Revista Peruana de Computación y Sistemas 2022 4(1):25-35
https://doi.org/10.15381/rpcs.v4i1.24125

ISSN Online: 2617-2003
Facultad de Ingeniería de Sistemas e Informática
Universidad Nacional Mayor de San Marcos

Artículo de Contribución

EasyRest: Generador automático de API Rest basado en Spring Framework y motor de plantillas Beetl

Automated Rest API generator based on Spring Framework and Beetl template engine

Alvaro Chavez Chavez [1,a], Lenis Wong Portillo [1,b]

[1] Universidad Nacional Mayor de Marcos, Facultad de Ingeniería de Sistemas e Informática. Lima, Perú

[a] Autor de correspondencia: alvaro.chavez4@unmsm.edu.pe - ORCID: https://orcid.org/0000-0001-7638-8636

[b] E-mail: lwongp@unmsm.edu.pe - ORCID: https://orcid.org/0000-0002-5032-3233

Resumen

En la actualidad *API REST* es el enfoque de desarrollo de aplicaciones más usado a nivel mundial, sin embargo, un gran porcentaje del desarrollo es repetitivo. La repetición en el desarrollo genera pérdidas de tiempo y dinero. En el presente estudio se propone *EasyRest*, un proyecto de generación automática de *API Rest* basado en Spring Framework. Para el desarrollo de esta propuesta se realizó la construcción de un arquetipo personalizado en *Apache Maven*. Además, se realizó la construcción de los *templates* basados en *Beetl*. La propuesta genera un proyecto *backend Rest*, que contiene *controladores*, *servicios* y *repositorios*. Los resultados obtenidos con la propuesta pudieron reducir significativamente el tiempo necesario para la construcción del *API Rest,* en comparación a una construcción manual. Como conclusión la propuesta *EasyRest* permite la creación automática de *API's Rest* en un mínimo de tiempo de desarrollo, automatizando el desarrollo de proyectos *backend*.

Palabras clave: Generador API Rest; Spring Framework Project; Generador Java Project Code; Generador de Proyecto MVC.

Abstract

API REST is currently the most widely used application development approach worldwide, however, a large percentage of development is repetitive. Repetition in development generates waste of time and money. For this purpose, *EasyRest* is proposed, a project for automatic generation of API Rest based on Spring Framework. For the development of this proposal, a custom archetype was built in *Apache Maven*. In addition, the construction of templates based on Beetl is performed. The proposal generates a Rest backend project, which contains *controllers*, *services* and *repositories*. The results obtained with the proposal were able to significantly reduce the time required for the construction of *API Rest*, compared to a manual construction. As a conclusion, the *EasyRest* proposal allows the automatic creation of API's Rest in a minimum of development time, automating the development of backend projects.

Keywords: API Rest Generator; Spring Framework Project; Java Project Code Generator; MVC Project Generator.

Correspondencia:
XXX

Recibido: 28/06/2022 - Aceptado: 04/10/2022 - Publicado: 28/11/2022

Citar como:
Chavez, A. & Wong, L. (2022) EasyRest: Generador automático de API Rest basado en Spring Framework y motor de plantillas Beetl. Revista Peruana de Computación y Sistemas, 4(1):25-35. https://doi.org/10.15381/rpcs.v4i1.24125

1. Introducción

En la actualidad para desarrollar software se necesita un alto nivel técnico y una buena especificación sin embargo muchas veces en el desarrollo de API Rest existen ciertos procesos que son repetitivos (operaciones CRUD, generación de reportes, búsquedas típicas en una BD, Test Unitarios, configuraciones) [1]. Realizar reiteradas veces dichas operaciones ocasiona pérdidas de tiempo y dinero [2], repetir código puede ocasionar problemas de mantenimiento donde al modificar código se tendría que modificar en todos los valores repetidos además de tener un código difícil de mantener [3]. En el desarrollo de software empresarial suceden situaciones que pueden ocasionar demora, el nivel de complejidad aumenta y con ello el tiempo de desarrollo [4] .

Por otra parte, desarrollar API's Rest con Spring Framework [1] es muy sencillo, nos provee distintas soluciones listas para usar y no tener que preocuparnos en escribir código de configuración, gestionar puertos o cargar un contenedor web [5]. A pesar de ello se tiene que construir distintas entidades (clase java que representa una tabla), servicios (clases encargadas de la lógica de negocio), repositorios (clase relacionada con las operaciones de persistencia) y controladores (clase que responde a acciones del usuario), archivos de configuración, utilitarios, documentación de las API's y manejo de excepciones, realizar estas implementaciones consumen tiempo además se vuelven repetitivas. Un generador de código basado en Spring Framework es CRUDLeaf [6] que permite automatizar el proceso de creación mediante meta modelos, si bien esta propuesta es muy interesante y reciente, agrega complejidad al usuario final obligando a entender la sintaxis de un nuevo lenguaje además de no contar con las validaciones correspondientes. Otro aporte muy interesante es Automatic Code Generation of MVC Web Applications [7] que utiliza diagramas UML para la captación de información previniendo errores al momento de ingresar la información y metamodelos como plantilla para generar código sin embargo los diagramas UML cada vez son menos utilizado depender de ello limita su uso. Para la generación de código se puede utilizar distintas técnicas una de ellas son los meta modelos, Ariel Arsaute y su equipo [8] desarrollaron un generador automático de API Rest basado en metamodelos donde utilizan un formato XMI, el aporte solo brinda las clases generadas mas no un proyecto con las configuraciones para su ejecución.

