Swagger and Its Practical Implementation in Spring Boot

Trainer name:

Shreyansh Kumar

1. Introduction to Swagger

In the modern software development landscape, APIs (Application Programming Interfaces) have become the backbone of system integration. As applications grow in complexity and scale, the need for clear, consistent, and accessible API documentation becomes paramount. This is where Swagger, now known as the OpenAPI Specification, comes into play.

Swagger is an open-source framework that helps developers design, build, document, and consume RESTful web services. It provides a standardized way to describe API endpoints, request parameters, response models, authentication methods, and other API characteristics. By using Swagger, development teams can maintain up-to-date documentation that evolves alongside the API implementation.

The significance of Swagger extends beyond mere documentation. It serves as a contract between different teams working on a project, ensuring that frontend developers, backend developers, and QA engineers have a shared understanding of how the API should function. Moreover, Swagger's interactive UI allows developers to test API endpoints directly from the documentation, streamlining the development and testing process.

In the context of Spring Boot, which has emerged as one of the most popular frameworks for building Java-based applications, Swagger integration provides a seamless way to document RESTful services. Spring Boot's convention-over-configuration approach aligns well with Swagger's goal of simplifying API development and documentation.

This comprehensive guide will explore the practical implementation of Swagger in Spring Boot applications, covering everything from basic setup to advanced configurations and best practices. By the end, you'll have a thorough understanding of how to leverage Swagger to enhance your Spring Boot APIs.

2. Understanding API Documentation

Before diving into Swagger specifics, it's essential to understand why API documentation is crucial in software development. Well-documented APIs offer several benefits:

Improved Developer Experience: Clear documentation reduces the learning curve for new developers joining a project. It provides examples, explains expected behaviors, and outlines potential error scenarios.

Reduced Development Time: With comprehensive documentation, developers spend less time figuring out how to use an API and more time building features that utilize it.

Enhanced Collaboration: Documentation serves as a common reference point for discussions between teams, reducing misunderstandings and ensuring everyone is on the same page.

Easier Maintenance: As APIs evolve, documentation helps track changes and understand the impact of modifications on existing integrations.

Better Testing: Testers can use documentation to understand expected behaviors and edge cases, leading to more thorough testing.

Traditional approaches to API documentation often involved maintaining separate documents, wikis, or README files. These methods had several drawbacks:

Synchronization Issues: Manual documentation often became outdated as the API evolved, leading to inconsistencies between the actual implementation and the documentation.

Lack of Standardization: Without a standardized format, documentation varied widely in quality and comprehensiveness across different projects.

Limited Interactivity: Static documentation didn't allow developers to interact with the API directly, making it harder to understand and test endpoints.

Swagger addresses these challenges by providing a standardized, interactive, and code-integrated approach to API documentation. It generates documentation directly from the codebase, ensuring that it stays in sync with the actual implementation. Additionally, Swagger's interactive UI allows developers to send requests to the API and view responses in real-time, enhancing understanding and facilitating testing.

3. Swagger Specification

The Swagger Specification, now officially known as the OpenAPI Specification (OAS), is a format for describing RESTful APIs. It defines a standard, language-agnostic interface to RESTful APIs, allowing both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic.

The specification has evolved over time:

Swagger 1.0: The initial version introduced the core concepts but had limitations in terms of flexibility and extensibility.

Swagger 2.0: Released in 2014, this version brought significant improvements and became widely adopted in the industry.

OpenAPI 3.0: In 2017, the Swagger Specification was renamed to the OpenAPI Specification and transferred to the OpenAPI Initiative, a consortium of industry experts. This version introduced enhanced features for describing complex APIs.

OpenAPI 3.1: The latest version, released in 2021, further refines the specification with improved JSON Schema support and other enhancements.

