**Sprint 1 Evaluation Project**

**Spring Boot RESTful API Documentation and Testing with Swagger 2**

[Swagger 2 in Spring Boot 1](#_Toc15615)

[The Spring Boot RESTful Application 3](#_Toc31661)

[Project Structure 3](#_Toc17951)

[Creating the Product Controller 4](#_Toc29955)

[Time to build out our services. 6](#_Toc6297)

[Setup the Repository 7](#_Toc19522)

[Add the Domain Object 7](#_Toc2168)

[Configuring Swagger 2 in the Application 7](#_Toc30447)

[Configuring application.properties 8](#_Toc2097)

[Running the application for the first time 9](#_Toc11433)

[Using Swagger to test the Restful endpoints 11](#_Toc11255)

[Save Product– /product/add 12](#_Toc18336)

[Get Products- /product/list 15](#_Toc18649)

[Get Product – /product/show/{id} 16](#_Toc9662)

[Update Product –  /product/update/{id} 17](#_Toc5585)

[Delete Product – /product/delete/{id} 19](#_Toc30723)

[Customizing Swagger 20](#_Toc9758)

[Swagger 2 Annotations for REST Endpoints 21](#_Toc6632)

[Swagger 2 Annotations for Model 23](#_Toc4005)

[Summary 24](#_Toc6348)

Spring Boot makes developing RESTful services ridiculously easy — and using Swagger makes documenting your RESTful services easy.

Building a back-end API layer introduces a whole new area of challenges that goes beyond implementing just endpoints. You now have clients which will be using your API. Your clients will need to know how to interact with your API. In SOAP-based web services, you had a WSDL to work with. This gave API developers an XML-based contract, which defined the API. However, with RESTFul web services, there is no WSDL. Thus your API documentation becomes more critical.

API documentation should be structured so that it’s informative, succinct, and easy to read.

In this project , you wil use [Swagger 2](http://swagger.io/" \t "https://dzone.com/articles/_blank)to test and generate REST API documentation for a Spring Boot project.

## 

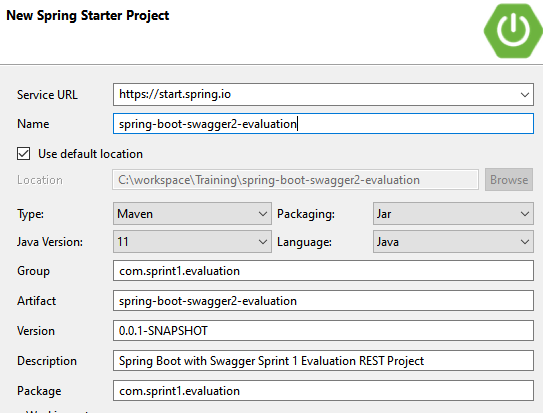
## **Swagger 2 in Spring Boot**

Swagger 2 is an open-source project used to describe and document RESTful APIs. Swagger 2 is language-agnostic and is extensible into new technologies and protocols beyond HTTP. The current version defines a set HTML, JavaScript, and CSS assets to dynamically generate documentation from a Swagger-compliant API. These files are bundled by the [Swagger UI](http://swagger.io/swagger-ui/" \t "https://dzone.com/articles/_blank) project to display the API on the browser. Besides rendering documentation, Swagger UI allows other API developers or consumers to interact with the API’s resources without having any of the implementation logic in place.

The Swagger 2 specification, which is known as [OpenAPI specification](https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md" \t "https://dzone.com/articles/_blank), has several implementations. Currently, [Springfox](https://springfox.github.io/springfox/" \t "https://dzone.com/articles/_blank) that has replaced Swagger-SpringMVC (Swagger 1.2 and older) is popular for Spring Boot applications. Springfox supports both Swagger 1.2 and 2.0.

We will be using Springfox in this project.

Create a Spring initializr project with the folowing entries:



Choose the following dependencies :

– Web

– H2

- Data JPA

To bring Springfox in, add the required dependency declarations in your Maven POM.

In addition to Springfox, we also require Swagger UI. Add the required dependency to your POM.

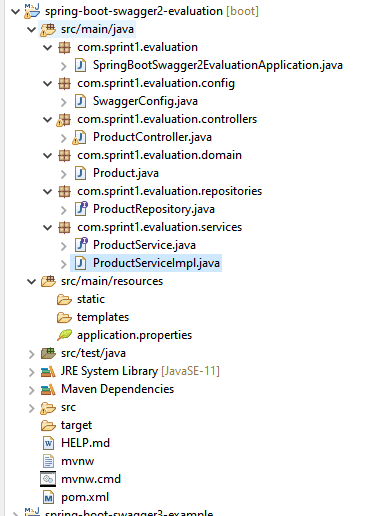
## **The Spring Boot RESTful Application**

Our application implements a set of REST endpoints to manage products. We have a Product JPA entity and a repository named ProductRepository that extends CrudRepository to perform CRUD operations on products against an in-memory H2 database.

The service layer is composed of a ProductService interface and a ProductServiceImpl implementation class.

## **Project Structure**

The project structure is as below:

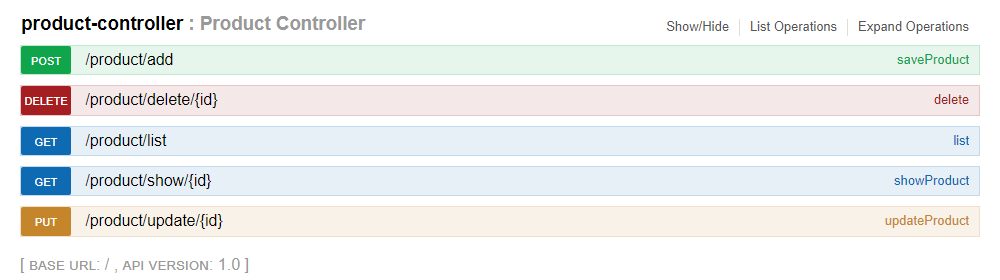


## **Creating the Product Controller**

The controller of the application, ProductController, defines the REST API endpoints. The stub code of ProductController is this:



Once your app is up and running, this is the list of operations which will be displayed in the Swagger UI.