Con la finalidad de agilizar el desarrollo de construcción de API's Rest se propone EasyRest, un generador basado en Spring Framework [1] y el motor de plantillas Beetl [9] donde se busca automatizar el proceso de desarrollo de APÍ's Rest permitiendo que a partir de la lógica de negocio se genere entidades, repositorios CRUD, servicios y controladores, así como archivos de configuración predefinidos, clases utilitarias y manejo de excepciones. Las clases generadas serán agregadas al arquetipo *Maven* que cuenta con una estructura definida y un archivo pom.xml con las dependencias necesarias para la ejecución además se generará la documentación automática mediante Swagger [10].

En el capítulo 3 se describe un resumen del Estado del Arte donde se realizaron 3 preguntas para la revisión y una descripción por cada fase explicando los artículos que más impactaron para el desarrollo de esta propuesta. En la Sección 4 se describe la propuesta, su estructura y el flujo para la generación de API's Rest. En la sección 5 se evalúa la propuesta desarrollada y finalmente en la Sección 6 se presenta la conclusión y trabajos futuros.

2. Trabajos relacionados

La revisión sistemática de la literatura tomando en consideración el trabajo de [11] la cual cuenta las siguientes fases: Planificación de revisión, desarrollo de la revisión y resultados y análisis de la revisión.

En la fase de planeamiento se consideraron tres preguntas, (P1) ¿Cuáles son las técnicas utilizadas para la generación de código?, (P2) ¿Qué tipo de propuestas están siendo implementadas?, (P3) ¿Qué formas de ingreso de requerimientos son utilizadas en los aportes?

Para realizar la búsqueda de los potenciales artículos se utilizaron las siguientes palabras claves "Code generator Java", "Code generator Api Rest", "Automatic generator Java", "Automatic Generator Api Rest", "Code generator Spring Boot", la búsqueda de los artículos se realizó en las bases de datos ScienceDirect, ResearchGate, IEEExplore, Springer Link.

Se establecieron los siguientes criterios de inclusión: Año de publicación entre 2018 y 2021, tipo de fuente Journals o proceedings y artículos relacionados que respondan alguna de las tres preguntas de investigación.

En la fase de desarrollo se realizó la búsqueda de los potenciales artículos teniendo en consideración los criterios de inclusión.

En la fase de resultados se obtuvieron 20 artículos que se analizaron a través de una taxonomía obteniendo las siguientes categorías: *técnicas utilizadas, tipos de propuestas, arquitecturas*, cada categoría está relacionada a una pregunta de investigación planteada.

Tabla 1. *Clasificación de estudios*

Taxonomía	Referencias	Total
Técnicas Utilizadas (P1)	[12], [6], [13], [14], [7], [15], [16], [17], [18], [19] [20], [8], [21], [22]	15
Tipos de Propuestas (P2)	[12], [6], [13], [14], [7], [15], [16], [17], [18], [19] [20], [8], [23], [25], [21], [22]	17
Formas Ingreso de Requerimientos (P3)	[12], [6], [13], [14], [7], [15], [16], [18], [19], [8], [24], [25], [21], [22]	14

En las técnicas utilizadas para generar la generación de código se lograron clasificar 4, generación de código

a partir de meta-modelos [6], [7], [15], [16], [18], [19], [8], [21], [22], reflexión [20], algoritmos de Inteligencia artificial [17], [18], [12], [13], [14]. El tipo de técnica a considerar es la generación de código a partir de meta-modelos ya que permite establecer plantillas que serán utilizadas para la generación de código [26]. En el artículo Automatic Code Generation of MVC Web Applications [7], se propone un software para traducir diagramas del lenguaje de modelado unificado (UML) en código orientado a objetos, la generación automática de código a través de la propuesta ayuda a los diseñadores a entregar software en intervalos cortos de tiempo. Para realizar la implementación utilizaron metamodelos para los casos de uso y clases además de la herramienta XGenerator [7] que transforma dichos modelos en clases java todo ello basado en una arquitectura Modelo-Vista-Controlador.

Dentro del tipo de propuestas como resultado de la revisión se lograron identificar 4, generación de API Rest [7], [15], [25], [24], [20], [8], generación de Casos de prueba [13], [14], [17], [18] , generación de Aplicaciones WEB [7], [16], [21] generación de código [12], [19], [22] dentro de los artículos destaca CRUDyLeaf [6] que desarrollaron un DSL basado en texto para construir API REST de Spring Boot mediante operaciones CRUD. A partir de una sintaxis del CRUDyLeaf DSL se definen las propiedades necesarias para la construcción del proyecto además utilizaron XTend [27], un lenguaje para escribir el generador de código a partir del DSL propuesto, para crear una API Rest que contenga diferentes operaciones CRUD se deberá seguir la gramática propuesta en el DSL, así como especificar el endpoint de salida de la API, el tipo de dato de los campos, groupId, artifactId y demás propiedades.

En las formas de ingreso de requerimientos del usuario como resultado de la revisión se lograron identificar 4, ingreso de información mediante código [6], [8], [12], [13], [14] [15], diagramas UML [7], [19], [22], formularios [16], [25] y en formato JSON [18], [21], [24], [25], destacando el ingreso de información en formato JSON al ser una forma de ingreso simple de utilizar sin tener que aprender algún lenguaje nuevo o depender de los diagramas, dentro de esta categoría destaca Automatic Generation of Test Cases for REST APIs : a Specification-Based Approach [25] que a partir de requerimientos de usuario utilizando metamodelos generan casos de prueba, un punto muy importante es la adaptación del ingreso de información a la especificación OpenAPI [28].

3. EasyRest: Generador API Rest

En el presenta trabajo proponemos EasyRest, un software novedoso para generar API Rest en Spring Framework y documentación automática en Swager a partir de datos ingresados en formato JSON mediante una petición Rest.

