

# Large-Scale and Multi-Structured Databases

***RESTful APIs***

***Java Spring Boot***

***OpenAPI***

***Swagger UI***

Eng. Alessio Schiavo

Prof Pietro Ducange

# Project Overview – Event Description Enhancer Service

- Objective: Introduce the Event Description Enhancer Service, a **REST Service implemented in Java using Spring Boot & MySQL**
- **Functional Requirements:**
  - Create, read, update, and delete events.
  - Auto-generate event descriptions using OpenAI API.
  - Admin-only feature to delete events.
- **Non-Functional Requirements:**
  - **RESTful architecture adhering to REST principles**, including correct usage of HTTP methods with well structured endpoint URLs
  - **Swagger** REST API documentation according to OpenAPI standard
  - **MySQL** for persistent storage.
  - Basic Authentication to secure access
  - **Testing with Swagger UI and Postman**

# Introduction to REST APIs

- **What are REST APIs?**
  - REST (Representational State Transfer) is an architectural style used to design networked applications, particularly web services
  - REST APIs facilitate communication between client and server **over HTTP by using standard HTTP methods** (GET, POST, PUT, DELETE) to operate on resources

**REST is not a specification.** It is a **loose set of rules** that has been **the de facto standard for building web API** since the early 2000s.

# Introduction to REST APIs

- An API that follows the REST standard is called a **RESTful API**. Some real-life examples are **Stripe**, and **Google Maps**.
- **Resources:**
  - A resource is any piece of information that can be named, such as a user, a document, or a product. Each resource is represented by a **URI** (Uniform Resource Identifier)
  - **A RESTful API organizes resources into a set of unique URIs or Uniform Resource Identifiers**

## URIs

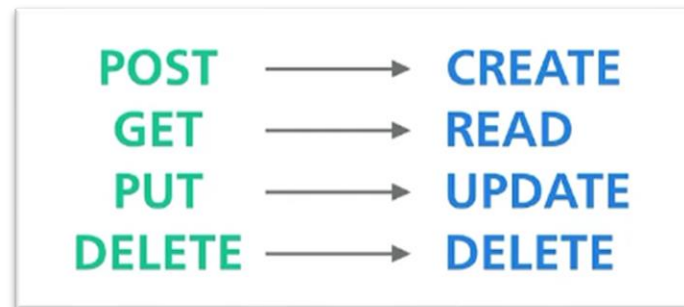
`https://example.com/api/v3/products`

`https://example.com/api/v3/users`

# HTTP Methods in REST APIs

- **HTTP Methods:**

- **GET:** Retrieve a resource (e.g., GET /events)
- **POST:** Create a new resource (e.g., POST /events)
- **PUT:** Update an existing resource (e.g., PUT /events/{id})
- **DELETE:** Remove a resource (e.g., DELETE /events/{id})



- **Additional Methods:**

- **PATCH:** For partial updates.
- **OPTIONS:** For available HTTP methods.

# Introduction to REST APIs

- The **resources should be grouped by noun and not verb**. An API to get all products should be:

Correct

`https://example.com/api/v3/products`

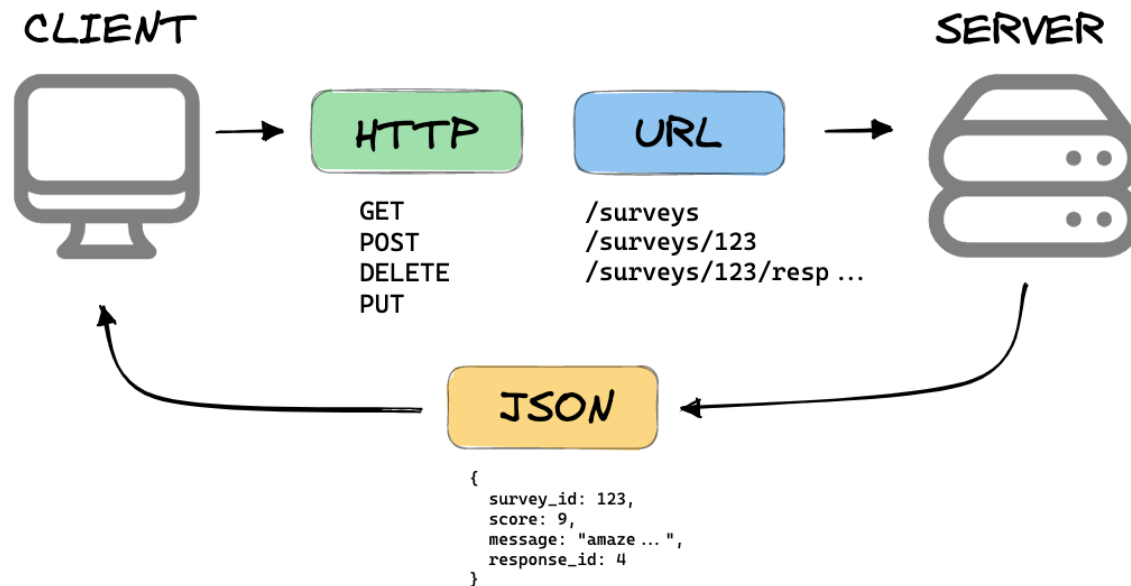
`https://example.com/api/v3/getAllProducts`

