KOENIG

step forward

# Spring Book with Full Stack Design and Development

# Prerequisites

- Basic understandings on Java Programming, Spring Framework, Spring Boot

# Introduction to REST API Principles

# Understanding REST Architecture

# Understanding REST Architecture

- Principles of REST: Statelessness, client-server separation, uniform interfaces.
- HTTP methods: GET, POST, PUT, DELETE, PATCH, and their appropriate use cases.
- API design best practices: Resource naming, versioning, and HATEOAS.

# What is REST Architecture?

- REST (Representational State Transfer) is an architectural style for designing networked applications.
- It is based on a set of principles that enable communication between clients (such as web browsers or mobile apps) and servers over the internet.
- RESTful systems use standard HTTP methods to perform operations on resources, making them simple, scalable, and stateless.
- The three key principles of REST are Statelessness, Client-Server Separation, and Uniform Interfaces.

# Principles of REST: Statelessness, client-server separation, uniform interfaces.

- Statelessness
  - Each request from a client to a server must contain all the necessary information to process it.
  - The server does not store any client-specific session data between requests.
- Key Characteristics of Statelessness:
  - No session storage on the server.
  - Each request is independent and self-contained.
  - Clients must include authentication details (like API keys, JWT tokens) with every request.
  - Improves scalability since the server does not maintain session state.

# Principles of REST: Statelessness, client-server separation, uniform interfaces.

- Client-Server Separation
  - The client (frontend) and server (backend) operate independently.
  - The client sends requests, the server processes them and responds with data.
- Key Characteristics of Client-Server Separation:
  - **Independence**: Clients and servers can evolve separately.
  - **Scalability**: Multiple clients (web, mobile) can consume the same REST API.
  - **Security**: Servers enforce authentication and authorization, keeping client-side logic separate.
- Example Architecture:
  - Client (React, Angular, iOS, Android) → Requests Data
  - Server (Node.js, Spring Boot) → Processes Requests and Returns Data

# Principles of REST: Statelessness, client-server separation, uniform interfaces.

- Uniform Interface
  - A REST API should have a consistent and standardized way of accessing resources, making it predictable and easy to use.
- Key Constraints of Uniform Interface:
  - Resource Identification: Every resource has a unique URI (e.g., /users/1).
  - Standard HTTP Methods: RESTful APIs use:
    - GET → Retrieve data
    - POST → Create a new resource
    - PUT → Update an existing resource
    - DELETE → Remove a resource
  - Self-descriptive Messages: Responses should contain sufficient information for the client to process them.

# HTTP methods: GET, POST, PUT, DELETE, PATCH, and their appropriate use cases.

- HTTP methods define the actions that can be performed on resources in a RESTful API.
- The most commonly used methods are GET, POST, PUT, DELETE, and PATCH.
- GET – Retrieve Data
  - Fetch data from the server without modifying it.
  - Use Cases:
    - Retrieve a list of resources (GET /products)
    - Retrieve a single resource (GET /products/1)

# HTTP methods: GET, POST, PUT, DELETE, PATCH, and their appropriate use cases.

- POST – Create a New Resource
  - Submit data to the server to create a new resource.
  - Use Cases:
    - Register a new user (POST /users)
    - Add a new product (POST /products)
- PUT – Update an Existing Resource (Full Update)
  - Replace an existing resource entirely with new data.
  - Use Cases:
    - Update user profile (PUT /users/10)
    - Change product details (PUT /products/1)

# HTTP methods: GET, POST, PUT, DELETE, PATCH, and their appropriate use cases.

- PATCH – Update a Resource (Partial Update)
  - Modify specific fields of an existing resource instead of replacing the entire resource.
  - Use Cases:
    - Update only the price of a product (PATCH /products/1)
    - Modify user details without affecting other fields (PATCH /users/10)
- DELETE – Remove a Resource
  - Delete an existing resource from the server.
  - Use Cases:
    - Remove a user account (DELETE /users/10)
    - Delete a product from the inventory (DELETE /products/1)

# API design best practices: Resource naming, versioning, and HATEOAS.

- A well-designed REST API is easy to use, scalable, and maintainable.
- The three key best practices for API design are resource naming, versioning, and HATEOAS (Hypermedia as the Engine of Application State).