La API Rest generada estará construida con el Framework Spring, Spring es un framework de código abierto para la creación de aplicaciones empresariales Java [1]. También se utilizará Maven [29] para la gestión de dependencias además de Lombok [30] para la reducción de código y Swagger [10] para la generación automática de la documentación.

En la Fig. 1 se ejemplifica el flujo de construcción de una API Rest mediante *EasyRest*. Como primer paso se realizará el ingreso de la información mediante una petición http, dicha petición que será interceptada por el Business Controller (2) que se encargará de procesar la petición e invocar a los Generadores (3).

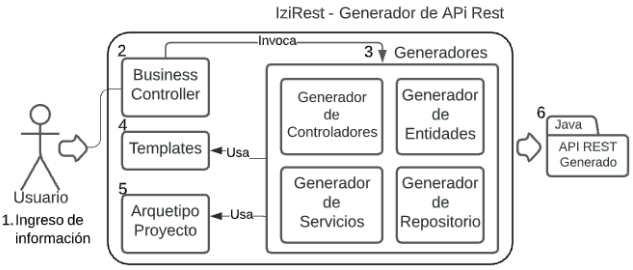

Fig. 1. Flujo de EasyRest

El módulo de generadores se divide en 4, cada división se encarga de generar una capa de la API Rest. El generador de controladores se encarga de generar los controladores necesarios para el funcionamiento de la API Rest. El generador de Entidades se encarga del mapeo entre objetos Java y de agregar las anotaciones necesarias para que la clase sea utilizada como una entidad que estará asignada a una tabla de base de datos. El generador de Servicios se encargará de manejar la lógica de negocio, así como las invocaciones del controlador y las peticiones al repositorio. El generador de repositorio se encargará de realizar las funciones básicas de CRUD a la base de datos mediante JpaRepository [31].

Para que los generadores creen las clases Java es necesario definir templates (4), para ello se dispondrá de una clase y una interfaz template por cada división del generador que funcionan como plantillas donde se agrega los datos ingresados por el usuario según la lógica de negocio. Las clases generadas deberán ser almacenadas en el Arquetipo de un Proyecto Spring (5), dicho arquetipo contendrá algunas clases por defecto como utilitarios, archivos de configuración, así como también contendrá el archivo POM con las dependencias necesarias para una compilación exitosa del API Rest generado.

Para una mejor comprensión del gráfico, se dividirá el flujo del sistema en 6 pasos que se explicaran a continuación.

3.1. Ingreso de información

El ingreso de información solicitada por el generador consta de 4 partes: propiedades del proyecto, propiedades para la documentación de las Apis Rest, propiedades de seguridad e información del negocio.

3.1.1 Propiedades del Proyecto. Las propiedades del proyecto están vinculadas al nombre de la compañía, nombre del proyecto, versión del proyecto, puerto que será utilizado por el proyecto y ruta base de las Apis del proyecto. Esta información será utilizada para definir la estructura del proyecto e información general.

3.1.2 Propiedades para la documentación. Es necesario información de contacto que será utilizada para documentar la Api Rest como descripción del proyecto, email, página web y tipo de licencia del proyecto.

3.1.3 Propiedades de seguridad. Para el consumo de las API Rest es necesario autentificarse es por ello que será necesario proporcionar las credenciales de acceso que se utilizará.

3.1.4 Información del negocio. Contiene la información del negocio, información como nombre del objeto, nombre de la tabla en la base de datos, descripción, ruta que será utilizada por el api Rest y los distintos campos como se visualiza en la Figura 2. La información proporcionada se utilizará para generar las entidades, controladores, servicios y repositorios que serán detallados en los siguientes capítulos.

3.2. Controlador de peticiones

Se realizará una petición http de tipo POST con la información solicitada al controlador del generador de API Rest como se muestra en la figura 3. El controlador de peticiones capturará la petición y transformará el JSON [32] enviado en un objeto Java e invoca al generador central para el procesamiento de la información y generación del proyecto Java, como respuesta de la petición se obtendrá un archivo con extensión zip que contendrá el proyecto generado.

3.3. Definición del arquetipo del proyecto

Un arquetipo es un artefacto muy simple, que contiene el prototipo del proyecto que desea crear. *Maven* provee distintos arquetipos para ser utilizado en proyectos Java [33].

```
"businessInformation": [
  {
    "name": "Product",
    "tableName": "PRODUCTO",
    "description": "Contendra la información del producto",
    "pathName": "product",
    "fields": [
      {
        "name": "id",
        "type":"Integer",
        "description": "identificador del producto",
        "defaultValue:":13,
        "autoIncrement":true,
        "columnName": "ID_PRODUCTO"
      },
      {
        "name": "name",
        "type":"String",
        "description": "nombre del producto",
        "columnName": "NOMBRE_PRODUCTO",
        "length": 70,
        "noNulleable": true
      },
```

Fig. 2. Información del negocio

```
@PostMapping(value = "/generated", produces = MediaType.APPLICATION_OCTET_STREAM_VALUE)
public ResponseEntity<StreamingResponseBody> generated(
        @RequestBody GenericProjectData genericProjectData) throws Exception {
    LOG.info("CentralController :: generated");
    ZipDataResponse zipDataResponse = apiModuleGenerator.generate(genericProjectData);
    return ResponseEntity.ok()
            .header( headerName: "Content-Disposition",
                    ...headerValues: "attachment;filename=" + zipDataResponse.getZipName())
            .contentType(MediaType.valueOf("application/zip"))
            .body(outputStream -> {
                apiModuleGenerator.downloadZip(zipDataResponse, outputStream);
            });
}
```

Fig. 3. Controlador peticiones

A continuación, se detalla la estructura del arquetipo.

3.3.1 Estructura del proyecto generado. La estructura del proyecto generado por *IziRest* estará dividido por paquetes donde cada paquete cumple una función determinada y la comunicación entre paquetes será mediante interfaces como se visualiza en la figura 4.