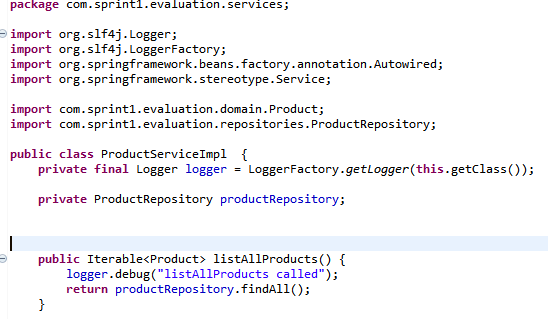
Study the above carefully and update the ProductController as follows:

* At the class level, add the required annotation to designate this class as a Rest Controller. Remember that this annotation works as a convenience annotation to annotate the class with the @Controller and @ResponseBody.
* At the class level, add the required annotation for mapping web requests from "/product" onto methods in this class
* Autowire the ProductService.
* Add the appropriate @RequestMapping annotations to each method so as to corresopond to the operations displayed in the image above.

**Note**: The  class-level annotation maps requests to /product onto the ProductController class. The method-level annotations map web requests to the handler methods of the controller.

## **Time to build out our services.**

By analyzing the Product Controller, code the appropriate methods in the ProductServiceImpl class. The first one is done for you.



Add in the methods corresponding to each operation in Swagger UI.

Autowire the Product Repository.

Use the logger to output the name of the method called as shown above for each method in the service.

ProductServiceImpl implements ProductService interface.

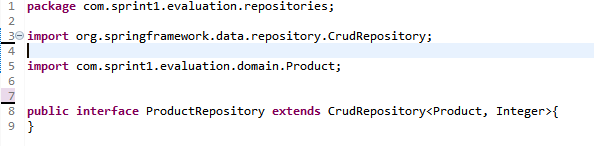
Extract the ProductService from ProductServiceImpl using your IDE.

Override annotations should be added automatically.

You can also build the ProductService interface manually.

## **Setup the Repository**

The Repository tier is simple. We will leverage the functionality provided by Spring Data JPA.



For this to function as a Repository, you need a class level annotation. Add the appropriate annotation.

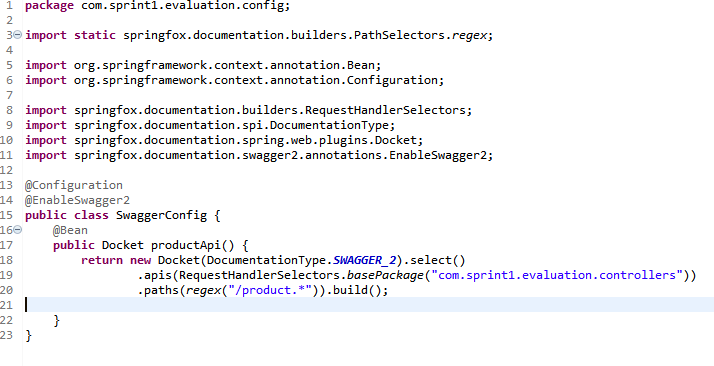
## **Add the Domain Object**

Create a class Product.java in the appropriate package.

* Analyze the updateProduct(…) method in the Product Contoller to ascertain the fields required.
* Annotate Product.java at the class level to designate it as a persistent class.
* Annotate the id variable appropriately to disignate it as the primary key. The Generation Type can be set to AUTO.
* Data Type of the variable price should allow decimal values.
* Add the appropriate Getters and Setters.
* Lastly, add a “**version**” field of type Integer for later use.

## **Configuring Swagger 2 in the Application**

For our application, we will create a Docket bean in a Spring Boot configuration to configure Swagger 2 for the application. A Springfox Docket instance provides the primary API configuration with sensible defaults and convenience methods for configuration. Our Spring Boot configuration class, SwaggerConfig is this.



In this configuration class, the @EnableSwagger2 annotation enables Swagger support in the class. The select() method called on the Docket bean instance returns an ApiSelectorBuilder, which provides the apis() and paths() methods that are used to filter the controllers and methods that are being documented using String predicates.

In the code, the RequestHandlerSelectors.basePackage predicate matches the com.sprint1.evaluation.controllers base package to filter the API. The regex parameter passed to paths()acts as an additional filter to generate documentation only for the path starting with /product.

## **Configuring application.properties**

In application.properties, configure the H2 database:

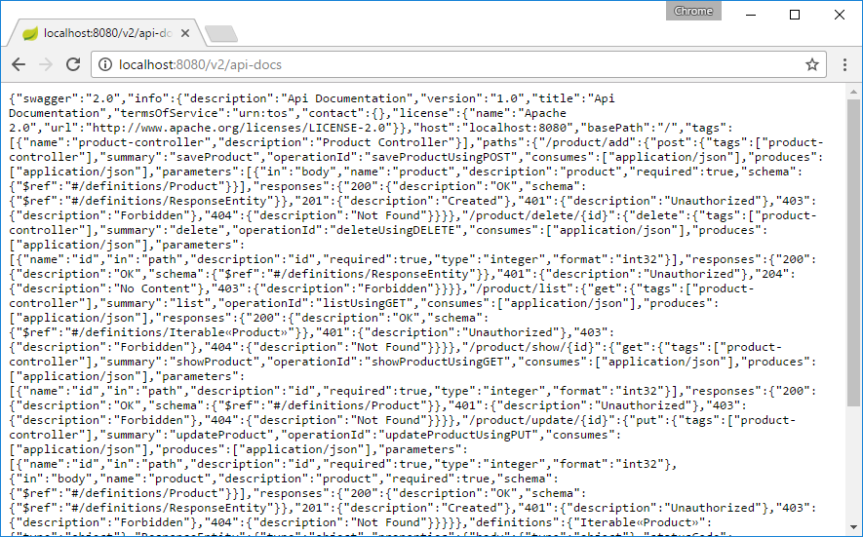
* Enable the H2 console.
* Configure the console path as “/h2”.
* Set “jdbc:h2:mem:test” as the URL of the data source.

One other property is required here before we run the app. In Spring applications AntPathMatcher is used to identify classpath, file system, remote, and other resources in Spring configuration. It has also been used in Spring MVC to match URL paths.

Add the required property for the AntPathMatcher.

## **Running the application for the first time**

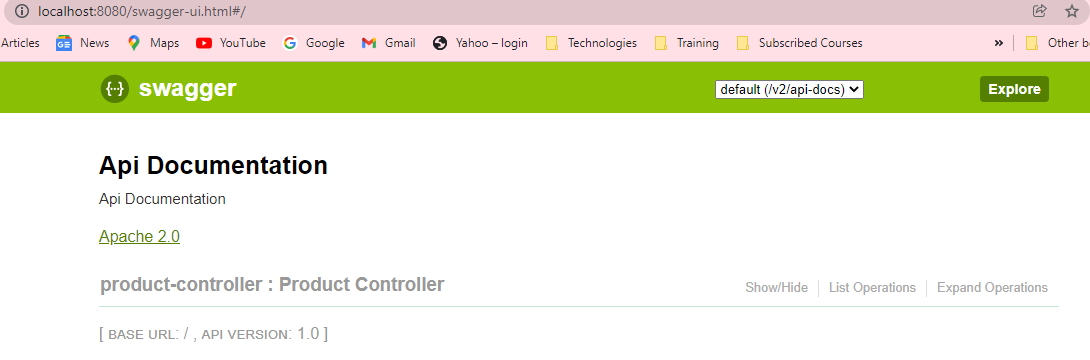
At this point, you should be able to test the configuration by starting the app and pointing your browser to *[http://localhost:8080/v2/api-docs](http://localhost:8080/v2/api-docs.)*[.](http://localhost:8080/v2/api-docs.)



