Swagger

jim stafford

Fall 2024 v2024-08-30: Built: 2024-11-19 21:33 EST

Table of Contents

Chapter 1. Introduction

The core charter of this course is to introduce you to framework solutions in Java and focus on core Spring and SpringBoot frameworks. **Details** of Web APIs, database design, and distributed application design are best covered in other courses. We have been covering a modest amount of Web API topics in these last set of modules to provide a functional front door to our application implementations. You know by now how to implement basic RMM level 2, CRUD Web APIs. I now want to wrap up the Web API coverage by introducing a functional way to call those Web APIs with minimal work using Swagger UI. Detailed aspects of configuring Swagger UI is considered out of scope for this course but many example implementation details are included in each API example and a detailed example in the Swagger Contest Example.

1.1. Goals

You will learn to:

- identify the items in the Swagger landscape and its central point — OpenAPI
- generate an Open API interface specification from Java code
- deploy and automatically configure a Swagger UI based on your Open API interface specification
- invoke Web API endpoint operations using Swagger UI
- generate a client library

1.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. generate a default Open API 3.0 interface specification using Springdoc
2. configure and deploy a Swagger UI that calls your Web API using the Open API specification generated by your API
3. make HTTP CRUD interface calls to your Web API using Swagger UI
4. identify the starting point to make configuration changes to Springdoc
5. generate a client API using the OpenAPI schema definition from a running application

Chapter 2. Swagger Landscape

The core portion of the Swagger landscape is made up of a line of standards and products geared towards HTTP-based APIs and supported by the company SmartBear. There are two types of things directly related to Swagger: the OpenAPI standard and tools. Although heavily focused on Java implementations, Swagger is generic to all HTTP API providers and not specific to Spring.

2.1. Open API Standard

OpenAPI — is an implementation-agnostic interface specification for HTTP-based APIs. This was originally baked into the Swagger tooling but donated to open source community in 2015 as a way to define and document interfaces.

- Open API 2.0 - released in 2014 as the last version prior to transitioning to open source. This is equivalent to the Swagger 2.0 Specification.
- Open API 3.x - released in 2017 as the first version after transitioning to open source.

2.2. Swagger-based Tools

Within the close Swagger umbrella, there are a set of Tools, both free/open source and commercial that are largely provided by Smartbear.

- Swagger Open Source Tools - these tools are primarily geared towards single API at a time uses.
  - Swagger UI — is a user interface that can be deployed remotely or within an application. This tool displays descriptive information and provides the ability to execute API methods based on a provided OpenAPI specification.
  - Swagger Editor - is a tool that can be used to create or augment an OpenAPI specification.
  - Swagger Codegen - is a tool that builds server stubs and client libraries for APIs defined using OpenAPI.
- Swagger Commercial Tools - these tools are primarily geared towards enterprise usage.
  - Swagger Inspector - a tool to create OpenAPI specifications using external call examples
  - Swagger Hub - repository of OpenAPI definitions

SmartBear offers another set of open source and commercial test tools called SoapUI which is geared at authoring and executing test cases against APIs and can read in OpenAPI as one of its API definition sources.

Our only requirement in going down this Swagger path is to have the capability to invoke HTTP methods of our endpoints with some ease. There are at least two libraries that focus on generating the Open API spec and packaging a version of the Swagger UI to document and invoke the API in Spring Boot applications: Springfox and Springdocs.

2.3. Springfox

Springfox is focused on delivering Swagger-based solutions to Spring-based API implementations but is not an official part of Spring, Spring Boot, or Smartbear. It is hard to even find a link to Springfox on the Spring documentation web pages.

Essentially Springfox is:

- a means to generate Open API specs using Java annotations
- a packaging and light configuring of the Swagger-provided swagger UI

Springfox has been around many years. I found the initial commit in 2012. It supported Open API 2.0 when I originally looked at it in June 2020 (Open API 3.0 was released in 2017). At that time, the Webflux branch was also still in SNAPSHOT. However, a few weeks later a flurry of releases went out that included Webflux support but no releases have occurred in the years since. Consider it deceased relative to Spring Boot 3.