Wrong

# Introduction to REST APIs

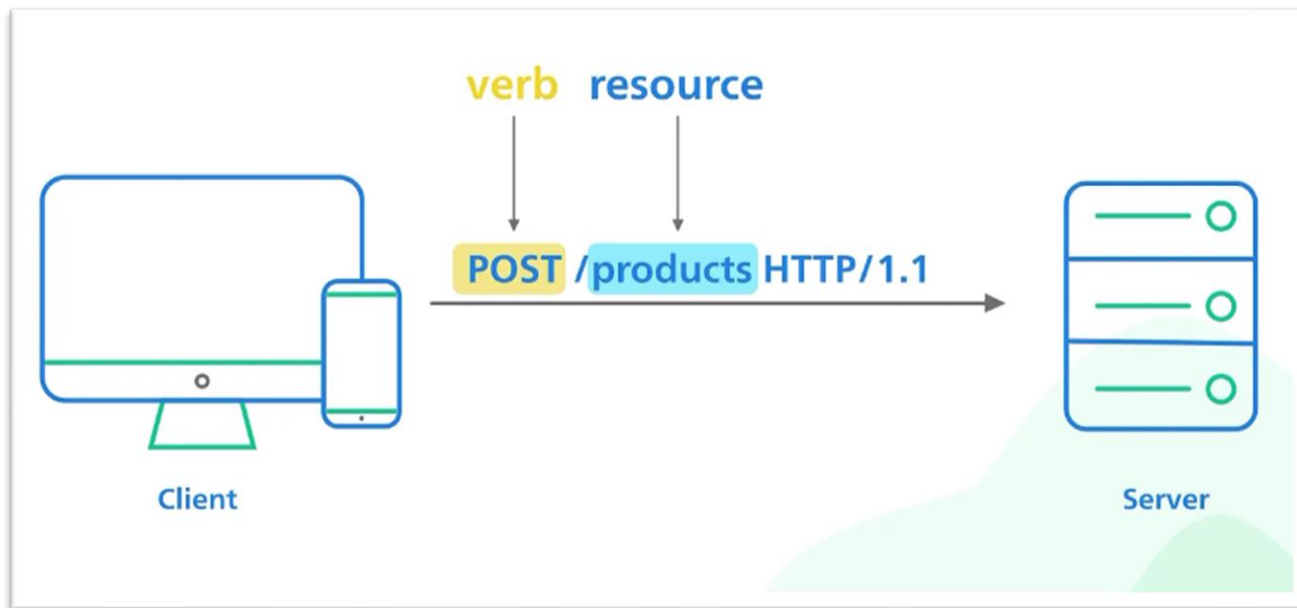
- **Clients** interact with resources by sending requests to manipulate or retrieve them, and the **server** responds with **the current state of the resource or the result of the action**

## WHAT IS A REST API?



# An example of REST request

- The line contains **the URI for the resource we'd like to access**. The URI is **preceded by an HTTP verb which tells the server what we want to do with the resource**:





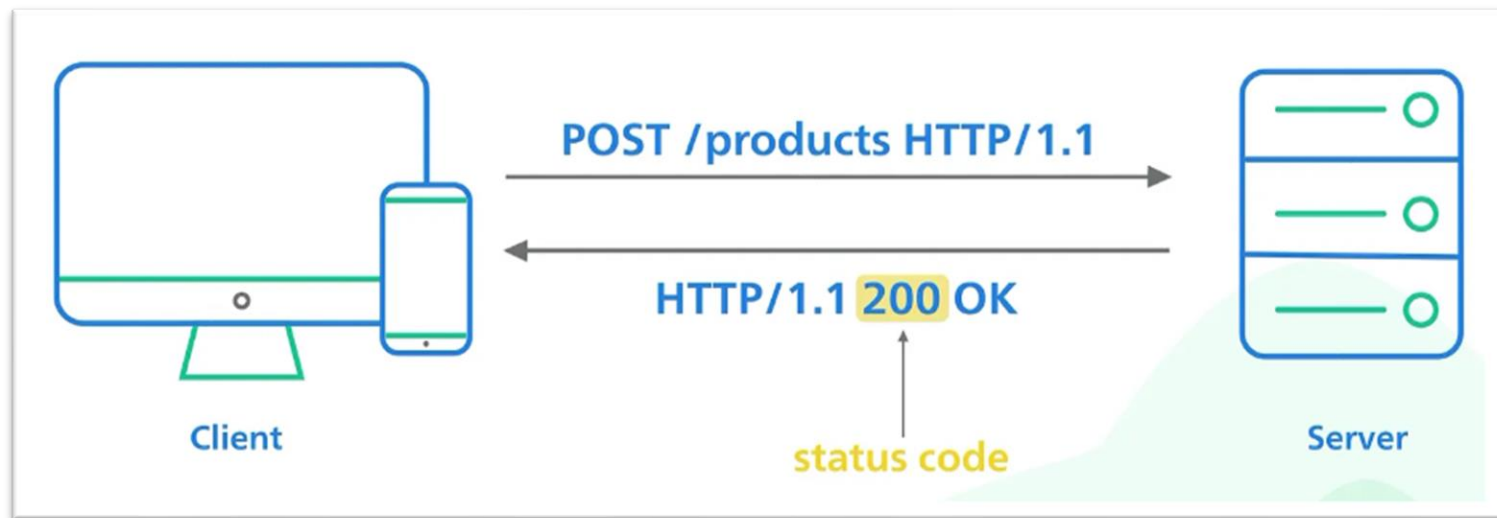
# An example of REST request

- In the body of these requests there could be an optional HTTP request **body** that contains a **custom payload of data**, usually **encoded in JSON**:



# An example of REST request

- The server receives the request, processes it, and **formats the result into a response**
- The **first line of the response contains the HTTP status code** to tell the client what happened to the request:



- **A well-implemented RESTful API returns proper HTTP status codes!**

# A REST implementation should be Stateless

- There is a critical attribute of REST that is worth discussing more
- **A REST implementation should be stateless.** It means that the two parties don't need to store any information about each other and **every request and response is independent from all others**
- This leads to web applications that **are easy to scale** and well-behaved

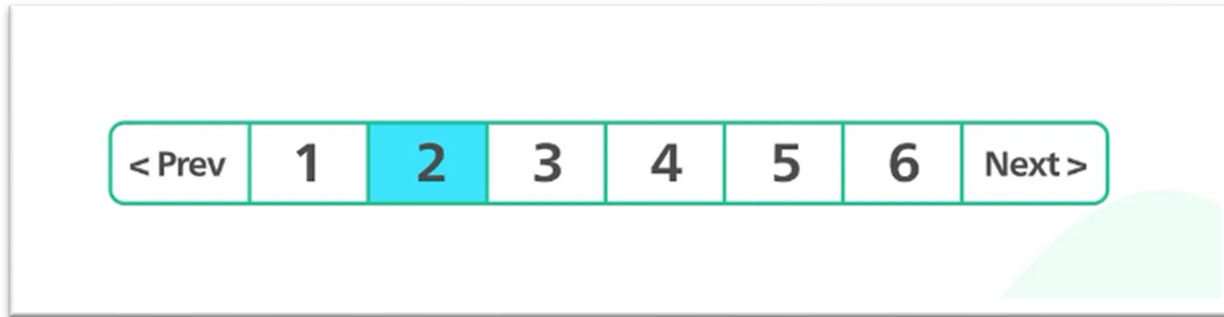
# Idempotence

- We said “could choose to retry” because **some actions are not idempotent** and those require extra care when retrying
- **When an API is idempotent, making multiple identical requests has the same effect as making a single request.** This is usually not the case for a POST request to create a new resource