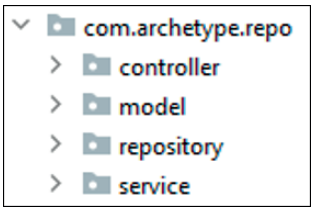

Fig. 4. Estructura del proyecto generado por EasyRest

- Controller: Contendrá las clases encargadas de responder a solicitudes Rest enviadas por el cliente, las clases y los métodos de este paquete utilizaran anotaciones definidas en Spring Framework para peticiones Http.
- Model: Contendrá las clases java que serán utilizadas como una entidad, dichas clases estarán relacionadas a una tabla en la base de datos.
- Service: Contendrá las clases encargadas de la lógica de negocio, desde esta capa se podrá invocar al repositorio o a otros servicios.
- Repository: Basado en el patrón repositorio contendrá las clases que serán utilizadas para la persistencia de los datos, así como el almacenamiento, obtención y búsqueda a una entidad.

3.3.2 Nomenclatura de clases e interfaces. Para nombrar clases e interfaces generadas se realizará el uso de convenciones de nomenclatura [34]. Por ejemplo, para nombrar a una interfaz del controlador de producto usaremos la letra "I" como prefijo seguido de la palabra Producto y "Controller" como sufijo. En la Tabla 2 se muestra los siguientes casos para la API Product.

Tabla 2. *Nomenclatura utilizada para las clases generadas*

Capa	Clase	Interfaz
Controller	ProductController	IProductController
Service	ProductService	IProductService
Entity	ProductEntity	
Repository	ProductRepository	IProductRepository

3.3.3 Arquetipo del proyecto generado. Como se mencionó anteriormente se creará un arquetipo Maven [33] personalizado basado en la estructura del proyecto detallada en el punto 3.3.1 dentro de cada paquete se generarán las clases con las nomenclaturas mencionadas en el punto 3.3.2 además en el arquetipo se agregará un paquete de utilitarios que contendrá clases de apoyo, así como los paquetes de configuración, manejo de excepciones y un archivFo con las credenciales a la base de datos. En la Fig. 5 se muestra la estructura del proyecto generado por EasyRest.

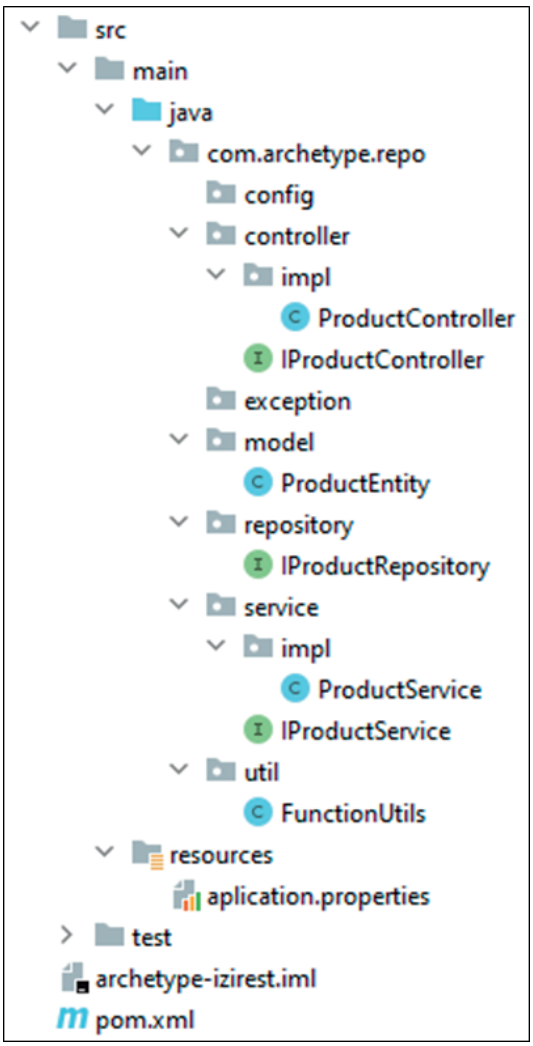

Fig. 5. Estructura de un proyecto generado con EasyRest

3.4. Definición y creación de templates

Si bien a partir del manejo de templates es posible generar código, el objetivo de esta propuesta no es generar la mayor cantidad de código sino generar la mínima cantidad de código que a su vez sea legible y aplique buenas prácticas de programación como principios SOLID [35] en el código generado. Para la creación y manejo de templates se utilizará Beetl [9], Beetl es un motor de plantillas con funciones completas, de rendimiento ultra alto y que tiene una gramática muy simple similar a Javascript además de una fácil integración con distintos marcos Web.

Beetl [9] funciona como una plantilla que es capaz de recibir información enviada mediante programación. Para la creación de los templates se creó distintas plantillas que serán detalladas a continuación.

3.4.1 Plantilla Entity. Una "entidad" es una clase que representa a una tabla, en la plantilla se puede establecer las características que tendrá la entidad para ello se presenta la Tabla 3.

Tabla 3. *Características plantilla Entity*

Característica	Descripción	Valores
Id de la tabla	permite identificar de forma única las filas de la tabla	Auto incremental
Nombre de la columna	-	-
Definición de la columna	Indica el tipo formato en el campo fecha	Fecha corta, larga
Obligatorio	Indica si un campo es obligatorio	Verdadero, falso
Único	Indica si el campo es único en la tabla	Verdadero, falso
Tamaño	Indica el tamaño del campo de tipo texto	-

La elección de una o más características depende de las necesidades del usuario. En la plantilla se puede asignar el nombre de la tabla, valores a los atributos de la clase y permite la reducción de código utilizando anotaciones de Lombok además permite crear relaciones entre entidades.

