

    private final UserRepository userRepository;

    public UserServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    @Transactional(readOnly = true)
    public User getUserInfo() throws Exception {
       Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

       if (authentication == null || !authentication.isAuthenticated()) ||
                                 authentication.getPrincipal().equals("anonymousUser")) {

          throw new AccessDeniedException("Authentication required");

       return (User) authentication.getPrinciple();
}
```


© luv2code LLC

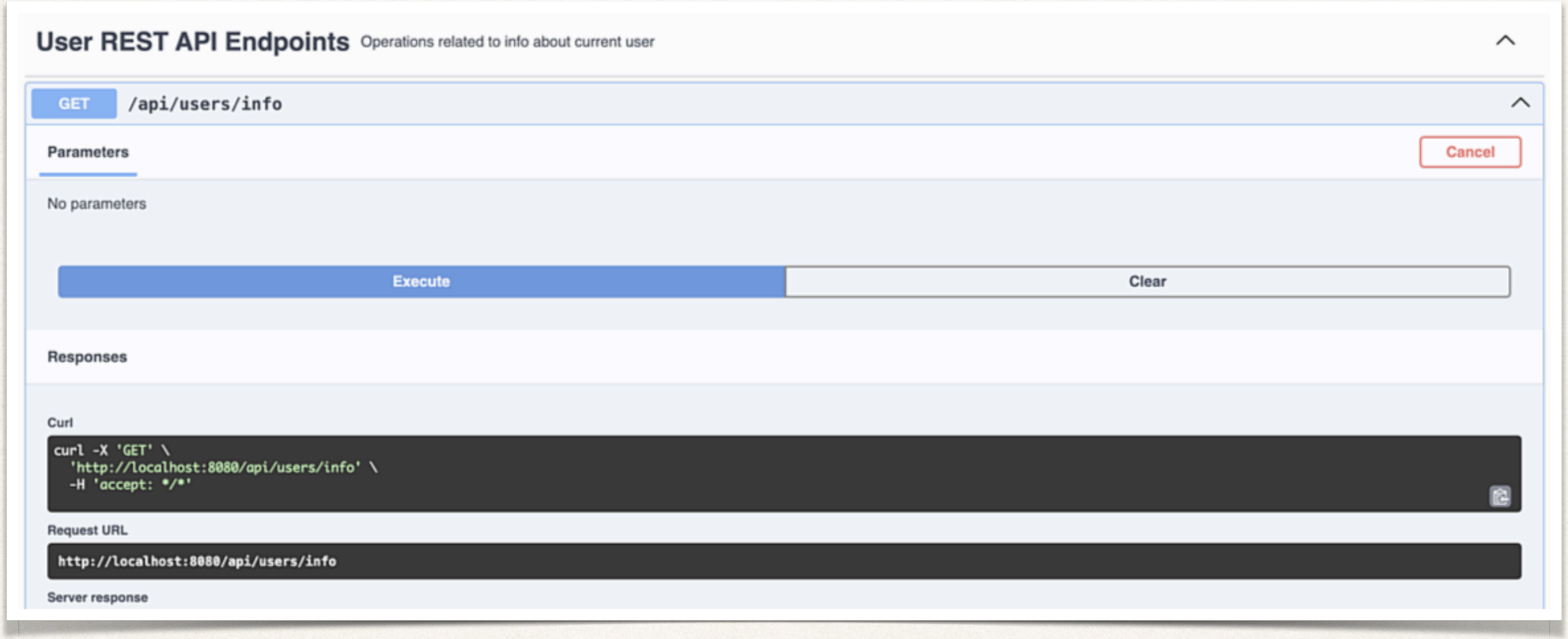
UserController

File: UserController.java

```
@RestController
@RequestMapping("/api/users")
@Tag(name = "User REST API Endpoints", description = "Operations related to info about current user")
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping("/info")
    public User getUserInfo() {
        return userService.getUserInfo();
    }
}
```


© luv2code LLC

Is there something missing in Swagger?



- No where to pass in JWT for validation.

© luv2code LLC

Security Config

File: SecurityConfig.java

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests(configurer ->
            configurer
                    .requestMatchers("/api/auth/**","/swagger-ui/**", "/v3/api-docs/**",
                            "/swagger-resources/**", "/webjars/**", "/docs").permitAll()
                                                                            .anyRequest().authenticated());

    http.csrf(csrf -> csrf.disable());

    http.exceptionHandling(exceptionHandling ->
            exceptionHandling
                    .authenticationEntryPoint(authenticationEntryPoint()));

    http.sessionManagement(session ->
            session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

    http.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

All other paths need authentication.

© luv2code LLC

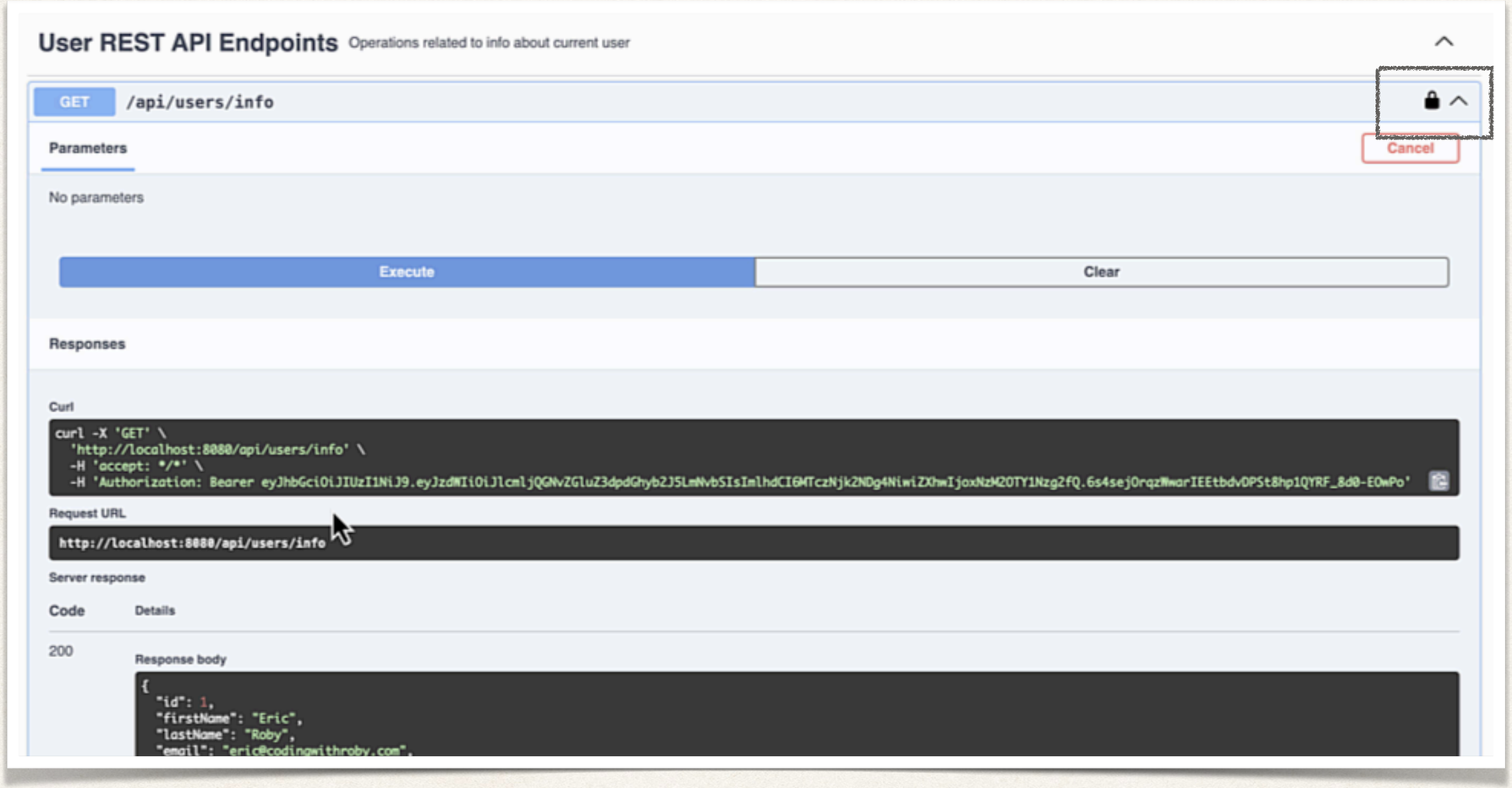
Swagger Config

File: SwaggerConfig.java