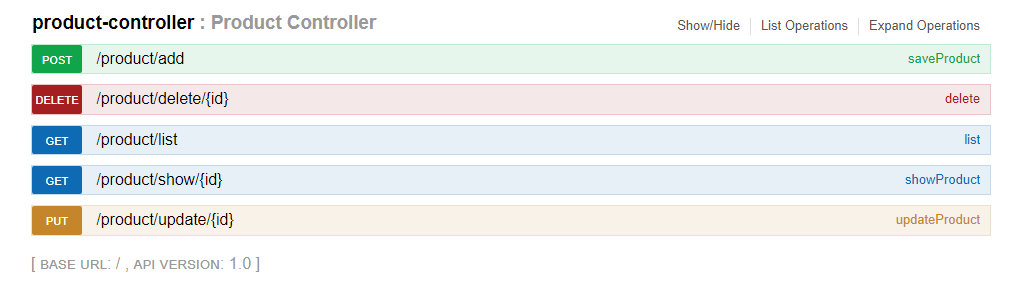
Obviously, the above JSON dump that Swagger 2 generates for our endpoints is not something we want.

What we want is some nice human readable structured documentation, and this is where Swagger UI takes over.

On pointing your browser to <http://localhost:8080/swagger-ui.html>, you will see the generated documentation rendered by Swagger UI, like this:



Click on Show/Hide or List Operations and the ProductControleer will expand to show the list of operations defined in your REST API.



As you can see, Swagger 2 used sensible defaults to generate the documentation of our ProductController.

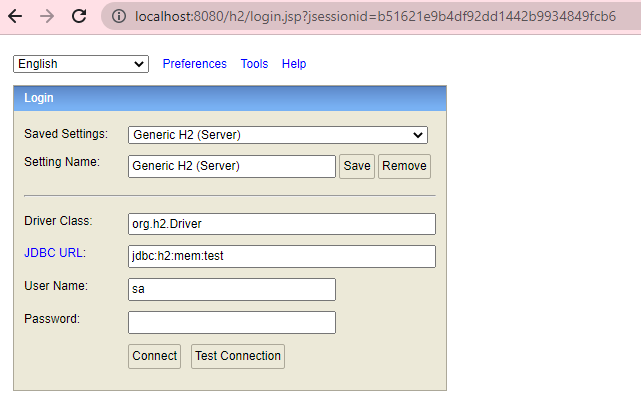
Then, Swagger UI wrapped everything up to provide us an intuitive UI. This was all done automatically. We did not write any code or other documentation to support Swagger.

**Accessing the H2 Console**

Open H2 database console by hitting following URL

**<http://localhost:8080/h2>**

It will show you the following screen :



Click on the Connect button and it will connect you to the H2 database. and you can see that the PRODUCT table has been created, but there is no data in this table, which is as we expected.

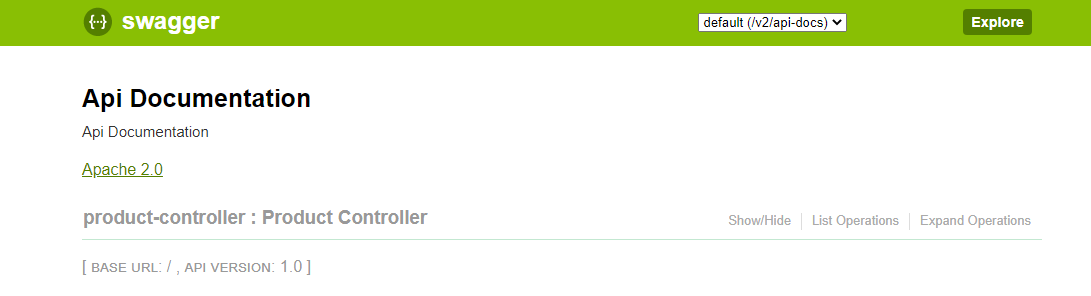


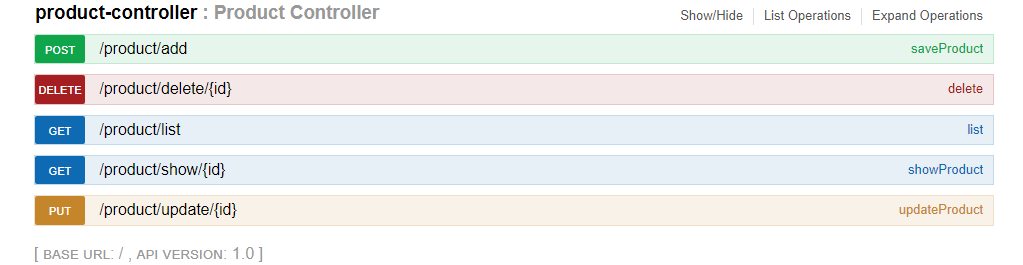
## Using Swagger to test the Restful endpoints

To test your rest endpoints, hit the Swagger URL as follows :

**<http://localhost:8080/swagger-ui.html>**

It will open the following page



Click on the product-controller link. It will show you the supported operations by this controller as follows:

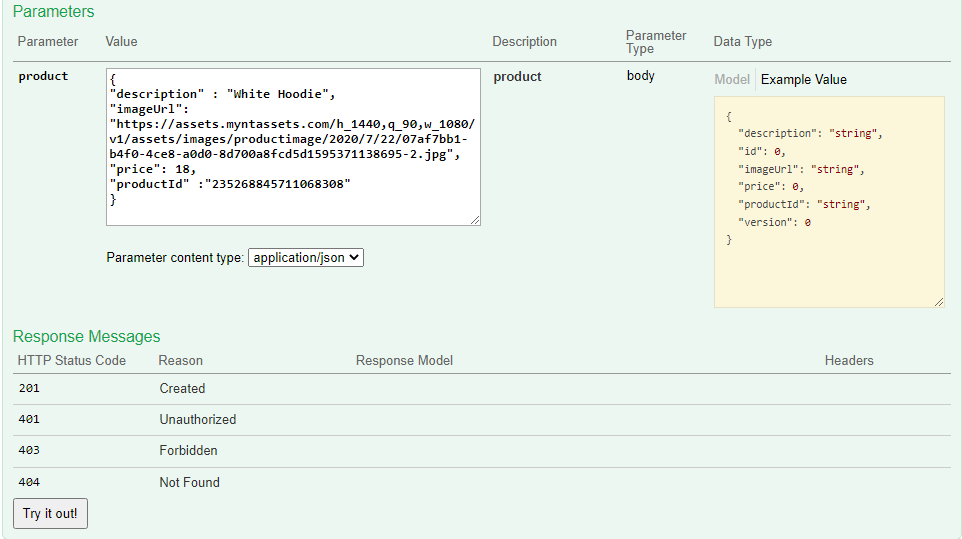
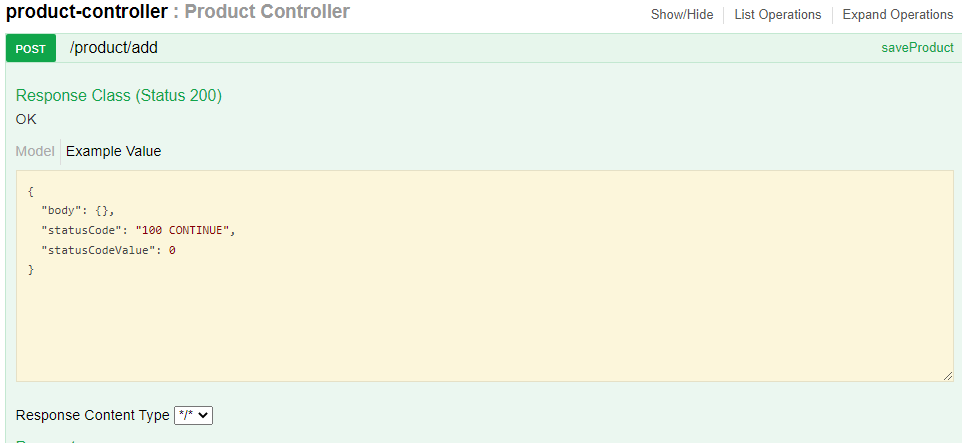
Now we can see five endpoints in the above screenshot. We will test them one by one.

### Save Product– /product/add

The first thing we need to do is to create a resource in the database.for that we will use POST operation and use /product/add endpoint.

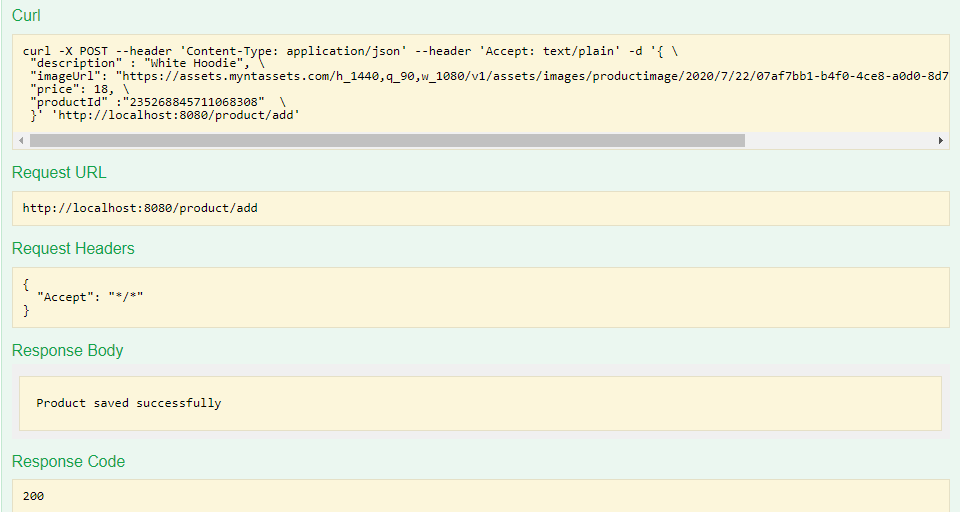


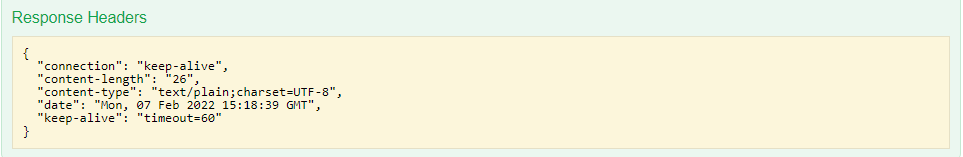
Click on the saveProduct and fill all the required data that we need to create a resource and then click on “Try it out” button.



**You can get any imageUrl from the net and put in in here.**

This is how your request and response will look like





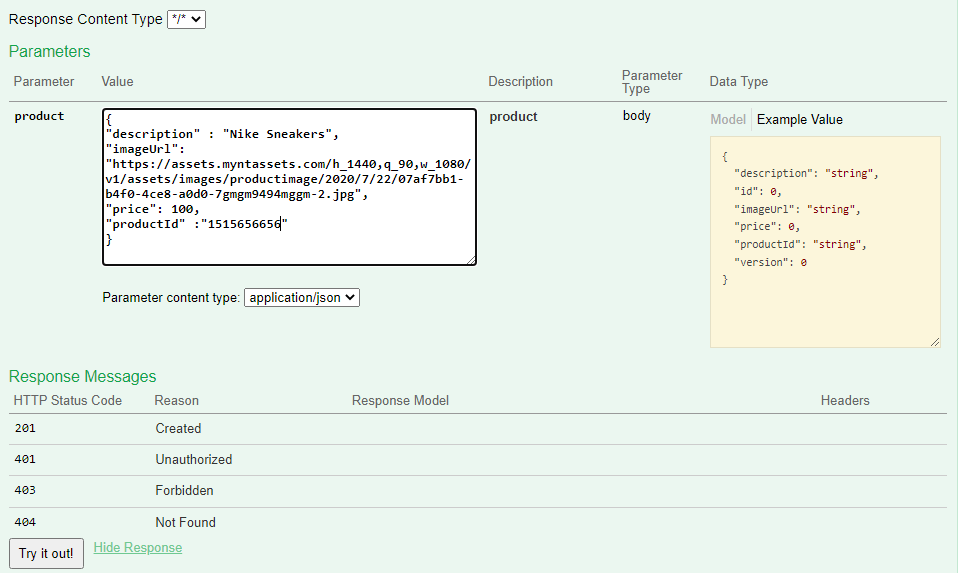
As you can see that the response code is 200, which means SUCCESS and hence our record should have been created in the H2 database.