3.4.2 Plantilla Repository. Spring Framework [5] facilita gestionar las operaciones de persistencia contra una tabla de la base de datos para ello nos provee distintas interfaces. Para la realización de la plantilla se creó una interfaz *IGenericRepository* donde se le envía el tipo de objeto y el id, esta clase extiende de *JpaRepository* [31] que contiene todas las funciones relacionadas a la persistencia.

La plantilla repository tendrá la estructura de una interfaz en Java el nombre de esta interfaz será manejada mediante la lógica de negocio además es necesario que la interfaz extienda de *IGenericRepository* para acceder a métodos de persistencia.

3.4.3 Plantillas para los servicios. Se ha dividido la generación de servicios en 2 plantillas, plantilla de la interfaz del servicio y plantilla de la implementación del servicio.

Debido a que todos los servicios generados implementaran los métodos de la Fig. 6 es necesario centralizar los métodos para ello se ha agregado al prototipo la clase abstracta *GenericServiceImpl* que se encarga de invocar al repositorio respectivo para ejecutar los métodos CRUD.

```
public interface GenericService<D, I> {
                                Tipo dato
    D save(D d);
    D update(D d);
    List<D> findAll();
    D findById(I id);                Id Entidad
    void delete(I id);
}
```

Fig. 6. Interfaz Genérica que provee métodos CRUD

La plantilla Interfaz Servicio deberá de extender la clase *IGenericService* enviando la entidad correspondiente (Fig. 7).

3.4.3.2 Plantilla Implementacion Servicio. La plantilla *ImplementacionServicio* deberá de extender de la clase abstracta *GenericServiceServiceImpl* enviando la entidad y el tipo de id además de sobrescribir el método *getRepo*() donde se enviará el repositorio respectivo. En la Fig. 8 se visualiza la plantilla implementación servicio.

```
public interface ${myUtil.interfaceClassName}
    extends IGenericService<${myUtil.className}Entity, Integer>{

}
```

Fig. 7. Plantilla Interfaz Servicio

```
@Service
public class ${myUtil.customClassName}
    extends GenericServiceImpl<${myUtil.className}Entity, Integer>
    implements ${myUtil.interfaceClassName}{

    @Autowired
    private ${myUtil.className}Repository repo;

    @Override
    protected GenericRepository<${myUtil.className}Entity, Integer> getRepo() {
        return repo;
    }
}
```

Fig. 8. Plantilla Implementación Servicio

3.5. Documentación mediante Swagger

Documentar una API es una tarea muy importante, pero a su vez laboriosa, por suerte existen muchas herramientas que facilitan generar documentación. Para generación automática de API Rest se utilizará la librería SpringFox que nos permite generar documentación basada en Swagger según la lógica de negocio. Además, agrega información adicional que brindaran mayor información al usuario final, para ello se hace uso del template de documentación. En la Fig. 9 muestra la información de contacto proporcionada por el usuario.

3.6. Seguridad en las API REST

Asegurar las API nos ayuda a protegernos de ataques externos ya que exponemos una gran cantidad de datos e información personal. Para la seguridad en las APIS se utilizará el módulo de seguridad de Spring Framework. Se implementará la seguridad de tipo Basic Authentication y se agregará las credenciales proporcionadas por el usuario.

*3.6.1. Autenticación de usuario***.** Para utilizar las APIS es necesario autenticarse con las credenciales proporcionadas por el usuario en el formulario que se muestra en la Fig. 10, si las credenciales no son correctas no podrá hacer uso de ninguna API.

4. Validación

Para la validación del aporte se desarrollará el backend Rest basado en Spring Framework de una tienda online de productos Agrícolas, para ello se realizará el desarrollo en dos casos de estudios, en el primer caso de estudio se realizará el desarrollo común del backend Rest y en el segundo caso se desarrollará con la propuesta.

En cada caso de estudio se tendrá las siguientes operaciones por cada API Rest: consulta por id, consultar todos, guardar, actualizar y eliminar. Las API Rest a desarrollar son Productos, Categorías, Usuarios y Órdenes tal como se muestra en la Tabla 4.

Como resultado de cada caso de estudio se obtendrá la misma arquitectura, estructura, nomenclatura del proyecto, nomenclatura de las clases en las distintas capas que fueron detalladas en el punto 3.3.2.

En la Tabla 5 mostrada a continuación se visualiza las clases java que se obtendrán por la lógica de negocio