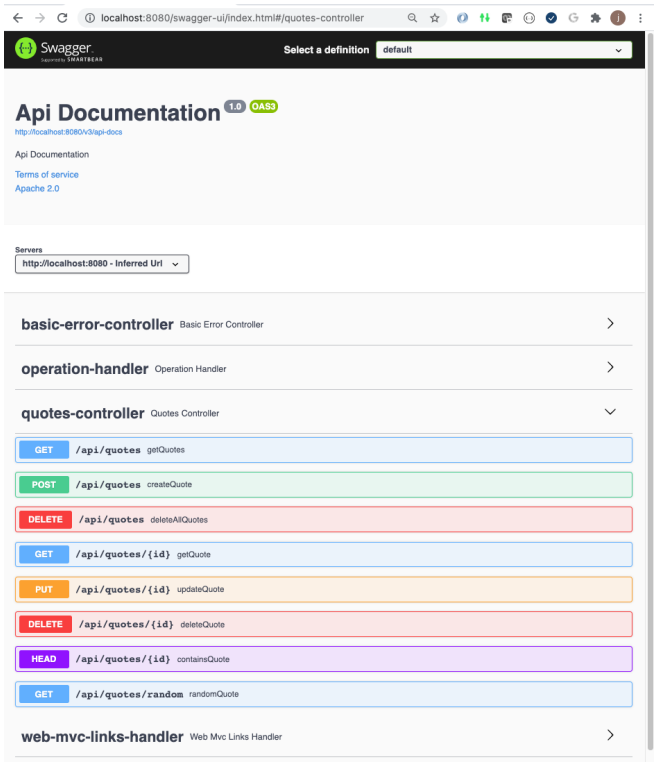


Figure 1. Example Springfox Swagger UI

Springfox does not work with >= Spring Boot 2.6

Springfox does not work with Spring Boot >= 2.6 where a patternParser was deprecated and causes an inspection error during initialization. We can work around the issue for demonstration — but serious use of Swagger (as of July 2022) is now limited to Springdoc.

Springfox is dead

There has not been a single git commit to Springfox since June 2020 and that version does not work with Spring Boot 3. We will consider Springfox dead and not speak much of it further.

2.4. Springdoc

Springdoc is an independent project focused on delivering Swagger-based solutions to Spring Boot APIs. Like Springfox, Springdoc has no official ties to Spring, Spring Boot, or Pivotal Software. The library was created because of Springfox's lack of support for Open API 3.x many years after its release.

Springdoc is relatively new compared to Springfox. I found its initial commit in July 2019 and has released several versions per month since — until 2023. That indicates to me that they had a lot of catch-up to do to complete the product and now are in a relative maintenance mode. They had the advantage of coming in when the standard was more mature and were able to bypass earlier Open API versions. Springdoc targets integration with the latest Spring Web API frameworks — including Spring MVC and Spring WebFlux.

The 1.x version of the library is compatible with Spring Boot 2.x. The 2.x version of the library has been updated to use jakarta dependencies and is required for Spring Boot 3.x.

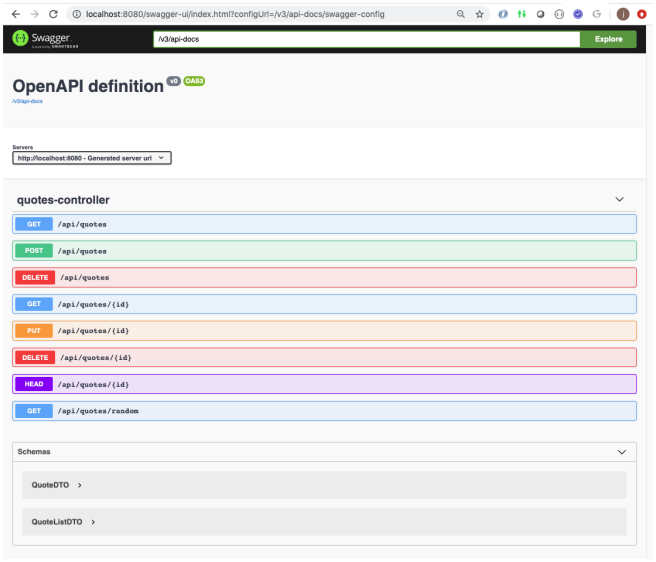


Figure 2. Example Springdoc SwaggerUI

Chapter 3. Minimal Configuration

My goal in bringing the Swagger topics into the course is solely to provide us with a convenient way to issue example calls to our API — which is driving our technology solution within the application. For that reason, I am going to show the least amount of setup required to enable a Swagger UI and have it do the default thing.

The minimal configuration will be missing descriptions for endpoint operations, parameters, models, and model properties. The content will rely solely on interpreting the Java classes of the controller, model/DTO classes referenced by the controller, and their annotations. Springdoc definitely does a better job at figuring out things automatically, but they are both functional in this state.

3.1. Springdoc Minimal Configuration

Springdoc minimal configuration is as simple as it gets. All that is required is a single Maven dependency.