**Let us check that.**

Open H2 web console and query the PRODUCT table and you can see the record that we pushed from Swagger UI.



Similarly, Insert one more product from Swagger UI with the following data :



Query database again and you will see two records in the database as follows :

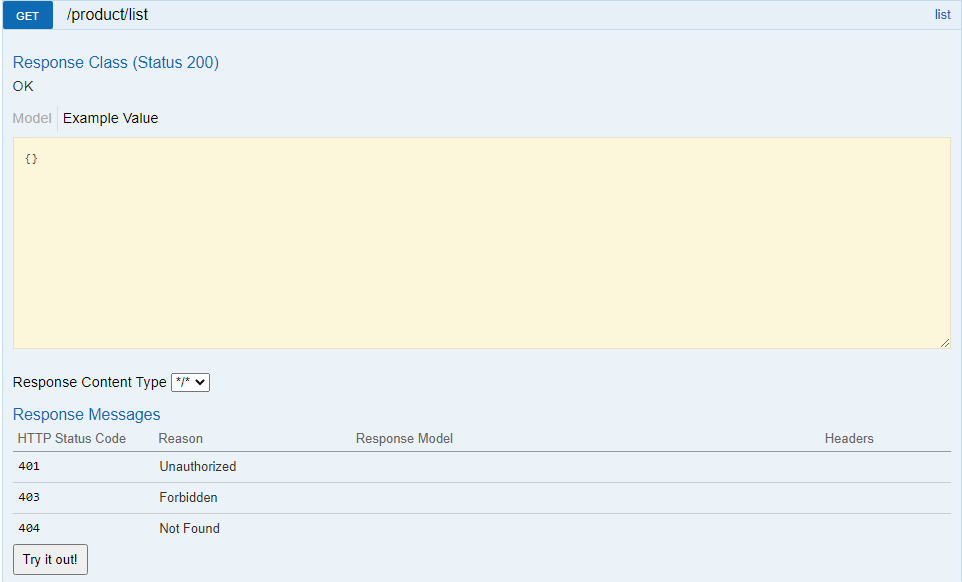


### Get Products- /product/list

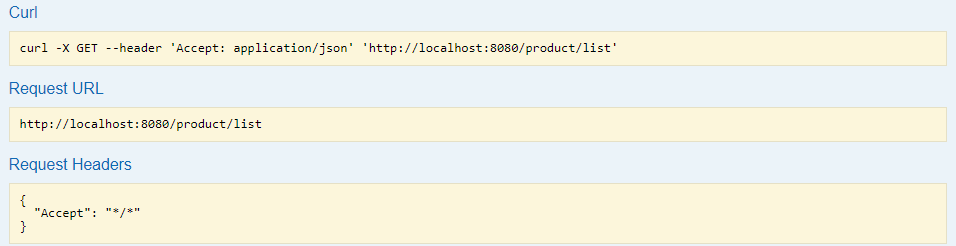
Now as we have already inserted two records  in the database, we will try to retrieve these records with the help of GET operation and using /product/list endpoint as follows :



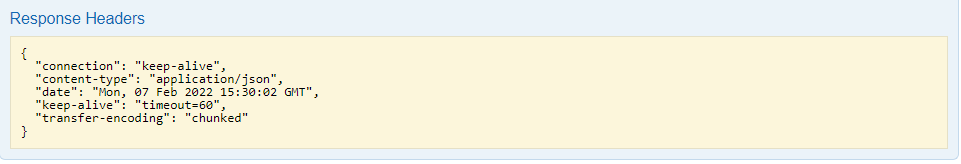
Click on the “**list**” and then as we want to retrieve the list of all products, we need not pass any parameter.



Hence just click on “Try it out” button and you will get a list of products in the response.





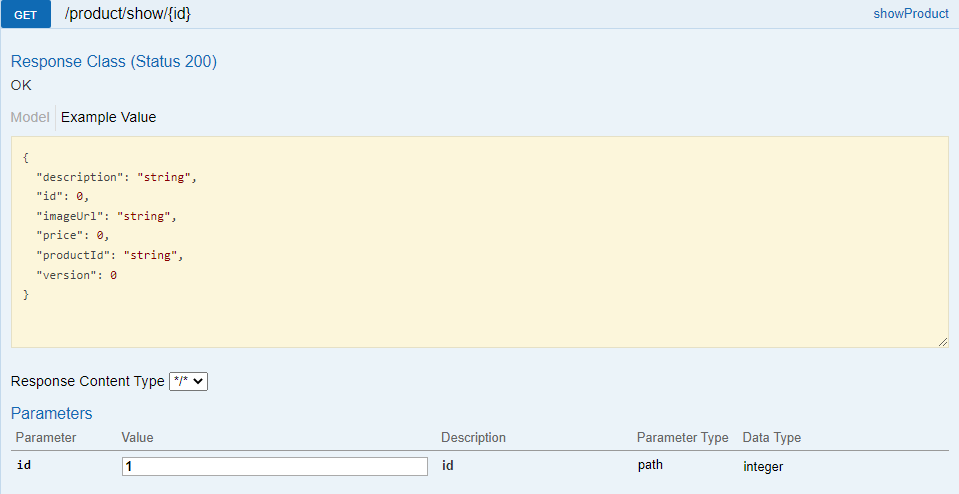


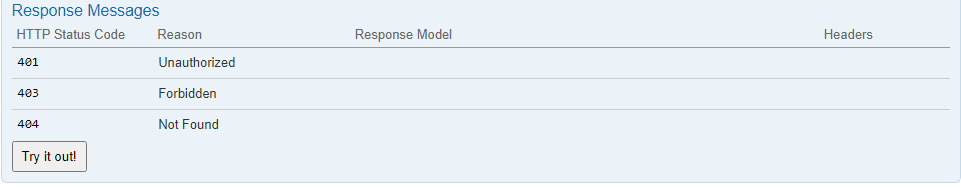
### Get Product **– [/product/show/{id}](http://localhost:8080/swagger-ui.html" \l "!/product45controller/showProductUsingGET)**

Next, we will retrieve only one **product** on the basis of input Id using GET operation. We will pass the Id to the rest endpoint **[/product/show/{id}](http://localhost:8080/swagger-ui.html" \l "!/product45controller/showProductUsingGET).**

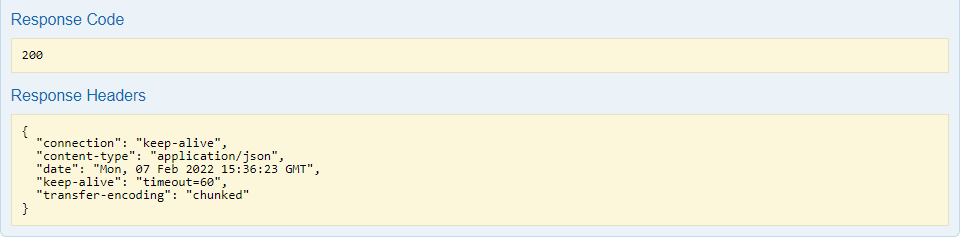


Click on the showProduct and fill id as 1, which means that we want to retrieve the product with id 1.