```
@Configuration
@OpenAPIDefinition(
        info = @Info(title = "API Documentation", version = "v1"),
        security = @SecurityRequirement(name = "bearerAuth")
)
@SecurityScheme(
        name = "bearerAuth",
        type = SecuritySchemeType.HTTP,
        scheme = "bearer",
        bearerFormat = "JWT"
)
public class SwaggerConfig {
    // Leave this blank
}
```


© luv2code LLC

Now we can add JWTs to Swagger



Add JWT here from login

© luv2code LLC

Return an entire user

Response body


```
{
  "id": 1,
  "firstName": "Eric",
  "lastName": "Roby",
  "email": "eric@codingwithroby.com",
  "password": "$2a$10$eORKtbIgt.CdftPLXd9kGuxKg5DKaFy4F4p4wkJOrMHEzQWX.U5n2",
  "authorities": [
    {
      "authority": null
    },
    {
      "authority": null
    }
  ],
  "enabled": true,
  "credentialsNonExpired": true,
  "username": "eric@codingwithroby.com",
  "accountNonExpired": true,
  "accountNonLocked": true
}
```


- Need to return only the items the user needs.

© luv2code LLC

User Response

File: UserResponse.java

```
public class UserResponse {

    private long id;
    private String fullName;
    private String email;
    private List<Authority> authorities;

    public UserResponse(long id, String fullName, String email, List<Authority> authorities) {
        this.id = id;
        this.fullName = fullName;
        this.email = email;
        this.authorities = authorities;
    }

<getters & setters>
}
```


© luv2code LLC

UserService

File: UserService.java

```
public interface UserService {
    UserResponse getUser();
}
```


© luv2code LLC

UserServiceImpl

File: UserServiceImpl.java

```
    @Override
    @Transactional(readOnly = true)
    public UserResponse getUserInfo() throws Exception {
       Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

       if (authentication == null || !authentication.isAuthenticated()) ||
                                 authentication.getPrincipal().equals("anonymousUser")) {

          throw new AccessDeniedException("Authentication required");

       User user = (User) authentication.getPrinciple();

       return new UserResponse(
          user.getId(),
          user.getFirstName() + " " + user.getLastName(),
          user.getEmail(),
          user.getAuthorities().stream().map(auth -> (Authority) auth).toList());
       )
    }
}
```


© luv2code LLC

UserController

File: UserController.java

```
@RestController
@RequestMapping("/api/users")
@Tag(name = "User REST API Endpoints", description = "Operations related to info about current user")
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping("/info")
    public UserResponse getUserInfo() {
        return userService.getUserInfo();
    }
}
```


© luv2code LLC

Return a User Response

```
{
  "id": 1,
  "fullName": "Eric Roby",
  "email": "eric@codingwithroby.com",
  "authorities": [
    {
      "authority": null
    },
    {
      "authority": null
    }
  ]
}
```

- Only the items that we need.

© luv2code LLC

Delete User

© luv2code LLC

Development Process

Step-By-Step

1. Add delete to UserService interface
   1. Add delete to UserServiceImpl
2. Write custom query to find how many admins are in the application.
3. Add delete to User Controller

© luv2code LLC

UserService

File: UserService.java

```
public interface UserService {
    UserResponse getUser();
    void deleteUser();
}
```

- Implement delete user business logic in our UserServiceImpl class.

© luv2code LLC

UserServiceImpl

File: UserServiceImpl.java

```
@Override
public void deleteUser() {

   Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

   if (authentication == null || !authentication.isAuthenticated()) ||
                              authentication.getPrincipal().equals("anonymousUser")) {
       throw new AccessDeniedException("Authentication required");
   }

   User user = (User) authentication.getPrinciple();


    if (isLastAdmin(user)) {
        throw new ResponseStatusException(HttpStatus.FORBIDDEN, "Admin cannot delete itself");
    }

    userRepository.delete(user);
}

private boolean isLastAdmin(User user) {
    boolean isAdmin = user.getAuthorities().stream()
            .anyMatch(authority -> "ROLE_ADMIN".equals(authority.getAuthority()));

    if (isAdmin) {
        long adminCount = userRepository.countAdminUsers();
        return adminCount <= 1;
    }

    return false;
}
```

Custom Query

© luv2code LLC

UserRepository

File: UserRepository.java

```
@Repository
public interface UserRepository extends CrudRepository<User, Long> {

    Optional<User> findByEmail(String email);

    @Query("SELECT COUNT(u) FROM User u JOIN u.authorities a WHERE a.authority = 'ROLE_ADMIN'")
    long countAdminUsers();
}
```


© luv2code LLC

UserController

File: UserController.java

```
@RestController
@RequestMapping("/api/users")
@Tag(name = "User REST API Endpoints", description = "Operations related to info about current user")
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping("/info")
    public UserResponse getUserInfo() {
        return userService.getUserInfo();
    }

    @DeleteMapping
    public void deleteUser() {
        userService.deleteUser();
    }
}
```


© luv2code LLC

Exception Handling

© luv2code LLC

Exceptions

- As of now we are not handling exceptions.
- We are returning to the user whatever the exception was which could be a security risk.
  - We do not want to give the user more information than they need.

© luv2code LLC

We really want this …

- Handle the exception and return error as JSON

