Annotations 1.5.X

frantuma edited this page on May 12 · 36 revisions

Swagger-Core Annotations

In order to generate the Swagger documentation, swagger-core offers a set of annotations to declare and manipulate the output. The swagger-core output is compliant with Swagger Specification. A user is not required to be familiar with the full aspects of the Swagger Specification in order to use it, but as a reference it may answer a few questions regarding the generated output.

This page introduces the annotations provided by swagger-core. They are grouped into three - the annotation to declare the resource, the set of annotations to declare an operation, and the set of annotations that declare API models.

The documentation for each annotation is meant as an overview of its usage. Each annotation also has links to its javadocs (both on the header and at the end of the overview). The javadocs provide you with additional information about each annotation, especially dealing with some edge cases.

At the very least:

`@Api` is required to declare an API resource.

`@javax.ws.rs.Path` is required at class level (since v1.5.8) to have Swagger scan root resoure, in compliance with JAX-RS spec.

Without having those two combined, no output will be generated, unless scanAllResources config option is not set, in which case also @Path annotatied classes with no @Api annotation will be scanned.

Servlets require `@ApiImplicitParam` to define the method parameters whereas JAX-RS based application can utilize the basic `@XxxxParam` annotations (`@QueryParam` , `@PathParam` ...).

The following table summarizes parsing behaviour depending on annotations and `ReaderConfig.scanAllResources` value

Annotations	Result
@Api	skip
@Path	skip
scanAllResources	skip
@Api hidden	skip
@Api @Path hidden	skip
@Api hidden scanAllResources	skip
@Api scanAllResources	skip
@Api @Path	parse

Clone this wiki locally

`https://github.com/swagger-`

Clone in Desktop

@Api hidden scanAllResources	skip
@Path scanAllResources	parse
subresource	parse
@Api subresource	parse

NOTE: in version 1.5.8 only, @Api annotation is not needed to have class scanned, @Path is sufficient, according to the following table:

Annotations	Result
@Api	skip
@Path	parse
scanAllResources	parse
@Api hidden	skip
@Api @Path hidden	skip
@Api hidden scanAllResources	skip
@Api scanAllResources	skip
@Api @Path	parse
@Api hidden scanAllResources	skip
@Path scanAllResources	parse
subresource	parse
@Api subresource	parse

Table of contents:

For your convenience, the javadocs are available as well.

Quick Annotation Overview

Name	Description
@Api	Marks a class as a Swagger resource.
@ApiImplicitParam	Represents a single parameter in an API Operation.
@ApiImplicitParams	A wrapper to allow a list of multiple ApiImplicitParam objects.
@ApiModel	Provides additional information about Swagger models.
@ApiModelProperty	Adds and manipulates data of a model property.
@ApiOperation	Describes an operation or typically a HTTP method against a specific path.

@ApiParam	Adds additional meta-data for operation parameters.
@ApiResponse	Describes a possible response of an operation.
@ApiResponses	A wrapper to allow a list of multiple ApiResponse objects.
@Authorization	Declares an authorization scheme to be used on a resource or an operation.
@AuthorizationScope	Describes an OAuth2 authorization scope.
@ResponseHeader	Represents a header that can be provided as part of the response.

The latest release also adds a number of annotations for adding extensions and metadata at the Swagger Definition level:

Name	Description
@SwaggerDefinition	Definition-level properties to be added to the generated Swagger definition
@Info	General metadata for a Swagger definition
@Contact	Properties to describe the contact person for a Swagger definition
@License	Properties to describe the license for a Swagger definition
@Extension	Adds an extension with contained properties
@ExtensionProperty	Adds custom properties to an extension

Resource API Declaration

@Api

In Swagger 2.0, resources were replaced by tags, and this impacts the `@Api` annotation. It is no longer used to declare a resource, and it is now used to apply definitions for all the operations defined under it.

