**Programación distribuida y concurrente**

**Protocolo HTTP**

El **Protocolo de transferencia de hipertexto** es el [protocolo](https://es.wikipedia.org/wiki/Protocolo_de_comunicaciones) de comunicación que permite las transferencias de información en internet. define la sintaxis y la semántica que utilizan los elementos de software de la arquitectura web (clientes, servidores, [proxies](https://es.wikipedia.org/wiki/Servidor_proxy)) para comunicarse.

HTTP es un [protocolo sin estado](https://es.wikipedia.org/wiki/Protocolo_sin_estado), es decir, no guarda ninguna información sobre conexiones anteriores. El desarrollo de aplicaciones web necesita frecuentemente mantener estado. Para esto se usan las [cookies](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)), que es información que un servidor puede almacenar en el sistema cliente. Esto le permite a las aplicaciones web instituir la noción de [sesión](https://es.wikipedia.org/wiki/Sesi%C3%B3n_(inform%C3%A1tica)), y también permite rastrear usuarios ya que las cookies pueden guardarse en el cliente por tiempo indeterminado.

Es un protocolo orientado a transacciones y sigue el esquema petición-respuesta entre un cliente y un servidor. El cliente realiza una petición enviando un mensaje, con cierto formato al servidor. El servidor le envía un mensaje de respuesta.

Los mensajes HTTP, son en texto plano lo que lo hace más legible y fácil de depurar. Esto tiene el inconveniente de hacer los mensajes más largos.

Los mensajes tienen la siguiente estructura:

* Línea inicial (termina con retorno de carro y un salto de línea) con
  + Para las peticiones: la acción requerida por el servidor ([método de petición](https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto#Métodos_de_petición)) seguido de la [URL](https://es.wikipedia.org/wiki/URL) del recurso y la versión HTTP que soporta el cliente.
  + Para respuestas: La versión del HTTP usado seguido del [código de respuesta](https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto#Códigos_de_respuesta) (que indica que ha pasado con la petición seguido de la [URL](https://es.wikipedia.org/wiki/URL) del recurso) y de la frase asociada a dicho retorno.
* Las [cabeceras](https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto#Cabeceras) del mensaje que terminan con una línea en blanco. Son [metadatos](https://es.wikipedia.org/wiki/Metadato). Estas cabeceras le dan gran flexibilidad al protocolo.
* Cuerpo del mensaje. Es opcional. Su presencia depende de la línea anterior del mensaje y del tipo de recurso al que hace referencia la URL. Típicamente tiene los datos que se intercambian cliente y servidor.

**Métodos de petición**

HTTP define una serie predefinida de métodos de petición ("verbos") que pueden utilizarse. Cada método indica la acción que desea que se efectúe sobre el recurso identificado.

GET: solicita una representación del recurso especificado. Las solicitudes que usan GET solo deben recuperar datos y no deben tener ningún otro efecto.

El método GET se emplea para leer una representación de un ***recurso*** En caso de respuesta positiva (200 OK), GET devuelve la representación en un formato concreto: HTML, XML, JSON o imágenes, JavaScript, CSS, etc.

POST: Envía los datos para que sean procesados por el recurso identificado. Los datos se incluirán en el cuerpo de la petición. Esto puede resultar en la creación de un nuevo recurso o de las actualizaciones de los recursos existentes o ambas cosas.

DELETE: Borra el recurso especificado en la **URL**. Si se elimina correctamente devuelve 200 junto con un *body response*, o 204 sin *body*. DELETE, al igual que PUT y GET, también es **idempotente**.

PUT: Sube, carga o realiza un *upload* de un recurso especificado (archivo o fichero) y es un camino más eficiente ya que POST utiliza un **mensaje multiparte** y el mensaje es decodificado por el servidor. En contraste, el método PUT permite escribir un archivo en una conexión *socket* establecida con el servidor. La desventaja del método PUT es que los servidores de alojamiento compartido no lo tienen habilitado.

OPTIONS: Devuelve los métodos HTTP que el servidor soporta para un URL específico. Esto puede ser utilizado para comprobar la funcionalidad de un servidor web mediante petición en lugar de un recurso específico.