```
DELETE /api/users/1
```

```
{
    “status”: 403,
    “message”: “Admin cannot delete itself,
    “timestamp”: “1526149650271”
}
```


© luv2code LLC

Spring REST Exception Handling

© luv2code LLC

Development Process

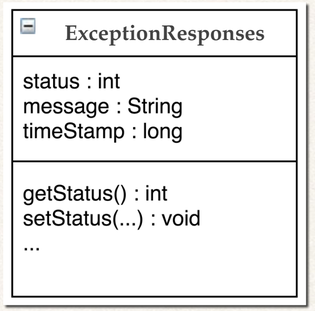
Step-By-Step

1. Create a custom error response class
2. Create a custom exception class with @ControllerAdvice
3. Add a exception handler methods using `@ExceptionHandler`
   1. One for our global Exception
   2. One for our ResponseStatusException

© 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

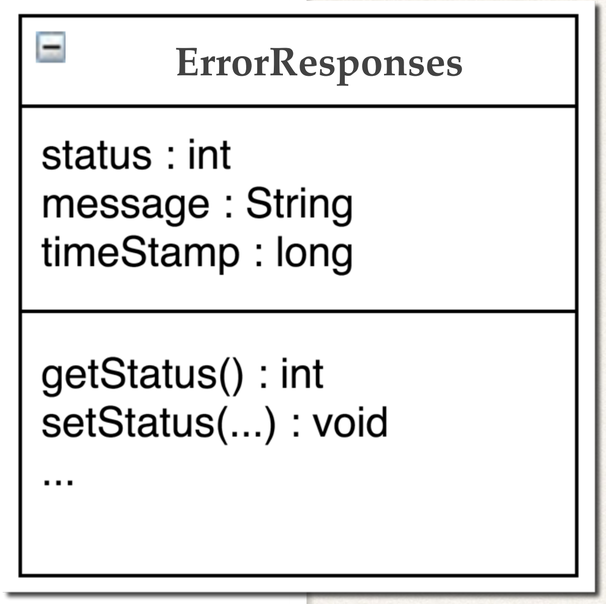
```
{
    "status": 403,
    "message": "Admin cannot delete itself,
    "timestamp": "1526149650271"
}
```


© luv2code LLC

Step 1: Create custom error response class

File: ErrorResponses.java

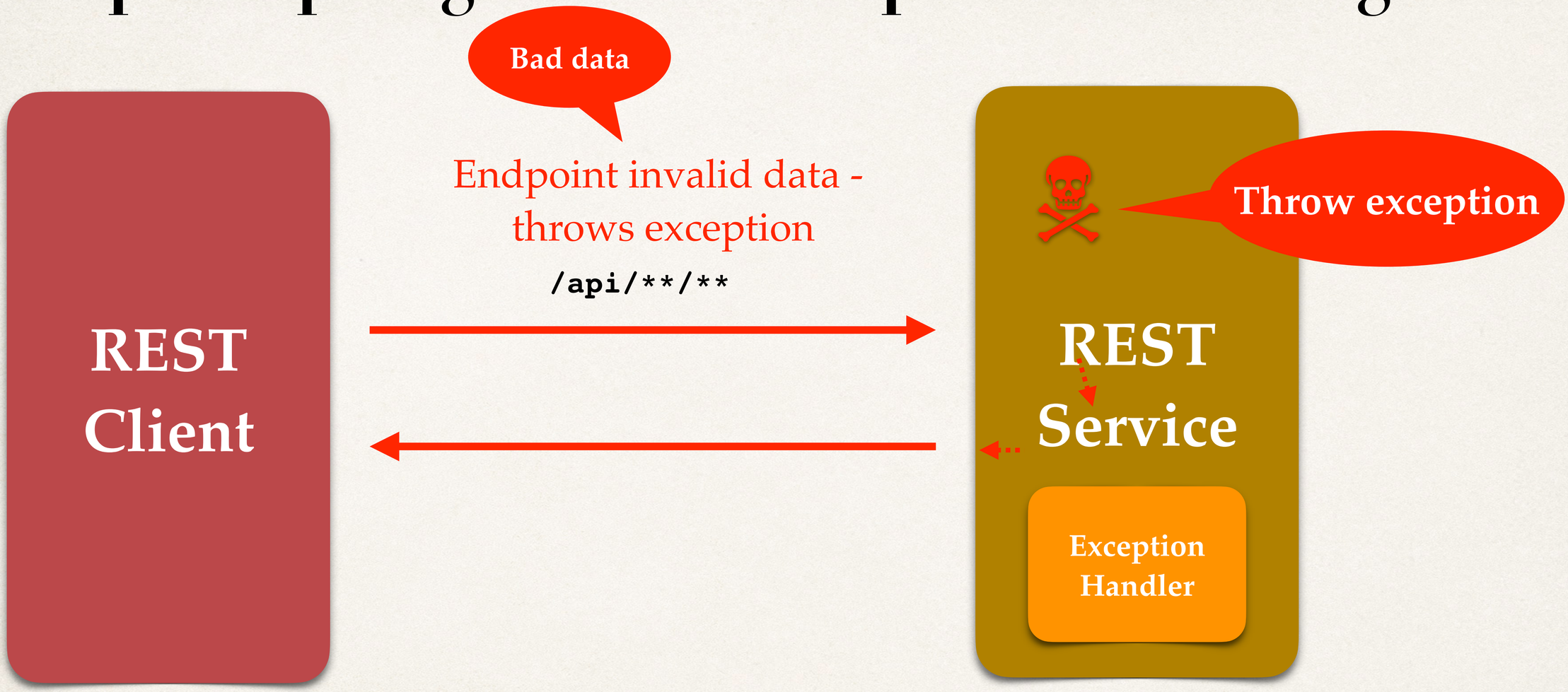
```
public class ErrorResponses {