POST	Not idempotent
GET	Idempotent
PUT	Idempotent
DELETE	Idempotent

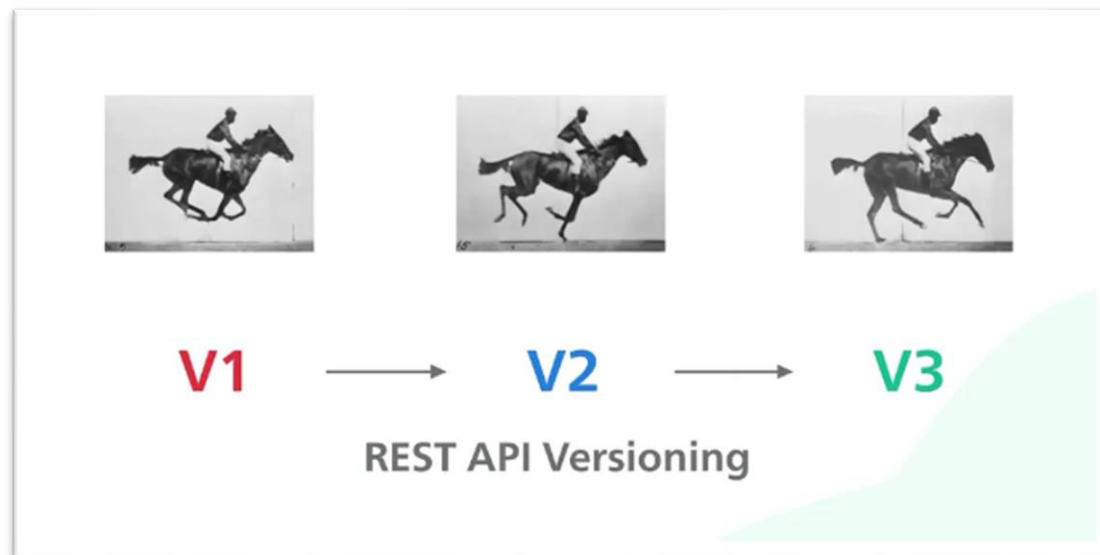
# Use pagination

- If an API endpoint returns a **huge amount of data**, use **pagination**
- A common pagination scheme uses **limit and offset**, If they are not specified, the **server should assume sensible default values**



# REST API Versioning

- Lastly, **versioning of an API is very important**
- Versioning allows an implementation **to provide backward compatibility** so that if we introduce breaking changes from one version to another, **consumers get enough time to move to the next version**
- There are many ways to version an API. **The most straightforward is to prefix the version before the resource on the URI**



# Best Practices for Implementing REST Services

- **Best Practices to implement RESTful APIs:**
  - **Use nouns** in endpoints (e.g., /events instead of /getEvents)
  - Leverage HTTP **status codes** (e.g., 200 OK, 201 Created, 400 Bad Request)

200-level	Success
400-level	Something wrong with our request
500-level	Something wrong at the server level

- Statelessness: Each request should be self-contained
- **Version your API** (e.g., /api/v1/events)
- **Secure your API** with proper authentication and authorization

# Java Spring Boot





# What is Spring Boot?

- Spring Boot is a **framework that simplifies the development of Java-based RESTful web services**
- Spring Boot is a convention-over-configuration framework, meaning **it reduces the need for boilerplate code and configuration, allowing developers to focus more on writing business logic**. With Spring Boot, you can set up and run an application with minimal configuration and setup.
- It **provides built-in support for HTTP methods, auto-configuration**, and embedded servers like Tomcat

# What is Spring Boot useful for?

- **Building RESTful Web Services:** It allows you to build REST APIs quickly and provides everything you need, from managing requests, routing them, to managing responses.
- **Microservices:** Spring Boot is popular for microservice architectures, where each service is designed as an independent, loosely-coupled component.