El método get, así como también el head y options, al ser métodos que realizan solo lecturas en el servidor; no llevan un body en la petición.

En el caso de POST lleva los datos a cargar en el body, lo mismo para PUT

En el caso del DELETE, la petición no lleva body, y en la respuesta puede tenerlo (código 200) o no (código 204).

**Códigos de estado respuesta**

El código de respuesta o retorno es un número que indica que ha pasado con la petición. El resto del contenido de la respuesta dependerá del valor de este código.

Cada código tiene un significado concreto. Sin embargo, el número de los códigos están elegidos de tal forma que según si pertenece a una centena u otra se pueda identificar el tipo de respuesta que ha dado el servidor:

* Códigos con formato 1xx: Respuestas informativas. Indica que la petición ha sido recibida y se está procesando.
* Códigos con formato 2xx: Respuestas correctas. Indica que la petición ha sido procesada correctamente.
* Códigos con formato 3xx: Respuestas de redirección. Indica que el cliente necesita realizar más acciones para finalizar la petición.
* Códigos con formato 4xx: Errores causados por el cliente. Indica que ha habido un error en el procesado de la petición a causa de que el cliente ha hecho algo mal.
* Códigos con formato 5xx: Errores causados por el servidor. Indica que ha habido un error en el procesado de la petición a causa de un fallo en el servidor.

Ejemplos de algunos códigos:

**200 OK**. El request es correcto. Esta es la respuesta estándar para respuestas correctas.

**201 Created**. El request se ha completado y se ha creado un nuevo recurso. Esto será el retorno correcto de un POST

**400 Bad Request**. El servidor no puede o no va a procesar el request por un error de sintaxis del cliente.

**401 Unauthorized (RFC 7235)**. Similar al error 403, pero se usa cuando se requiere una autentificación y ha fallado o todavía no se ha facilitado.

**403 Forbidden**. El request fue válido pero **el servidor se niega a responder**.

**404 Not Found**. El recurso del request no se ha podido encontrar, pero podría estar disponible en el futuro. Se permiten **requests subsecuentes** por parte del cliente.

**500 Internal Server Error**. Error genérico, cuando se ha dado una condición no esperada y no se puede concretar el mensaje.

**Content-type**

Es un campo dentro del header de una petición http y en el se encuentra el Tipo MIME del cuerpo; pertinente para las solicitudes POST y PUT.

EJEMPLO: Content-Type: application/x\_222-form-urlencoded

En el header de una respuesta del servidor contendrá: Tipo MIME del archivo solicitado

EJEMPLO: Content-Type: text/tml; charset=utf-8

**JSON  
JSON** (acrónimo de **JavaScript Object Notation**, «notación de objeto de JavaScript») es un formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de [objetos](https://es.wikipedia.org/wiki/Objeto_(programaci%C3%B3n)) de [JavaScript](https://es.wikipedia.org/wiki/JavaScript), aunque, debido a su amplia adopción como alternativa a [XML](https://es.wikipedia.org/wiki/XML), se considera (año [2019](https://es.wikipedia.org/wiki/2019)) un formato independiente del lenguaje.

Una de las supuestas ventajas de JSON sobre XML como formato de intercambio de datos es que resulta mucho más sencillo escribir un [analizador sintáctico](https://es.wikipedia.org/wiki/Analizador_sint%C3%A1ctico) (parser) para él.  Además, para un mismo dato a enviar, ocupa menos caracteres, por lo tanto, menos espacio.

Los tipos de datos disponibles con JSON son:

* Números: Se permiten números negativos y opcionalmente pueden contener parte fraccional separada por puntos. Ejemplo: 123.456
* Cadenas: Representan secuencias de cero o más caracteres. Se ponen entre doble comilla y se permiten cadenas de escape. Ejemplo: "Hola"
* Booleanos: Representan valores booleanos y pueden tener dos valores: true y false
* null: Representan el valor nulo.
* Array: Representa una lista ordenada de cero o más valores los cuales pueden ser de cualquier tipo. Los valores se separan por comas y el vector se mete entre corchetes. Ejemplo ["juan", "pedro", "jacinto"]
* Objetos: Son colecciones no ordenadas de pares de la forma **<nombre>:<valor>** separados por comas y puestas entre llaves. El nombre tiene que ser una cadena entre comillas dobles. El valor puede ser de cualquier tipo. Ejemplo:

{

**"departamento"**:8,

**"nombredepto"**:"Ventas",

**"director"**: "Juan Rodríguez",

**"empleados"**:[

{

**"nombre"**:"Pedro",

**"apellido"**:"Fernández"

},{

**"nombre"**:"Jacinto",

**"apellido"**:"Benavente"

}

]

}

**Spring**

Nace como un Framework MVC para Java, sin embargo, Hoy es un conjunto mucho más amplio de productos.

Permite desarrollar Servlets MVC, pero los nuevos proyectos van para otro lado: Programación Reactiva, microservicios, REST.

Spring Boot: Permite correr una aplicación web stand-alone, generar un war o publicar en la nube.

Spring Initializr: permite armar una aplicación desde la web con las dependencias necesarias

Utiliza Inversión de control (IoC) e inyección de dependencias para definir componentes de la aplicación. Los componentes de una aplicación Spring se especifican con anotaciones: @Configuration, @Controller, @Service, @Repository

Cada requerimiento recibe un Model que se utiliza para almacenar la información que se va a mostrar en la vista. Funciona dentro del mismo contenedor, es decir sirve para aplicaciones web donde la vista se genera con el mismo contenedor. Utiliza una clase manager para obtener los datos. Esto puede ser un servicio o cualquier otra clase de la lógica de negocio. Sirve solo para generar contenido web

Spring también incluye contenedores para generar contenido REST. Los Frameworks JS se pueden alimentar de servicios REST y también las apps móviles, Por ende, puedo tener el mismo controlador para aplicaciones web y móviles

**annotations**

Agrega etiquetas que incorporan metadatos para indicar ciertas características o funcionalidad que cumplirán nuestras clases métodos o cualquier elemento que tenga la etiqueta, estas anotaciones reemplazan a los viejos archivos config XML

El framework, mediante la recursividad, busca etiquetas y cuando las encuentra agrega el bean de forma automática al contenedor spring

**@override**

sobreescritura de método

**@Component** (“nombre referencia”)

registro un bean para su posterior uso. Si no le pongo nombre de referencia toma el nombre de la clase, pero con la primera letra en minúscula. es el estereotipo principal e indica que una clase con esta anotación es un Component o Bean de Spring.

**@Repository**, **@Service** y **@Controller** son especificaciones de la anotación **@Component** para casos concretos, por ejemplo, para la persistencia de datos, servicios o para la capa de presentación respectivamente.

Por lo tanto, puedes anotar tus beans con la anotación **@Component**, pero al anotarlas con **@Repository**, **@Service** o **@Controller** obtendrás beneficios adicionales ya que algunos de los Spring modules procesan de manera distinta estas anotaciones.

**@Controller**

Anotación que registra el controlador para Spring MVC. El uso de anotaciones no implica que únicamente tengamos un único controlador, sino que nos permite agrupar un conjunto de urls que estén asociadas a nivel de negocio en un controlador especifico. La aplicación soportará **n controladores.**

**@RequestMapping**

se encarga de relacionar un método con una petición http. Para saber el método HTTP usado es añadir a la anotación @RequestMapping otro atributo llamado method con el método HTTP que debe tener la petición para que llamen a ese método.

@RequestMapping (value="/SeguroMedico”, method=RequestMethod.GET)

en el que indicamos que cuando se llame a esa URL con ese método HTTP llamemos a dicho el método Java.