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

  // constructors

  // getters / setters

}
```




```
{
    "status": 403,
    "message": "Admin cannot delete itself,
    "timestamp": "1526149650271"
}
```


© luv2code LLC

Step 2: Spring REST Exception Handling



© luv2code LLC

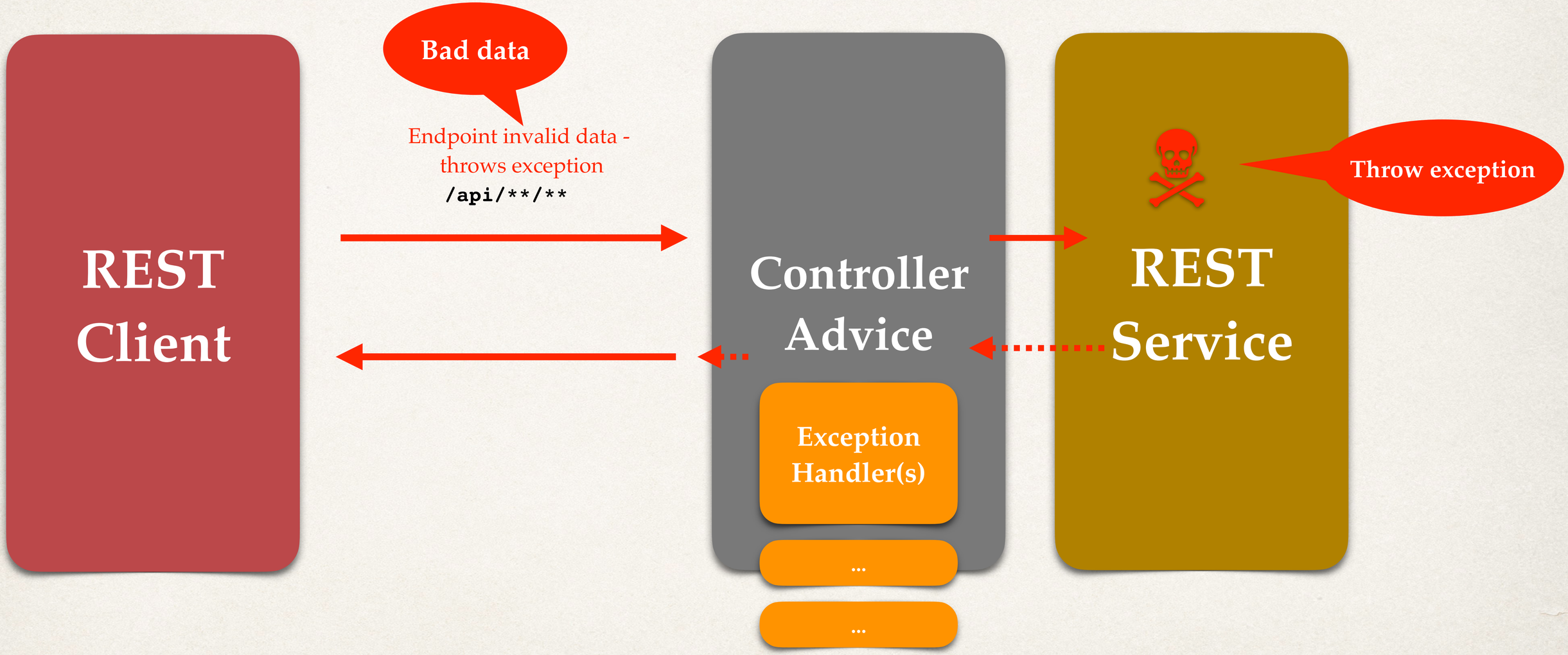
Spring @ControllerAdvice

- `@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

Step 2: Create new @ControllerAdvice

File: ExceptionHandler.java

```
@ControllerAdvice
public class ExceptionHandlers {

  ...

}
```


© luv2code LLC

Step 2: Create new @ControllerAdvice

File: ExceptionHandler.java

```
@ControllerAdvice
public class ExceptionHandlers {

    private ResponseEntity<ExceptionResponses> buildResponseEntity(Exception exc, HttpStatus status) {
        ExceptionResponses error = new ExceptionResponses();
        error.setStatus(status.value());
        error.setMessage(exc.getMessage());
        error.setTimeStamp(System.currentTimeMillis());
        return new ResponseEntity<>(error, status);
    }
}
```


© luv2code LLC

Step 3: Add exception handler to @ControllerAdvice

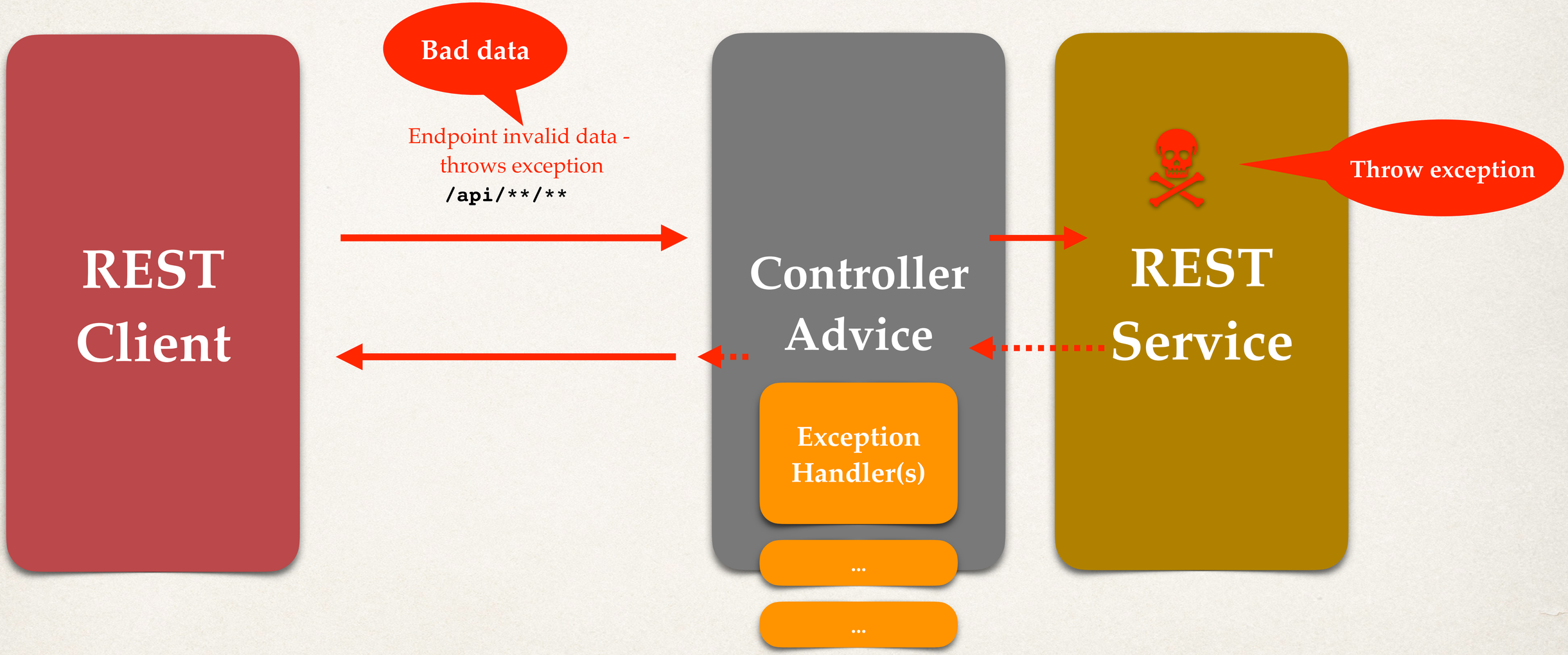
File: ExceptionHandlers.java

Exception Handlers

```
@ControllerAdvice
public class ExceptionHandlers {

    @ExceptionHandler(ResponseStatusException.class)
    public ResponseEntity<ExceptionResponses> handleException(ResponseStatusException exc) {
        return buildResponseEntity(exc, HttpStatus.valueOf(exc.getStatusCode().value()));
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ExceptionResponses> handleException(Exception exc) {
        return buildResponseEntity(exc, HttpStatus.BAD_REQUEST);
    }