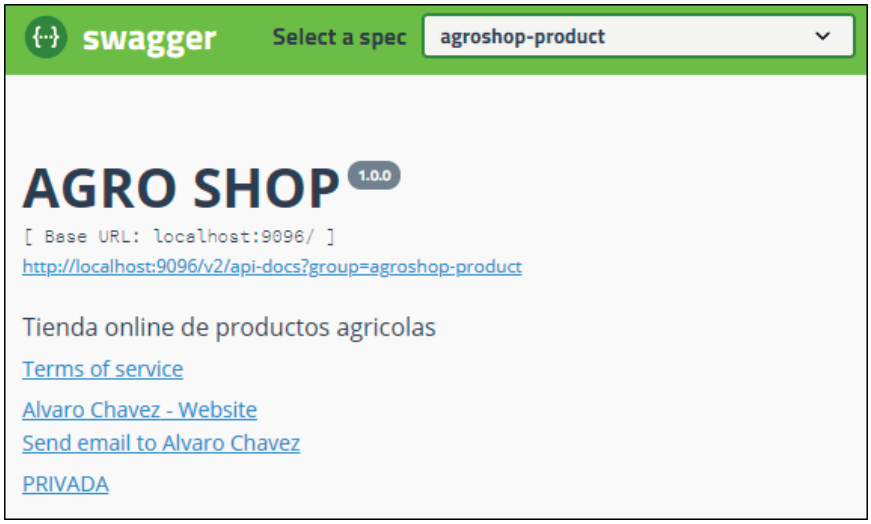

Fig. 9. Información de contacto documentación API Rest

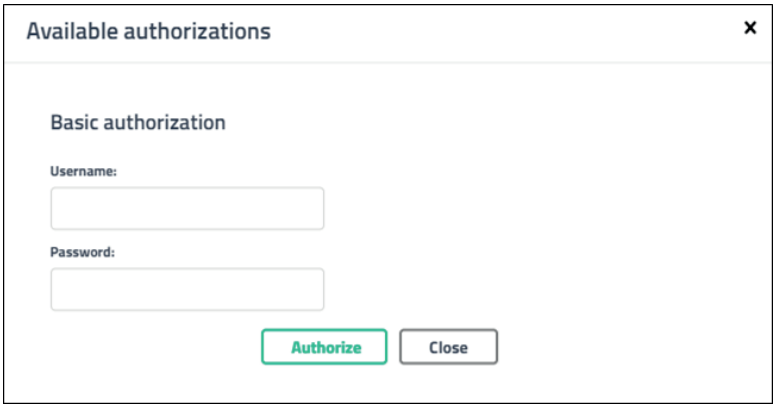

Fig.10. Autenticación usuario.

Tabla 4. *Funcionalidades por cada caso de estudio*

Caso de Estudio	Sujeto de estudio	APIS Rest a validar	Operaciones por cada API Rest	Métricas
Con arquetipo	Tienda online de productos agrícolas	Producto, categorías, usuarios, órdenes	Consultar por id, consultar todos, guardar, actualizar y eliminar	Tiempo de ciclo, Tasa de productividad por LOC
Con el generador API Rest	Tienda online de productos agrícolas	Producto, categorías, usuarios, órdenes	Consultar por id, consultar todos, guardar, actualizar y eliminar	Tiempo de ciclo, Tasa de productividad por LOC

Tabla 5. *Clases generadas en las distintas capas*

Entidad/ Capa	Controller	Service	Repository	Model
Producto	ProductController, IProductController	ProductService, IProductService	IProductRepository	Product
Categoría	CategoriaController, ICategoriaController	CategoriaService, ICategoriaService	ICategoryRepository	Category
Usuario	UserController, IUserController	UserService, IUserService	IUserRepository	User
Orden	OrderController, IOrderController	OrderService, IOrderService	IOrderRepository	Order

para ambos casos de estudio, cabe mencionar que en la Tabla 5 no se tomó en consideración las clases de configuración, seguridad, utilitarios y excepciones.

Para el caso de estudio 1 se utilizará un arquetipo maven que contendrá las clases utilitarias, excepciones, configuraciones y dependencias necesarias para el desarrollo del software, este arquetipo también se encuentra incluido dentro del generador de API Rest. El objetivo de utilizar un arquetipo es centrarnos en el desarrollo de la lógica de negocio y no en clases o archivos de configuración.

Para el caso de estudio 2 se utilizará el software generador de API Rest desarrollado en el aporte. El software generador de API Rest necesita de entrada un JSON con la estructura detallada en el punto 3.1, la información enviada será transformada a una clase Java y será utilizada para generar el proyecto backend Rest según la lógica de negocio.

Posteriormente se procederá a realizar la petición http de tipo POST con la información solicitada en formato de JSON.

5. Resultados y discusión

A continuación, se detallan los resultados obtenidos del proceso de validación realizado al aporte propuesto en el caso de estudio 1.

5.1. Con el generador de API REST

El tiempo que demoro en generar el proyecto backend rest con el aporte fue de 8827 milisegundos. En la Tabla 6 mostrada a continuación se visualiza la cantidad de líneas de código generadas según el tipo de archivo.

Tabla 6. *Cantidad de líneas por tipo de archivo*

Tipos Archivos	Cantidad Líneas Código Fuente (LOC)
Java	1229
Properties	14
XML	140
Total	**1383**

El aporte género las siguientes cantidades de líneas por paquete como se visualiza en la Tabla 7.

Tabla 7. *Cantidad de líneas generadas por tipo paquete.*

Paquete	LOC Generadas
CONFIG	98
CONTROLLER	415
EXCEPTION	147
REPOSITORY	31
SECURITY	70
SERVICE	154
ÚTIL	160
APP	12
RESOURCES	14
POM	140
ENTITY	142
Total	**1383**

Tiempo de ciclo. El tiempo de ciclo se calcula como la diferencia entre la fecha de terminación (FT) y la fecha de inicio (FI). Con el sistema propuesto el tiempo de ciclo fue de 9 segundos aproximadamente (ver ecuación 1).

$$Tiempo\ Ciclo = FT - F\,I \quad (1)$$

Entonces:

$$Tiempo\ Ciclo = 9\ segundos$$

Tasa de automatización por LOC para cada Loc. La tasa de automatización por cada linea de código ingresada es de 6 lineas de código obtenidas, dicho valor se obtuvo dividiendo la cantidad de líneas de código generadas entre la cantidad de líneas de información ingresadas, tal como se observa en la ecuación 2.

$$T.\,Automatizacion\ xLOC = \frac{Cant\ Lineas\ Gen}{Cant\ Lineas\ Ingresadas} \quad (2)$$

Entonces:

$$T.Automatizacion\ xLOC=6$$

A continuación, en la Tabla 8 se visualiza las clases *java* generadas y la cantidad de líneas generadas por cada clase.