Para evitar conflictos de rutas repetidas en diferentes controladores, es recomendable utilizar, un @RequestMapping como ruta relativa a toda la clase del xcontroller que pertenezca, así, aunque sus rutas de los métodos sean iguales la relativa será distinta y no habrá errores. Para esto se pone @RequestMapping abajo del @Controller y el nombre de la ruta relativa

Ahora existen anotaciones para cada método http: **@GetMapping**(“ruta”) lo mismo para el resto de verbos

@**Service**

debe de ser empleada para la capa de servicios (*facade*) y lógica de negocio. se encarga de gestionar las operaciones de negocio más importantes a nivel de la aplicación y **aglutina llamadas a varios repositorios de forma simultánea**. Su tarea fundamental es la de **agregador.**

Algunas URL incluyen una parte variable como idSeguroMedico. Para poder acceder debemos hacer 2 cosas:

* Indicar en la URL que esa parte es variable, poniendo entre { } la parte variable
* Añadiendo un parámetro al método de Java y usando una anotación @PathVariable con el nombre de la parte variable para que lo relacione con la URL.

@RequestMapping(value="/SeguroMedico/{idSeguroMedico}”, method=RequestMethod.GET)

    public void read(@PathVariable("idSeguroMedico") int idSeguroMedico)

Se ha puesto entre { } la parte de idSeguroMedico para indicar que esa parte es variable.

Ahora el método Java tiene un parámetro de tipo int que se llama idSeguroMedico y lo relacionamos con la URL con la anotación @PathVariable(“idSeguroMedico”). Siendo el valor que hay en @PathVariable exactamente el mismo que hay entre { } de la URL.

**@Entity**

declara una clase como una entidad Hibernate, donde actuara como una entidad de un modelo E-R que se usa en las bd relacionales, si no se declara nada mas

**@Table**

(name=” nombre tabla”) mapea un objeto a la tabla indica

**@Column**

(name=” nombre columna”) mapea un atributo con un campo indicado

**@Transient**

indica que este atributo no se persista en BD.

**@ID**

representa el PK de una base de datos, en una clase java en un determinado atributo

**@GeneratedValue**

(strategy = GenerationType.***AUTO***) int autoincremental

**@RequestBody**

la anotación que indica un parámetro de método debe vincularse al cuerpo de la solicitud HTTP.

**@RequestParam**

parámetros anotados **@RequestParam** se vinculan a parámetros específicos de solicitud de servlet. Los valores de parámetro se convierten al tipo de argumento de método declarado. Esta anotación indica que un parámetro de método debe vincularse a un parámetro de solicitud web.

**public** String procesarForm(@RequestParam("usuario") String nombre, Model model) {

queda en la variable nombre lo que obtengo del RequestParam, donde usuario, es el campo id-name del html

**@requestHeader**

parámetros de método se vinculan al header de la solicitud http

**@Autowired**

le indica a Spring dónde debe ocurrir una inyección de dependencias de la clase indicada como dependencia. Puede ser mediante constructor, setter, getter, o incluso, en un atributo mismo.

**@Qualifier**

sirve para que el autowired de una interfaz con múltiples clases que la implementan funcione y sepa de que clase en particular debe inyectar. Se pone después del @Autowired

@Autowired

@Qualifier("informeFinancieroTrim3")

**private** InformeFinanciero informeFinanciero;

**@Scope**(“prototype”)

es para que spring al crean bean lo haga con el patrón prototype (varias instancias) y no con singleton, el cual es el default (una instancia). no se usa

**@PostContruct**

ejecuta algo cuando se crea el bean, antes de que esté disponible

**@PreDestroy**

ejecuta algo antes de que muera el bean

**@Configuration**

nos permite reemplazar el archivo config.xml

**@ComponentScan**(“paquete”)

escanea en busca de annotations

**@Bean**

declarar beans para su posterior uso

**@PropertySource**

nos permite importar valores de un file externo

**@Value**

mediante pares clave valor

**@ModelAttribute** (“nombre atributo”) Tipo nombre

consigo guardar en la variable nombre el atributo que estaba en el model referenciado

“para los campos en tymaleyf hay que poner la palabra clave th: field=” \*{nombre atributo}” para que los inputs se asocien al objeto. IMPORTANTE no olvidarse poner el th: object=” ${nombre objeto}” en el form” para los radiobuttons select option y checkboxes no hace falta ya solito se aviva