A JAX-RS usage would be:

```
@Path("/pet")
@Api(value = "pet", authorizations = {
      @Authorization(value="sampleoauth", scopes = {})
    })
@Produces({"application/json", "application/xml"})
public class PetResource {
 ...
}
```

In this example, we're saying that the tag for the operations under this class is `pet` (so they would all be grouped together). Swagger will pick up on the `@Produces` annotation but you can override this value if you wish.

`@Api` can also be used to declare authorization at the resource-level. These definitions apply to all operations under this resource, but can be overridden at the operation level if needed. In the example above, we're adding a previously-declared OAuth2 authorization scheme without any scopes. For further details, check the @Authorization annotation.

Instead of using the `value()`, you can use the `tags()` property which allows you to set multiple tags for the operations. For example:

```
@Api(tags = {"external_info","user_info"})
```

Note that in this case, `value()` would be ignored even if it exists.

The boolean `hidden` property can be used to entirely hide an @Api even if it declared. This is especially useful when using sub-resources to remove unwanted artifacts.

In swagger-core 1.5.X, `description()`, `basePath()`, and `position()` are no longer used.

For further details about this annotation, usage and edge cases, check out the javadocs.

Operation Declaration

@ApiOperation

The `@ApiOperation` is used to declare a single operation. An operation is considered a unique combination of a path and a HTTP method.

A JAX-RS usage would be:

```
 @GET
 @Path("/findByStatus")
 @ApiOperation(value = "Finds Pets by status",
    notes = "Multiple status values can be provided with comma seperated strings",
    response = Pet.class,
    responseContainer = "List")
 public Response findPetsByStatus(...) { ... }
```

The `value` of the annotation is a short description on the API. Since this is displayed in the list of operations in Swagger-UI and the location is limited in size, this should be kept short (preferably shorter than 120 characters). The `notes` allows you to give significantly more details about the operations. `response` is the return type of the method. Notice that the actual method declaration returns a `Response` but that is a general-purpose JAX-RS class and not the actual response sent to the user. If the returned object is the actual result, it can be used directly instead of declaring it in the annotation. Since we want to return a list of pets, we declare that using the `responseContainer`. Keep in mind that Java has type erasure, so using generics in the return type may not be parsed properly, and the `response` should be used directly. The `@GET` JAX-RS annotation will be used as the (HTTP) `method` field of the operation, and the `@Path` would tell us the path of the operation (operations are grouped under the same path, one for each HTTP method used).

The output would be:

```
    "/pet/findByStatus": {
  "get": {
    "tags": [
      "pet"
    ],
    "summary": "Finds Pets by status",
    "description": "Multiple status values can be provided with comma seperated strings",
    "responses": {
      "200": {
        "description": "successful operation",
        "schema": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/Pet"
          }
        }
      },
      .
      .
```

.

For further details about this annotation, usage and edge cases, check out the javadocs.

@ApiResponses, @ApiResponse

It's a common practice to return errors (or other success messages) using HTTP status codes. While the general return type of an operation is defined in the @ApiOperation, the rest of the return codes should be described using these annotations.

The `@ApiResponse` describes a concrete possible response. It cannot be used directly on the method or class/interface and needs to be included in the array value of `@ApiResponses` (whether there's one response or more).

If the response is accompanied with a body, the body model can be described as well (one model per response).

```
@ApiResponses(value = {
    @ApiResponse(code = 400, message = "Invalid ID supplied",
                 responseHeaders = @ResponseHeader(name = "X-Rack-Cache", description = "E
    @ApiResponse(code = 404, message = "Pet not found") })
public Response getPetById(...) {...}
```

In swagger-core 1.5.X, you can also add description of response headers as seen in the example above.

For further details about this annotation, usage and edge cases, check out the javadocs (@ApiResponses, @ApiResponse).

@Authorization, @AuthorizationScope

These annotations are used as input to @Api and @ApiOperation only, and not directly on the resources and operations. Once you've declared and configured which authorization schemes you support in your API, you can use these annotation to note which authorization scheme is required on a resource or a specific operation. The `@AuthorizationScope` is specific to the case of an OAuth2 authorization scheme where you may want to specify specific supported scopes.

The @Authorization and @AuthorizationScope translate to the Security Requirement Object.