Tabla 8. *Cantidad de líneas de código fuente por clase generada*

#	Clase	LOC
1	App.java	12
2	CategoryProductController.java	71
3	CategoryProductEntity.java	23
4	CategoryProductService.java	18
5	ClientController.java	71
6	ClientEntity.java	25
7	ClientService.java	18
8	CORS.java	38
9	ErrorResponse.java	31
10	FileStorageException.java	9
11	GenericResponse.java	52
12	GenericServiceImpl.java	30
13	GlobalExceptionHandler.java	85
14	ICategoryProductController.java	12
15	ICategoryProductRepository.java	5
16	ICategoryProductService.java	5
17	IClientController.java	12
18	IClientRepository.java	5
19	IClientService.java	5
20	IGenericRepository.java	6
21	IGenericService.java	9
22	IOrderController.java	12
23	IOrderDetailController.java	12
24	IOrderDetailRepository.java	5
25	IOrderDetailService.java	5
26	IOrderRepository.java	5
27	IOrderService.java	5
28	IProductController.java	12
29	IProductRepository.java	5
30	IProductService.java	5
31	MessageResponse.java	57
32	MyFileNotFoundException.java	12
33	NoSuchElementFoundException.java	10
34	ObjectMapperJSON.java	13
35	OrderController.java	71
36	OrderDetailController.java	71
37	OrderDetailEntity.java	29
38	OrderDetailService.java	18
39	OrderEntity.java	35
40	OrderService.java	18
41	ProductController.java	71
42	ProductEntity.java	30
43	ProductService.java	18
44	SecurityConfiguration.java	42
45	SecurityProperties.java	28
46	SwaggerConfig.java	98

5.2. Sin el generador de API REST

Para realizar la implementación de la tienda sin el aporte se deberá crear servicios, controladores y repositorios por cada entidad además es necesario comprender las relaciones entre entidades y realizar el mapeo. Este proceso suele ser repetitivo y tedioso cuando se necesita crear una gran cantidad de API Rest

Tasa de automatización por LOC para cada Loc. La tasa de automatización por LOC para cada LOC utilizando la ecuación 2 fue de 1 ya que la construcción del proyecto fue manual y no se utiliza alguna herramienta que permita automatizar algunos procesos.

T.Automatizacion xLOC=1

*Tasa de productividad por LOC***.** Para calcular el tiempo de ciclo fue necesario revisar algunos estudios para encontrar tasas de productividad por líneas de código fuente, estableciendo en promedio 20 LOC por cada hora.

Tasa productividad por LOC =20 LOC x hora

Tiempo de ciclo. Con el generador de *API Rest* se obtuvo 1383 líneas de código fuente (LOC), para generar dichas cantidades de líneas requerirá 69.15 horas aproximadamente, que serían alrededor de 8 días con 3 horas laborales.

Tiempo Ciclo=69.15 horas

5.3. Discusión

En la Tabla 9 mostrada a continuación se visualiza los resultados obtenidos al construir el proyecto *backend rest* con y sin el aporte.

Tabla 9. *Resultados obtenidos según el caso de estudio*

Caso de Estudio	LOC X LOC	Tiempo de ciclo	TR LOC por 1h	TA x LOC para cada LOC
Con el generador	1383	9 segundos	553 200 LOC	6
Sin el generador	1383	69.15 horas	20 LOC	1

Como se puede observar, el aporte solo necesito de 9 segundos aproximadamente para generar 1383 líneas de código fuente a comparación de la construcción manual donde se necesitó 69.15 horas. Además, el aporte reflejó una tasa de automatización 6 LOC por cada LOC ingresado, si bien el valor no es tan alto, esto se debe principalmente al método de ingreso de información ya que al usar un formato JSON es necesario escribir algunas líneas adicionales, cabe mencionar que el principal aporte de este generador no es generar la mayor cantidad de líneas sino generar según las necesidades del negocio una aplicación REST basado en la arquitectura de capas y utilizando plantillas que sirve como iniciación al momento de desarrollar aplicaciones con Spring Framework.

6. Conclusiones

En el presente trabajo se desarrolló un generador de *API REST* basado en plantillas que brinda distintos métodos para ser consumidos con el fin de automatizar el proceso de construcción de proyectos backend con tecnologías Java y Spring Framework. El proyecto generado con el aporte está constituido en distintos paquetes (servicios, controladores, repositorio, entidades) además de la implementación de la seguridad y la documentación automática mediante Swagger. El aporte nos permite enfocarnos en la lógica de negocio dejando de lado tareas repetitivas y archivos de configuración.

El ingreso de la información en formato JSON facilitó la transformación a objetos java permitiendo un fácil tratamiento de los datos.

Maven como herramienta para la gestión y construcción del aporte fue fundamental, ya que permitió la construcción de arquetipos customizados que fueron utilizados en el estudio. Además de brindar un fácil manejo de dependencias y facilitó la compilación y empaquetado del proyecto generado.

Beetl como motor de plantillas permitió la comunicación mediante código Java, además la velocidad de captura de información a la plantilla fue rápida.

7. Trabajos futuros

Como futuros trabajos se propone realizar la generación automática de las *pruebas unitarias,* que es una actividad fundamental en el desarrollo de software y la inclusión de archivos *Dockerfile* para un fácil despliegue. Además de agrupar este tipo de proyecto en dos categorías: monolitos y microservicios

8. Referencias

[1] Gupta A., Suri B. & Misra S. A Systematic Literature Review: Code Bad Smells in Java Source Code. 665-682. 10.1007/978-3-319-62404-4_49, «A Systematic Literature Review: Code Bad Smells in Java Source Code,» *Springer,* 2017.

[2] C. McGregor, «Craftsmanship for Software,» 25 Mayo 2015. [En línea]. Available: https://craftsmanshipforsoftware.com/2015/05/25/the-cost-of-duplicated-code/.