  <…Code from last slide…>
}
```


© luv2code LLC

Spring REST Exception Handling



© luv2code LLC

Utilities

© luv2code LLC

Utilities

- We have two pieces of code using the same snippet wrote twice.
  - getUserInfo
  - deleteUser

```
Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

if (authentication == null || !authentication.isAuthenticated()) ||
                            authentication.getPrincipal().equals("anonymousUser")) {
    throw new AccessDeniedException("Authentication required");
}

User user = (User) authentication.getPrinciple();
```


© luv2code LLC

FindAuthenticatedUser

File: FindAuthenticatedUser

```
public interface FindAuthenticatedUser {

    User getAuthenticatedUser();
}
```


© luv2code LLC

Utilities

```
public class FindAuthenticatedUserImpl implements FindAuthenticatedUser {

    @Override
    public User getAuthenticatedUser() {

        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

        if (authentication == null || !authentication.isAuthenticated()) ||
                                    authentication.getPrincipal().equals("anonymousUser")) {
            throw new AccessDeniedException("Authentication required");
        }

        return (User) authentication.getPrinciple();

    }
}
```


© luv2code LLC

Update Password

© luv2code LLC

Development Process

Step-By-Step

1. Create PasswordUpdateRequest DTO
2. Add update password to UserService interface
   1. Add update password to UserServiceImpl
3. Add update password to User Controller

© luv2code LLC

PasswordUpdate Request

File: PasswordUpdateRequest.java

```
public class PasswordUpdateRequest {

    @NotEmpty(message = "Old password is mandatory")
    @Size(min = 5, max = 30, message = "Old password must be at least 5 characters long")
    private String oldPassword;

    @NotEmpty(message = "New password is mandatory")
    @Size(min = 5, max = 30, message = "New password must be at least 5 characters long")
    private String newPassword;

    @NotEmpty(message = "New password is mandatory")
    @Size(min = 5, max = 30, message = "New password must be at least 5 characters long")
    private String newPassword2;

    public PasswordUpdateRequest(String oldPassword, String newPassword, String newPassword2) {
        this.oldPassword = oldPassword;
        this.newPassword = newPassword;
        this.newPassword2 = newPassword2;
    }

<…getters & setters…>
```


© luv2code LLC

UserService

File: UserService.java

```
public interface UserService {
    UserResponse getUser();
    void deleteUser();
    Void updatePassword(PasswordUpdateRequest passwordUpdateRequest);
```

- Implement update password business logic in our UserServiceImpl class.

© luv2code LLC

Update Password Checks

- Is the oldPassword, correct?
  - We need to check if the oldPassword matches what we have in the database.
- Is newPassword and newPassword2 the same?
  - We need to validate the user typed in the correct information both times.
- Is newPassword not the same as oldPassword?
  - We only want to implement this logic if the user actually adding a new password.

© luv2code LLC

UserServiceImpl

File: UserServiceImpl.java

Old password correct?

```
private boolean isOldPasswordCorrect(String currentPassword, String oldPassword) {
    return passwordEncoder.matches(oldPassword, currentPassword);
}

private boolean isNewPasswordConfirmed(String newPassword, String newPasswordConfirmation) {
    return newPassword.equals(newPasswordConfirmation);
}

private boolean isNewPasswordDifferent(String oldPassword, String newPassword) {
    return !oldPassword.equals(newPassword);
}
```

Passwords match?

New password diff from old password?

© luv2code LLC

UserServiceImpl

File: UserServiceImpl.java

```
@Override
@Transactional
public void updatePassword(PasswordUpdateRequest passwordUpdateRequest) {
    User user = findAuthenticatedUser.getAuthenticatedUser();

    if (!isOldPasswordCorrect(user.getPassword(), passwordUpdateRequest.getOldPassword())) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Current password is incorrect");
    }

    if (!isNewPasswordConfirmed(passwordUpdateRequest.getNewPassword(),
            passwordUpdateRequest.getNewPassword2())) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "New passwords do not match");
    }

    if (!isNewPasswordDifferent(passwordUpdateRequest.getOldPassword(),
            passwordUpdateRequest.getNewPassword())) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Old and new passwords must be different");
    }

    user.setPassword(passwordEncoder.encode(passwordUpdateRequest.getNewPassword()));
    userRepository.save(user);
}
```

Old password correct?

Passwords match?

New password diff from old password?

© luv2code LLC

UserController

File: UserController.java

```
@RestController
@RequestMapping("/api/users")
@Tag(name = "User REST API Endpoints", description = "Operations related to info about current user")
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    <… Other endpoints …>

    @PutMapping("/password")
    public void passwordUpdate(@Valid @RequestBody PasswordUpdateRequest passwordUpdateRequest)
                throws Exception {
        userService.updatePassword(passwordUpdateRequest);
    }
}
```


© luv2code LLC

Todos

© luv2code LLC

Development Process

Step-By-Step

1. Create the TodoResponse & TodoRequest DTO objects
2. Create TodoRepository for CRUD operations.
3. Create TodoService interface (create todo)
    1. Create TodoServiceImpl for business logic (create todo)
4. Create TodoController for creating a todo

© luv2code LLC

TodoResponse DTO

File: TodoResponse.java

```
public class TodoResponse {

    private long id;

    private String title;

    private String description;

    private int priority;

    private boolean complete;

    public TodoResponse(long id, String title, String description, int priority, boolean complete) {
        this.id = id;
        this.title = title;
        this.description = description;
        this.priority = priority;
        this.boolean = boolean
    }

<…getters & setters…>
```


© luv2code LLC

TodoRequest DTO

File: TodoRequest.java

```
public class TodoRequest {

    @NotEmpty(message = "Title is mandatory")
    @Size(min = 3, max = 30, message = "Title must be at least 3 characters long")
    private String title;

    @NotEmpty(message = "Description is mandatory")
    @Size(min = 3, max = 30, message = "Description must be at least 3 characters long")
    private String description;

    @Min(1)
    @Max(5)
    private int priority;

    public TodoRequest(String title, String description, int priority) {
        this.title = title;
        this.description = description;
        this.priority = priority;
    }

<…getters & setters…>
```


© luv2code LLC

TodoRepository

File: TodoRepository.java

```
public interface TodoRepository extends CrudRepository<Todo, Long> {
}
```

- Automatically adds CRUD operations for the Todo entity.

© luv2code LLC

TodoService

File: TodoService.java

```
public interface TodoService {
    TodoResponse createTodo(TodoRequest todoRequest);
}
```

- Implement create todo business logic in our TodoServiceImpl class.

© luv2code LLC

TodoServiceImpl

File: TodoServiceImpl.java

```
@Service
public class TodoServiceImpl implements TodoService {

    private final TodoRepository todoRepository;
    private final FindAuthenticatedUser findAuthenticatedUser;

    public TodoServiceImpl(TodoRepository todoRepository, FindAuthenticatedUser findAuthenticatedUser) {
        this.todoRepository = todoRepository;
        this.findAuthenticatedUser = findAuthenticatedUser;
    }