# Resource Naming Best Practices

- Proper resource naming ensures clarity and consistency in API endpoints.
- Best Practices for Naming Resources:
  - Use **nouns**, not verbs (Resources represent entities, not actions).
  - Use **plural names** for collections and singular for specific resources.
  - Use **hyphens (-)** to separate words, not underscores (GET /user-profile, not GET /user_profile).
  - Use **lowercase** letters for URLs (GET /users, not GET /Users).

# Resource Naming Best Practices

- Examples of Well-Designed Resource Names

| Resource | Correct | Incorrect |
|---|---|---|
| **Users Collection** | GET /users | GET /getUsers |
| **Single User** | GET /users/10 | GET /users?id=10 |
| **User's Orders** | GET /users/10/orders | GET /users/orders?id=10 |
| **Search (with query parameters)** | GET /products?category=electronics | GET /searchProducts/electronics |

# API Versioning Best Practices

- Versioning allows you to introduce new features or modify existing ones without breaking client applications.
- Best Practices for Versioning
  - Use **explicit versioning** in the URL (/v1/, /v2/), not implicit.
  - Use **major versions** only (e.g., v1, v2), not minor versions (v1.1).
  - Provide **backward compatibility** when updating APIs.
  - Deprecate old versions with a clear timeline.

# API Versioning Best Practices

- Common API Versioning Methods

| Versioning Method | Example | Pros | Cons |
|---|---|---|---|
| **URL Versioning** | GET /v1/products | Easy to implement, clear | Requires updating URLs when version changes |
| **Header Versioning** | GET /products + Accept: application/vnd.company.v1+json | Clean URLs, flexible | Clients must send correct headers |
| **Query Parameter Versioning** | GET /products?version=1 | Easy for testing | Messy, less commonly used |

- **Recommended: URL versioning** (/v1/) for clarity and ease of use.

# HATEOAS (Hypermedia as the Engine of Application State)

- HATEOAS enhances REST APIs by including links in responses, guiding clients on available actions.
- Best Practices for HATEOAS
  - Include links to related resources in API responses.
  - Use self-descriptive links (self, update, delete).
  - Reduce the need for hardcoded URLs in client applications.

# HATEOAS (Hypermedia as the Engine of Application State)

- Example of a HATEOAS-Enabled Response

```
{
 "id": 1,
 "name": "Laptop",
 "price": 1200,
 "links": {
  "self": "/products/1",
  "update": "/products/1/update",
  "delete": "/products/1/delete"
 }
}
```

- Benefits: Clients dynamically discover available actions instead of relying on documentation.

# Building REST APIs with Spring Boot

- Setting up a Spring Boot project.
- Creating RESTful endpoints and mapping HTTP methods to Java methods.
- Handling request parameters, path variables, and request bodies.

# Setting up a Spring Boot project.

1. Go to Spring Initializer: https://start.spring.io/
2. Select Project Settings:
   - Project: Maven
   - Language: Java
   - Spring Boot Version: Choose the latest stable version
   - Packaging: Jar
   - Java Version: 21
3. Define Project Metadata:
   - Group: com.example
   - Artifact: my-spring-boot-app
   - Name: my-spring-boot-app
   - Package Name: com.example.myspringbootapp

# Setting up a Spring Boot project.

4. Select Dependencies (for a basic MVC setup with MS SQL):
   - Spring Web (for REST APIs)
   - Spring Boot DevTools (for auto-reloading)
5. Generate & Download the Project:
   - Click "Generate", then extract the ZIP file.
6. Open the Project in IntelliJ IDEA
   - Open IntelliJ IDEA Community Edition
   - Click "Open" and select the extracted my-spring-boot-app folder.
   - Wait for Maven dependencies to download.

