**Pasos para el mapeo de cada tabla en ontimize boot.**

Para este tutorial utlizaré como ejemplo la tabla cliente, con las columnas ID\_CLIENTE, CLI\_NOMBRE,\_CLI\_APELLIDOS y CLI\_TELEFONO;

1. **Creación de los dao.**

Los dao se encargan de mapear cada tabla a una clase java, es decir, convierten cada registro de una tabla dada en un objeto java. Se componen de dos archivos, un archivo xml al que se le indica el campo clave mediante el cual se realizarán las operaciones de delete y update, a través de las siguientes etiquetas contenidas en el archivo xml :

**<DeleteKeys>**

**<Column>**ID\_CLIENTE**</Column>**  
 **</DeleteKeys>**

**<UpdateKeys>**  
 **<Column>**ID\_CLIENTE**</Column>**  
 **</UpdateKeys>**

Quedando el archivo xml final de la siguiente manera :

<?xml version="1.0" encoding="UTF-8"?>  
<JdbcEntitySetup  
 xmlns="http://www.ontimize.com/schema/jdbc"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.ontimize.com/schema/jdbc http://www.ontimize.com/schema/jdbc/ontimize-jdbc-dao.xsd"  
 catalog="" schema="${mainschema}" table="CLIENTE"  
 datasource="mainDataSource" sqlhandler="dbSQLStatementHandler"**>**  
 **<DeleteKeys>**  
 **<Column>**ID\_CLIENTE**</Column>**  
 **</DeleteKeys>**  
 **<UpdateKeys>**  
 **<Column>**ID\_CLIENTE**</Column>**  
 **</UpdateKeys>**  
 **<GeneratedKey>**ID**</GeneratedKey>**  
**</JdbcEntitySetup>**

Este archivo habrá que crearo en la siguiente ruta : /hr-model/src/main/resources/dao

Todas las clases java que componen los dao hay que crearlas en el paquete **com.ontimize.hr.model.core.dao de hr-model** están formadas por el siguiente código :

@Repository("ClienteDao")  
@Lazy  
@ConfigurationFile(configurationFile = "dao/ClienteDao.xml", configurationFilePlaceholder = "dao/placeholders.properties")  
public **class** **ClienteDao** **extends** OntimizeJdbcDaoSupport {

La anotación @Repository("ClienteDao") le indica a ontimize el repositorio que utilizará para realizar operaciones contra la base de datos, es decir, en spring boot los repositorios son los encargados de hacer las operaciones create, read, update y delete en la base de datos. Cada dao o tabla de la base de datos tiene su propio repository.

@Lazy no sé exactamente como explicarlo, pero de momento le implementaremos esta anotación a cada dao.

La anotación @ConfigurationFile(configurationFile = "dao/ClienteDao.xml", configurationFilePlaceholder = "dao/placeholders.properties") le indica el archivo xml que hemos creado en el paso anterior y un segundo parámetro apuntando a un archivo de tipo properties que en este momento creo que en él están las configuraciones comunes a todos los dao. Un archivo properties está formado por parejas de clave – valor, spring los utiliza como archivos de configuración.

Todas las clases java que forman los dao heredan o extienden de OntimizeJdbcDaoSupport, esto creo que es debido que al hacer esto se autoconfiguran en otras partes de la aplicación.

Por último, y ya para terminar con esta parte de creación de los dao, el cuerpo de la clase se compone de una serie de variables que se corresponden con cada columna de la base de datos, todos ellos de tipo public, static, final Y String, en mayúscula y comenzando por la palabra ATTR. En nuestro ejemplo de la tabla CLIENTE con las columnas ID\_CLIENTE, CLI\_NOMBRE, CLI\_APELLIDO Y CLI\_TELEFONO quedaría así :

**public** **static** **final** String ATTR\_ID\_CLIENTE = "ID\_CLIENTE";

**public** **static** **final** String ATTR\_CLI\_NOMBRE = "CLI\_NOMBRE";

**public** **static** **final** String ATTR\_CLI\_APELLIDO = "CLI\_APELLIDO";  
**public** **static** **final** String ATTR\_CLI\_TELEFONO = "CLI\_TELEFONO";

La clase completa quedaría de la siguiente manera :

@Repository("ClienteDao")  
@Lazy  
@ConfigurationFile(configurationFile = "dao/ClienteDao.xml", configurationFilePlaceholder = "dao/placeholders.properties")  
public **class** **ClienteDao** **extends** OntimizeJdbcDaoSupport {

**public** **static** **final** String ATTR\_ID\_CLIENTE = "ID\_CLIENTE";

**public** **static** **final** String ATTR\_CLI\_NOMBRE = "CLI\_NOMBRE";

**public** **static** **final** String ATTR\_CLI\_APELLIDO = "CLI\_APELLIDO";  
**public** **static** **final** String ATTR\_CLI\_TELEFONO = "CLI\_TELEFONO";

}

**El servicio**

El servicio se compone de dos partes, una interfaz en la que simplemente se declara el nombre de los métodos public y una lista de parámetros que recibe y una clase que implementa esta interfaz, que es donde se desarrolla la lógica de los métodos declarados en la interfaz. Para hacer uso del servicio en otra clase, por ejemplo el controlador, se declara una variable del tipo de la interfaz y no se instancia con el operador new, como hacemos normalmente para crear un objeto java, sino que para crear el objeto de esta clase se hace mediante la anotación @Autowired. Con esta anotación spring inyecta una implementación de esta interfaz, por ejemplo dada la interfaz con la que seguiremos este ejemplo **IClienteService** y su implementación **ClienteService para hacer uso del servicio en otra clase, se declararía una variable de la siguiente manera :**

@Autowired

Private IClienteService clienteService;

Sin necesidad de instanciar la clase con el operador new. Spring automáticamente buscará una clase que implemente esta interfaz, en este caso la clase ClienteService y creará el objeto clienteService, declarado más arriba con esta clase.