At its core, the Swagger/OpenAPI Specification is a JSON or YAML document that describes various aspects of an API:

```yaml
openapi: 3.0.0
info:
  title: Sample API
  description: A sample API to illustrate OpenAPI concepts
  version: 1.0.0
servers:
  - url: https://api.example.com/v1
paths:
  /users:
    get:
      summary: Returns a list of users
      responses:
        '200':
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

This example demonstrates a simple OpenAPI document describing an API with a single endpoint (`/users`) that returns a list of user names. The specification includes:

Metadata: Information about the API, such as its title, description, and version.

Servers: The base URLs where the API is hosted.

Paths: The available endpoints and the HTTP methods they support.

Components: Reusable objects used by the API, such as schemas for request and response bodies.

Security: Authentication methods required to access the API.

Understanding the Swagger/OpenAPI Specification is crucial for effectively implementing Swagger in Spring Boot applications, as it forms the foundation for how your API will be documented and presented to consumers.

4. SpringFox and SpringDoc

When implementing Swagger in Spring Boot applications, developers typically use one of two popular libraries: SpringFox or SpringDoc. Both libraries facilitate the integration of Swagger/OpenAPI with Spring Boot, but they have different approaches and features.

SpringFox

SpringFox was one of the first libraries to provide Swagger integration for Spring applications. It automatically generates Swagger documentation based on Spring annotations, making it relatively easy to implement.

Key features of SpringFox include:

Automatic Documentation Generation: SpringFox scans Spring controllers and models to generate API documentation.

Customization Options: It offers various ways to customize the generated documentation through configuration classes and annotations.

Support for Swagger UI: SpringFox includes Swagger UI, allowing developers to interact with the API directly from the browser.

Integration with Spring Security: It can incorporate security requirements into the documentation.

However, SpringFox has some limitations:

Limited OpenAPI 3.0 Support: While SpringFox initially announced support for OpenAPI 3.0, the implementation was incomplete, and development has been relatively inactive in recent years.

Compatibility Issues: Some users have reported compatibility issues with newer versions of Spring Boot.

Maintenance Concerns: The project has seen periods of inactivity, raising concerns about long-term maintenance.

SpringDoc

SpringDoc emerged as an alternative to SpringFox, focusing on providing better support for OpenAPI 3.0 and addressing some of SpringFox's limitations.

Key features of SpringDoc include:

Full OpenAPI 3.0 Support: SpringDoc was designed from the ground up to support the OpenAPI 3.0 specification.

Active Development: The project is actively maintained, with regular updates and improvements.

Spring Boot Starter: It offers a Spring Boot starter for easy integration.

Customization Options: Like SpringFox, it provides various ways to customize the generated documentation.

Swagger UI and ReDoc Support: It includes both Swagger UI and ReDoc for visualizing and interacting with the API.

Choosing Between SpringFox and SpringDoc

When deciding which library to use, consider the following factors:

OpenAPI Version: If you need OpenAPI 3.0 support, SpringDoc is the better choice.

Spring Boot Version: Check compatibility with your Spring Boot version. SpringDoc generally has better compatibility with newer Spring Boot versions.

Active Development: SpringDoc is more actively maintained, which may be important for long-term projects.

Migration Effort: If you're already using SpringFox, consider the effort required to migrate to SpringDoc.

For new projects, SpringDoc is generally recommended due to its better OpenAPI 3.0 support and active maintenance. However, if you're working with an existing project that uses SpringFox and doesn't require OpenAPI 3.0 features, continuing with SpringFox might be more practical.

In the following sections, we'll focus primarily on implementing Swagger using SpringDoc, as it represents the more modern and actively maintained approach.

5. Setting Up Swagger in Spring Boot

Implementing Swagger in a Spring Boot application involves several steps, from adding dependencies to configuring the Swagger documentation. This section provides a detailed guide on setting up Swagger using SpringDoc.

Step 1: Add Dependencies

First, add the necessary dependencies to your project. For a Maven project, add the following to your `pom.xml`:

```
<!-- For Spring Boot 3.x with Jakarta EE -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.3.0</version>
</dependency>

<!-- OR for Spring Boot 2.x with Java EE -->
<!--
```

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.7.0</version>
</dependency>
-->
```

For a Gradle project, add this to your `build.gradle`:

```
implementation 'org.springdoc:springdoc-openapi-ui:1.6.9'
```

These dependencies include:

- The core SpringDoc OpenAPI library
- Swagger UI for the interactive documentation interface

Step 2: Create a Basic Configuration

Create a configuration class to customize the OpenAPI documentation:

```
package com.example.demo.config;

import io.swagger.v3.oas.models.OpenAPI;
import io.swagger.v3.oas.models.info.Info;
import io.swagger.v3.oas.models.info.License;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class OpenApiConfig {

    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
                .info(new Info()
                        .title("My API")
                        .version("1.0")
                        .description("This is a sample Spring Boot RESTful
service using SpringDoc OpenAPI")
                        .termsOfService("http://swagger.io/terms/")
                        .license(new License().name("Apache
2.0").url("http://springdoc.org")));
    }
}
```

This configuration class creates a bean that customizes the OpenAPI documentation with metadata such as the API title, version, and description.

