Índice

1. Introducción

El presente documento trata de describir aspectos técnicos importantes de la implementación de la aplicación **biesca-crud-spring-jpa.**

La aplicación ha sido implementada haciendo uso del framework Spring y tiene como objetivo la gestión de un conjunto de tareas.

2. Tecnologías aplicadas

2.1 Código

- Plataforma de desarrollo Java: OpenJDK 11.
- Uso de funciones lambda.

2.2 PostgreSQL

La base de datos seleccionada ha sido un modelo Entidad-Relación con PostgreSQL. Para la correcta ejecución del servicio será necesaria la conexión con la base de datos/esquema.

El esquema definido por defecto es ***biesca_task_instance***

Dentro de la aplicación proporcionada existe un fichero application.properties, donde se podrán configurar los datos de conexión.

En el proyecto adjunto podemos encontrar la siguiente configuración:

```
#############################
##DATA BASE CONFIG##
#############################

spring.datasource.url=jdbc:postgresql://localhost:5432/biesca
spring.datasource.username=postgres
spring.datasource.password=postgres
```

Donde el nombre de la base de datos es *biesca* sobre *localhost* y bajo el puerto *5432. A*demás, las credenciales actuales por defecto son *postgres* tanto para el usuario como para la contraseña, por lo que en el caso de que el usuario desee probar la aplicación con otros datos de conexión estos deberán de ser cambiados antes de la ejecución del servicio.

2.3 Liquibase: Despliegue automático de modelo de datos

El modelo de datos se desplegará automáticamente al iniciarse la aplicación por primera vez.

Liquibase (https://www.liquibase.org/) permite una gestión de los scripts ejecutados en la base de datos, existiendo un registro por cada uno de los ficheros ejecutados. Estos scripts han sido creados como ficheros yaml.

En el proyecto implementado podremos encontrar dos scripts, uno para la creación del modelo de datos como tablas, claves, secuencias, etc (**db.changelog01_biesca.yaml**) y un segundo script para la inserción de datos iniciales (**db.changelog02_ biesca.yaml**), de forma que el usuario pueda comprobar el funcionamiento de la aplicación desde el momento en que el servicio esté levantado por primera vez. Estos Scripts insertarán:

- 5 posibles estados (task_status) que contendrán una columna (done) que indicará si el estado implica que la tarea está finalizada.

- 3 tareas con estados NEW, DOING y DONE.

Al igual que en el anterior punto, los datos de la conexión vendrán definidos en el fichero properties.

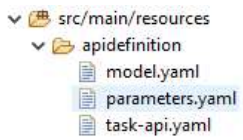
2.4 ModelMapper: Transformación de objetos

El uso de la librería modelMapper ((http://modelmapper.org/) permite la transformación de objetos de entidad (Base de datos) a objeto DTO (negocio) de manera automática, evitando así la transformación manual de cada uno de los atributos de ambas clases.

2.5 Swagger Codegen

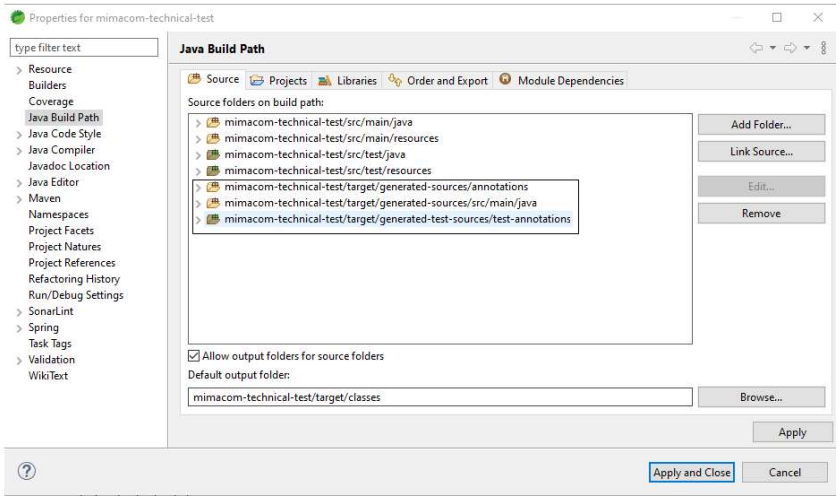
La herramienta swagger CodeGen (https://swagger.io/tools/swagger-codegen/) permite la generación automática de APIS y objetos de negocio (DTO) a partir de ficheros yaml. Es una utilidad muy importante para garantizar la homogeneidad de los objetos compartidos entre varias aplicaciones.

Los ficheros yaml definidos son los siguientes:



Para la generación automática del código será necesario:

- Indicar al IDE que el código generado (carpeta target) formará parte de la aplicación, para ello habrá que acceder a *las propiedades del proyecto/Java BuildPath/pestaña Source*, quedando de la siguiente manera:



- Realizar un *mvn clean install*, de forma que se genere automáticamente el código de los end-points y DTOS.