**@NotNull**

permite que no se inserten campos null en un determinado atributo

**@NotBlank**

verifica que el campo string no sea null ni vacío

**@Size (min=n, message=”msg error”)**

permite determinar un tamaño especifico a un atributo

**@Min** (value=n, message=”msg error”)

permite limitar un input numero a un valor >= a n

**@Max** (value=n, message=”msg error”)

permite limitar un input número a un valor <= a n

**@InitBinder**

comprueba información antes de que llegue al Controller mediante la ejecución de un determinado método antes de que lo atienda el controller

StringTrimmerEditor es un objeto que me permite tratas campos inputs e eliminarle los espacios en blancos transformándolo a null

Cuando se envía un formulario los resultados de las verificaciones se almacenan en el objeto BindingResult

En el objeto bindingresult hay métodos como hasErrors () que me devuelve un boolean diciendo si hubo errores en el formulario o no

IMPORTANTE además de poner como argumento, en el método del Controller, el bindingresult también poner como primer argumento @Valid para indicarle que usa validaciones

**@Pattern (regexp=” [rango de valores]{cantidad}”, message=”msg error”)**

@Pattern(regexp="[0-9]{5}”, message="solo 5 valores numéricos")

EXPRESIONES REGULARES me permite declarar algún tipo de datos que cumpla una determinada estructura para luego hacer que se cumpla eso a un determinado input. Por ejemplo: el CBU que tiene un determinado número de dígitos que siguen un estándar con una estructura de que sebe respetar

MENSAJES DE ERROR PERSONALIZADOS  
se debe crear una carpeta resources, la cual contenga, un archivo messages.properties

Donde podremos poner el siguiente código:

typeMismatch.nombreobjeto.nombreatributo=”nuevo msg error”

typeMismatch.alumno.edad="solo valores numericos"

**Spring Boot**

Spring Boot busca que el desarrollador solo si centre en el desarrollo de la solución, olvidándose por completo de la compleja configuración que actualmente tiene Spring Core para poder funcionar.

Spring Boot centra su éxito en las siguientes características que lo hacen extremadamente fácil de utilizar:

* **Configuración:** Spring Boot cuenta con un complejo módulo que autoconfigura todos los aspectos de nuestra aplicación para poder simplemente ejecutar la aplicación, sin tener que definir absolutamente nada.
* **Resolución de dependencias:** Con Spring Boot solo hay que determinar que tipo de proyecto estaremos utilizando y el se encarga de resolver todas las librerías/dependencias para que la aplicación funcione.
* **Despliegue:** Spring Boot se puede ejecutar como una aplicación Stand-alone, pero también es posible ejecutar aplicaciones web, ya que es posible desplegar las aplicaciones mediante un servidor web integrado, como es el caso de Tomcat, Jetty o Undertow.
* **Métricas:** Por defecto, Spring Boot cuenta con servicios que permite consultar el estado de salud de la aplicación, permitiendo saber si la aplicación está prendida o apagada, memoria utilizada y disponible, número y detalle de los Bean’s creado por la aplicación, controles para el prendido y apagado, etc.
* **Extensible:** Spring Boot permite la creación de complementos, los cuales ayudan a que la comunidad de Software Libre cree nuevos módulos que faciliten aún más el desarrollo.

**JPA**es un ORM provisto por java. JPA (Java Persistence API - API de Persistencia en Java) es una abstracción por sobre JDBC que permite realizar en forma sencilla una correlación entre objetos y registros de base de datos.

JPA es una especificación de Java EE basada en interfaces que son implementadas por diferentes proveedores de persistencia como Hibernate o Eclipse Link entre otros. Es el proveedor quién realiza el trabajo, pero siempre funcionando bajo la API de JPA.

requiere meta-información en las clases Java para poder realizar el mapeo objeto-relacional. Se pueden añadir metadatos de dos maneras: Archivos XML o Anotaciones (es más simple y utilizado)

Es meta información que se escribe en el código de una manera determinada que permite al intérprete para que en run-time realice instrucciones: @algo. Ejemplos: @entity , @table, @id, @column , @generatedvalue, @OneToOne

En el caso particular de JPA sirven para relacionar cuestiones de objetos con cuestiones de base de datos:

Clases con tablas, Variables con campos, Pertenencia con relaciones

JPA también permite seguir el sentido inverso, creando objetos a partir de las tablas de una base de datos también de forma transparente. Estos objetos se denominan entities

**JAXRS JERSEY**

**@GET**

Es un *request method designator*. El método Java anotado con @GET procesa peticiones HTTP GET.