The behavior between the implementations (JAX-RS, Servlets or otherwise) is the same:

```
@ApiOperation(value = "Add a new pet to the store",
  authorizations = {
    @Authorization(
        value="petoauth",
        scopes = { @AuthorizationScope(scope = "add:pet") }
        )
  }
)
public Response addPet(...) {...}
```

In this case we declare that the `addPet` operation uses the `petoauth` authorization scheme (we'll assume it is an OAuth2 authorization scheme). Then using the `@AuthorizationScope` we fine-tune the definition by saying it requires the `add:pet` scope. As mentioned above, you can see that `@AuthorizationScope` is used as an input to `@Authorization`, and that in turn is used as input to `@ApiOperation`. Remember, these annotations can only be used as input to `@Api` and `@ApiOperation`. Using any of them directly on a class or a method will be ignored.

The output would be:

```
  "security": [
    {
      "petoauth": [
        "add:pet"
      ]
    }
  ]
```

For further details about this annotation, usage and edge cases, check out the javadocs (@Authorization, @AuthorizationScope).

@ApiParam

The `@ApiParam` is used solely with the JAX-RS parameter annotations (`@PathParam` , `@QueryParam` , `@HeaderParam` , `@FormParam` and in JAX-RS 2, `@BeanParam`). While swagger-core scans these annotations by default, the `@ApiParam` can be used to add more details on the parameters or change the values as they are read from the code.

In the Swagger Specification, this translates to the Parameter Object.

Swagger will pick up the `value()` of these annotations and use them as the parameter name, and based on the the annotation it will also set the parameter type.

Swagger will also use the value of `@DefaultValue` as the default value property if one exists.

```
   @Path("/{username}")
   @ApiOperation(value = "Updated user",
      notes = "This can only be done by the logged in user.")
  public Response updateUser(
        @ApiParam(value = "name that need to be updated", required = true) @PathParam("usernam
        @ApiParam(value = "Updated user object", required = true) User user) {...}
```

Here we have two parameters. The first, `username` which is a part of the path. The second is the body, in this case a User object. Note that both parameters have the `required` property set to `true` . For the @PathParam, this is redundant as it is mandatory by default and cannot be overridden.

The output would be:

```
   "parameters": [
    {
      "name": "username",
      "in": "path",
      "description": "name that need to be deleted",
      "required": true,
      "type": "string"
    },
    {
      "in": "body",
      "name": "body",
      "description": "Updated user object",
      "required": false,
      "schema": {
        "$ref": "#/definitions/User"
      }
    }
  ]
```

For further details about this annotation, usage and edge cases, check out the javadocs.

@ApiImplicitParam, @ApiImplicitParams

You may wish you describe operation parameters manually. This can be for various reasons, for example:

- Using Servlets which don't use JAX-RS annotations.
- Wanting to hide a parameter as it is defined and override it with a completely different definition.
- Describe a parameter that is used by a filter or another resource prior to reaching the JAX-RS implementation.

Since there can be several parameters to be included, the `@ApiImplicitParams` allows for multiple `@ApiImplicitParam` definitions.

In the Swagger Specification, these translate to the Parameter Object.

When defining parameters implicitly, it's important to set `name`, `dataType` and `paramType` for Swagger's definitions to be proper.

```
@ApiImplicitParams({
   ApiImplicitParam(name = "name", value = "User's name", required = true, dataType = "stri
   ApiImplicitParam(name = "email", value = "User's email", required = false, dataType = "s
   ApiImplicitParam(name = "id", value = "User ID", required = true, dataType = "long", par
 })
public void doPost(HttpServletRequest request, HttpServletResponse response) throws Servlet
```

In the above sample we can see a Servlet definition with several parameters. The `dataType` can be either a primitive or a class name. The `paramType` can be any of the parameter types that are supported by Swagger (refer to the javadocs or the spec for further details).

```
"parameters": [
 {
   "name": "name",
   "description": "User's name",
   "required": true,
   "type": "string",
   "in": "query"
 },
 {
   "name": "email",
   "description": "User's email",
   "required": false,
   "type": "string",
   "in": "query"
 },
 {
   "name": "id",
   "description": "User ID",
   "required": true,
   "type": "integer",
   "format": "int64",
   "in": "query"
 }
]
```

For further details about this annotation, usage and edge cases, check out the javadocs (@ApiImplicitParam, @ApiImplicitParams).