Step 3: Configure Application Properties

Add the following properties to your `application.properties` or `application.yml` file to customize the Swagger UI:

```
# Swagger UI path
springdoc.swagger-ui.path=/swagger-ui.html

# Sort API paths alphabetically
springdoc.swagger-ui.operationsSorter=alpha

# Enable or disable Swagger UI
springdoc.swagger-ui.enabled=true

# API docs path
springdoc.api-docs.path=/api-docs
```

These properties configure:

- The path where Swagger UI will be available
- How operations are sorted in the UI
- Whether Swagger UI is enabled
- The path for the OpenAPI JSON documentation

Step 4: Create a Simple REST Controller

To demonstrate Swagger documentation, create a simple REST controller:

```java
package com.example.demo.controller;

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

import java.util.ArrayList;
import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController {

    private List<User> users = new ArrayList<>();

    @GetMapping
    public List<User> getAllUsers() {
        return users;
    }

    @GetMapping("/{id}")
    public User getUserById(@PathVariable Long id) {
        return users.stream()
                .filter(user -> user.getId().equals(id))
                .findFirst()
                .orElseThrow(() -> new RuntimeException("User not found"));
    }

    @PostMapping
```

```
    public User createUser(@RequestBody User user) {
        users.add(user);
        return user;
    }

    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User
updatedUser) {
        User existingUser = getUserById(id);
        existingUser.setName(updatedUser.getName());
        existingUser.setEmail(updatedUser.getEmail());
        return existingUser;
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        User user = getUserById(id);
        users.remove(user);
    }
}

class User {
    private Long id;
    private String name;
    private String email;

    // Getters and setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
```

```
    }
}
```

Step 5: Run the Application

Start your Spring Boot application. Once it's running, you can access the Swagger UI at:

`http://localhost:8080/swagger-ui.html`

And the OpenAPI JSON documentation at:

`http://localhost:8080/api-docs`

Verification

When you access the Swagger UI, you should see a page displaying your API's endpoints, organized by controller. You can expand each endpoint to see details about:

- The HTTP method (GET, POST, PUT, DELETE)
- The path parameters, query parameters, and request body
- The response structure and status codes
- The ability to try out the API directly from the UI

This basic setup provides a functional Swagger implementation for your Spring Boot application. In the following sections, we'll explore how to enhance and customize this implementation to better document your API.

6. Customizing Swagger Documentation

While the basic Swagger setup provides useful documentation, customizing it can significantly enhance its value. This section covers various ways to tailor Swagger documentation to your specific needs.

Global Documentation Customization

You can extend the OpenAPI configuration to include more detailed information:

```
@Configuration
public class OpenApiConfig {

    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
                .info(new Info()
                        .title("Customer Management API")
                        .version("2.0")
                        .description("This API allows managing customer
information")
                        .termsOfService("http://example.com/terms/")
                        .contact(new Contact()
                                .name("API Support")
```

```
                                .url("http://example.com/support")
                                .email("support@example.com"))
                        .license(new License()
                                .name("Apache 2.0")
                                .url("http://www.apache.org/licenses/LICENSE-
2.0.html")))
                .externalDocs(new ExternalDocumentation()
                        .description("More Documentation")
                        .url("http://example.com/docs"))
                .servers(List.of(
                        new
Server().url("https://api.example.com").description("Production server"),
                        new Server().url("https://staging-
api.example.com").description("Staging server"),
                        new
Server().url("http://localhost:8080").description("Development server")
                ));
    }
}
```

This enhanced configuration includes:

Contact Information: Details about who to contact for API support.

External Documentation: Links to additional documentation resources.

Server Information: Different environments where the API is deployed.

Grouping APIs

For larger applications with many endpoints, grouping APIs can improve organization:

```
@Configuration
public class OpenApiConfig {

    @Bean
    public GroupedOpenApi publicApi() {
        return GroupedOpenApi.builder()
                .group("public-api")
                .pathsToMatch("/api/public/**")
                .build();
    }

    @Bean
    public GroupedOpenApi adminApi() {
        return GroupedOpenApi.builder()
                .group("admin-api")
                .pathsToMatch("/api/admin/**")
                .build();
    }

    @Bean
```

```
    public OpenAPI customOpenAPI() {
        // ... (as before)
    }
}
```

This configuration creates two API groups:

- "public-api" for endpoints under `/api/public/`
- "admin-api" for endpoints under `/api/admin/`

Users can switch between these groups in the Swagger UI.