[3] Amjed T., Jens D., Steve C., Sherlock L & Aiko Y, «A large scale study on how developers discuss code smells and anti-pattern in Stack Exchange,» *Information and Software Technology,* 2020.

[4] Huari Casas M., «Revisión sistemática sobre generadores de código fuente y patrones de arquitectura,» *Pontificia Universidad Católica del Perú,* 2020.

[5] Spring io, «Spring Framework,» 2021. [En línea]. Available: https://spring.io/why-spring.

[6] Gómez O., Rosero R. & Cortés K, «CRUDyLeaf: A DSL for Generating Spring Boot REST APIs from Entity CRUD Operations,» *CYBERNETICS AND INFORMATION TECHNOLOGIES,* vol. 20, nº 3, 2020.

[7] Paolone G., Marinelli M., Paesani R. & Di Felice P., «Automatic Code Generation of MVC Web Applications,» *Computers,* 2020.

[8] Arsaute A., Zorzan F., Daniele M., González A. & Frutos M., «Generación automática de API REST a partir de API Java, basada en,» *XX Workshop de Investigadores en Ciencias de la Computaci´on,* 2018.

[9] L. J. (. D. Fu), «Beetl 3,» 2021. [En línea]. Available: http://ibeetl.com/guide/#/beetl.

[10] S. Software, «Swagger IO,» 2021. [En línea]. Available: https://swagger.io/.

[11] Wong L., Mauricio D. & Rodriguez G. «Una revisión sistemática de la literatura sobre la obtención de requisitos de software,» *Revista de ciencia y tecnología de la ingeniería,* vol. 12, nº 2, 2017.

[12] Wei B., Li G., Xia X., Fu Z. & Jin Z, «Code Generation as a Dual Task of Code Summarization,» *33rd Conference on Neural Information Processing Systems,* 2019.

[13] Arcuri A, «RESTful API Automated Test Case Generation,» *IEEE,* Vols. %1 de %2International Conference on Software Quality, Reliability and Security, 2017.

[14] Atlidakis V., Godefroid P. & Polishchuk M. «RESTler: Stateful REST API Fuzzing,» *IEEE/ACEM 41st International Conference on Software Engineering (ICSE),* 2019.

[15] P. Rantanen, «REST API Example Generation Using Javadoc,» *Computer Science and Information Systems,* 2019.

[16] Tijana L., Zeljko. & Gordana M, «A Meta-Model and Code Generator for Evolving Software Product Lines,» 2021.

[17] Arcuri A., «Automated Blackbox and Whitebox Testing of RESTful APIs with EvoMaster,» *IEEE Explore,* 2020.

[18] Ed-douibi H., Cánovas J. & Cabot J., «Automatic Generation of Test Cases for REST, » *IEEE 22nd International Enterprise Distributed Object Computing Conference,* 2018.

[19] Sunitha E. & Philip S, «Automatic Code Generation From UML State Chart Diagrams, » *IEEE Xplore,* vol. 7, 2019.

[20] Nakamaru T. & Chiba S, «Generating a Generic Fluent API in Java,» *The Art, Science, and Engineering of Programming,* vol. 4, nº 3, p. 23 pages, 2020.

[21] Fajardo A., «Método para complementar la generación de códigos de aplicaciones web desde el diagrama de clases UML,» *Ciencia y Tecnologia,* 2019.

[22] Radermacher A., Gérard S., Cam Pham V. & Li S. «Complete Code Generation from UML State Machine,» *Science and Technology Publications,* Vols. %1 de %2In Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, p. 208'2019, 2017.

[23] Venkatraj S., Rajiv V., Vasughi V., Vengatesan K. & Rajesh M. «Development of Test Automation Framework for REST API Testing, » *Computational and Theoretical Nanoscience,* vol. 16, pp. 453-457, 2019.

[24] Hanyang C., Jean-Rémy F. & Xavier B. «Automated Generation of REST API Specification from Plain HTML Documentation, » *Springer International Publishing,* p. 453=461, 2017.

[25] Sohan S., Craig A. & Frank M., «Automated Example Oriented REST API Documentation at Cisco, » *IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track,* 2017.

[26] Fajardo Adolfo, «Método para complementar la generación de códigos de aplicaciones web desde el diagrama de clases UML,» *Ciencia y Tecnología,* nº 19, pp. 35 - 63, 2019.

[27] Eclipse Fundation, «LANGUAGE ENGINEERING FOR EVERYONE!,» 2021. [En línea]. Available: https://www.eclipse.org/Xtext/.

[28] The Linux Foundation, «OPENAPI,» 2021. [En línea]. Available: https://www.openapis.org/about.

[29] The Apache Software Foundation, «Welcome to Apache Maven,» 2021. [En línea]. Available: https://maven.apache.org/.

[30] R. S. Reinier Zwitserloot, «Project Lombok,» 2021. [En línea]. Available: https://projectlombok.org/.

[31] T. D. C. S. M. P. J. B. Oliver Gierke, «Spring Data JPA - Reference Documentation,» [En línea]. Available: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference.

[32] JSON Org, «Introducción a JSON,» [En línea]. Available: https://www.json.org/json-es.html.

[33] The Apache Software Foundation, «Introduction to Archetypes,» 2021. [En línea]. Available: https://maven.apache.org/guides/introduction/introduction-to-archetypes.html.

[34] Rodriguez P., «Convenciones sobre nombres de clases, métodos, interfaces, variables y constantes en java.,» 18 Julio 2020. [En línea]. Available: https://sacavix.com/2020/07/18/convenciones-sobre-nombres-de-clases-metodos-interfaces-variables-y-constantes-en-java/.

[35] Martin C., «Principios SOLID,» 03 Abril 2019. [En línea]. Available: https://enmilocalfunciona.io/principios-solid/.