    @Override
    @Transactional
    public TodoResponse createTodo(TodoRequest todoRequest) {
        User currentUser = findAuthenticatedUser.getAuthenticatedUser();

        Todo todo = <Convert request into Todo Entity>

        Todo savedTodo = todoRepository.save(todo);

        TodoResponse todoResponse = <Convert savedTodo into a TodoRequest>

        return todoResponse;
    }
}
```


© luv2code LLC

TodoController

File: TodoController.java

```
@RestController
@RequestMapping("/api/todos")
@Tag(name = "Todo REST API Endpoints", description = "Operations for managing user todos")
public class TodoController {

    private final TodoService todoService;

    public TodoController(TodoService todoService) {
        this.todoService = todoService;
    }

    @Operation(summary="Create todo for user", description="Create todo for the signed in user")
    @ResponseStatus(HttpStatus.CREATED)
    @PostMapping
    public TodoResponse createTodo(@Valid @RequestBody TodoRequest todoRequest) {
        return todoService.createTodo(todoRequest);
    }
}
```


© luv2code LLC

Todos Fetch

© luv2code LLC

Development Process

Step-By-Step

1. Update TodoRepository to find all todos based on authenticated user.
2. Update TodoService for getAllTodos
3. Update TodoServiceImpl to add business logic for getAllTodos
4. Update TodoController to fetch all todos.

© luv2code LLC

TodoRepository

File: TodoRepository.java

```
public interface TodoRepository extends CrudRepository<Todo, Long> {
     List<Todo> findByOwner(User owner);
}
```

- Find all todos based on the authenticated user.

© luv2code LLC

TodoService

File: TodoService.java

```
public interface TodoService {
    List<TodoResponse> getAllTodos();
    TodoResponse createTodo(TodoRequest todoRequest);
}
```

- Implement getAllTodos business logic in our TodoServiceImpl class.

© luv2code LLC

TodoServiceImpl

File: TodoServiceImpl.java

```
@Service
public class TodoServiceImpl implements TodoService {

    private final TodoRepository todoRepository;
    private final FindAuthenticatedUser findAuthenticatedUser;

    public TodoServiceImpl(TodoRepository todoRepository, FindAuthenticatedUser findAuthenticatedUser) {
        this.todoRepository = todoRepository;
        this.findAuthenticatedUser = findAuthenticatedUser;
    }

    @Override
    @Transactional(readOnly = true)
    public List<TodoResponse> getAllTodos() {
        User currentUser = findAuthenticatedUser.getAuthenticatedUser();

        return todoRepository.findByOwner(currentUser)
                .stream()
                .map(this::convertToTodoResponse)
                .toList();
    }

<…Other Code…>

}
```


© luv2code LLC

TodoController

File: TodoController.java

```
@RestController
@RequestMapping("/api/todos")
@Tag(name = "Todo REST API Endpoints", description = "Operations for managing user todos")
public class TodoController {

    private final TodoService todoService;

    public TodoController(TodoService todoService) {
        this.todoService = todoService;
    }

    @Operation(summary = "Get all todos for user", description = "Fetch all todos from signed in user")
    @ResponseStatus(HttpStatus.OK)
    @GetMapping
    public List<TodoResponse> getAllTodos() {
        return todoService.getAllTodos();
    }

<…Other Code…>

}
```


© luv2code LLC

Todos Complete

© luv2code LLC

Development Process

Step-By-Step

1. Update TodoRepository to find specific todo by ID and owner.
2. Update TodoService for toggleTodoCompletion
3. Update TodoServiceImpl to add business logic for toggleTodoCompletion
4. Update TodoController to have this new endpoint

© luv2code LLC

TodoRepository

File: TodoRepository.java

```
public interface TodoRepository extends CrudRepository<Todo, Long> {
     List<Todo> findByOwner(User owner);
     Optional<Todo> findByIdAndOwner(Long id, User owner);
}
```

- Find specific todo based on id and owner.

© luv2code LLC

TodoService

File: TodoService.java

```
public interface TodoService {
    List<TodoResponse> getAllTodos();
    TodoResponse createTodo(TodoRequest todoRequest);
    TodoResponse toggleTodoCompletion(long id);
}
```

- Implement toggleTodoCompletion business logic in our TodoServiceImpl class.

© luv2code LLC

TodoServiceImpl

File: TodoServiceImpl.java

```
@Override
@Transactional
public TodoResponse toggleTodoCompletion(long id) {
    User currentUser = findAuthenticatedUser.getAuthenticatedUser();

    Optional<Todo> todo = todoRepository.findByIdAndOwner(id, currentUser);

    if (todo.isEmpty()) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Todo not found");
    }

    todo.get().setComplete(!todo.get().isComplete());
    Todo updatedTodo = todoRepository.save(todo.get());

    return convertToTodoResponse(updatedTodo);
}
```


© luv2code LLC

TodoController

File: TodoController.java

```
@RestController
@RequestMapping("/api/todos")
@Tag(name = "Todo REST API Endpoints", description = "Operations for managing user todos")
public class TodoController {

    private final TodoService todoService;

    public TodoController(TodoService todoService) {
        this.todoService = todoService;
    }

    @Operation(summary = "Update todo for user", description = "Update todo for the signed in user")
    @ResponseStatus(HttpStatus.OK)
    @PutMapping("/{id}")
    public TodoResponse toggleTodoCompletion(@PathVariable @Min(1) long id) {
        return todoService.toggleTodoCompletion(id);
    }

<…Other Code…>

}
```


© luv2code LLC

Todos Delete

© luv2code LLC

Development Process

Step-By-Step

1. Update TodoService for delete.
2. Update TodoServiceImpl to add business logic for delete.
3. Update TodoController to have this new endpoint

© luv2code LLC

TodoService

File: TodoService.java

```
public interface TodoService {
    List<TodoResponse> getAllTodos();
    TodoResponse createTodo(TodoRequest todoRequest);
    TodoResponse toggleTodoCompletion(long id);
    void deleteTodo(long id);
}
```

- Implement delete business logic in our TodoServiceImpl class.

© luv2code LLC

TodoServiceImpl

File: TodoServiceImpl.java

```
@Override
@Transactional
public void deleteTodo(long id) {
    User currentUser = findAuthenticatedUser.getAuthenticatedUser();

    Optional<Todo> todo = todoRepository.findByIdAndOwner(id, currentUser);

    if (todo.isEmpty()) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Todo not found");
    }
    todoRepository.delete(todo.get());
}
```


© luv2code LLC

TodoController

File: TodoController.java

```
@RestController
@RequestMapping("/api/todos")
@Tag(name = "Todo REST API Endpoints", description = "Operations for managing user todos")
public class TodoController {

    private final TodoService todoService;

    public TodoController(TodoService todoService) {
        this.todoService = todoService;
    }