Customizing Models

You can enhance the documentation of your data models using annotations:

```
public class User {
    @Schema(description = "Unique identifier of the user", example = "1")
    private Long id;

    @Schema(description = "User's full name", example = "John Doe", required
= true)
    private String name;

    @Schema(description = "User's email address", example =
"john.doe@example.com", format = "email", required = true)
    private String email;

    @Schema(description = "User's role in the system", example = "ADMIN",
allowableValues = {"USER", "ADMIN", "MANAGER"})
    private String role;

    @Schema(description = "Date when the user was created", format = "date-
time")
    private LocalDateTime createdAt;

    // Getters and setters
}
```

These annotations provide:

- Descriptions for each field
- Example values
- Format information
- Required field indicators
- Allowed values for enumerated fields

Customizing Response Documentation

You can document different response scenarios:

```java
@RestController
@RequestMapping("/api/users")
public class UserController {

    @Operation(summary = "Get all users", description = "Retrieves a list of
all users in the system")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Successfully
retrieved users",
                content = @Content(array = @ArraySchema(schema =
@Schema(implementation = User.class)))),
        @ApiResponse(responseCode = "401", description = "Not authorized to
view users"),
        @ApiResponse(responseCode = "403", description = "Forbidden from
viewing users"),
        @ApiResponse(responseCode = "500", description = "Internal server
error")
    })
    @GetMapping
    public List<User> getAllUsers() {
        return users;
    }

    // Other methods...
}
```

This documentation specifies:

- A summary and description of the operation
- Different response codes that might be returned
- The content type and structure of successful responses
- Descriptions for error responses

Security Documentation

If your API requires authentication, you can document security requirements:

```java
@Configuration
public class OpenApiConfig {

    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
                // ... other configuration
                .components(new Components()
                        .addSecuritySchemes("bearer-jwt", new
SecurityScheme()
                                .type(SecurityScheme.Type.HTTP)
                                .scheme("bearer")
                                .bearerFormat("JWT")
                                .in(SecurityScheme.In.HEADER)
```

```
                                .name("Authorization"))
                        .addSecuritySchemes("api-key", new SecurityScheme()
                                .type(SecurityScheme.Type.APIKEY)
                                .in(SecurityScheme.In.HEADER)
                                .name("X-API-KEY")))
                .addSecurityItem(new SecurityRequirement()
                        .addList("bearer-jwt", Arrays.asList("read",
"write"))
                        .addList("api-key"));
    }
}
```

Then, at the controller or method level:

```
@RestController
@RequestMapping("/api/users")
@SecurityRequirement(name = "bearer-jwt")
public class UserController {
    // Methods...
}
```

This configuration:

- Defines different security schemes (JWT and API key)
- Specifies which endpoints require which security schemes
- Documents the required scopes for OAuth-based authentication

By applying these customizations, you can create comprehensive, clear, and user-friendly API documentation that accurately reflects your API's capabilities and requirements.

7. Swagger Annotations

Swagger annotations are crucial for enriching API documentation. They allow you to provide detailed information about endpoints, parameters, responses, and models directly in your code. This section covers the most important annotations and how to use them effectively.

Controller and Operation Annotations

These annotations document the purpose and behavior of controllers and their methods:

```
@RestController
@RequestMapping("/api/products")
@Tag(name = "Product Management", description = "APIs for managing product
information")
public class ProductController {

    @Operation(
        summary = "Get product by ID",
        description = "Retrieves a specific product based on its unique
identifier",
```

```
        operationId = "getProductById"
    )
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(
            @Parameter(description = "ID of the product to retrieve",
required = true, example = "123")
            @PathVariable Long id) {
        // Implementation
        return ResponseEntity.ok(new Product());
    }

    @Operation(
        summary = "Create new product",
        description = "Creates a new product in the system"
    )
    @PostMapping
    public ResponseEntity<Product> createProduct(
            @RequestBody Product product) {
        // Implementation
        return ResponseEntity.status(HttpStatus.CREATED).body(product);
    }
}
```

Key annotations:

@Tag: Groups operations by a logical category.

@Operation: Describes an API endpoint, including its purpose and behavior.

@Parameter: Documents a method parameter, including its description, whether it's required, and example values.