# Creating RESTful endpoints and mapping HTTP methods to Java methods.

- Create a Spring Boot REST Controller
  - Spring Boot uses the @RestController annotation to define RESTful APIs.

| HTTP Method | Java Annotation | Example URL |
|---|---|---|
| **GET** | @GetMapping | /users |
| **GET (by ID)** | @GetMapping("/{id}") | /users/1 |
| **POST** | @PostMapping | /users |
| **PUT** | @PutMapping("/{id}") | /users/1 |
| **PATCH** | @PatchMapping("/{id}") | /users/1?name=Updated Name |
| **DELETE** | @DeleteMapping("/{id}") | /users/1 |

# Handling request parameters, path variables, and request bodies.

- Handling Path Variables (@PathVariable)
  - Path variables are used to extract values from the URL path.
- Handling Request Parameters (@RequestParam)
  - Request parameters are used to pass optional or query parameters in the URL.
- Handling Request Body (@RequestBody)
  - Request bodies are used in POST, PUT, and PATCH requests to send JSON data.

# Error Handling in REST APIs

- Custom exception handling in Spring Boot.
- Designing standard error responses.
- Implementing global exception handling using @ControllerAdvice.

# Custom exception handling in Spring Boot.

- Spring Boot provides a structured way to handle exceptions in REST APIs using @ExceptionHandler, and @RestControllerAdvice
- Basic Exception Handling Using @ExceptionHandler
  - The @ExceptionHandler annotation is used inside a controller to handle specific exceptions.
- Global Exception Handling Using @RestControllerAdvice
  - Instead of handling exceptions per controller, you can use @RestControllerAdvice to centralize exception handling.

# Designing standard error responses.

- A well-structured error response improves the API’s usability and helps clients debug issues effectively.
- Structure of a Standard Error Response
- A standard error response should include the following fields:
  - timestamp → When the error occurred
  - status → HTTP status code (e.g., 400, 404, 500)
  - error → HTTP status message (e.g., "Bad Request", "Not Found")
  - message → A human-readable error message
  - path → The requested endpoint that caused the error

# Implementing global exception handling using @ControllerAdvice

- Global exception handling allows you to manage errors centrally, improving maintainability and user experience.
- Why Use @ControllerAdvice or @RestControllerAdvice?
  - Centralized error handling for all controllers.
  - Ensures consistent error response format.
  - Separates business logic from error handling.

# Advanced REST API Development

# Securing REST APIs

- Authentication and Authorization: OAuth2 and JWT basics.
- Implementing security in Spring Boot using Spring Security.
- Protecting APIs from common vulnerabilities (e.g., CSRF, XSS).

# Authentication and Authorization: OAuth2 and JWT basics.

- To secure a Spring Boot REST API use JWT (JSON Web Token) for authentication and authorization.
- Understanding Authentication vs. Authorization
  - **Authentication:** Verifies user identity (e.g., username & password).
  - **Authorization:** Grants or denies access to resources based on roles/permissions.
- What is JWT (JSON Web Token)?
  - JWT is a secure, compact token used for stateless authentication in REST APIs.
- Structure of a JWT Token: Header.Payload.Signature
  - Header: Contains token type (JWT) and signing algorithm (e.g., HS256).
  - Payload: Contains claims (user details, roles, expiration time, etc.).
  - Signature: Ensures the token is valid and untampered.

# Implementing security in Spring Boot using Spring Security.

- Spring Security is a powerful framework that helps secure Spring Boot applications by handling authentication, authorization, and other security features.
- Enabling Spring Security
  - Edit pom.xml and add spring-boot-starter-security

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

  - This will automagically secure all endpoints for application

# Implementing security in Spring Boot using Spring Security.

- When you add spring-boot-starter-security, Spring Boot applies default security settings:
  - All endpoints require authentication.
  - The default user is "user", and the password is auto-generated at startup (found in the logs).
- To test this behavior: Start your application and check the logs for:
  - Using generated security password: abc123xyz
- You can override default user name and generated password