    @Operation(summary = "Delete todo for user", description = "Delete todo for the signed in user")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    @DeleteMapping("/{id}")
    public void deleteTodo(@PathVariable @Min(1) long id) {
        todoService.deleteTodo(id);
    }

<…Other Code…>

}
```


© luv2code LLC

Admins

© luv2code LLC

Development Process

Step-By-Step

1. Create new AdminService
2. Create AdminServiceImpl for business logic.
3. Update security config to check for Admin role for admin endpoints
4. Create AdminController to have this new endpoint

© luv2code LLC

AdminService

File: AdminService.java

```
public interface AdminService {
    List<UserResponse> getAllUsers();
    UserResponse promoteToAdmin(long userId);
    void deleteNonAdminUser(long id);
}
```

- Admin users will have granted permissions to these three new functions.
- We now need to add the business logic for these by creating a new class AdminServiceImpl.

© luv2code LLC

AdminServiceImpl

File: AdminServiceImpl.java

```
@Service
public class AdminServiceImpl implements AdminService {

    private final UserRepository userRepository;

    public AdminServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    @Transactional(readOnly = true)
    public List<UserResponse> getAllUsers() {
        return StreamSupport.stream(userRepository.findAll().spliterator(), false)
                .map(this::convertToUserResponse).toList();
    }
}
```


© luv2code LLC

AdminServiceImpl

File: AdminServiceImpl.java

```
@Override
@Transactional
public UserResponse promoteToAdmin(long userId) {
    Optional<User> user = userRepository.findById(userId);

    if (user.isEmpty() || user.get().getAuthorities().stream().anyMatch(authority -> "ROLE_ADMIN"
            .equals(authority.getAuthority()))) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "User does not exist or already an admin");
    }

    List<Authority> authorities = new ArrayList<>();
    authorities.add(new Authority("ROLE_EMPLOYEE"));
    authorities.add(new Authority("ROLE_ADMIN"));
    user.get().setAuthorities(authorities);

    User savedUser = userRepository.save(user.get());

    return convertToUserResponse(savedUser);
}
```


© luv2code LLC

AdminServiceImpl

File: AdminServiceImpl.java

```
@Override
@Transactional
public void deleteNonAdminUser(long userId) {

    Optional<User> user = userRepository.findById(userId);

    if (user.isEmpty() || user.get().getAuthorities().stream().anyMatch(authority -> "ROLE_ADMIN"
            .equals(authority.getAuthority()))) {

        throw new ResponseStatusException(HttpStatus.BAD_REQUEST,
                "User does not exist or already an admin");
    }

    userRepository.delete(user.get());
}
```


© luv2code LLC

SecurityConfig

File: SecurityConfig.java

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests(configurer ->
            configurer
                    .requestMatchers("/api/auth/**", "/swagger-ui/**", "/v3/api-docs/**",
                            "/swagger-resources/**", "/webjars/**", "/docs").permitAll()
                    .requestMatchers("/api/admin/**").hasRole("ADMIN").anyRequest().authenticated());

    http.csrf(csrf -> csrf.disable());

    http.exceptionHandling(exceptionHandling -
            exceptionHandling
                    .authenticationEntryPoint(  enticationEntryPoint()));

    http.sessionManagement(session ->
            session.sessionCreationPo                    STATELESS))

    http.addFilterBefore(jwtAuthentica                    ordAuthentic

    return http.build();
}
```

NEW requestMatcher

NEW hasRole

© luv2code LLC

AdminController

File: AdminController.java

```
@Tag(name = "Admin REST API Endpoints", description = "Operations related to a admin")
@RestController
@RequestMapping("/api/admin")
public class AdminController {

    private final AdminService adminService;

    public AdminController(AdminService adminService) {
        this.adminService = adminService;
    }

    @Operation(summary = "Get all users", description = "Retrieve a list of all users in the system")
    @ResponseStatus(HttpStatus.OK)
    @GetMapping
    public List<UserResponse> getAllUsers() {
        return adminService.getAllUsers();
    }
}
```


© luv2code LLC

AdminController

File: AdminController.java

```
@Tag(name = "Admin REST API Endpoints", description = "Operations related to a admin")
@RestController
@RequestMapping("/api/admin")
public class AdminController {

    private final AdminService adminService;

    public AdminController(AdminService adminService) {
        this.adminService = adminService;
    }

    @Operation(summary = "Promote user to admin", description = "Promote user to admin role")
    @ResponseStatus(HttpStatus.OK)
    @PutMapping("/{userId}/role")
    public UserResponse promoteToAdmin(@PathVariable @Min(1) long userId) {
        return adminService.promoteToAdmin(userId);
    }

<… Other Code …>
}
```


© luv2code LLC

AdminController

File: AdminController.java

```
@Tag(name = "Admin REST API Endpoints", description = "Operations related to a admin")
@RestController
@RequestMapping("/api/admin")
public class AdminController {

    private final AdminService adminService;

    public AdminController(AdminService adminService) {
        this.adminService = adminService;
    }

    @Operation(summary = "Delete user", description = "Delete a non-admin user from the system")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    @DeleteMapping("/{userId}")
    public void deleteUser(@PathVariable @Min(1) long userId) {
        adminService.deleteNonAdminUser(userId);
    }

<… Other Code …>
}
```


© luv2code LLC

Thank you!