@ResponseHeader

If you want to describe a response header, you can simply add it to your @ApiOperation or @ApiResponse, while supplying the name of the header, a description and a type.

For example, in a given response, it would look as follows:

```
    @ApiResponses(value = {
        @ApiResponse(code = 400, message = "Invalid ID supplied",
                     responseHeaders = @ResponseHeader(name = "X-Rack-Cache", description = "E
        @ApiResponse(code = 404, message = "Pet not found") })
    public Response getPetById(...) {...}
```

For further details about this annotation, usage and edge cases, check out the javadocs.

Model Declaration

@ApiModel

Swagger-core builds the model definitions based on the references to them throughout the API introspection. The `@ApiModel` allows you to manipulate the meta data of a model from a simple description or name change to a definition of polymorphism.

This translates to the Schema Object in the Swagger Specification.

At its basic functionality, you an use `@ApiModel` to change the name of the model and add a description to it:

```
@ApiModel(value="DifferentModel", description="Sample model for the documentation")
class OriginalModel {...}
```

Here we change the name of the model from OriginalModel to DifferentModel.

The output would be:

```
 "DifferentModel": {
      "description": "Sample model for the documentation",
      .
      .
  }
```

You can also use `@ApiModel` to implement model composition, by specifying subtypes like:

```
@ApiModel(value = "Pet", subTypes = {Cat.class})
public class Pet {

}
```

An example of this scenario is available in swagger-samples:

https://github.com/swagger-api/swagger-samples/blob/master/java/java-jaxrs/src/main/java/io/swagger/sample/model/Pet.java

https://github.com/swagger-api/swagger-samples/blob/master/java/java-jaxrs/src/main/java/io/swagger/sample/model/Vehicle.java

For further details about this annotation, usage and edge cases, check out the javadocs.

@ApiModelProperty

While swagger-core will introspect fields and setters/getters, it will also read and process JAXB annotations. The `@ApiModelProperty` allows controlling Swagger-specific definitions such as allowed values, and additional notes. It also offers additional filtering properties in case you want to hide the property in certain scenarios.

```
  @ApiModelProperty(value = "pet status in the store", allowableValues = "available,pending,
  public String getStatus() {
    return status;
  }
```

This is a simple example of adding a short description to the model property. It can also be observed that while `status` is a String, we document it as having only three possible values.

The output of it would be:

```
  "properties": {
        ...,
        "status": {
          "type": "string",
          "description": "pet status in the store",
          "enum": [
            "available",
            "pending",
            "sold"
          ]
        }
      }
```

For further details about this annotation, usage and edge cases, check out the javadocs.

Swagger Definition

@SwaggerDefinition

The SwaggerDefinition annotation provides properties corresponding to many (but not all) top-level properties of the Swagger object, allowing you to set these for your auto-generated definition. The annotation can be on any class scanned during the Swagger auto-configuration process, i.e. it does not have to be on a JAX-RS API class but could just be on a marker/config interface, for example:

```
@SwaggerDefinition(
        info = @Info(
                description = "Gets the weather",
                version = "V12.0.12",
                title = "The Weather API",
                termsOfService = "http://theweatherapi.io/terms.html",
                contact = @Contact(
                   name = "Rain Moore",
                   email = "rain.moore@theweatherapi.io",
                   url = "http://theweatherapi.io"
                ),
                license = @License(
                   name = "Apache 2.0",
                   url = "http://www.apache.org/licenses/LICENSE-2.0"
                )
        ),
        consumes = {"application/json", "application/xml"},
        produces = {"application/json", "application/xml"},
        schemes = {SwaggerDefinition.Scheme.HTTP, SwaggerDefinition.Scheme.HTTPS},
        tags = {
                @Tag(name = "Private", description = "Tag used to denote operations as priva
        },
        externalDocs = @ExternalDocs(value = "Meteorology", url = "http://theweatherapi.io/m
)
public interface TheWeatherApiConfig {
}
```

The properties shown above will result in the corresponding metadata to be added to the generates swagger.json / swagger.yaml file

If you have multiple @SwaggerDefinition annotations they will be aggregated in the order they are found - any duplicate annotation properties will overwrite previous ones.