# Spring Boot Project Structure

When you create a Spring Boot project, you typically follow a standard project structure that **helps organize your code in a clean and modular way**:

```
your-project/
├── src
│   ├── main
│   │   ├── java
│   │   │   ├── com
│   │   │   │   ├── example
│   │   │   │   │   ├── yourproject
│   │   │   │   │   │   ├── controller
│   │   │   │   │   │   ├── model
│   │   │   │   │   │   ├── repository
│   │   │   │   │   │   ├── service
│   │   │   │   │   │   ├── config
│   │   │   │   │   │   └── YourProjectApplication.java
│   │   │   └── resources
│   │   │       ├── application.properties (or application.yml)
│   │   │       ├── static (optional)
│   │   │       ├── templates (optional)
│   │   │       └── schema.sql and data.sql (optional)
```

# Purpose of Each Package in Spring Boot Project

- **Controller:** Maps API requests to handler methods
- **Model:** Entity classes that map to database tables
- **Service:** Contains business logic
- **Repository:** Interfaces for CRUD operations via Spring Data JPA, provides data access logic
- **Config:** Configuration classes
- **Resources:** External configurations (e.g., application.properties).

# Spring Boot Application Main Class

## 1. `src/main/java:`

This folder contains all the Java source code for your Spring Boot application. **All the components such as controllers, services, repositories, and configurations go here. It follows a hierarchical structure with packages to organize different layers of the application**

## - `com/example/yourproject/YourProjectApplication.java:`

This is **the main class of your Spring Boot application, annotated with `@SpringBootApplication`. It acts as the entry point of the application and contains the `main()` method that bootstraps the Spring Boot application.**

# Spring Boot Application Main Class

Main Boot String class **example:**

```
3  import org.springframework.boot.SpringApplication;
4  import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6  @SpringBootApplication
7  public class EventDescriptionEnhancerApplication {
8
9      public static void main(String[] args) {
10         SpringApplication.run(EventDescriptionEnhancerApplication.class, args);
11     }
12
13 }
14
15
```

# Spring Boot Project Structure: controller

## controller/

This package contains the **REST controllers** that handle incoming HTTP requests and return responses. It's where you define the logic of REST request handlers, using the main annotations in Spring Boot:

- **@RestController**: indicates that the class will handle REST requests and return data as JSON or XML
- **@GetMapping, @PostMapping, @RequestMapping, @DeleteMapping**: Map HTTP methods to Java methods
- **@RequestBody** and **@ResponseBody**: Handle JSON data in the request body and response body
- **@PathVariable** and **@RequestParam**: Capture path variables and query parameters from the URL

# Spring Boot Project Structure: controller

Controller class **example**:

```
11  @RestController
12  @RequestMapping("/api/events")
13  @RequiredArgsConstructor
14  public class EventController {
15      private final EventService eventService;
16
17      @GetMapping
18      public List<Event> getAllEvents() {
19          // logic here
20      }
21
22      @GetMapping("/{id}")
23      public ResponseEntity<Event> getEventById(@PathVariable Long id) {
24          // logic here
25      }
```



# Example Spring Boot GET Method

```
@GetMapping("/api/events")  
public List<Event> getAllEvents() {  
    return eventService.getAllEvents();  
}
```

Explanation: Maps GET requests to retrieve a list of events.

# Example Spring Boot POST Method

```
@PostMapping("/api/events")  
public Event createEvent(@RequestBody Event event) {  
    return eventService.createEvent(event);  
}
```

Explanation: Maps POST requests to create a new event.

# Example Spring Boot PUT Method

```
@PutMapping("/api/events/{id}")
public ResponseEntity<Event> updateEvent(@PathVariable Long
id, @RequestBody Event updatedEvent) {
    Event event = eventService.updateEvent(id, updatedEvent);
    return ResponseEntity.ok(event);
}
```

Explanation: Maps PUT requests to update an event by ID.

# Example Spring Boot DELETE Method

```
@DeleteMapping("/api/events/{id}")
public ResponseEntity<Void> deleteEvent(@PathVariable Long
id) {
    eventService.deleteEvent(id);
    return ResponseEntity.noContent().build();
}
```

Explanation: Maps DELETE requests to remove an event by ID.