© luv2code LLC

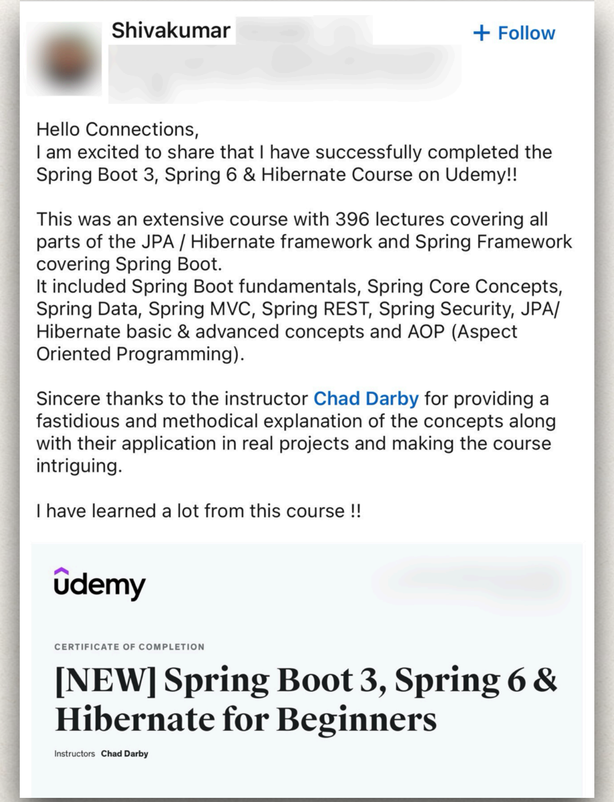
Accomplishments

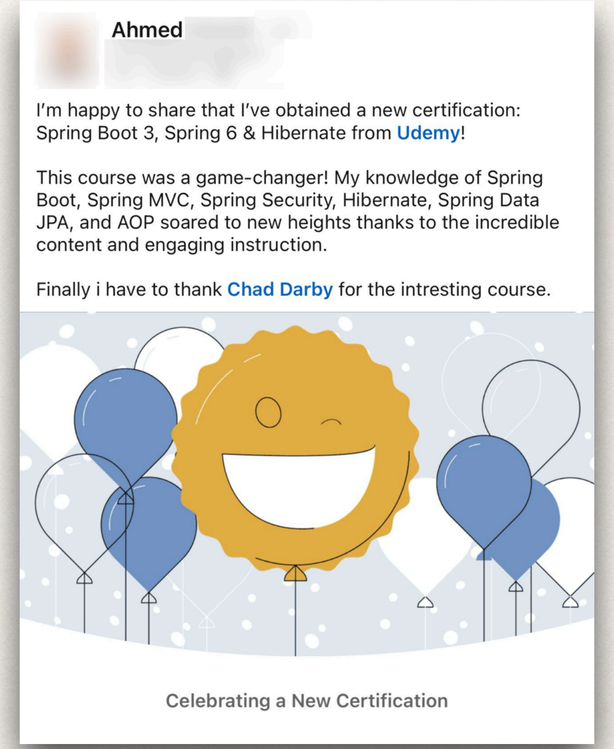
- Create REST APIs / Web Services with Spring
- Discuss REST concepts, JSON and HTTP messaging
- Install REST client tool: Swagger
- Develop REST APIs / Web Services with **@RestController**
- Build a CRUD interface to the database with Spring REST
- Create and use JWTs for authentication & authorization
- Handle roles and admin status

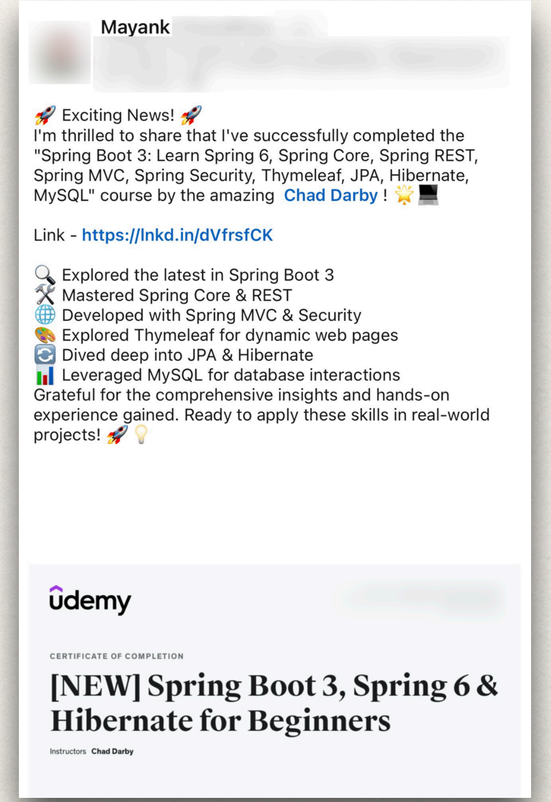
© luv2code LLC

Share Your Accomplishment

- Download your course certificate and share it on social media!

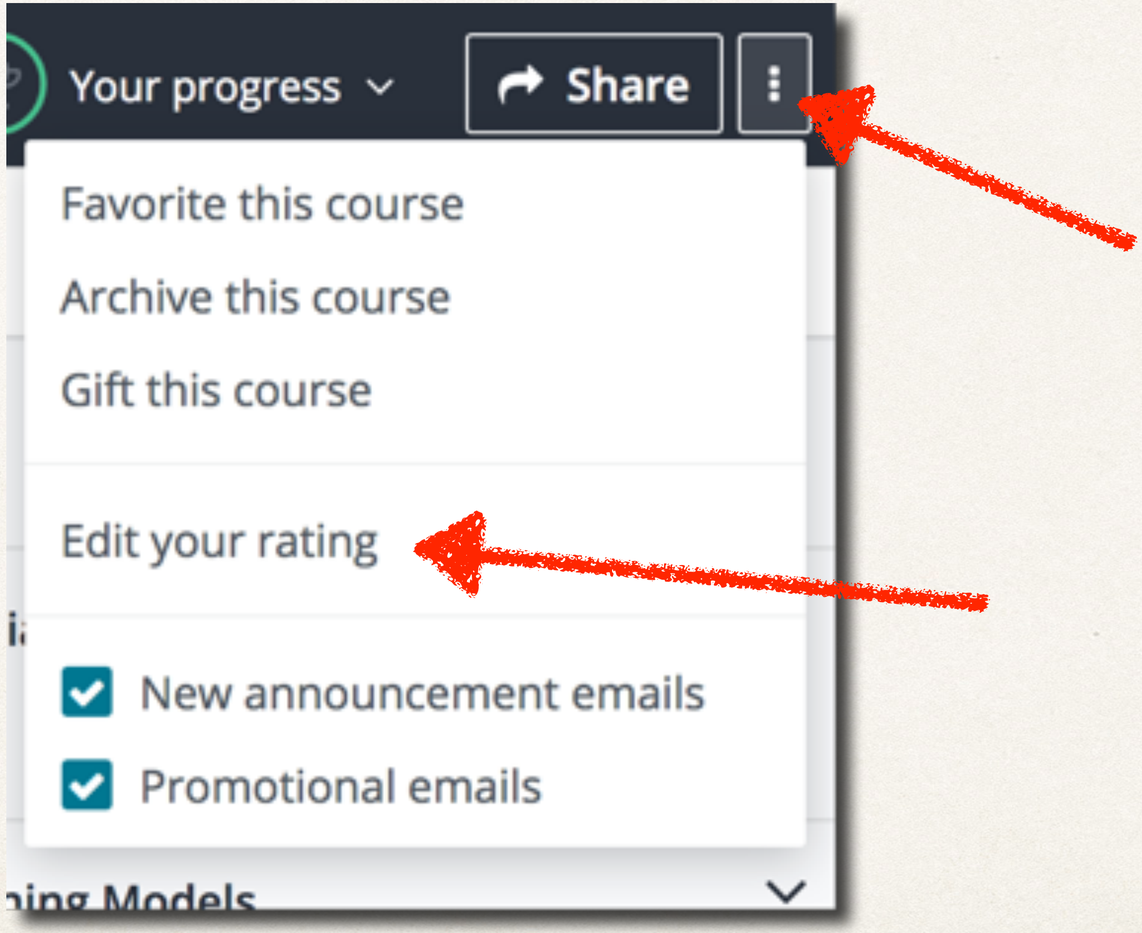






© luv2code LLC

Please Rate the Course



Contact Us

Eric Roby

www.codingwithroby.com

Chád Darby

www.luv2code.com