3.1.1. Springdoc Maven Dependency

Springdoc has a single top-level dependency that brings in many lower-level dependencies. This specific artifact changed between Spring Boot 2 and 3

Springdoc Spring Boot 2 Maven Dependency

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

Springdoc Spring Boot 3 Maven Dependency

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

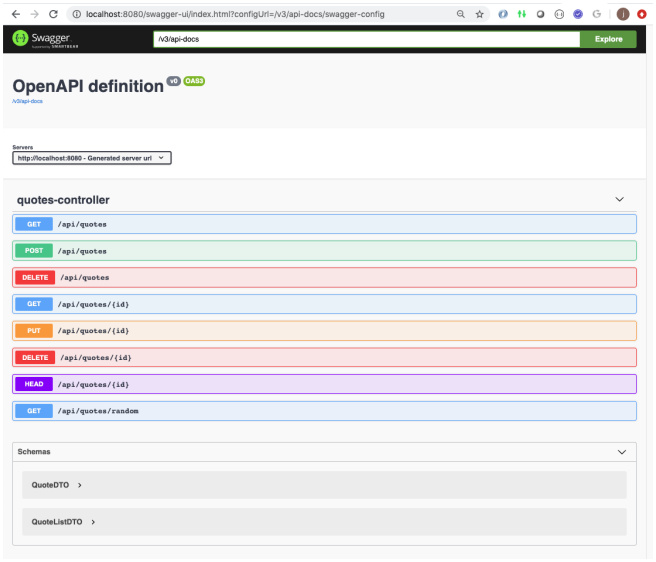
We will focus on using the Spring Boot 3 version, but the basics are pretty much the same with Spring Boot 2.

3.1.2. Springdoc Access

Once that is in place, you can access

- Open API spec: http://localhost:8080/v3/api-docs
- Swagger UI: http://localhost:8080/swagger-ui.html

The minimally configured Springdoc automatically provides an interface for our exposed web API.



3.1.3. Springdoc Starting Customization

The starting point for adjusting the overall interface for Springdoc is done through the definition of one or more GroupedOpenApi objects. From here, we can control the path and countless other options. The specific option shown will reduce the operations shown to those that have paths that start with "/api/".

Springdoc Starting Customization

```
...
import org.springdoc.core.models.GroupedOpenApi;
import org.springdoc.core.utils.SpringDocUtils;

@Configuration
public class SwaggerConfig {
   @Bean
    public GroupedOpenApi api() {
        SpringDocUtils.getConfig();
        //...
        return GroupedOpenApi.builder()
                .group("contests")
                .pathsToMatch("/api/**")
                .build();
    }
}
```

Textual descriptions are primarily added to the annotations of each controller and model/DTO class.

Chapter 4. Example Use

By this point in time we are past-ready for a live demo. You are invited to start the Springdoc version of the Contests Application and poke around. The following commands are being run from the parent `swagger-contest-example` directory. They can also be run within the IDE.

Start Springdoc Demo App

```
$ mvn spring-boot:run -f springdoc-contest-svc \①
-Dspring-boot.run.arguments=--server.port=8080 ②
```

① option to use custom port number

② passes arguments from command line, though Maven, to the Spring Boot application

