(/)

# Implementación del servidor API abierto con OpenAPI Generator

Última modificación: 31 de marzo de 2021

por baeldung (https://www.baeldung.com/author/baeldung/)
(https://www.baeldung.com/author/baeldung/)

**Java (https://www.baeldung.com/category/java/) +**

**Primavera (https://www.baeldung.com/category/spring/) +**

# 1. Información general

fr

Como sugiere el nombre, OpenAPI Generator (https://github.com/OpenAPITools/openapi-generator) genera código a partir de una especificación de OpenAPI (/spring-rest-openapi-documentation) . Puede crear código para bibliotecas cliente, stubs de servidor, documentación y configuración.

e

Es compatible con varios lenguajes y marcos. En particular, hay soporte para C ++, C #, Java, PHP, Python, Ruby, Scala, casi todos los más utilizados (https://openapi-generator.tech/docs/generators/) .

En este tutorial, aprenderemos **cómo implementar un stub de servidor basado en Spring usando OpenAPI Generator a través de su complemento maven** .

o

Otras formas de usar el generador son a través de su CLI (https://openapi-generator.tech/docs/installation/) o herramientas en línea (http://api.openapi-generator.tech/index.html) .

st

# 2. Archivo YAML

Para empezar, necesitaremos un archivo YAML que especifique la API. Lo daremos como entrada a nuestro generador para producir un código auxiliar del servidor.

d

Aquí hay un fragmento de nuestro *petstore.yml* :

.c

'freestar.com/?
ce=branding&utm_name=baeldung_adhesion)

o

m

```
openapi: "3.0.0"
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
          ...
      responses:
        ...
    post:
      summary: Create a pet
      operationId: createPets
      ...
  /pets/{petId}:
    get:
      summary: Info for a specific pet
      operationId: showPetById
      ...
components:
  schemas:
    Pet:
      type: object
      required:
        - id
        - name
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
        tag:
          type: string
    Error:
      type: object
      required:
        - code
        - message
      properties:
        code:
          type: integer
          format: int32
        message:
          type: string
```

# 3. Dependencias de Maven

di

## 3.1. Complemento para OpenAPI Generator

A continuación, agreguemos la dependencia de Maven (https://search.maven.org/search?q=a:openapi-generator-maven-plugin%20AND%20g:%20org.openapitools) para el complemento del generador:

```
<plugin>
    <groupId>org.openapitools</groupId>
    <artifactId>openapi-generator-maven-plugin</artifactId>
    <version>5.1.0</version>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
            <configuration>
                <inputSpec>
                    ${project.basedir}/src/main/resources/petstore.yml
                </inputSpec>
                <generatorName>spring</generatorName>
                <apiPackage>com.baeldung.openapi.api</apiPackage>
                <modelPackage>com.baeldung.openapi.model</modelPackage>
                <supportingFilesToGenerate>
                    ApiUtil.java
                </supportingFilesToGenerate>
                <configOptions>
                    <delegatePattern>true</delegatePattern>
                </configOptions>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Como podemos ver, pasamos el archivo YAML como *inputSpec* . Después de eso, dado que necesitamos un servidor basado en Spring, usamos el *generatorName* como *spring* .

Luego, *apiPackage* especifica el nombre del paquete en el que se generará la API. A continuación, tenemos el *modelPackage* donde el generador coloca los modelos de datos. Con *delegatePattern* establecido en *true* , estamos solicitando crear una interfaz que se pueda implementar como una clase *@Service* (/spring-bean-annotations#service) personalizada .

Es importante destacar que las **opciones para OpenAPI Generator (https://openapi-generator.tech/docs/generators/spring/) son las mismas ya sea que esté usando CLI, complementos de Maven / Gradle o opciones de generación en línea** .

## 3.2. Dependencias de primavera


Como generaremos un servidor Spring, **también necesitamos sus dependencias ( Spring Boot Starter Web (https://search.maven.org/search?q=a:spring-boot-starter-web%20AND%20g:org.springframework.boot) y Spring Data JPA (https://search.maven.org/search?q=a:spring-data-jpa%20AND%20g:org.springframework.data) ) para que el código generado se compile y se ejecute como se esperaba** :

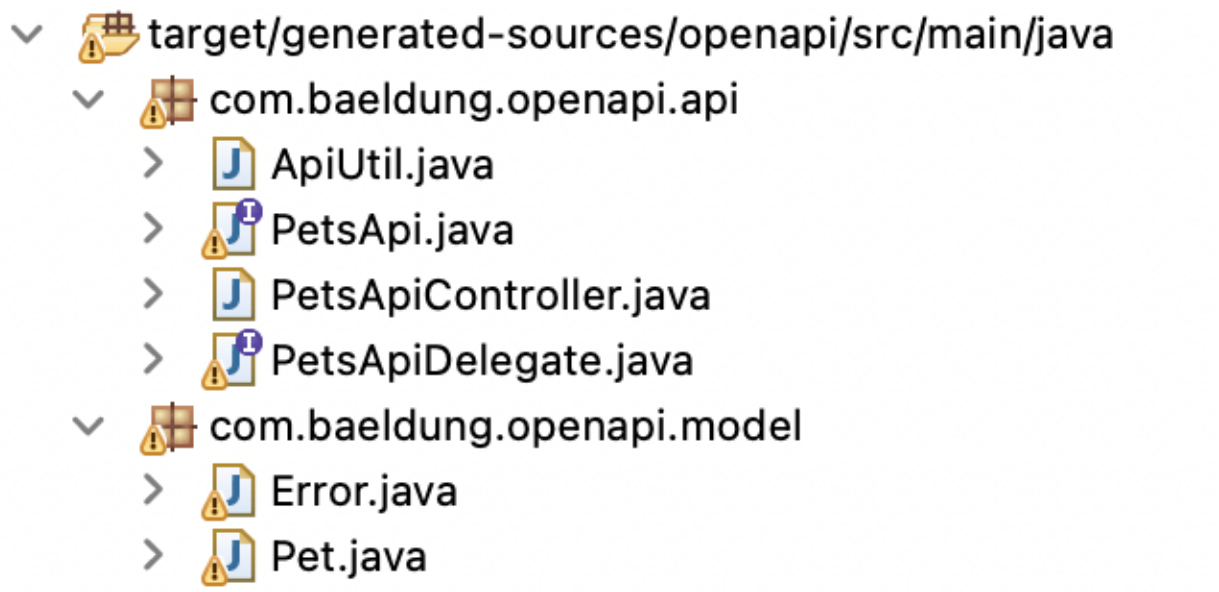
```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>2.4.4</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-jpa</artifactId>
        <version>2.4.6</version>
    </dependency>
</dependencies>
```

# 4. Generación de código

Para generar el stub del servidor, simplemente necesitamos ejecutar:

```
mvn clean install
```

Como resultado, obtenemos:




(/wp-content/uploads/2021/03/OpenAPI-generatedCode.png)

Ahora echemos un vistazo al código, comenzando con el contenido de *apiPackage* .

Primero, **obtenemos una interfaz API llamada *PetsApi*** , que contiene todas las asignaciones de solicitudes como se define en la especificación YAML. Aquí está el fragmento:

```
@javax.annotation.Generated(value =
"org.openapitools.codegen.languages.SpringCodegen",
  date = "2021-03-22T23:26:32.308871+05:30[Asia/Kolkata]")
@Validated
@Api(value = "pets", description = "the pets API")
public interface PetsApi {
    /**
     * GET /pets : List all pets
     *
     * @param limit How many items to return at one time (max 100) (optional)
     * @return A paged array of pets (status code 200)
     *         or unexpected error (status code 200)
     */
    @ApiOperation(value = "List all pets", nickname = "listPets", notes = "",
      response = Pet.class, responseContainer = "List", tags={ "pets", })
    @ApiResponses(value = { @ApiResponse(code = 200, message = "A paged array
of pets",
      response = Pet.class, responseContainer = "List"),
      @ApiResponse(code = 200, message = "unexpected error", response =
Error.class) })
    @GetMapping(value = "/pets", produces = { "application/json" })
    default ResponseEntity<List> listPets(@ApiParam(
      value = "How many items to return at one time (max 100)")
      @Valid @RequestParam(value = "limit", required = false) Integer limit)
{
        return getDelegate().listPets(limit);
    }

    // other generated methods
}
```

En segundo lugar, dado que estamos usando el patrón de delegado, OpenAPI también genera una interfaz de delegador llamada *PetsApiDelegate* para nosotros. En particular, los **métodos declarados en esta interfaz devuelven un estado HTTP de 501 No implementado de forma predeterminada** :

```
@javax.annotation.Generated(value =
"org.openapitools.codegen.languages.SpringCodegen",
  date = "2021-03-22T23:26:32.308871+05:30[Asia/Kolkata]")
public interface PetsApiDelegate {
    /**
     * GET /pets : List all pets
     *
     * @param limit How many items to return at one time (max 100) (optional)
     * @return A paged array of pets (status code 200)
     *         or unexpected error (status code 200)
     * @see PetsApi#listPets
     */
    default ResponseEntity<List<Pet>> listPets(Integer limit) {
        getRequest().ifPresent(request -> {
            for (MediaType mediaType:
MediaType.parseMediaTypes(request.getHeader("Accept"))) {
                if
(mediaType.isCompatibleWith(MediaType.valueOf("application/json"))) {
                    String exampleString = "{ \"name\" : \"name\", \"id\" :
0, \"tag\" : \"tag\" }";
                    ApiUtil.setExampleResponse(request, "application/json",
exampleString);
                    break;
                }
            }
        });
        return new ResponseEntity<>(HttpStatus.NOT_IMPLEMENTED);
    }

