Documenting APIs with Swagger and AsciiDoc

Table of Contents

This guide explains how to use the Swagger specification and the AsciiDoc markup language to generate reference documentation for your RESTful APIs. The document source is available at:

https://github.com/dcporter65/swagger-asciidoc-primer

Process overview

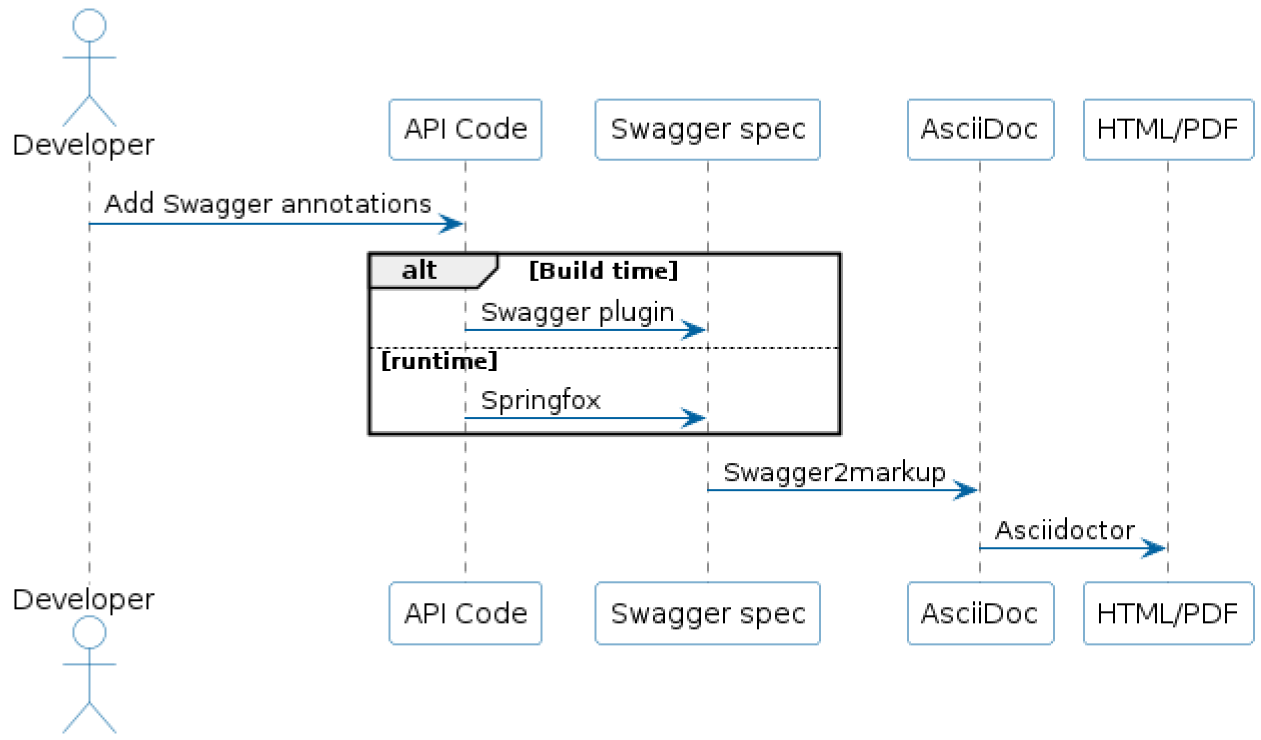


Figure 1. Generating documentation

Generating a Swagger spec

Swagger is an open source framework backed by a large ecosystem of tools that helps developers design, build, document, and consume RESTful Web services. At its heart is the Swagger Specification which is used to describe a RESTful API in JSON or YAML format.

The Swagger spec became the OpenApi Specification in version 3.0. The examples shown here use Swagger 2.0, since the swagger2markup tool used to generate the documentation has not yet caught up with the 3.0 standard.

You can choose to generate a Swagger specification from your API at either build time or runtime.

Design first or code first?

Swagger can take either a design-first or code-first approach to API development.

The design-first approach involves writing a Swagger or OpenAPI specification in JSON or YAML, circulating the specification to stakeholders for approval, and generating code from the specification. In this scenario, the API specification is the single source of truth for the API.