Request Body Documentation

For documenting request bodies:

```
@PostMapping
@Operation(summary = "Create new product")
public ResponseEntity<Product> createProduct(
        @io.swagger.v3.oas.annotations.parameters.RequestBody(
            description = "Product information for creation",
            required = true,
            content = @Content(
                mediaType = "application/json",
                schema = @Schema(implementation = Product.class),
                examples = {
                    @ExampleObject(
                        name = "Basic Product",
                        summary = "A basic product example",
                        value = "{ \"name\": \"Smartphone\", \"price\":
699.99, \"category\": \"Electronics\" }"
                    ),
```

```
                    @ExampleObject(
                        name = "Detailed Product",
                        summary = "A more detailed product example",
                        value = "{ \"name\": \"Smartphone XL\", \"price\":
899.99, \"category\": \"Electronics\", \"description\": \"Latest model with
enhanced features\", \"inStock\": true, \"tags\": [\"mobile\", \"5G\",
\"premium\"] }"
                    )
                }
            )
        )
        @RequestBody Product product) {
    // Implementation
    return ResponseEntity.status(HttpStatus.CREATED).body(product);
}
```

This annotation provides:

- A description of the request body
- Whether it's required
- The expected media type
- The schema of the request body
- Example request bodies with explanations

Response Documentation

For documenting responses:

```
@GetMapping("/{id}")
@Operation(summary = "Get product by ID")
@ApiResponses(value = {
    @ApiResponse(
        responseCode = "200",
        description = "Successfully retrieved the product",
        content = @Content(
            mediaType = "application/json",
            schema = @Schema(implementation = Product.class),
            examples = @ExampleObject(
                value = "{ \"id\": 123, \"name\": \"Smartphone\", \"price\":
699.99, \"category\": \"Electronics\" }"
            )
        )
    ),
    @ApiResponse(
        responseCode = "404",
        description = "Product not found",
        content = @Content(
            mediaType = "application/json",
            schema = @Schema(implementation = ErrorResponse.class),
            examples = @ExampleObject(
                value = "{ \"status\": 404, \"message\": \"Product with ID
```

```
123 not found\", \"timestamp\": \"2023-01-15T14:30:00Z\" }"
            )
        )
    ),
    @ApiResponse(
        responseCode = "500",
        description = "Internal server error",
        content = @Content(
            mediaType = "application/json",
            schema = @Schema(implementation = ErrorResponse.class)
        )
    )
})
public ResponseEntity<Product> getProductById(@PathVariable Long id) {
    // Implementation
    return ResponseEntity.ok(new Product());
}
```

This documentation specifies:

- Different response codes
- Descriptions for each response
- The content type and structure of responses
- Example responses

Model Annotations

For documenting data models:

```
@Schema(description = "Product information")
public class Product {

    @Schema(description = "Unique identifier of the product", example =
"123")
    private Long id;

    @Schema(description = "Name of the product", example = "Smartphone XL",
required = true)
    private String name;

    @Schema(description = "Price in USD", example = "699.99", minimum = "0")
    private BigDecimal price;

    @Schema(description = "Product category", example = "Electronics",
required = true)
    private String category;

    @Schema(description = "Detailed product description", example = "Latest
smartphone model with enhanced features")
    private String description;
```

```
    @Schema(description = "Whether the product is currently in stock",
example = "true", defaultValue = "false")
    private boolean inStock;

    @Schema(description = "Tags associated with the product")
    private List<String> tags;

    // Getters and setters
}
```

These annotations provide:

- A description of the model itself
- Descriptions for each field
- Example values
- Required field indicators
- Minimum/maximum values for numeric fields
- Default values

Hidden Operations and Parameters

Sometimes you might want to hide certain operations or parameters from the documentation:

```
@Hidden
@GetMapping("/internal/metrics")
public Map<String, Object> getInternalMetrics() {
    // Implementation
    return new HashMap<>();
}

@GetMapping("/search")
public List<Product> searchProducts(
        @Parameter(description = "Search query") @RequestParam String query,
        @Parameter(hidden = true) @RequestParam(required = false) String
internalParam) {
    // Implementation
    return new ArrayList<>();
}
```

The `@Hidden` annotation can be applied to:

- Controllers to hide all their operations
- Methods to hide specific operations
- Parameters to hide them from the documentation

Deprecated Operations

For APIs that are being phased out:

```java
@Deprecated
@Operation(
    summary = "Get product by code [DEPRECATED]",
    description = "This operation is deprecated. Use 'Get product by ID'
instead.",
    deprecated = true
)
@GetMapping("/by-code/{code}")
public ResponseEntity<Product> getProductByCode(@PathVariable String code) {
    // Implementation
    return ResponseEntity.ok(new Product());
}
```

This clearly indicates to API consumers that the endpoint is deprecated and should not be used in new integrations.