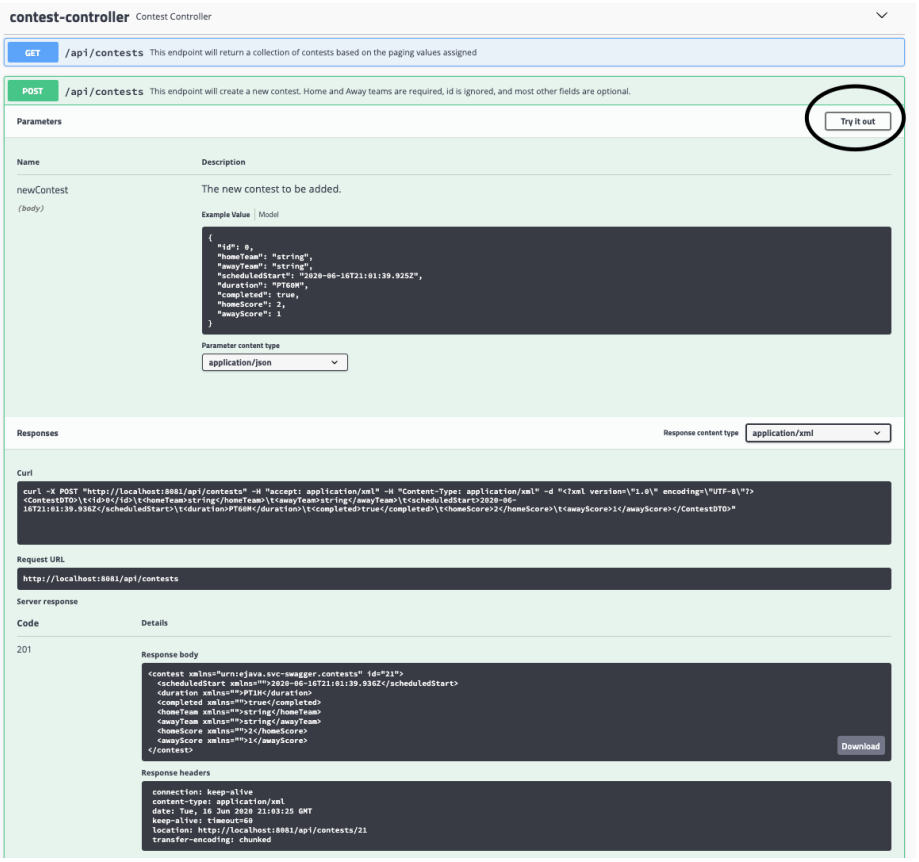
Access the application using

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

I will show an example thread.

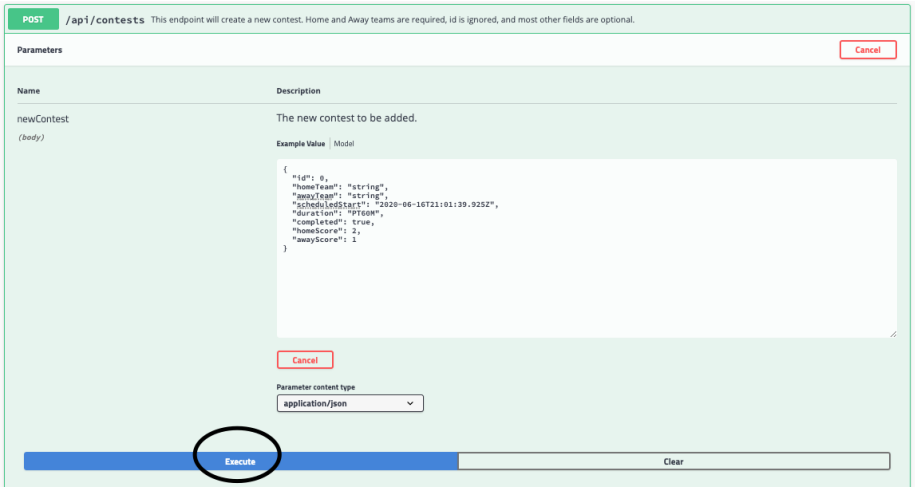
4.1. Access Contest Controller POST Command

1. click on the `POST /api/contests` line and the details of the operation will be displayed.
2. select a content type (`application/json` or `application/xml`)
3. click on the "Try it out" button.



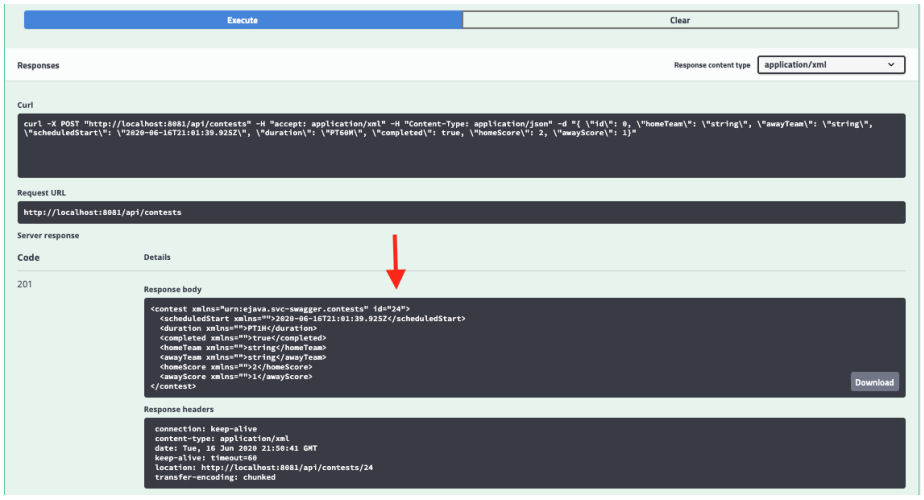
4.2. Invoke Contest Controller POST Command

1. Overwrite the values of the example with your values.
2. Select the desired response content type (`application/json` or `application/xml`)
3. press "Execute" to invoke the command



4.3. View Contest Controller POST Command Results

1. look below the "Execute" button to view the results of the command. There will be a payload and some response headers.
2. notice the payload returned will have an ID assigned
3. notice the headers returned will have a location header with a URL to the created contest
4. if your payload was missing a required field (e.g., home or away team), a 422/UNPROCESSABLE_ENTITY status is returned with a message payload containing the error text.



Chapter 5. Useful Configurations

I have created a set of examples under the `Swagger Contest Example` that provide a significant amount of annotations to add descriptions, provide accurate responses to dynamic outcomes, etc. using Springdoc to get a sense of how they performed.

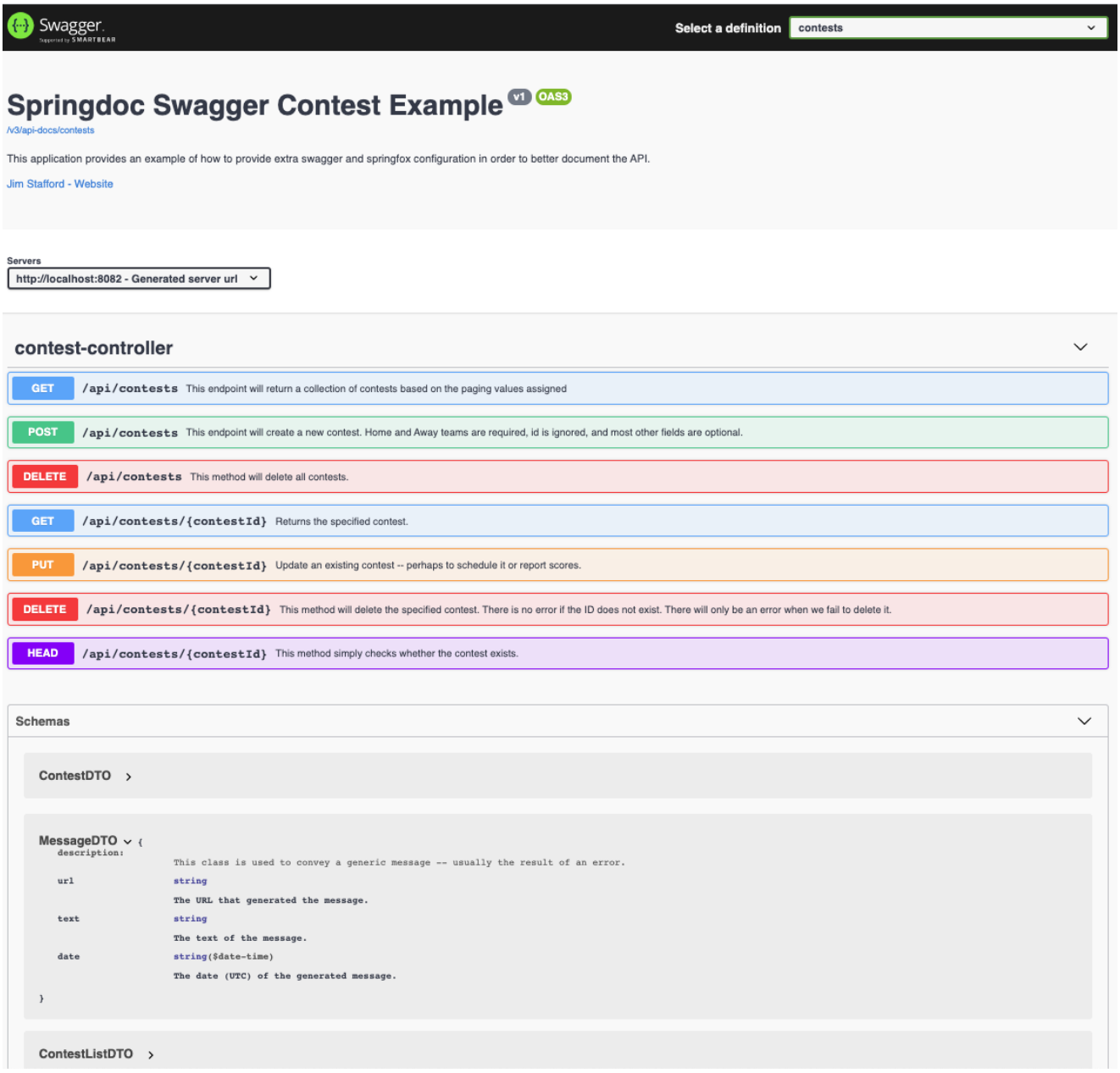


Figure 3. Fully Configured Springdoc Example

That is a lot of detail work and too much to cover here for what we are looking for. Feel free to look at the examples (controller, dtos) for details. However, I did encounter a required modification that made a feature go from unusable to usable and will show you that customization in order to give you a sense of how you might add other changes.

5.1. Customizing Type Expressions

`java.time.Duration` has a simple ISO string format expression that looks like `PT60M` or `PT3H` for periods of time.

5.1.1. OpenAPI 3 Model Property Annotations

The following snippet shows the Duration property enhanced with Open API 3 annotations, without any rendering hints.

ContestDTO Snippet with Duration without Rendering Hints

```
package info.ejava.examples.svc.springdoc.contests.dto;

import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlProperty;
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlRootElement;
import io.swagger.v3.oas.annotations.media.Schema;

@JacksonXmlRootElement(localName="contest", namespace=ContestDTO.CONTEST_NAMESPACE)
@Schema(description="This class describes a contest between a home and away team, "+
        "either in the past or future.")
public class ContestDTO {
    @JsonProperty(required = false)
    @Schema(
        description="Each scheduled contest should have a period of time specified "+
            "that identifies the duration of the contest. e.g., PT60M, PT2H")
    private Duration duration;
```

5.1.2. Default Duration Example Renderings

Springdoc's example rendering for `java.util.Duration` for JSON and XML is not desirable. It is basically trying to render an unknown type using reflection of the `Duration` class.

Springdoc Default Duration JSON Expression

```
"duration": {
"seconds": 0,
"nano": 0,
"negative": true,
"zero": true,
"units": [
  {
    "dateBased": true,
    "timeBased": true,
    "duration": {
      "seconds": 0,
      "nano": 0,
      "negative": true,
      "zero": true
    },
    "durationEstimated": true
  }
]
}
```

Springdoc Default Duration XML Expression

```
<duration>
    <seconds>0</seconds>
    <negative>true</negative>
    <zero>true</zero>
    <units>
        <durationEstimated>
true</durationEstimated>
        <duration>
            <seconds>0</seconds>
            <negative>true</negative>
            <zero>true</zero>
            <nano>0</nano>
        </duration>
        <dateBased>true</dateBased>
        <timeBased>true</timeBased>
    </units>
    <nano>0</nano>
</duration>
```

5.1.3. OpenAPI 3 Model Property Annotations

We can add an `example` to the `@Schema` annotation to override the produced value. Here we are expressing a `PT60M` example value be used.

ContestDTO Snippet with Duration and OpenAPI 3 Annotations

```
@JsonProperty(required = false)
@Schema(
    example = "PT60M",
    description="Each scheduled contest should have a period of time specified "+
        "that identifies the duration of the contest. e.g., PT60M, PT2H")
private Duration duration;
```

5.1.4. Simple Duration Example Renderings

The `example` attribute worked for JSON but did not render well for XML. The example text was displayed for XML without its `<duration></duration>` tags.

Springdoc-wide Duration Mapped to String JSON Expression

```
{
  "id": 0,
  "scheduledStart": "2023-08-
02T12:48:29.769Z",
  "duration": "PT60M", ①
  "completed": true,
  "homeTeam": "string",
  "awayTeam": "string",
  "homeScore": 2,
  "awayScore": 1
}
```

① correctly rendered provided `PT60M` example

Springdoc Property-specific Duration Mapped to String XML Expression

```
<?xml version="1.0" encoding="UTF-8"?>
<ContestDTO>
    <scheduledStart>2023-08-
02T13:36:20.867Z</scheduledStart>
PT60M<completed>true</completed> ①
    <homeTeam>string</homeTeam>
    <awayTeam>string</awayTeam>
    <homeScore>2</homeScore>
    <awayScore>1</awayScore>
</ContestDTO>
```

① incorrectly rendered XML for duration element

5.1.5. Schema Sub-Type Renderings

I was able to correct JSON and XML examples using a "StringSchema example". The following example snippets show expressing the schema at the property and global level.

Example Springdoc Map Property-specific to Schema Example

```
@JsonProperty(required = false)
@Schema(//position = 4,
        example = "PT60M",
        type = "string", ①
        description = "Each scheduled contest should have a period of time specified
that " +
                "identifies the duration of the contest. e.g., PT60M, PT2H")
private Duration duration;
```

① `type` allows us to subtype `Schema` for string expression

A global override can be configured at application startup.

Example Springdoc-wide Map Class to Schema Example

```
SpringDocUtils.getConfig()
        .replaceWithSchema(Duration.class,
                new StringSchema().example("PT120M")); ①
```

① `StringSchema` is a subclass of `Schema`

The subtypes of `Schema` were important to get both the JSON and XML rendered appropriately.

5.1.6. StringSchema Duration Example Renderings

The examples below show the payloads expressed in a usable form and potentially a valuable

source default example. Global configuration overrides property-specific configuration if both are used.