# Spring Boot Project Structure: model

## model/

This package holds the **domain models** or **entity classes** representing the business objects in your application. If you are using a database, this is where you define **JPA entities**. **Example:**

```
3  import lombok.Data;
4      import jakarta.persistence.*;
5  import java.time.LocalDate;
6
7  @Data
8  @Entity
9  public class Event {
10      @Id
11      @GeneratedValue(strategy = GenerationType.IDENTITY)
12      private Long id;
13
14      private String title;
15
16      @Column(columnDefinition = "TEXT")
17      private String description;
18
19      private LocalDate date;
20  }
```

# Spring Boot Project Structure: repository

## repository/

This package is where the **repository interfaces** live. These interfaces are typically based on **Spring Data JPA** and provide data access logic (e.g., database operations like CRUD). **Example:**

```
public interface EventRepository extends JpaRepository<Event, Long> {  
}
```

**Usage example** within the business logic class under the service package:

```
public List<Event> getAllEvents() {  
    return eventRepository.findAll();  
}  
  
public void deleteAllEvents() {  
    eventRepository.deleteAll();  
}
```

# Spring Boot Project Structure: service

service/

This **package contains the business logic** and is often used to mediate between the controllers and repositories. Services **usually contain the main logic for handling data processing** and are annotated with `@Service`. **Example:**

```
17  @Service
18  @RequiredArgsConstructor
19  public class EventService {
20      private final EventRepository eventRepository;
21
22      // for OpenAI API calls
23      public static final String API_URL = "https://ap:
24      public static final String API_KEY = "sk-proj-F2A
25
26      public List<Event> getAllEvents() {
27          return eventRepository.findAll();
28      }
29
30      public void deleteAllEvents() {
31          eventRepository.deleteAll();
32      }
```

# Spring Boot Project Structure: config

## config/

This **package is where you put configuration classes**. These could be related to **things like security**, custom beans, or Swagger configuration. You can use `@Configuration` annotations in these classes.

The `config/` package in a Spring Boot project is **primarily used to centralize and organize all the configuration classes of the application**.

It **helps separate the configuration logic** (which handles the application's specific settings and behavior) **from the business logic of the application itself**.

If for example you want to add an authentication mechanism to your REST Service you need to implement a class under this package.



# OpenAPI



# What is OpenAPI ?

- The **OpenAPI Specification** (OAS) is a **standardized format** for describing RESTful APIs.
- It allows developers to define the endpoints, request and response structures, data types, authentication methods, and other essential aspects of an API in a machine-readable format (typically JSON or YAML)
- **This specification is often used to generate API documentation, client SDKs, and server stubs automatically.**

# What is Springdoc OpenAPI ?

- Springdoc OpenAPI is a library that helps **integrate the OpenAPI specification into Spring Boot applications**
- It **automatically generates OpenAPI-compliant API documentation for your REST services**, based on the annotations in your Spring Boot code (like `@RestController`, `@GetMapping`, etc.)
- It **also provides an interactive Swagger UI for testing and interacting with the API directly from a browser**

# Add Springdoc OpenAPI dependency

- **To use Springdoc OpenAPI in a Spring Boot Project**, you need to add the following dependency to your pom.xml:

```
<!-- Swagger for API documentation -->  
<dependency>  
  <groupId>org.springdoc</groupId>  
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>  
  <version>2.3.0</version>  
</dependency>
```

# Swagger



# Introduction to Swagger

- **What is Swagger?**
  - Swagger is a set of open-source tools built around the OpenAPI Specification that help design, build, document, and consume RESTful APIs
  - In essence, Swagger provides a way to generate interactive documentation for your APIs and makes it easy for developers to explore, understand, and test the API endpoints
- **Why use Swagger?**
  - Simplifies testing and documenting APIs
  - Provides an **interactive Swagger UI for API testing**

# Swagger UI

## Swagger UI

Swagger UI allows anyone — be it your development team or your end consumers — to visualize and interact with the API's resources without having any of the implementation logic in place. It's automatically generated from your OpenAPI (formerly known as Swagger) Specification, with the visual documentation making it easy for back end implementation and client side consumption.

<https://swagger.io/tools/swagger-ui/>

# Swagger UI Demo: Swagger Petstore

<b>pet</b> Everything about your Pets		
POST	/pet/{petId}/uploadImage	uploads an image
POST	/pet	Add a new pet to the store
PUT	/pet	Update an existing pet
GET	/pet/findByStatus	Finds Pets by status
GET	<del>/pet/findByTags</del>	Finds Pets by tags
GET	/pet/{petId}	Find pet by ID
POST	/pet/{petId}	Updates a pet in the store with form data
DELETE	/pet/{petId}	Deletes a pet

[https://petstore.swagger.io/?\\_gl=1\\*1e49e4u\\*\\_gcl\\_au\\*MTg0NzcwMzQwNS4xNzI4OTIyNTI5#/](https://petstore.swagger.io/?_gl=1*1e49e4u*_gcl_au*MTg0NzcwMzQwNS4xNzI4OTIyNTI5#/)



# Swagger UI Demo: Swagger Petstore

## store Access to Petstore orders

**GET** `/store/inventory` Returns pet inventories by status

**POST** `/store/order` Place an order for a pet

**GET** `/store/order/{orderId}` Find purchase order by ID

**DELETE** `/store/order/{orderId}` Delete purchase order by ID

[https://petstore.swagger.io/?\\_gl=1\\*1e49e4u\\*\\_gcl\\_au\\*MTg0NzcwMzQwNS4xNzI4OTIyNTI5#/](https://petstore.swagger.io/?_gl=1*1e49e4u*_gcl_au*MTg0NzcwMzQwNS4xNzI4OTIyNTI5#/)

# Swagger live demo



# Home Assignment:

## Be Prepared for the Next Lesson



# Home Assignment #1

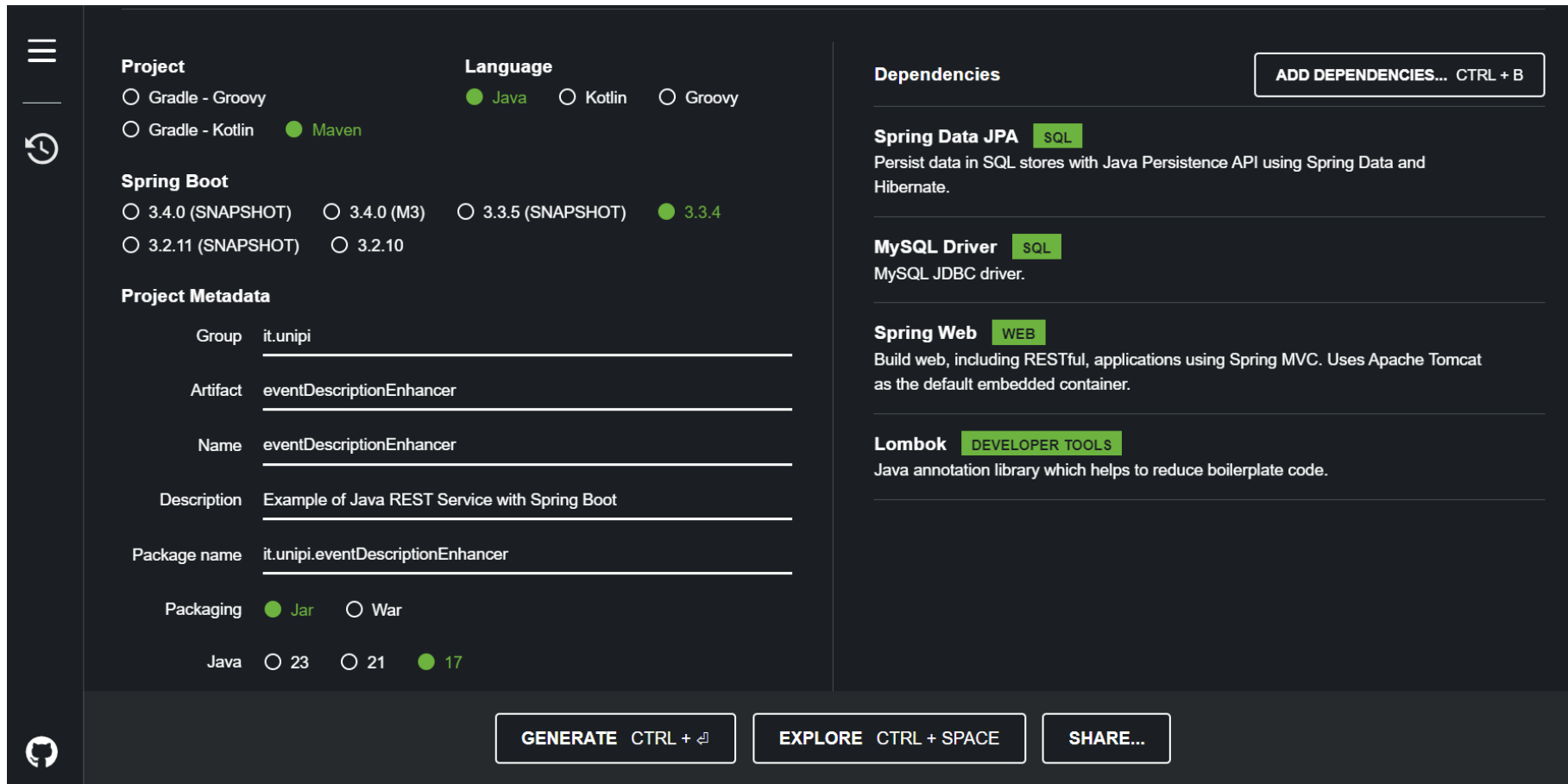
- **1. Explore RESTful API Design:**
  - Review REST API design principles (HTTP methods, status codes, statelessness, and resource-based URIs)
  - **Task:** Write down **three examples of REST API endpoints** for a hypothetical "Task Management" system.  
**Explain which HTTP method would be used and why.** For example:
    - **GET /tasks** - Retrieve all tasks
    - **POST /tasks** - Create a new task
    - **DELETE /tasks/{id}** - Delete a specific task

