**SERVIDOR**

Para la realización del servidor encargado de servir los datos a la aplicación web, se ha decidido desarrollar en Java 10, utilizando para ello principalmente dos frameworks, Spring y MyBatis. El proyecto además es compilado a través de Graddle.

* **Spring:** Proporciona una cómoda solución para crear una API Rest, pudiendo definir fácilmente las rutas y un diseño Vista-Controlador. Además, con Spring Security ofrece la posibilidad de implementar niveles de seguridad y controles de acceso a cada ruta, pudiendo identificar al usuario y sus niveles de permisos. Permite al desarrollador centrarse en lo importante y dejar en manos de Spring la conexión y el manejo de la comunicación Cliente-Servidor.
* **Mybatis:** Proporciona una solución a la hora de conectarse a una base de datos y mapear consultas con objetos de Java. A través de interfaces, se definen una serie de métodos, que llevan asociados consultas en ficheros XML, y a la hora de trabajar, desde el código no nos preocupamos de realizar la conexión con la base de datos, ni de realizar las consultas y parsear posteriormente los resultados al objeto deseado. En dichos ficheros XML primeramente relacionamos una columna o un nombre del resultado con un atributo del objeto, pudiendo mapear desde los objetos básicos hasta objetos propios definidos por el desarrollador. Luego podemos definir todas las consultas, especificando el tipo de objeto que devuelve y que recibe, en caso de tener parámetros. Podemos definir parámetros para utilizar en la consulta, como puede ser una clave primaria para la cláusula del *WHERE* o incluso un objeto, pudiendo acceder a los atributos del mismo y utilizarlos en la consulta.

A su vez, el proyecto se divide en dos grandes paquetes:

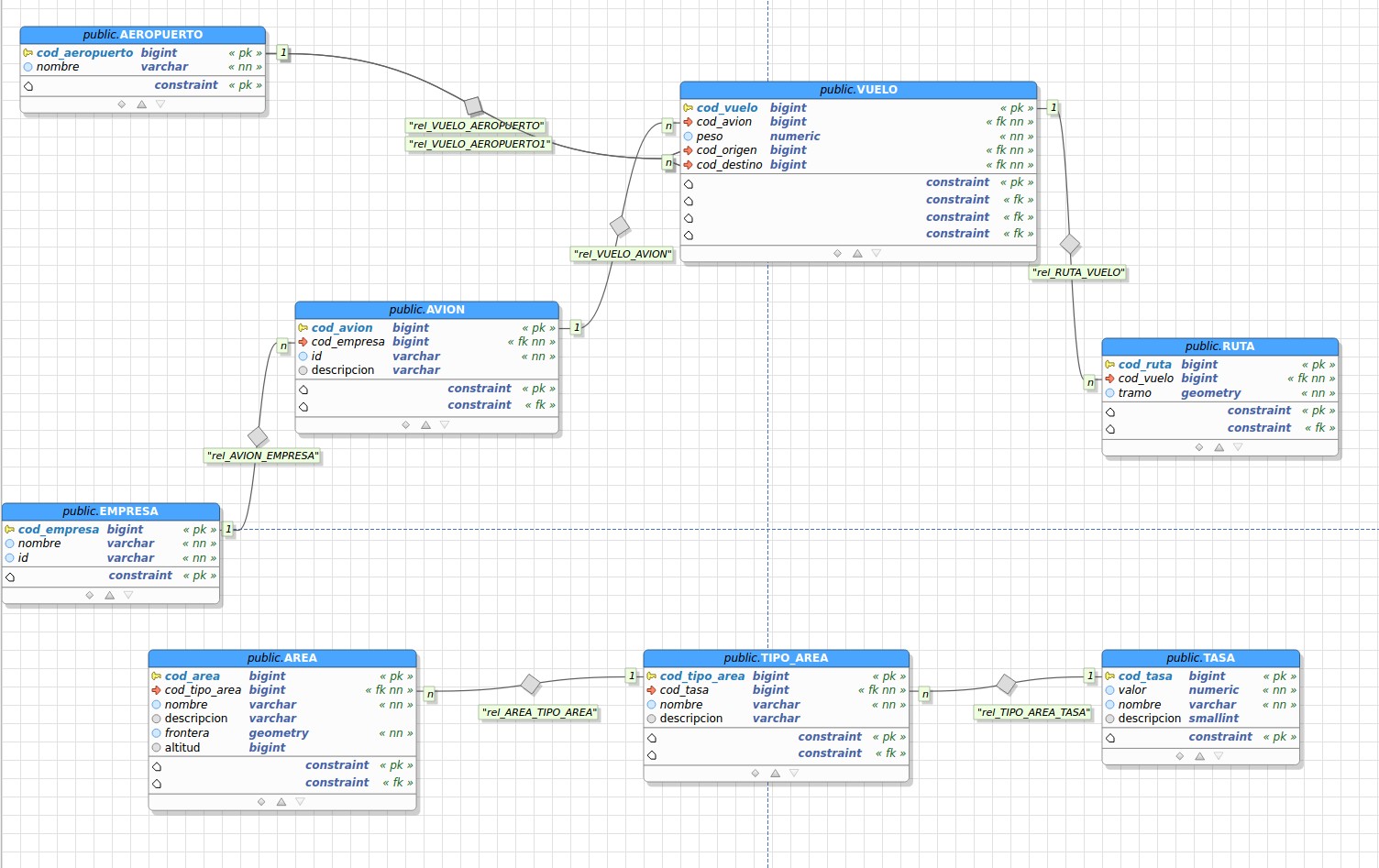
* **src/main/java:**
  + *org.uah.mmaa.config.\*:*Se encuentran las diferentes configuraciones tanto de la seguridad, como de MyBatis, etc. Esto nos permite automatizar la conexión a base de datos, la forma de manejar las peticiones y el cómo respondemos a estas peticiones. Un ejemplo de ello es el interceptado de las peticiones para examinar la presencia de una token de autenticación en las cabeceras, o de automáticamente serializar y deserializar, utilizando la líbrería Gson, los objetos Java a formato JSON para las respuestas.
  + *org.uah.mmaa.core.\*:* Formado por el conjunto de configuraciones relativas a los logs, excepciones, útiles, etc. Aquí es donde se inicializa el servidor, especificando las distintas configuraciones que tiene que seguir, así como la manera en la que debe escribir el log y una serie de respuestas a dar según el error que se pueda producir durante la ejecución.
  + *org.uah.mmaa.features.\*:* Se encuentran las diferentes funcionalidad que aporta el sistema como son lo referente a los aviones,vuelos y rutas. Cada paquete aquí contiene todo lo relacionado con su módulo, desde los objetos Java (*Area, TipoArea,...*) hasta el Controller correspondiente encargado de mapear las distintas direcciones e invocar a los métodos correspondientes, como la interfaz para trabajar con la base de datos.
* **src/main/resources:**
  + *mybatis:* Contiene el fichero jdbc.properties donde se configura el acceso a la base de datos, definiendo la dirección donde se encuentra y las credenciales para el acceso.
  + *mybatis.mappers.\*:* Aquí es donde se encuentran todos los ficheros XML con los mapeos correspondientes y las consultas, todo métodos definidos en la interfaz que se encuentra en el paquete *org.uah.mmaa.features* con el que comparta nombre.
  + *security:* Contiene el fichero jwt.properties para la configuración de la seguridad del sistema, con las propiedades para la expedición de tokens a los usuarios.