Code-first means that developers write the APIs using an appropriate RESTful coding framework, such as JAX-RS or Spring MVC, and add Swagger annotations to the code during the development process. A Swagger specification is then generated from the code either at build time or at runtime. In this case, the code is the single source of truth.

This document deals with the code-first approach.

Swagger annotations

You add Swagger annotations to your API code to ensure that a Swagger spec is generated at run or build time.

You declare the dependency on the Swagger Annotations library in the `build.gradle` in the project's API server module as follows:

Swagger Annotations dependency in build.gradle

```
dependencies {

  compile group: 'io.swagger', name: 'swagger-annotations', version: '1.5.17'

}
```

And then you add Swagger annotations to the resource classes, for JAX-RS projects, or to the controller classes in Spring MVC. For example:

Annotating a controller class

```
@Controller
@RequestMapping(value = "/pets", produces = "application/json")
@Api(value = "/pets", tags = "Pets", description = "Operations about pets")
public class PetController {

@RequestMapping(method = POST)
  @ApiOperation(value = "Add a new pet to the store")
  @ApiResponses(value = {@ApiResponse(code = 405, message = "Invalid input")})
  public ResponseEntity<String> addPet(
          @ApiParam(value = "Pet object that needs to be added to the store", required
= true) @RequestBody Pet pet) {
    petData.add(pet);
    return Responses.ok("SUCCESS");
  }
}
```

Here are some of the most useful Swagger annotations:

Annotation	Description	Attributes
`@Api`	Marks the class as a Swagger resource.	`tags` – Groups all the APIs in this class under a particular heading in the documentation `description` – Adds a further description for this group of APIs

Annotation	Description	Attributes
`@ApiOperation`	Used to document an individual API, usually a unique combination of HTTP method and resource path	`value` – The name of the API. Appears as a section title in the generated documentation. For example "Get widget". Does not end with a full stop (period). `notes` – A more detailed description of what the API does, written as one or more sentences. For example, "Returns the widget specified by the widgetUid parameter." `nickname` – Often required where the API method name is not unique across the project. For example, some projects use the method names `get`, `create`, and `update` in every resource class. For example, "getWidget" `response` – Where the response object cannot be inferred from the response statement in the code, you may need to use this annotation to set the name of the response class. `code` – The HTTP response code that is returned when this API is called successfully. Defaults to 200, so you need to set this for APIs that return 201, 202, and 204. Alternatively, you can set this in the @ApiResponse annotation.
@ApiParam	Used to document parameters that can be sent when the API is called.	`value` – A description of the parameter that appears in the generated documentation. `required` – A boolean showing whether the parameter is required. Defaults to false. `allowableValues` – A comma-separated list of allowable values.
`@ApiResponses`	Use this annotation where the response to a successful API call cannot be inferred from the code.	None. Contains a list of `@ApiResponse` annotations.
`@ApiResponse`	Defines an API response	`code` – The HTTP response code that is returned when this API is called successfully. `message` – The message that accompanies the HTTP response, for example 'No content'. `response` – The name of the response class.

Where an API consumes a JSON object in the body of a request or produces JSON in the response, the model of the JSON object is included in the generated `swagger.json`. You can add documentation to the model classes using the `@ApiModel` and `@ApiModelProperty` annotations. For example:

Annotating a model class

```
class CredentialsModel {
  @ApiModelProperty(required = true) String username
  @ApiModelProperty(required = true) String password
}
```

See the Swagger Annotations documentation for more details.

Generating Swagger at build time

The Swagger Gradle Plugin allows you to generate a `swagger.json` file at build time.

NOTE If you are using Maven as your build tool, you can use the Swagger Maven Plugin.

The dependency on this plugin and its configuration is declared in the `build.gradle` for each project's API server module.

Swagger Gradle Plugin config in build.gradle

```
plugins {
    id 'com.benjaminsproule.swagger'
    version '1.0.4'
}

swagger {
    apiSource {
        springmvc = false
        locations = ['com.acme.widget.api.resources']
        schemes = ['http', 'https']
        host = '{{host}}'
        basePath = '/api'

        info {
            title = 'Widget API'
            version = 'v1'
            description = 'This is the API for the Widget service.'
            termsOfService = 'http://www.example.com/termsOfService'
            contact {
                email = 'engineering@acme.com'
                name = 'Widget'
                url = 'http://www.widgetinc.com/'
            }
        }
        swaggerDirectory = "${project(':widget-docs').buildDir}/swagger"
    }
}
```

```
generateSwaggerDocumentation.dependsOn(compileGroovy)
```

The `swagger.json` is generated by the `generateSwaggerDocumentation` Gradle task and is written to a `swagger` folder under the docs module’s `build` directory.

Generating Swagger at runtime

There are a number of libraries that allow you to generate a Swagger or OpenAPI specification, as well as a Swagger UI API console at runtime. Here are a few that are used in the Java space:

- Swagger Core
- Springfox - for projects that rely on the Spring Framework
- springdoc-openapi - for projects that use Spring Boot

All these libraries expose a `swagger.json` or `openapi.json` at an HTTP URL when the API server is running. You can create a test that writes the JSON to a file in your project, so that you can proceed to the next step.

Converting Swagger to AsciiDoc

Once you have generated the Swagger specification you can use the Swagger2markup library to convert it to AsciiDoc.

About AsciiDoc

AsciiDoc is an easy-to-use, lightweight markup language that is well supported by a wide range of open source tools. It is similar to Markdown but is sophisticated enough to support many of the book-authoring features provided by more complicated specifications such as DocBook and DITA. Asciidoctor is a toolchain used for processing AsciiDoc and converting it to various formats including HTML and PDF.

Swagger2markup

The Swagger-to-AsciiDoc conversion is handled by the Swagger2markup Gradle plugin.

NOTE See the Swagger2markup Maven plugin if you are using Maven.

Example 1. Configuring the Swagger2markup Gradle plugin

build.gradle

```
buildscript {

    dependencies {
        classpath "io.github.swagger2markup:swagger2markup-gradle-plugin:1.3.3"
        classpath "io.github.swagger2markup:swagger2markup-import-files-ext:1.3.3"
    }
}

apply plugin: 'io.github.swagger2markup'

convertSwagger2markup {
    dependsOn(':mm-api-server:generateSwaggerDocumentation')
    swaggerInput "${project.buildDir}/swagger/swagger.json"
    outputDir file("${buildDir}/asciidoc/generated")
    config = [
        'swagger2markup.pathsGroupedBy': 'TAGS',
        'swagger2markup.extensions.dynamicPaths.contentPath':
file('src/asciidoc/extensions/paths').absolutePath
    ]
}
```

build.gradle.kts

```
import io.github.swagger2markup.tasks.Swagger2MarkupTask
```

```
buildscript {
    dependencies {
        classpath("io.github.swagger2markup:swagger2markup-gradle-plugin:1.3.3")
        classpath("io.github.swagger2markup:swagger2markup-import-files-
ext:1.3.3")
    }
}

apply(plugin = "io.github.swagger2markup")

tasks.withType<Swagger2MarkupTask> {
        swaggerInputFile = file("${project.buildDir}/swagger/swagger.json")
        outputDir = file("${buildDir}/asciidoc/generated")
        config = mapOf(
                "swagger2markup.pathsGroupedBy" to "TAGS",
                "swagger2markup.extensions.dynamicPaths.contentPath" to
file("asciidoc/extensions/paths").absolutePath
        )
}
```

The dependency on `swagger2markup-import-files-ext` and the `swagger2markup.extensions.dynamicPaths.contentPath` configuration relate to how sample requests and responses are included in the generated documentation.

To convert the Swagger to AsciiDoc, run:

`./gradlew convertSwagger2markup`

Including sample requests and responses

It is good practice to include a sample request and response in the documentation for each your your APIs.

You can use the Swagger2markup Dynamic file import extension and the PathsDocumentExtension point to import sample requests and responses into the generated API documentation.

You add the samples as follows:

1. Ensure that the `swagger2markup.extensions.dynamicPaths.contentPath` property is set in the docs module's`build.gradle`. Typically this is:

   ```
   src/asciidoc/extensions/paths
   ```

2. Create a directory under the content path and name it the same as the API method name (if unique) or the `ApiOperation` nickname.
3. In the new directory, create an AsciiDoc file called `operation-end-sample`.
4. Call the API either by running an acceptance test or using cURL or Postman.

5. Copy the output into the `operation-end-sample` file.

Sample request and response

```
[[getWidget.sample]]
== Sample request and response markup

[source,role="primary"]
.Request
----
GET https://api.example.com/api/widgets/6cef9e60-8d30-40d5-bbab-
c22e6eca9927/users/f6486748-4f6b-4274-9cd4-769ef0671e24
Accept: application/json
----
[source,role="secondary"]
.Response
----
200
Content-Type: application/json
Date: Mon, 10 Feb 2020 14:34:50 GMT
{
"uid": "f6486748-4f6b-4274-9cd4-769ef0671e24",
"foo": "bar"
}
----
```

Generating HTML and PDF

Once you have an AsciiDoc version of your API reference, you can combine it with other manually-maintained AsciiDoc documentation, such as an introductory chapter, and generate the final output as HTML or PDF.

To do this, you use the Asciidoctor Gradle plugin. Once again, you declare the dependency in the docs module’s Gradle build script:

Example 2. Configuring the Asciidoctor Gradle plugin

build.gradle

```
plugins {
    id 'org.asciidoctor.jvm.convert' version '3.3.0'
    id 'org.asciidoctor.jvm.pdf' version '3.3.0'
}

repositories {
    maven {
        url "https://repo.spring.io/release"
    }
    mavenCentral()
}

configurations {
    asciidoctorExt
}

dependencies {
    asciidoctorExt 'io.spring.asciidoctor:spring-asciidoctor-extensions-block-
switch:0.5.0'
}

ext {
    asciiDocOutputDir = file("${buildDir}/asciidoc/generated")
    docBuildDir = file("${project.buildDir}/doc")
}

asciidoctor {
    configurations 'asciidoctorExt'
    sourceDir = file("src/asciidoc/")
    baseDirFollowsSourceFile()
    sources {
        include 'widget-apiguide.adoc'
    }
    outputOptions{
        backends = ['html5', 'pdf']
        separateOutputDirs = false
    }
    resources {
        from(sourceDir) {
            include 'images/**'
        }
        from(sourceDir) {
            include 'assets/js/**'
        }
    }

    outputDir docBuildDir
    backends = ['html5', 'pdf']
    attributes = [
        'docinfo': 'shared',
        'doctype': 'book',
        'generated': asciiDocOutputDir,
        'pdf-stylesdir': 'theme',
        'pdf-style': 'acme',
        'revnumber': version,
        'source-highlighter': 'prettify',
        'toc': 'left',
        'toclevels': '3'
    ]
}
```

build.gradle.kts

```
import org.asciidoctor.gradle.jvm.AsciidoctorTask

plugins {
    id("org.asciidoctor.jvm.convert") version "3.3.0"
    id("org.asciidoctor.jvm.pdf")  version "3.3.0"
}

repositories {
    maven {
        url = uri("https://repo.spring.io/release")
    }
    mavenCentral()
}

val asciidoctorExt by configurations.creating

dependencies {
    asciidoctorExt("io.spring.asciidoctor:spring-asciidoctor-extensions-block-
switch:0.5.0")
}

tasks.withType<AsciidoctorTask> {
    configurations("asciidoctorExt")
    dependsOn(tasks.getByName("clean").name,
tasks.getByName("convertSwagger2markup").name)
    setSourceDir(file("${project.rootDir}/docs/asciidoc"))
    outputs.upToDateWhen { false }
    outputOptions {
        backends("html5", "pdf")
        separateOutputDirs = false
    }
    sources(delegateClosureOf<PatternSet> {
        include("widget-apiguide.adoc")
    })

    baseDirFollowsSourceDir()
    this.attributes(
            mapOf(
                    "docinfo"            to "shared",
                    "doctype"            to "book",
                    "generated"          to
file("${buildDir}/asciidoc/generated"),
                    "pdf-stylesdir"      to "theme",
                    "pdf-style"          to "acme",
                    "sectanchors"        to "true",
                    "source-highlighter" to "prettify",
                    "toc"                to "left",
                    "toclevels"          to "3")
    )
}
```

You generate the HTML and PDF by running the `asciidoctor` Gradle task.