@Info

The @Info annotation adds general metadata properties for a Swagger definition - corresponding to the Info object in the specification. As in the example above:

```
@SwaggerDefinition(
        info = @Info(
                description = "Gets the weather",
                version = "V12.0.12",
                title = "The Weather API",
                termsOfService = "http://theweatherapi.io/terms.html",
                ...
        ),
...
```

See the javadoc for a complete list of supported properties.

@Contact

The @Contact annotation adds contact properties to the @Info section of a Swagger definition - corresponding to the Contact object in the specification. As in the example above:

```
@SwaggerDefinition(
        info = @Info(
                ...
                contact = @Contact(
                   name = "Rain Moore",
                   email = "rain.moore@theweatherapi.io",
                   url = "http://theweatherapi.io"
                ),
                ...
        ),
...
```

See the javadoc for a list of supported properties.

@License

The @License annotation adds license properties to the @Info section of a Swagger definition - corresponding to the License object in the specification. As in the example above:

```
@SwaggerDefinition(
        info = @Info(
                ...
                license = @License(
                   name = "Apache 2.0",
                   url = "http://www.apache.org/licenses/LICENSE-2.0"
                )
        ),
...
```

See the javadoc for a list of supported properties.

@Extension

The extension annotation allows for adding of extension properties to a Swagger definition. It is currently supported within the @ApiOperation, @Info and @Tag annotations. There are two ways to use it:

```
...
   extensions = {
       @Extension(properties = {
           @ExtensionProperty(name = "test1", value = "value1"),
           @ExtensionProperty(name = "test2", value = "value2")
       })
    }
...
```

which will result in the following json:

```
...
   "x-test1" : "value1",
   "x-test2" : "value2"
...
```

The property name will automatically be prefixed with "x-" if it isn't done so explicitly in the annotation.

Alternatively you can name the extension:

```
...
   extensions = {
       @Extension( name = "my-extension", properties = {
           @ExtensionProperty(name = "test1", value = "value1"),
           @ExtensionProperty(name = "test2", value = "value2")
       })
    }
...
```

which will result in the following json:

```
...
   "x-my-extension" : {
      "test1" : "value1",
      "test2" : "value2"
   }
...
```

which wraps the contained extension properties in a JSON object.

@ExtensionProperty

An individual property within an extension - see previous @Extension section for examples.

Customising the Swagger Definition

If you for any reason want to customise the generated Swagger definition beyond what is possible with the annotations, you can provide the Swagger engine with a ReaderListener that provides the corresponding callbacks:

```
public interface ReaderListener {

    /**
     * Called before the Swagger definition gets populated from scanned classes. Use this me
     * pre-process the Swagger definition before it gets populated.
     *
     * @param reader the reader used to read annotations and build the Swagger definition
     * @param swagger the initial swagger definition
     */

    void beforeScan(Reader reader, Swagger swagger);

    /**
     * Called after a Swagger definition has been populated from scanned classes. Use this m
     * post-process Swagger definitions.
     *
     * @param reader the reader used to read annotations and build the Swagger definition
     * @param swagger the configured Swagger definition
     */

    void afterScan(Reader reader, Swagger swagger);
}
```

Any class found during resource scanning with this annotation will be instantiated and invoked correspondingly. For example the following class:

```
public class BasePathModifier implements ReaderListener {
    void beforeScan(Reader reader, Swagger swagger){}

    void afterScan(Reader reader, Swagger swagger){
        swagger.setBasePath( System.getProperty( "swagger.basepath", swagger.getBasePath() )
    }
}
```

Would allow you to override the generated basePath from a system-property.