Al ser un sistema de muestra, se encuentra preparado para un uso más avanzado, pero se encuentra sin implementar totalmente. Es decir, está definido la manera para realizar un acceso seguro, mediante JWT (*JSON Web Token*), pudiendo tener usuarios con distintos roles. Actualmente, en el fichero de configuración se encuentra desactivada la autenticación para las peticiones, pero el cambio a un modelo seguro sería instantáneo al encontrarse ya las herramientas para ello desarrolladas en el proyecto.

Por último, indicar que en la carpeta */modelo-bd/* se puede encontrar un fichero llamado *modelo\_bbddaa.dbm* en el que está el modelo de la base de datos realizado en el programa pgModeler para su visualización.

**BASE DE DATOS**

El modelo de base de datos desarrollado se puede observar en la siguiente imagen



Entidades:

* **TASA:** Contiene el valor fijo a aplicar y un nombre para identificarlo.
* **TIPO\_AREA:** Define los distintos tipos de área existentes (FIR/UIR Península o los grupos de TMAs) y se relacionan con una tasa concreta.
* **AREA:** Todas las distintas áreas en el sistema, cada una perteneciente a un tipo, aquí se define mediante una *geometry* sus límites, un *Polygon*, y en caso de ser aplicable, la altitud hasta la que surten efecto, así como un nombre para identificarlas.
* **EMPRESA:** Simplemente empresas dadas de alta y su nombre e identificativo.
* **AVIÓN:** Contiene los datos básicos de los aviones y la empresa a la que pertenecen.
* **AEROPUERTO:** Nombres de aeropuertos.
* **VUELO:** Alberga los datos referentes a un vuelo realizado por un avión, como el aeropuerto de origen y destino y el peso con el que ha despegado.
* **RUTA:** Los distintos tramos que ha ido recorriendo el avión durante el vuelo y la altitud en la que lo ha hecho. Estos tramos están definidos como una *geometry*, una *LineString*.

Se han mantenido los datos necesarios para la demostración que nos atañe, pero al igual que el servidor, está planteado para un desarrollo más extensivo, pudiendo llegar a tener información más detallada de aeropuertos, como su localización para representarlos en los mapas, poder desarrollar funciones de cálculos para todos los vuelos de un avión o de una empresa concreta y realizar una facturación completa en tramos para dicha empresa, incluir distintos tipos de áreas sin influir en el resto del modelo al estar tan desacoplado, etc.

Al igual que, simplemente insertando una nueva tabla *USUARIO* y relacionándolo con *EMPRESA*, se podría permitir visualizar los datos a trabajadores respectivos a su empresa y seguir ampliando funciones. Es una buena base de cara a seguir construyendo sobre ella sin tener que realizar cambios significativos.