By effectively using these annotations, you can create detailed, accurate, and helpful API documentation that makes it easier for developers to understand and use your API.

8. Swagger UI

Swagger UI is a powerful tool that transforms your OpenAPI specification into an interactive documentation interface. This section explores how to customize and effectively use Swagger UI in your Spring Boot application.

Accessing Swagger UI

By default, when you include the SpringDoc OpenAPI UI dependency, Swagger UI is available at:

`http://localhost:8080/swagger-ui.html`

You can customize this path in your `application.properties` or `application.yml`:

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

Customizing the UI

You can customize various aspects of the Swagger UI through application properties:

```
# Change the UI layout to optimize space
springdoc.swagger-ui.docExpansion=none

# Sort operations alphabetically
springdoc.swagger-ui.operationsSorter=alpha

# Sort tags alphabetically
springdoc.swagger-ui.tagsSorter=alpha

# Display operation id in the UI
springdoc.swagger-ui.displayOperationId=true
```

```
# Display request duration in the UI
springdoc.swagger-ui.displayRequestDuration=true

# Enable deep linking for easier sharing of specific operations
springdoc.swagger-ui.deepLinking=true

# Disable the default models section
springdoc.swagger-ui.defaultModelsExpandDepth=-1

# Set the default models expansion depth
springdoc.swagger-ui.defaultModelExpandDepth=2

# Set the default expansion depth for models in operations
springdoc.swagger-ui.defaultModelRendering=model

# Display request headers in the UI
springdoc.swagger-ui.showCommonExtensions=true

# Enable syntax highlighting for code examples
springdoc.swagger-ui.syntaxHighlight.activated=true
```

Customizing the UI Theme

You can also customize the UI theme by providing your own CSS. Create a file named `swagger-ui-custom.css` in your `src/main/resources/static/css` directory:

```css
.swagger-ui .topbar {
    background-color: #1e88e5;
}

.swagger-ui .info .title {
    color: #0d47a1;
}

.swagger-ui .opblock.opblock-get {
    background: rgba(97, 175, 254, 0.1);
}

.swagger-ui .opblock.opblock-post {
    background: rgba(73, 204, 144, 0.1);
}

.swagger-ui .opblock.opblock-put {
    background: rgba(252, 161, 48, 0.1);
}

.swagger-ui .opblock.opblock-delete {
    background: rgba(249, 62, 62, 0.1);
}
```

```css
.swagger-ui .btn.execute {
    background-color: #1e88e5;
}
```

Then, configure SpringDoc to use this CSS file:

```
springdoc.swagger-ui.css-location=classpath:static/css/swagger-ui-custom.css
```

Adding a Custom Logo

You can replace the default Swagger logo with your own:

1. Add your logo image (e.g., `company-logo.png`) to `src/main/resources/static/images/`
2. Configure SpringDoc to use this logo:

```
springdoc.swagger-ui.image-url=/images/company-logo.png
```

Interactive Features

Swagger UI provides several interactive features that make it easier to understand and test your API:

Try It Out: For each endpoint, users can click the "Try it out" button to:

- Enter parameter values
- Provide request bodies
- Execute the request against your API
- View the response, including status code, headers, and body

Models: The UI displays the structure of request and response models, making it clear what data is expected and returned.

Authentication: If your API requires authentication, Swagger UI provides interfaces to enter API keys, tokens, or other credentials.

Practical Usage Tips

To get the most out of Swagger UI:

Organize with Tags: Use the `@Tag` annotation to group related operations, making the documentation more navigable.

Provide Examples: Include example values for parameters and request bodies to help users understand what data is expected.

Document Error Responses: Document all possible response codes, especially error responses, so users know what to expect when things go wrong.

Use Meaningful Operation IDs: Set clear operation IDs to make it easier to reference specific endpoints in discussions.

Link to External Resources: If you have additional documentation, tutorials, or guides, link to them from your API description.

Swagger UI Diagram

Here's a visual representation of how Swagger UI integrates with your Spring Boot application: