Swagger

Swagger and OpenApi

Swagger is a specification for documenting REST API. It specifies the format (URL, method, and representation) to describe REST web services. The goal is to enable the service producer to update the service documentation in real time so that client (consumer) can get up-to-date information about the service structure (request/response, model, etc). With **swagger**, documentation systems are moving at the same pace as the server because all methods, parameters, and models description are tightly integrated into the server code, thereby maintaining the synchronization in APIs and its documentation.

Swagger is very helpful for automating the documentation of your APIs, and I always using it for every Spring API Projects. For this article, you'll need a **Spring Boot** application with Rest Controller(s).

Swagger 3 url:

http://localhost:8080/swagger-ui/

```
@Configuration
@EnableSwagger2WebMvc
public class SwaggerConfig {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
          .select()
          .apis(RequestHandlerSelectors.any())
          .paths(PathSelectors.any())
          .build();
    }
}
```

- `@Configuration` annotation indicates that the class has @Bean definition, and tells Spring to scan this class and wire up the bean in the context.
- `@EnableSwagger2` annotation is used to enable the **Swagger2** for your **Spring Boot** application .
- We create a Docket bean and annotate it with `@Bean`.
- `DocumentationType.SWAGGER_2` tells the Docket bean that we are using version 2 of Swagger specification.
- The Docket bean’s `select()` creates a builder (an instance of ApiSelectorBuilder to control the end-points that are exposed by **Swagger**. This is achieved by defining which controllers and which of their methods should be included in the generated documentation.

)
- ApiSelectorBuilder's `apis()` defines the classes (controller and model classes) to be included. Request Handlers can be configured using `RequestHandlerSelectors` and `PathSelectors`.
- ApiSelectorBuilder's `paths()` allow you to define which controller's methods should be included based on their `PathSelectors` (path mappings).

In our sample above we include all by using `any()` for both. It will enable the entire API to be available for **Swagger,** but you can limit them by a base package, class annotations and more.

Swagger with Spring Data Rest

Springfox provides support for Spring Data REST through its *springfox-data-rest* library.

Spring Boot will take care of the auto-configuration if it discovers the *spring-boot-starter-data-rest* on the classpath.

Now let's create an entity named *User*:

```
@Entity
public class User {
    @Id
    private Long id;
    private String firstName;
    private int age;
    private String email;
    // getters and setters
}
```

Then we'll create the *UserRepository* to add CRUD operations on the *User* entity:

```
@Repository
public interface UserRepository extends CrudRepository<User, Long> {}
```

Last, we'll import the *SpringDataRestConfiguration* class to the *SpringFoxConfig* class:

```
@EnableSwagger2WebMvc
@Import(SpringDataRestConfiguration.class)
public class SpringFoxConfig {
   //...
}
```

Note: We've used the *@EnableSwagger2WebMvc* annotation to enable Swagger, as it has replaced the *@EnableSwagger2* annotation in version 3 of the libraries.

Using Swagger-Core Annotations

In order to generate the Swagger documentation, swagger-core offers a set of annotations to declare and manipulate the output. Here a list of most used of Swagger-Core annotations:

Copy of Untitled

Name	Description
@Api	Marks a class as a Swagger resource.
@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.

Name	Description
@ApiResponses	A wrapper to allow a list of multiple ApiResponse objects.

```
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import io.swagger.annotations.ApiParam;
import io.swagger.annotations.ApiResponse;
import io.swagger.annotations.ApiResponses;
...

@Api(description = "Endpoints for Creating, Retrieving, Updating and Deleting of Contacts.",
        tags = {"contact"})
@RestController
@RequestMapping("/api")
public class ContactController {

    ...

    @ApiOperation(value = "Find Contacts by name", notes = "Name search by %name% format", tags = { "contact" })
    @ApiResponses(value = {
        @ApiResponse(code = 200, message = "successful operation", response=List.class )  })
    @GetMapping(value = "/contacts")
    public ResponseEntity<List<Contact>> findAll(
            @ApiParam("Page number, default is 1") @RequestParam(value="page", defaultValue="1") int pageNumber,
            @ApiParam("Name of the contact for search.") @RequestParam(required=false) String name) {
        ...
  return ...
    }

    @ApiOperation(value = "Find contact by ID", notes = "Returns a single contact", tags = { "contact" })
    @ApiResponses(value = {
        @ApiResponse(code = 200, message = "successful operation", response=Contact.class),
        @ApiResponse(code = 404, message = "Contact not found") })
    @GetMapping(value = "/contacts/{contactId}", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Contact> findContactById(
            @ApiParam("Id of the contact to be obtained. Cannot be empty.")
            @PathVariable long contactId) {
        ...
  return ...
    }

    @ApiOperation(value = "Add a new contact", tags = { "contact" })
    @ApiResponses(value = {
        @ApiResponse(code = 201, message = "Contact created"),
        @ApiResponse(code = 400, message = "Invalid input"),
        @ApiResponse(code = 409, message = "Contact already exists") })
    @PostMapping(value = "/contacts")
    public ResponseEntity<Contact> addContact(
            @ApiParam("Contact to add. Cannot null or empty.")
            @Valid @RequestBody Contact contact)
            throws URISyntaxException {
        ...
  return ...
    }

    @ApiOperation(value = "Update an existing contact", tags = { "contact" })
    @ApiResponses(value = {
        @ApiResponse(code = 200, message = "successful operation"),
        @ApiResponse(code = 400, message = "Invalid ID supplied"),
        @ApiResponse(code = 404, message = "Contact not found"),
        @ApiResponse(code = 405, message = "Validation exception") })
    @PutMapping(value = "/contacts/{contactId}")
    public ResponseEntity<Contact> updateContact(
            @ApiParam("Id of the contact to be update. Cannot be empty.")
            @PathVariable long contactId,
            @ApiParam("Contact to update. Cannot null or empty.")
            @Valid @RequestBody Contact contact) {
        ...
  return ...
    }

    @ApiOperation(value = "Update an existing contact's address", tags = { "contact" })
    @ApiResponses(value = {
        @ApiResponse(code = 200, message = "successful operation"),
        @ApiResponse(code = 404, message = "Contact not found") })
    @PatchMapping("/contacts/{contactId}")
    public ResponseEntity<Void> updateAddress(
            @ApiParam("Id of the contact to be update. Cannot be empty.")
            @PathVariable long contactId,
            @ApiParam("Contact's address to update.")
            @RequestBody Address address) {
        ...
  return ...
    }

    @ApiOperation(value = "Deletes a contact", tags = { "contact" })
    @ApiResponses(value = {
        @ApiResponse(code = 200, message = "successful operation"),
        @ApiResponse(code = 404, message = "Contact not found") })
    @DeleteMapping(path="/contacts/{contactId}")
    public ResponseEntity<Void> deleteContactById(
            @ApiParam("Id of the contact to be delete. Cannot be empty.")
            @PathVariable long contactId) {
        ...
  return ...
    }
}
```

```
@ApiModel(description = "Class representing a contact in the application.")
@Entity
@Table(name = "contact")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@Getter
@Setter
public class Contact implements Serializable {

    private static final long serialVersionUID = 4048798961366546485L;

    @ApiModelProperty(notes = "Unique identifier of the Contact.",
            example = "1", required = true, position = 0)
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    @ApiModelProperty(notes = "Name of the contact.",
            example = "Jessica Abigail", required = true, position = 1)
    @NotBlank
    private String name;

    @ApiModelProperty(notes = "Phone number of the contact.",
            example = "62482211", required = false, position = 2)
    private String phone;

    @ApiModelProperty(notes = "Email address of the contact.",
            example = "jessica@ngilang.com", required = false, position = 3)
    private String email;

    @ApiModelProperty(notes = "Address line 1 of the contact.",
            example = "888 Constantine Ave, #54", required = false, position = 4)
    private String address1;

    @ApiModelProperty(notes = "Address line 2 of the contact.",
            example = "San Angeles", required = false, position = 5)
    private String address2;

    @ApiModelProperty(notes = "Address line 3 of the contact.",
            example = "Florida", required = false, position = 6)
    private String address3;

    @ApiModelProperty(notes = "Postal code of the contact.",
            example = "32106", required = false, position = 7)
    private String postalCode;

    @ApiModelProperty(notes = "Notes about the contact.",
            example = "Meet her at Spring Boot Conference", required = false, position = 8)
    @Column(length = 4000)
    private String note;
}
```