**CONSULTAS A DESTACAR**

A la hora de trabajar, hay tres consultas relacionadas con *postgis* que merecen la pena destacar:

* **Inserción de datos de las *geometries*:** En el sistema se trabajan con dos *geometries*, los *Polygons* de las áreas y las *LineString* de las rutas. En el primer caso, a la hora de insertar, en el campo hay que introducir “*ST\_GeometryFromText('Polygon((03.1000000 42.260000, 03.2339000 42.013200, ..., 03.1000000 42.260000))')*”. Esto son las coordenadas de cada vértice del polígono que delimita la frontera del área especificada. A través de la función *ST\_GeometryFromText* creamos una *geometry* empleando el formato de texto *WKT* (*Well-Known text*). De forma similar, a la hora de insertar el las rutas, utilizamos “*ST\_MakeLine(ST\_GeomFromText('POINT(-0.524100 51.477200)'), ST\_GeomFromText('POINT(-0.546800 51.474600)'))*”. Primero, con la misma función que para el polígono, definimos los dos puntos necesarios para el inicio y fin de la línea, y con esas dos *geometries* y la función *ST\_MakeLine* ya lo tenemos listo.
* **Selección de los datos de las *geometries*:** A la hora de servir los datos de las áreas o las rutas, debemos enviar las coordenadas para poder representarlas en el mapa. Para ello, existe una función de *postgis* muy útil: “*ST\_AsGeoJSON*”. Esto te devuelve al pasarle una *geometry* una cadena en formato JSON con el tipo de geometría que es y sus coordenadas. Como lo único que nos interesa son las coordenadas, a través de una función de PostgreSQL extraemos ese atributo en concreto, quedando de esta manera “*json\_extract\_path(ST\_AsGeoJSON(frontera)::json, 'coordinates')*”. Con esto conseguimos una cadena de texto con solamente las coordenadas, listas para mandar. En el caso de que nos pidan toda la ruta realizada en un vuelo, lo que nos interesa es la unión de todos los tramos. Convenientemente, la función con la que habíamos insertado los datos de la ruta, *ST\_MakeLine*, funciona también como una función de agregación al recibir un set de resultados. Así que de la misma forma que antes, extraemos las coordenadas resultantes de agregar todos los puntos previamente ordenados para poder dibujar la ruta en el mapa.
* **Cálculo del recorrido del vuelo en las distintas áreas:** Cuando queremos calcular cuánto ha recorrido el avión por las distintas zonas, y cuál es el área vigente para el cobro, empleamos la siguiente consulta:

SELECT a.cod\_area AS a\_cod\_area, a.nombre AS a\_nombre, ta.cod\_tipo\_area AS a\_ta\_cod\_tipo\_area, ta.nombre AS a\_ta\_nombre, t.valor AS a\_ta\_t\_valor, t.cod\_tasa AS a\_ta\_t\_cod\_tasa,

sum(st\_length(st\_intersection(r.tramo, a.frontera)::geography)) / 1000 AS recorrido

FROM "RUTA" r, "AREA" a

LEFT JOIN "TIPO\_AREA" ta USING(cod\_tipo\_area)

LEFT JOIN "TASA" t USING(cod\_tasa)

WHERE r.cod\_vuelo = #{codVuelo} AND st\_intersects(r.tramo, a.frontera) AND ((t.cod\_tasa > 2 AND r.altitud < a.altitud) OR t.cod\_tasa < 3)

GROUP BY cod\_area, a.nombre, ta.cod\_tipo\_area, t.valor, t.cod\_tasa

La consulta se queda con aquellos tramos de la ruta de un vuelo concreto (*WHERE r.cod\_vuelo = #{codVuelo}*) que se encuentran contenidos en un área, gracias a la función *st\_intersects*, que recibiendo dos geometrías devuelve true si las dos comparten un espacio en común (*AND st\_intersects(r.tramo, a.frontera)*), y discierne según la altura si debe ser el área de aproximación la que se debe de aplicar o no (*AND ((t.cod\_tasa > 2 AND r.altitud < a.altitud) OR t.cod\_tasa < 3)*). En esta consulta, el resultado es una sola línea por cada área por la que se ha pasado y un recorrido total en dicha área, de ahí la cláusula del *GROUP BY*. El cálculo de ese recorrido lo conseguimos gracias a la combinación de las funciones *st\_intersection*, la cual al recibir dos geometrías devuelve otra con el espacio que tienen en común, y *st\_length*, que calcula la longitud de esa geometría devuelta. Como la unidad que nos interesa es kilómetros, una manera de transformarlo es transformar la *geometry* a *geography* antes de pasarselo a *st\_length*, y eso ya nos lo devuelve en metros. Solo quedaría dividirlo entre 1.000, y como estamos calculando un total, pasarlo a la función de agregación *sum*, ya obteniendo el recorrido total en kilómetros en un área tras agrupar (*sum(st\_length(st\_intersection(r.tramo, a.frontera)::geography)) / 1000*).