Click on try it out button and you will see the data of product with Id 1 in the response as follows:

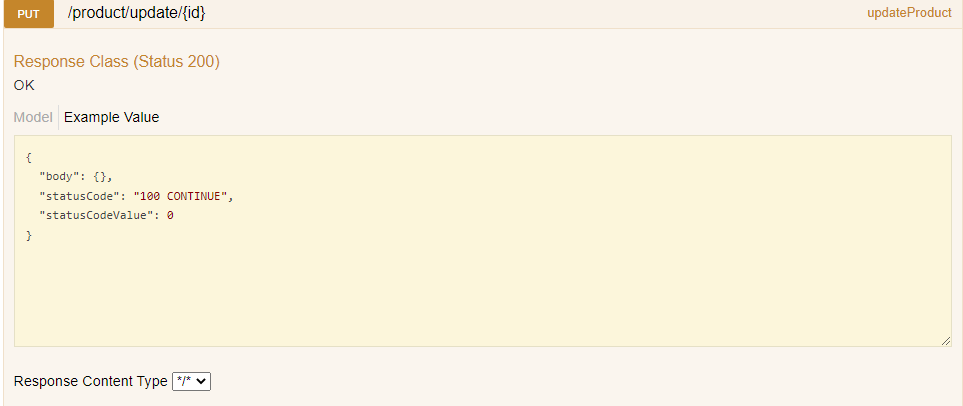


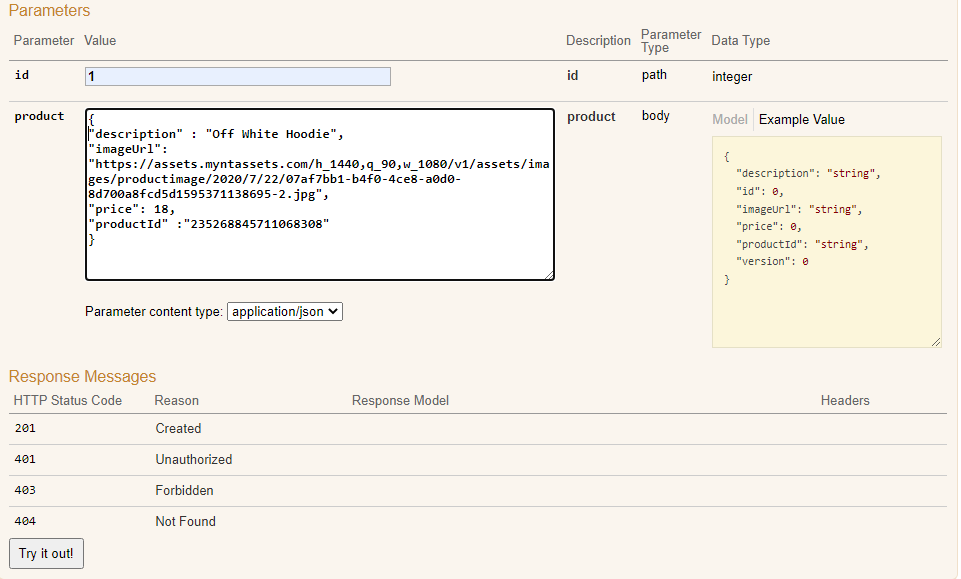
### Update Product –  **[/product/update/{id}](http://localhost:8080/swagger-ui.html" \l "!/product45controller/updateProductUsingPUT)**

Next, we will test the update product rest endpoint using PUT operation and by using **[/product/update/{id}](http://localhost:8080/swagger-ui.html" \l "!/product45controller/updateProductUsingPUT)** endpoint.



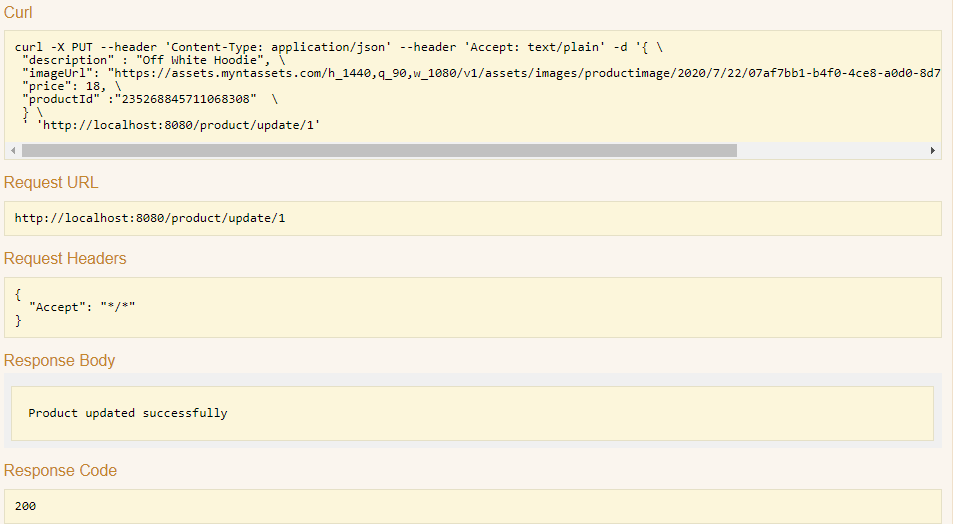
Click on the updateProduct link. Paste one of the product json and put corresponding id as follows :





Change the data in one or more fields.

Click on “Try it out” button and you will see following response with response code 200(SUCCESS).





Verify updated record with updated description from "White Hoodie" to "Off White Hoodie" in the Product table in H2 database for product with id 1.