Springdoc-wide Duration Mapped to String JSON Expression

```
{
  "id": 0,
  "scheduledStart": "2023-08-
02T12:48:29.769Z",
  "duration": "PT120M",
  "completed": true,
  "homeTeam": "string",
  "awayTeam": "string",
  "homeScore": 2,
  "awayScore": 1
}
```

Springdoc Property-specific Duration Mapped to String XML Expression

```
<?xml version="1.0" encoding="UTF-8"?>
<ContestDTO>
    <scheduledStart>2023-08-
02T13:36:20.867Z</scheduledStart>
    <duration>PT60M</duration>
    <completed>true</completed>
    <homeTeam>string</homeTeam>
    <awayTeam>string</awayTeam>
    <homeScore>2</homeScore>
    <awayScore>1</awayScore>
</ContestDTO>
```

The example explored above was good enough to get my quick example usable, but shows how common it can be to encounter a bad example rendered document. Swagger-core has many options to address this that could be an entire set of lectures itself. The basic message is that Swagger provides the means to express usable examples to users, but you have to dig.

Chapter 6. Client Generation

We have seen where the OpenAPI schema definition was used to generate a UI rendering. We can also use it to generate a client. I won't be covering any details here, but will state there is a generation example in the class examples that produces a Java jar that can be used to communicate with our server — based in the OpenAPI definition produced. The example uses the OpenAPI Maven Plugin that looks extremely similar to the Swagger CodeGen Maven Plugin. There repository page has a ton of configuration options. This example uses just the basics.

6.1. API Schema Definition

The first step should be to capture the schema definition from the server into a CM artifact. Running locally, this would come from http://localhost:8080/v3/api-docs/contests.

OpenAPI Interface Definition File

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "Springdoc Swagger Contest Example",
    "description": "This application provides an example of how to provide extra
swagger and springfox configuration in order to better document the API.",
    "contact": {
    },
    "version": "v1"
  },
  "servers": [
    {
      "url": "http://localhost:8080",
      "description": "Generated server url"
    }
  ],
  "tags": [
    {
      "name": "contest-controller",
      "description": "manages contests"
    }
  ],
  "paths": {
    "/api/contests/{contestId}": {
...
```

6.2. API Client Source Tree

We can capture that file in a module source tree used for generation.

OpenAPI Client Source Tree

```
|-- pom.xml
`-- src
    `-- main
        `-- resources
            `-- contests-api.json
```

6.3. OpenAPI Maven Plugin

We add the Maven plugin to compile the API schema definition. The following shows a minimal skeletal plugin definition (version supplied by parent). Refer to the plugin web pages for options that can control building the source.

OpenAPI Maven Plugin

```
<plugin>
    <groupId>org.openapitools</groupId>
    <artifactId>openapi-generator-maven-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
            <configuration>
                <inputSpec>${project.basedir}/src/main/resources/contests-
api.json</inputSpec>
                <generatorName>java</generatorName>
                <configOptions>
                    <sourceFolder>src/gen/java/main</sourceFolder>
                </configOptions>
            </configuration>
        </execution>
    </executions>
</plugin>
```

6.4. OpenAPI Generated Target Tree

With the plugin declared, we can execute `mvn clean generate-sources`, to get a generated source tree with Java files containing some familiar names — like `model.ContestDTO`, `ContestListDTO`, and `MessageDTO`.

OpenAPI Generated Target Tree

```
`-- target
    `-- generated-sources
        `-- openapi
            |-- README.md
```

```
...
           `-- src
               |-- gen
               |   `-- java
               |       `-- main
               |           `-- org
               |               `-- openapitools
               |                   `-- client
               |                       |-- ApiCallback.java
...
               |                       `-- model
               |                           |-- ContestDTO.java
               |                           |-- ContestListDTO.java
               |                           `-- MessageDTO.java
```

6.5. OpenAPI Compilation Dependencies

To compile the generated source, we are going to have to add some dependencies to the module. In my quick read through this capability, I was surprised that the dependencies where not more obviously identified and easier to add. The following snippet shows my result of manually resolving all compilation dependencies.

```
<!-- com.google.gson.Gson -->
<dependency>
    <groupId>io.gsonfire</groupId>
    <artifactId>gson-fire</artifactId>
</dependency>

<!-- javax.annotation.Nullable -->
<dependency>
    <groupId>com.google.code.findbugs</groupId>
    <artifactId>jsr305</artifactId>
</dependency>

<!-- javax.annotation.Generated -->
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
</dependency>

<!-- okhttp3.internal.http.HttpMethod -->
<dependency>
    <groupId>com.squareup.okhttp3</groupId>
    <artifactId>okhttp</artifactId>
</dependency>

<!-- okhttp3.logging.HttpLoggingInterceptor -->
<dependency>
    <groupId>com.squareup.okhttp3</groupId>
```

```
    <artifactId>logging-interceptor</artifactId>
</dependency>
```

6.6. OpenAPI Client Build

With API definition in place, plugin defined and declared, and dependencies added, we can now generate the client JAR.