The Asciidoctor plugin depends on two other libraries:

- AsciidoctorJPDF is used for PDF generation
- Spring Asciidoctor Extensions post-processes Asciidoctor's HTML output to collapse multiple code blocks into one that provides side-by-side code samples.

The extensions are stored in the repo.spring.io Artifactory repository, which is included in the`repositories` section of the Gradle build script.

Including side-by-side code snippets

Instead of displaying code samples one below the other, as is the default, it can look good to render them in a side-by-side tabbed view in the generated HTML.

For this you can use the block switch extension in the Spring Asciidoctor Extensions library.

Add the following dependency as a configuration to your Gradle build script:

build.gradle

```
configurations {
    asciidoctorExt
}

dependencies {
     asciidoctorExt 'io.spring.asciidoctor:spring-asciidoctor-extensions-block-
switch:0.5.0'
}
```

build.gradle.kts

```
val asciidoctorExt by configurations.creating

dependencies {
    asciidoctorExt("io.spring.asciidoctor:spring-asciidoctor-extensions-block-
switch:0.5.0")
}
```

And add the configuration to the `asciidoctor` task:

build.gradle

```
asciidoctor {
    configurations 'asciidoctorExt'
}
```

build.gradle.kts

```
tasks.withType<AsciidoctorTask> {
    configurations("asciidoctorExt")
}
```

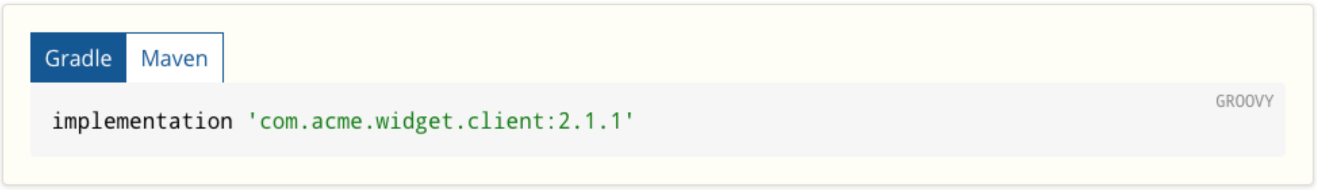
You also need to add a `role` attribute to each of the source code blocks. There must be one 'primary' role and at least one 'secondary' role.

So this markup:

```
====
[source,groovy,role="primary"]
.Gradle
----
implementation 'com.acme.widget.client:2.1.1'
----
```

```
[source,xml,role="secondary"]
.Maven
----
<dependency>
  <groupId>com.acme.widget</groupId>
  <artifactId>client</artifactId>
  <version>2.1.1</version>
</dependency>
----
====
```

… is rendered like this:



```
implementation 'com.acme.widget.client:2.1.1'
```

Styling and themes

You use a docinfo file to override Asciidoctor's default CSS styling for HTML output.

The file should be named `docinfo.html` and be stored alongside your AsciiDoc source document.

Example docinfo.html

```
<style>
    a:hover, a:active, a:focus {
        text-decoration:none;
        color:#009fe3;
    }
    .listingblock .switch {
        border-color: #0061a0;
    }
    .switch .switch--item {
        color: #0061a0;
    }
    .switch .switch--item.selected {
        background-color: #0061a0;
        color: #ffffff;
    }
    .switch .switch--item:not(:first-child) {
        border-color: #0061a0;
    }
</style>
```

For PDF output you use the theming system. These two lines in the Asciidoctor configuration in the `build.gradle` govern the themes used:

Theming configuration

```
'pdf-stylesdir' : 'theme',
'pdf-style' : 'my_theme'
```

Including an expandable TOC

The default HTML output displays the table of contents as a simple list in the left-hand navigation frame. You can use the Tocbot JavaScript library to render an expandable TOC.