    // other generated method declarations
}
```

Después de eso, vemos que **hay una clase *PetsApiController* , que simplemente *conecta* el delegador** :

```
@javax.annotation.Generated(value =
"org.openapitools.codegen.languages.SpringCodegen",
  date = "2021-03-22T23:26:32.308871+05:30[Asia/Kolkata]")
@Controller
@RequestMapping("${openapi.swaggerPetstore.base-path:}")
public class PetsApiController implements PetsApi {

    private final PetsApiDelegate delegate;

    public PetsApiController(
      @org.springframework.beans.factory.annotation.Autowired(required =
false) PetsApiDelegate delegate) {
        this.delegate = Optional.ofNullable(delegate).orElse(new
PetsApiDelegate() {});
    }

    @Override
    public PetsApiDelegate getDelegate() {
        return delegate;
    }
}
```

En el *modelPackage* , un **par de POJOs modelo de datos llamado *error* y *mascotas* se generan** , sobre la base de los *esquemas* definidos en nuestra entrada YAML.

Veamos uno de ellos - *Mascota* :

```
@javax.annotation.Generated(value =
"org.openapitools.codegen.languages.SpringCodegen",
  date = "2021-03-22T23:26:32.308871+05:30[Asia/Kolkata]")
public class Pet {
  @JsonProperty("id")
  private Long id;

  @JsonProperty("name")
  private String name;

  @JsonProperty("tag")
  private String tag;

  // constructor

  @ApiModelProperty(required = true, value = "")
  @NotNull
  public Long getId() {
    return id;
  }

  // other getters and setters

  // equals, hashcode, and toString methods
}
```

# 5. Prueba del servidor


Ahora, todo lo que se requiere para que el stub del servidor funcione como servidor es agregar una implementación de la interfaz de delegador.

Para simplificar las cosas, aquí no haremos eso y solo probaremos el código auxiliar. Además, antes de hacer eso, necesitaremos una *aplicación* Spring :

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

## 5.1. Prueba usando *curl*

Después de iniciar la aplicación, simplemente ejecutaremos el comando:

```
curl -I http://localhost:8080/pets/
```

Y aquí está el resultado esperado:

```
HTTP/1.1 501
Content-Length: 0
Date: Fri, 26 Mar 2021 17:29:25 GMT
Connection: close
```

## 5.2. Pruebas de integración

Alternativamente, podemos escribir una prueba de integración (/spring-boot-testing#integration-testing-with-springboottest) simple para lo mismo:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class OpenApiPetsIntegrationTest {
    private static final String PETS_PATH = "/pets/";

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void whenReadAll_thenStatusIsNotImplemented() throws Exception {
this.mockMvc.perform(get(PETS_PATH)).andExpect(status().isNotImplemented());
    }

    @Test
    public void whenReadOne_thenStatusIsNotImplemented() throws Exception {
        this.mockMvc.perform(get(PETS_PATH +
1)).andExpect(status().isNotImplemented());
    }
}
```

# 6. Conclusión

En este tutorial, **vimos cómo generar un stub de servidor basado en Spring a partir de una especificación YAML utilizando el complemento maven del generador OpenAPI** .

Como siguiente paso, también podemos usarlo para generar un cliente (/spring-boot-rest-client-swagger-codegen#generate-rest-client-with-openapi-generator) .

Como siempre, el código fuente está disponible en GitHub (https://github.com/eugenp/tutorials/tree/master/spring-boot-modules/spring-boot-libraries-2) .

**>> EL CURSO (/ls-course-end)**


¿Está aprendiendo a "construir su API **con Spring** "?

Ingrese su dirección de correo electrónico

**>> Obtenga el eBook**

Acceso (https://www.baeldung.com/wp-login.php?redirect_to=https%3A%2F%2Fwww.baeldung.com%2Fjava-openapi-generator-server)

*Be the First to Comment!*

0 COMENTARIOS

## CATEGORIAS

PRIMAVERA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
LADO DEL CLIENTE HTTP (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

## SERIE

TUTORIAL "VOLVER A LO BÁSICO" DE JAVA (/JAVA-TUTORIAL)
TUTORIAL DE JACKSON JSON (/JACKSON)
TUTORIAL DE HTTPCLIENT 4 (/HTTPCLIENT-GUIDE)
DESCANSO CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
TUTORIAL DE PERSISTENCIA DE PRIMAVERA (/PERSISTENCE-WITH-SPRING-SERIES)
SEGURIDAD CON SPRING (/SECURITY-SPRING)
TUTORIALES REACTIVOS DE PRIMAVERA (/SPRING-REACTIVE-GUIDE)

## ACERCA DE

SOBRE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJOS (/TAG/ACTIVE-JOB/)

EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBE PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
ANÚNCIESE EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)
CONTACTO (/CONTACT)