```
$ mvn clean install
...
[INFO] --- openapi-generator-maven-plugin:7.8.0:generate (default) @ generated-
contest-client ---
[INFO] Generating with dryRun=false
[INFO] OpenAPI Generator: java (client)
...
[INFO] Processing operation getContest
[INFO] Processing operation updateContest
[INFO] Processing operation doesContestExist
[INFO] Processing operation deleteContest
[INFO] Processing operation getContests
[INFO] Processing operation createContest
[INFO] Processing operation deleteAllContests
[INFO] writing file .../svc/svc-api/swagger-contest-example/generated-contest-
client/target/generated-
sources/openapi/src/gen/java/main/org/openapitools/client/model/ContestDTO.java
...
################################################################################
# Thanks for using OpenAPI Generator.                                          #
# Please consider donation to help us maintain this project 🙏                 #
# https://opencollective.com/openapi_generator/donate                          #
################################################################################
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  6.814 s
```

This was light coverage of the OpenAPI client generation capability. It gives you insight into why OpenAPI is useful for building external clients as well as the Swagger UI capability.

Chapter 7. Springdoc Summary

Springdoc is surprisingly functional right out of the box with minimal configuration — except for some complex types. In early June 2020, Springdoc definitely understood the purpose of the Java code better than Springfox and quickly overtook Springfox before the older option eventually stopped turning out releases. Migrating Springdoc from Spring Boot 2 to 3 was primarily a Maven dependency and a few package name changes for their configuration classes.

It took some digging, but I was able to find a solution to my specific `Duration` rendering problem. Along the way, I saw examples of how I could provide example payloads and complex objects (with values) that could be rendered into example payloads. A custom example is quite helpful if the model class has a lot of optional fields that are rarely used and unlikely to be used by someone using the Swagger UI.

Springfox has better documentation that shows you features ahead of time in logical order. Springdoc's documentation is primarily a Q&A FAQ that shows features in random order. I could not locate a good Springdoc example when I got started — but after implementing with Springfox first, the translation was extremely easy. Following the provided example in this lecture should provide you with a good starting point.

Springfox had been around a long time but with the change from Open API 2 to 3, the addition of Webflux, and their slow rate of making changes — that library soon showed to not be a good choice for Open API or Webflux users. Springdoc seemed like it was having some early learning pains in 2020 — where features may have worked easier but didn't always work 100%, lack of documentation and examples to help correct, and their existing FAQ samples did not always match the code. However, it was quite usable already in early versions, already supports Spring Boot 3 in 2023, and they continue to issuing releases. By the time you read this much may have changed.

One thing I found after adding annotations for the technical frameworks (e.g., Lombok, WebMVC, Jackson JSON, Jackson XML) and then trying to document every corner of the API for Swagger in order to flesh out issues — it was hard to locate the actual code. My recommendation is to continue to make the good names for controllers, models/DTO classes, parameters, and properties that are immediately understandable to save on the extra overhead of Open API annotations. Skip the obvious descriptions one can derive from the name and type, but still make it document the interface and usable to developers learning your API.

Chapter 8. Summary

In this module we:

- learned that Swagger is a landscape of items geared at delivering HTTP-based APIs
- learned that the company Smartbear originated Swagger and then divided up the landscape into a standard interface, open source tools, and commercial tools
- learned that the Swagger standard interface was released to open source at version 2 and is now Open API version 3
- learned that two tools — Springfox and Springdoc — were focused on implementing Open API for Spring and Spring Boot applications and provided a packaging of the Swagger UI
- learned that Springfox and Springdoc have no formal ties to Spring, Spring Boot, Pivotal, Smartbear, etc. They are their own toolset and are not as polished (in 2020) as we have come to expect from the Spring suite of libraries.
- learned that Springfox is older, originally supported Open API 2 and SpringMVC for many years, and now supports Open API 3, Spring Boot 2, and WebFlux. Springfox has stopped producing releases since July 2020.
- learned that Springdoc is newer, active, and supports Open API 3, SpringMVC, Webflux, and Spring Boot 3
- learned how to minimally configure Springdoc into our web application in order to provide the simple ability to invoke our HTTP endpoint operations
- learned how to minimally setup API generation using the OpenAPI schema definition from a running application and a Maven plugin