  File: src/main/resources/application.properties

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

# Protecting APIs from common vulnerabilities (e.g., CSRF, XSS).

- What is CSRF?
  - CSRF is an attack where a malicious website tricks users into making unwanted requests to your API while authenticated.
- Solution: Enable CSRF Protection
  - Spring Security enables CSRF protection by default.
  - However, for stateless APIs (JWT-based authentication), we usually disable CSRF since tokens protect against CSRF.
  - Default CSRF protection (for form-based authentication):

```
http.csrf(csrf -> csrf.enable()) // Keep CSRF enabled for session-based authentication
```

  - Disable CSRF for JWT-based authentication:

```
http.csrf(csrf -> csrf.disable()) // Disable for JWT authentication
```

# Protecting APIs from common vulnerabilities (e.g., CSRF, XSS).

- Preventing XSS (Cross-Site Scripting)
- What is XSS?
  - XSS allows attackers to inject malicious JavaScript into web applications, which can steal sensitive information.
- Solution: Use Content Security Policy (CSP)
  - Spring Boot can set CSP headers to prevent XSS.
  - Modify your security configuration:

```
http.headers(headers -> headers
        .contentSecurityPolicy(csp -> csp.policyDirectives("default-src 'self'"))
    )
```

# Protecting APIs from common vulnerabilities (e.g., CSRF, XSS).

- Preventing SQL Injection
- What is SQL Injection?
  - SQL injection occurs when user inputs are directly concatenated into SQL queries, allowing attackers to execute harmful queries.
- Solution: Use Prepared Statements
  - Instead of:

    ```
    @Query("SELECT * FROM users WHERE username = '" + username + "'"")
    ```

  - Use:

    ```
    @Query("SELECT u FROM User u WHERE u.username = :username")
    User findByUsername(@Param("username") String username);
    ```

  - Spring Data JPA automatically prevents SQL injection when using @Query with parameters.

# Protecting APIs from common vulnerabilities (e.g., CSRF, XSS).

- Secure API Endpoints with Proper CORS Configuration
- Why CORS?
  - CORS (Cross-Origin Resource Sharing) prevents unauthorized websites from making API calls.
- Solution: Configure Allowed Origins

```
public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
            .allowedOrigins("https://trusted.com")
            .allowedMethods("GET", "POST", "PUT", "DELETE");
      }
    };
```

- Only allow API calls from trusted domains.

# Data Serialization and Validation

- Using Jackson for JSON serialization and deserialization.
- Validating API requests with @Valid and custom annotations.

# Using Jackson for JSON serialization and deserialization.

- Jackson is the default JSON processing library used by Spring Boot for serializing Java objects to JSON and deserializing JSON to Java objects.
- Adding Jackson to a Spring Boot Project
  - Spring Boot automatically includes Jackson when using spring-boot-starter-web.
- JSON Serialization (Java Object → JSON)
  - When we return Java Object as response, Spring Boot automatically converts it to JSON.
- JSON Deserialization (JSON → Java Object)
  - When we send JSON data in a request, Spring Boot automatically converts it into a Java object.

# Validating API requests with @Valid and custom annotations.

- Using @Valid for Request Validation
- Spring Boot integrates Jakarta Bean Validation (formerly Javax Validation) to validate API request payloads.
- Add Validation Dependency

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

- Validating Request Data with @Valid
  - Spring Boot uses @Valid to apply validation constraints defined in the model.

# Validating API requests with @Valid and custom annotations.

- Creating Custom Validation Annotations
  - Spring Boot allows custom validation annotations when built-in ones are not enough.
- Example: Custom Annotation to Validate Product Name
  - Create the Annotation (@ValidProductName)

```
@Documented
@Constraint(validatedBy = ProductNameValidator.class)
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidProductName {
    String message() default "Product name must start with an uppercase letter";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

# Validating API requests with @Valid and custom annotations.

- Create the Validator Class