# Home Assignment #2

- **2. Hands-On with Spring Boot**
  - **Set Up a Simple REST API:** Create a simple Spring Boot project using the **Spring Initializer** (<https://start.spring.io/>), **MySQL** as db and **Spingdoc OpenAPI** to automatically generate OpenAPI-compliant API documentation
    - **Objective:** Create a REST controller for a "Product" entity with basic CRUD operations (GET, POST, PUT, DELETE)
    - Use `@RestController`, `@GetMapping`, `@PostMapping`, `@PutMapping`, and `@DeleteMapping` to implement the REST endpoints
    - **Try using Swagger UI:** Test your API endpoints by making requests using Swagger UI

# Home Assignment #2

**Starting point:** generate Spring Boot project with Spring Initializr and import it your IDE



The screenshot shows the Spring Initializr web application interface. It is divided into several sections: Project, Language, Spring Boot, Project Metadata, Dependencies, and a bottom bar with action buttons.

**Project**

- ☐ Gradle - Groovy
- ☐ Gradle - Kotlin
- ☒ Maven

**Language**

- ☒ Java
- ☐ Kotlin
- ☐ Groovy

**Spring Boot**

- ☐ 3.4.0 (SNAPSHOT)
- ☐ 3.4.0 (M3)
- ☐ 3.3.5 (SNAPSHOT)
- ☒ 3.3.4
- ☐ 3.2.11 (SNAPSHOT)
- ☐ 3.2.10

**Project Metadata**

Group:

Artifact:

Name:

Description:

Package name:

Packaging: ☒ Jar ☐ War

Java: ☐ 23 ☐ 21 ☒ 17

**Dependencies**

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

**MySQL Driver** SQL  
MySQL JDBC driver.

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

**Lombok** DEVELOPER TOOLS  
Java annotation library which helps to reduce boilerplate code.

**Buttons:**

- ADD DEPENDENCIES... CTRL + B
- GENERATE CTRL + G
- EXPLORE CTRL + SPACE
- SHARE...

# Suggested documentation

<https://www.redhat.com/it/topics/api/what-is-a-rest-api>

<https://swagger.io/resources/articles/best-practices-in-api-design/>

<https://swagger.io/specification/>

<https://spring.io/projects/spring-boot>