2.6 Hibernate JPA/SpringData

Se ha usado JPA para definir las relaciones de las entidades de base de datos, asimismo se ha utilizado SpringData para la definición de Querys, Transaccionalidad, etc.

2.7 Testing junit/ Mockito

Se ha creado una clase de Testing para garantizar el correcto funcionamiento de la aplicación en la fase de clean install.

Los test han sido implementados con jUnit haciendo uso de la etiqueta org.junit.runner.RunWith y la inyección de Mocks de los end-points generados.

La **cobertura** de código lograda en los test es de un **86.6%.** Creo que esto es un aspecto importante en la implementación de aplicaciones, evitando así código no probado o no referenciado.

La fase de *testing* se ejecutará en una **base de datos en memoria h2**, la cual será desplegada automáticamente haciendo uso de la herramienta **hibernate-tools**, de modo que a la hora de ejecutar los test se tendrán en cuenta 2 ficheros:

schema.sql: Permite la creación del esquema en tiempo de ejecución del test.
import.sql: Permite la inserción de datos en tiempo de ejecución del test. Estos datos serán usados posteriormente en cada uno de los @Test definidos.

Se han incluido transformaciones de ficheros JSON a objetos de negocio en la fase de Test haciendo uso de la librería *fasterxml.jackson* de forma que no sea necesario crear los objetos JAVA manualmente, sino que las tareas a añadir en el sistema vendrán definidas en un JSON llamado *TASK001_* biesca*.json*, acercándose más a la realidad de un sistema de esta naturaleza.

2.8 Logs

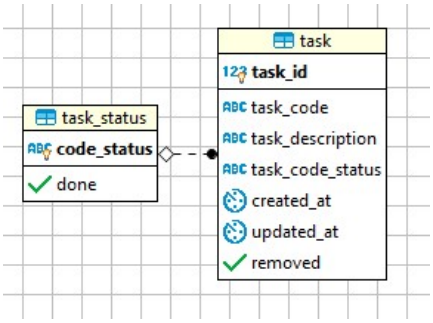
La aplicación creará los logs automáticamente. Dichos logs podrán ser consultados a la altura de la carpeta inicial del proyecto en los ficheros con nombre “technical-test-ms.log”, estos ficheros serán comprimidos al alcanzar un tamaño de forma que existan rotaciones de ficheros.

3. Modelo de datos

El modelo de datos consta de dos tablas:

1. Task: Tabla que contiene las tareas del sistema
2. Task Status: Tabla que contiene los posibles estados de una tarea.

No llegarían a ser necesarias dos tablas para este modelo, pero me ha parecido interesante tenerlas para poder aportar valor añadido y poder realizar operaciones JPA de hibernate con objetos relacionados `@ManyToOne/@OneToMany`



4. End-points

A continuación, se detallan los endpoints (@Controller) implementados en la aplicación:

End-point	Tipo	Descripción	Parámetros

/tasks/add/	POST	Permite añadir una tarea al sistema. Todas las tareas añadidas tendrán asociado el estado NEW	{
			"taskCode": "string",
			"taskDescription": "string"
			}

/tasks/delete/	DELETE	Permite el borrado lógico de una tarea, lo que implica asignar el estado **REMOVED** a la tarea que cumpla con el código de tarea definido en el parámetro	Task-code:String.
/tasks/getAll/	GET	Permite obtener todas las tareas definidas en el sistema	
/tasks/search/done	GET	Permite obtener todas las tareas terminadas en el sistema. Las tareas terminadas son aquellas tareas que tienen el estado **DONE**	
/tasks/search/unfinished	GET	Permite obtener todas las tareas sin terminar en el sistema. Las tareas sin terminar son aquellas tareas que tienen un estado **diferente** a **DONE**	
/tasks/updateTaskStatus/	PATCH	Permite actualizar el estado de una tarea. Los valores posibles vienen definidos en el propio Swagger.	Task-code:String. Task-status: String.
/tasksStatus/find/	GET	Permite obtener los datos propios de un estado	Task-status: String.
/tasksStatus/getAll/	GET	Permite obtener los datos propios de los estados	

5. Pasos para la ejecución

- Crear base de datos *biesca* o reutilizar alguna existente por el usuario, en este segundo caso deberá tenerse en cuenta la configuración en los properties.
- Crear esquema de base de datos con el nombre **biesca *_task_instance.***
- Importar proyecto como tipo maven en el IDE.
- Configurar proyecto para que el código generado automáticamente sea parte del propio proyecto.
- Realizar un mvn clean install del proyecto (esta operación incluirá la ejecución de los tests definidos).
- Ejecutar proyecto Sprint Boot, clase *Application.java,* en este momento el modelo de base de datos se desplegará automáticamente con liquibase.
- Navegar a la URL *http://localhost:8080/swagger-ui.html#/ ,* donde se podrán encontrar los diferentes *end-points.* El puerto **8080** es el puerto por defecto, si es necesario podrá cambiarse en el propio properties del proyecto.
- ¡Jugar!