La interfaz.

La interfaces se crean en el módulo hr-api, y en concreto en el paquete **com.ontimize.hr.api.core.service de este módulo. Como mínimo deben estar declarados los métodos para realizar las cuatro operaciones de CRUD básicas, es decir create, read, update y delete, de la siguiente manera :**

**public** **interface** **IClienteService** {  
  
 // CLIENTE  
 **public** EntityResult candidateQuery(Map<String, Object> keyMap, List<String> attrList) **throws** OntimizeJEERuntimeException;  
 **public** EntityResult candidateInsert(Map<String, Object> attrMap) **throws** OntimizeJEERuntimeException;  
 **public** EntityResult candidateUpdate(Map<String, Object> attrMap, Map<String, Object> keyMap) **throws** OntimizeJEERuntimeException;  
 **public** EntityResult candidateDelete(Map<String, Object> keyMap) **throws** OntimizeJEERuntimeException;  
  
}

Todos los métodos de tipo public y devolviendo un objeto de tipo EntityResult, que en este momento creo que internamente optimize mapea cualquier tabla a este tipo de objeto.

La implementación.

Llegados a este punto voy intentar explicar como funcionan las peticiones http, para que se entienda la función del servicio. Cuando estamos haciendo una petición con postman, esta se compone de tres partes :

La url, que le indica a nuestra aplicación que controlador va a procesar la petición y el método del controlador que va a llamar.

Por ejemplo en la petición GET localhost:33333/candidates/candidate?columns=ID,NAME,SURNAME localhost se refiere a que está llamando al mismo ordenador en el que estamos trabajando, es decir, si tenemos nuestra aplicación corriendo en nuestro ordenador y estamos haciendo la petición con postman también con nuestro ordenador, en la primera parte habría que escribir localhost. Si tuviéramos corriendo nuestra aplicación ontimize en un ordenador diferente al nuestro, en lugar de localhost habría que escribir la ip del ordenador en el que está corriendo la aplicación.

La siguiente parte, :33333, es el puerto por el cual el controlador está escuchando peticiones, que se puede configurar a nuestro gusto. Creo que es el puerto del servidor de aplicaciones tomcat sobre el que corren todas las aplicaciones de spring boot. Tanto el servidor como el puerto son comunes a todas las peticiones que hagamos a la aplicación, indiferentemente de la tabla de la que queramos obtener información y del tipo de petición que se haga.

La parte que sigue al puerto es /candidates, con la cual le estamos indicando a qué controlador queremos llamar. Los controladores tienes la anotación @RequestMapping("/candidates") a nivel de clase. Con este requestMapping le estamos diciendo al controlador que escuche peticiones en la url localhost:33333/candidates. Por ejemplo, en nuestra tablas clientes de ejemplo, cuando creemos el controlador tendríamos que cambiar /candidates por /clientes, es decir, tendríamos que poner esta anotación @RequestMapping("/clientes") a nivel de clase, con lo cual cuando hagamos una petición a la URL localhost:33333/clientes haremos consultas a la tabla clientes.

La última parte de la petición sería /candidate?columns=ID,NAME,SURNAME, con /candidate le estamos diciendo que queremos llamar al método que hace la operación read, este método no se muestra porque al crear los controladores estamos heredando de la clase ORestController, que es en esta clase donde están implementados los métodos que procesan las peticiones. Esta clase ORestController tiene cuatro métodos, una por cada operación de create, read, update y delete. La última parte de la URL, ?columns=ID,NAME,SURNAME, es necesaria para que nos devuelva las columnas que deseemos en la respuesta. En este caso devuelve las columnas ID,NAME,SURNAME, pero, por ejemplo en nuestra tabla cliente, si queremos que nos devuelva las columnas ID\_CLIENTE, CLI\_NOMBRE y CLI\_APELLIDO, habría que indicárselo en esta última parte de la petición, quedando nuestro ejemplo de la siguiente manera :

localhost:33333/clientes/cliente?columns=ID\_CLIENTE,CLI\_NOMBRE,CLI\_APELLIDO.

El controlador se limita a escuchar peticiones y recoger los parámetros de las peticiones, en nuestro ejemplo estos parámetros serían ID\_CLIENTE,CLI\_NOMBRE,CLI\_APELLIDO, y estos parámetros se los pasa al servicio que es quien procesa las peticiones. Una el método del servicio procesa la petición, este método devuelve unos resultados, que los recoge el controlador y los envía al usuario en formato JSON, que son los resultados que vemos en la ventana de respuestas de postman. En spring toda la lógica de las peticiones se escribe en el servicio, limitándose el controlador únicamente a escuchar peticiones y devolver los resultados al usuario.