```
public class ProductNameValidator implements ConstraintValidator<ValidProductName, String>
{
    @Override
    public boolean isValid(String name, ConstraintValidatorContext context) {
        return name != null && Character.isUpperCase(name.charAt(0));
    }
}
```

- Global Exception Handling for Validation Errors
  - Spring Boot automatically throws MethodArgumentNotValidException when @Valid fails.
  - To return a custom error response, use @RestControllerAdvice.

# Optimizing REST APIs

- Pagination and filtering for large datasets.
- Caching responses to improve performance.
- Using asynchronous processing for long-running requests.

# Pagination and filtering for large datasets.

- When handling large datasets in REST APIs, fetching all records at once can lead to performance issues and high memory usage.
- Pagination and filtering help optimize API responses, improving efficiency and user experience.
- Implementing Pagination & filtering in Spring Boot
  - Spring Boot supports pagination using Spring Data JPA's Pageable interface.
  - Add Spring Data JPA Dependency (If Missing)

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

# Caching responses to improve performance.

- Caching helps reduce database queries and response times by storing frequently accessed data in memory.
- Spring Boot provides caching support via Spring Cache Abstraction, with implementations like EhCache, Redis, Caffeine, and more.
- Enabling Caching in Spring Boot
  - Spring Boot requires enabling caching at the application level.
    - Add @EnableCaching in the Main Class
  - Caching API Responses using @Cacheable
    - @Cacheable stores method responses in cache so repeated calls return cached data instead of querying the database.
    - Apply Caching to the Service Layer

# Caching responses to improve performance.

- Clearing Cache with @CacheEvict
  - When a product is added, updated, or deleted, we need to invalidate cache so new data is retrieved.
  - Cache is invalidated when a new record is added. The next request fetches fresh data from the database.
- Summary of Caching Annotations

| Annotation | Description |
|---|---|
| **@EnableCaching** | Enables caching in the Spring Boot app |
| **@Cacheable(value = "cacheName")** | Caches method results |
| **@CacheEvict(value = "cacheName", allEntries = true)** | Clears cache on data update |

# Using asynchronous processing for long-running requests.

- Long-running requests can slow down API response times, leading to poor user experience and server overload.
- Asynchronous processing allows Spring Boot APIs to handle requests in the background while freeing up resources for other tasks.
- The @Async annotation allows methods to execute asynchronously in a separate thread.
- The method executes asynchronously without blocking the main thread.
- CompletableFuture<String> returns a future result when processing completes.

# Testing and Documentation

- Writing unit tests for REST APIs using JUnit and Mockito.
- Automating API testing with Postman and REST Assured.
- Generating API documentation with Swagger/OpenAPI.

# Writing unit tests for REST APIs using JUnit and Mockito.

- Unit testing ensures that individual components work as expected.
- For testing Spring Boot REST APIs, we use JUnit and Mockito to mock dependencies.
- Dependencies for Testing

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

- spring-boot-starter-test includes JUnit and Mockito.

# Automating API testing with Postman and REST Assured.

- Automating API testing ensures the stability of your REST endpoints.
- Automating API Testing with Postman
  - Step 1: Install Postman
    - Download and install Postman.(https://www.postman.com/downloads/)
  - Step 2: Create a Collection
    - Open Postman → Click "New Collection" → Name it "Product API Tests".
    - Add requests like GET /products, POST /products, etc.
  - Step 3: Add a Test Script
    - Open the request (GET /products).
    - Click "Tests" and add this script

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
```

  - Step 4: Run Automated Tests
    - Click Runner → Select "Product API Tests" → Run.
  - Postman Collection Runner executes multiple API tests at once.

# Automating API testing with Postman and REST Assured.

- Automating API Testing with REST Assured (Java)
  - REST Assured is a Java library for testing REST APIs.
  - Add REST Assured Dependency

```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <scope>test</scope>
</dependency>
```

# Generating API documentation with Swagger/OpenAPI.

- Swagger (OpenAPI) is a powerful tool for documenting REST APIs, making them easy to understand, test, and consume.
- Add Swagger Dependencies

```
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
   <version>2.8.5</version>
</dependency>
```

- Enable OpenAPI in Spring Boot
  - Spring Boot automatically configures Swagger when the dependency is added.
  - You can access the documentation at: http://localhost:8080/swagger-ui/index.html

Happy Learning :)