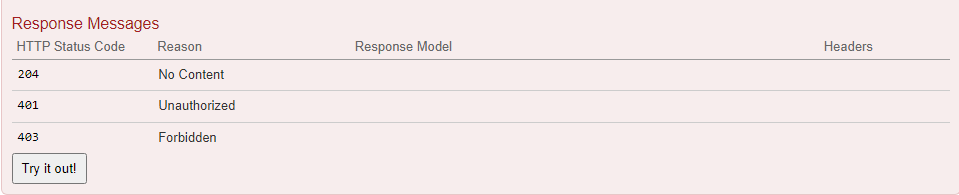


### Delete Product – **[/product/delete/{id}](http://localhost:8080/swagger-ui.html" \l "!/product45controller/deleteUsingDELETE)**

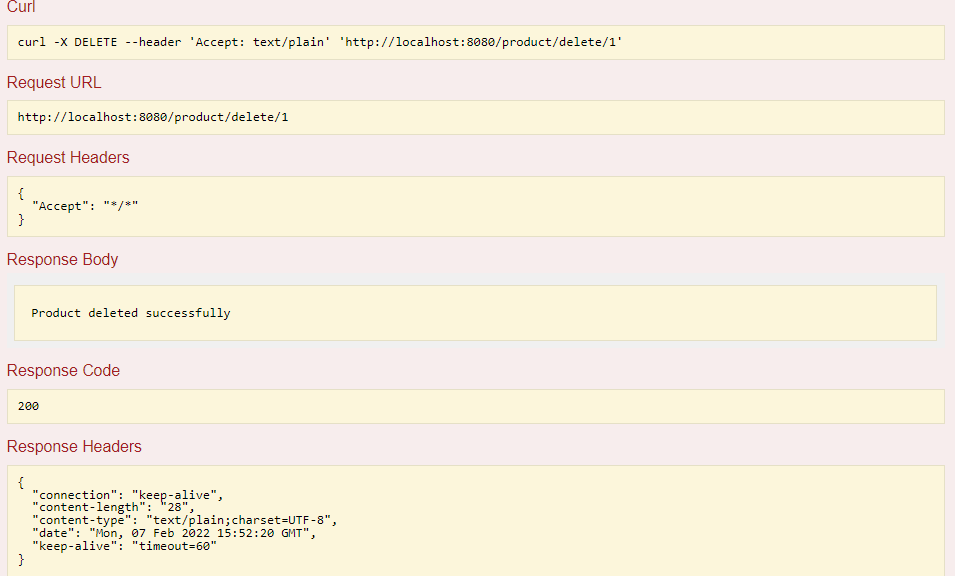
Next, we will test the delete Employee rest endpoint using DELETE operation and be using /product/delete/{id} endpoint.



Click on delete link and fill id 1, which means that we want to delete the product with id 1.



Click on “Try it out” button and you will get response code 200, which means the request is successfully processed.



Let us verify if a product with id 1 has been successfully deleted from the database by opening H2 console and querying the database.



As we can see above that we have the only product with id 2 in the PRODUCT table, hence product with id 1 has been successfully deleted.

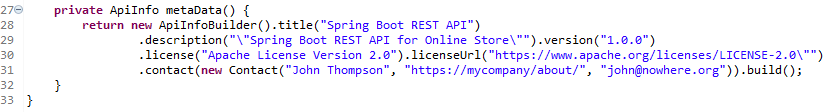
## **Customizing Swagger**

So far, we’ve been looking at Swagger documentation as it comes out of the box — but Swagger 2 has some great customization options.

Let’s start customizing Swagger by providing information about our API in the SwaggerConfig class like this.

SwaggerConfig.java:

Add the following method :-

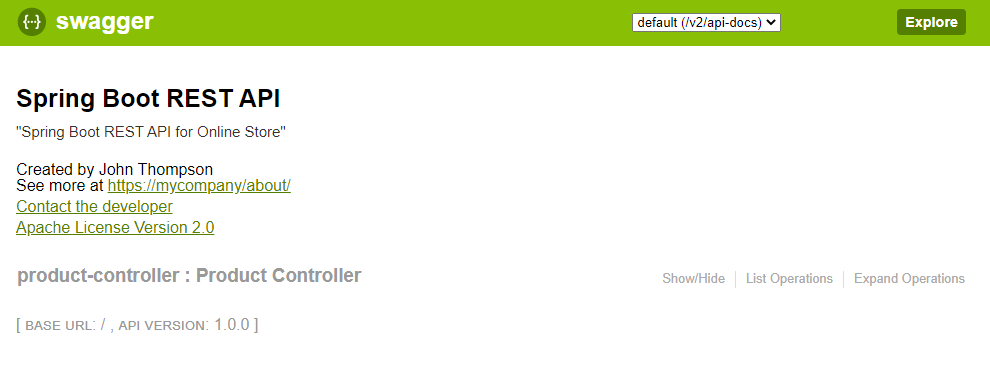


Choose the correct import for the Contact object.

In the SwaggerConfig class, we have added a metaData() method that returns an ApiInfo object initialized with information about our API. Line 23 initializes the Docket with the new information.

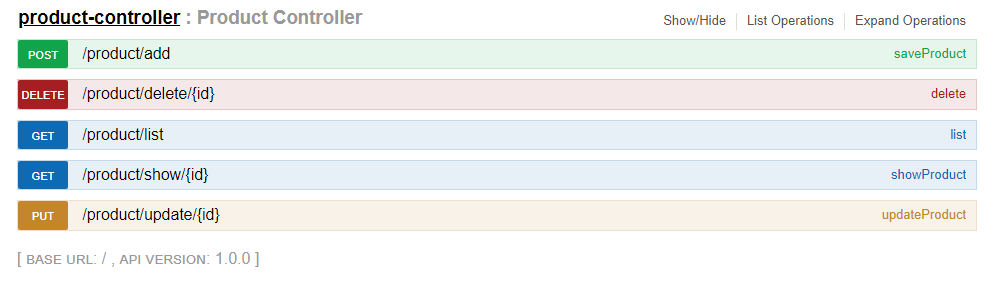


The Swagger 2-generated documentation now looks similar to this:



## **Swagger 2 Annotations for REST Endpoints**

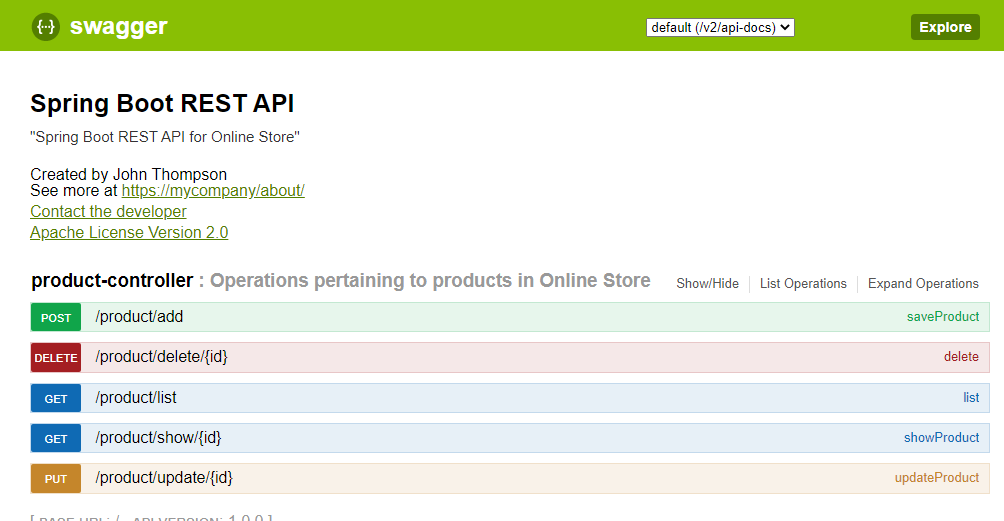
At this point, if you click the product controller link, Swagger UI will display the documentation of our operation endpoints, like this:



We can use the @Api annotation on our ProductController class to describe our API.



The Swagger UI-generated documentation will reflect the description and now looks like this:



For each of our operation endpoints, we can use the @ApiOperation annotation to describe the endpoint and its response type. Add the following to the top of the **list** method:



Swagger 2 also allows overriding the default response messages of HTTP methods. You can use the @ApiResponse annotation to document other responses, in addition to the regular HTTP 200 OK.

Add the following just below the previous annotation on the **list** method:



On the showProduct() method, add the following annotation:



On the saveProduct() method, add the following annotation:



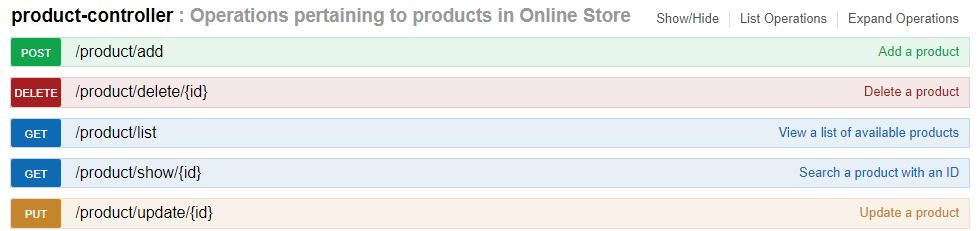
On the updateProduct() method, add the following annotation:

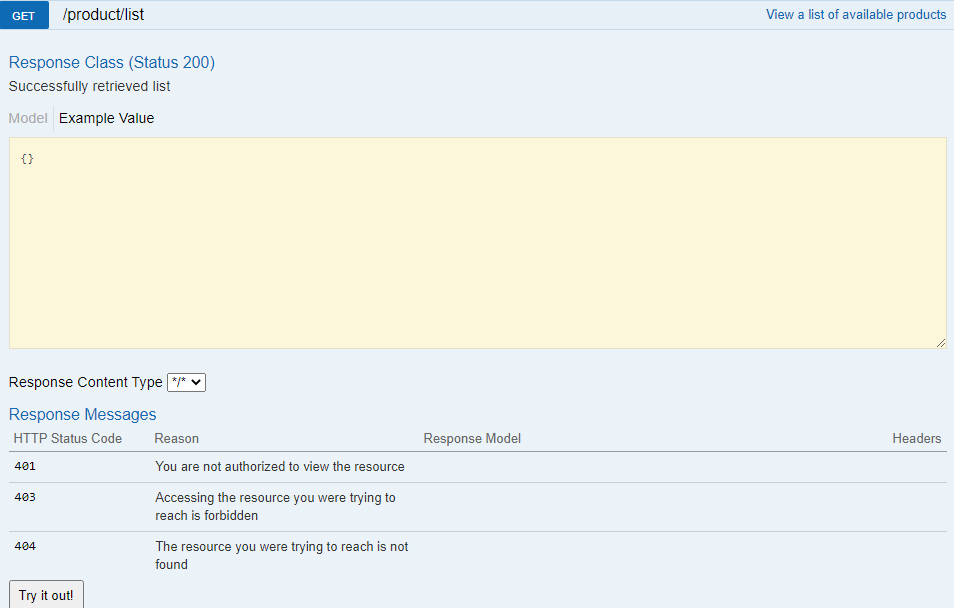


On the delete() method, add the following annotation:



The output of the operation endpoints on the browser is this:





The current documentation is missing one thing: documentation of the Product JPA entity. We will generate documentation for our model next.

## **Swagger 2 Annotations for Model**

You can use the @ApiModelProperty annotation to describe the properties of the Product model. With @ApiModelProperty, you can also document a property as required.

At the moment, the code of yourProduct classshould contain 6 fields.

The following task is pretty simple.

Choose the appropriate annotation from below and place it on the fields - one annotation per field.





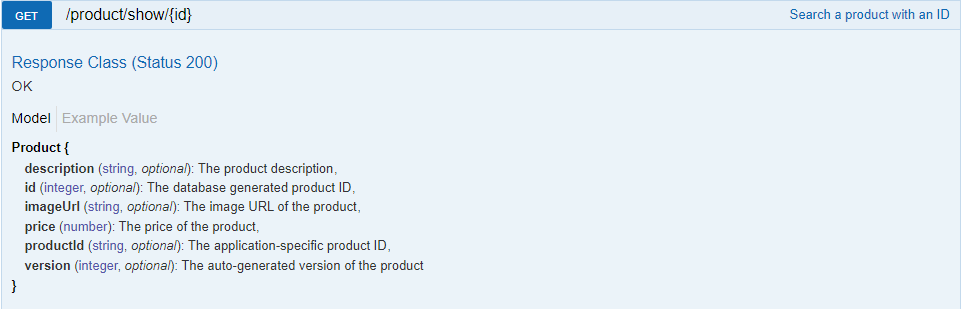








The Swagger 2 generated documentation for Product is this:



## **Summary**

Besides REST API documentation and presentation with Swagger Core and Swagger UI, Swagger 2 has a whole lot of other uses beyond the scope of this post. One of my favorites is [Swagger Editor](http://swagger.io/swagger-editor/" \t "https://dzone.com/articles/_blank), a tool to design new APIs or edit existing ones. The editor visually renders your Swagger definition and provides real-time error-feedback. Another one is [Swagger Codegen](http://swagger.io/swagger-codegen/" \t "https://dzone.com/articles/_blank), a code generation framework for building Client SDKs, servers, and documentation from Swagger definitions.

Swagger 2 also supports Swagger definition through JSON and YAML files. It is something you should try if you want to avoid implementation-specific code in your codebase by externalizing them in JSON and YAML files.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*