**@POST**

Es un *request method designator*. El método Java anotado con @POST procesa peticiones HTTP POST.

**@PUT**

Es un *request method designator*. El método Java anotado con @PUT procesa peticiones HTTP PUT. El comportamiento del recurso es determinado por el método HTTP al que responde el recurso

**@DELETE**

Es un *request method designator*. El método Java anotado con @DELETE procesa peticiones HTTP DELETE. El comportamiento del recurso es determinado por el método HTTP al que responde el recurso

**@Path**

identifica la plantilla de path para la URI del recurso al que se accede y se puede especificar a nivel de clase o a nivel de método de dicho recurso. es relativa a la URI base del servidor en el que se despliega el recurso, a la raíz del contexto de la aplicación, y al patrón URL al que responde el *runtime* de JAX-RS. puede incluir variables entre llaves, que serán sustituidas en tiempo de ejecución dependiendo del valor que se indique en la llamada al recurso. por ejemplo:

@Path("/users/{username}")

**@Produces**

Se utiliza para especificar el tipo MIME de las representaciones que un recurso puede proporcionar y enviar al cliente, por ejemplo "text/plain”

Esta anotación funciona conjuntamente con @GET, @POST y @PUT. Indica al *framework* qué tipo de representación se envía de vuelta al cliente. De forma más específica, el cliente envía una petición HTTP junto con una cabecera *Accept* HTTP que se mapea directamente con el *Content-Type* que el método produce. Por lo tanto, si el valor de la cabecera *Accept* HTTP es application/xml, el método que gestiona la petición devuelve un *string* de tipo MIME application/xml. Esta anotación también puede utilizarse en más de un método en la misma clase de recurso.

@Produces("application/json")

**@Consumes**

Se utiliza para especificar el tipo MIME de las representaciones que un recurso puede consumir cuando ésta es enviada desde el cliente.

Esta anotación funciona conjuntamente con @POST y @PUT. Le indica al *framework* (librerías Jersey) a qué método se debe delegar la petición de entrada. Específicamente, el cliente fija la cabecera HTTP *Content-Type* y el *framework* delega la petición al correspondiente método capaz de manejar dicho contenido.

Si @Consumes se aplica a la clase, por defecto los métodos de respuesta aceptan los tipos especificados de tipo MIME. Si se aplica a nivel de método, se ignora cualquier anotación @Consumes a nivel de clase para dicho método.

@Consumes("application/json")

**@PathParam**

Es un tipo de parámetro que puede extraerse para utilizarse en la clase del recurso. Los parámetros del path de la URI se extraen de la URI de la petición, y los nombres de los parámetros se corresponden con los nombres de las variables de las plantillas del path de la URI especificados en la anotación @Path a nivel de clase

@Path("/users/{username}")

public String getUser(@PathParam("username") String userName)

crea una variable userName dentro del método que tendrá el valor obtenido de “username” que viene en la URI de la petición

**@QueryParam**

Es un tipo de parámetro que puede extraerse para utilizarse en la clase del recurso. Los parámetros de la *query* de la URI se extraen de los parámetros de *query* de la URI de la petición

@Path("/figures")

public String handleGETRequest(@QueryParam("element")

En este caso, el parámetro color tomará como valor, el valor especificado como *query* en la petición *http* correspondiente, como por ejemplo:

http://ruta/servicio/web/figures?color="blue"

**@HeaderParam**

Enlaza una cabecera http al parámetro de un método

public Response get(@HeaderParam("User-Agent") String agente)

**Core Response**

Define el contrato entre una instancia devuelta y el tiempo de ejecución cuando una aplicación necesita proporcionar metadatos al tiempo de ejecución.

Una clase de aplicación no debe extender esta clase directamente. Response (La clase) está reservada para una extensión por los proveedores de implementación JAX-RS. Una aplicación debe usar uno de los métodos estáticos para crear una Response instancia usando un ResponseBuilder.

Método status: es para devolver los códigos de estados definidos por HTTP

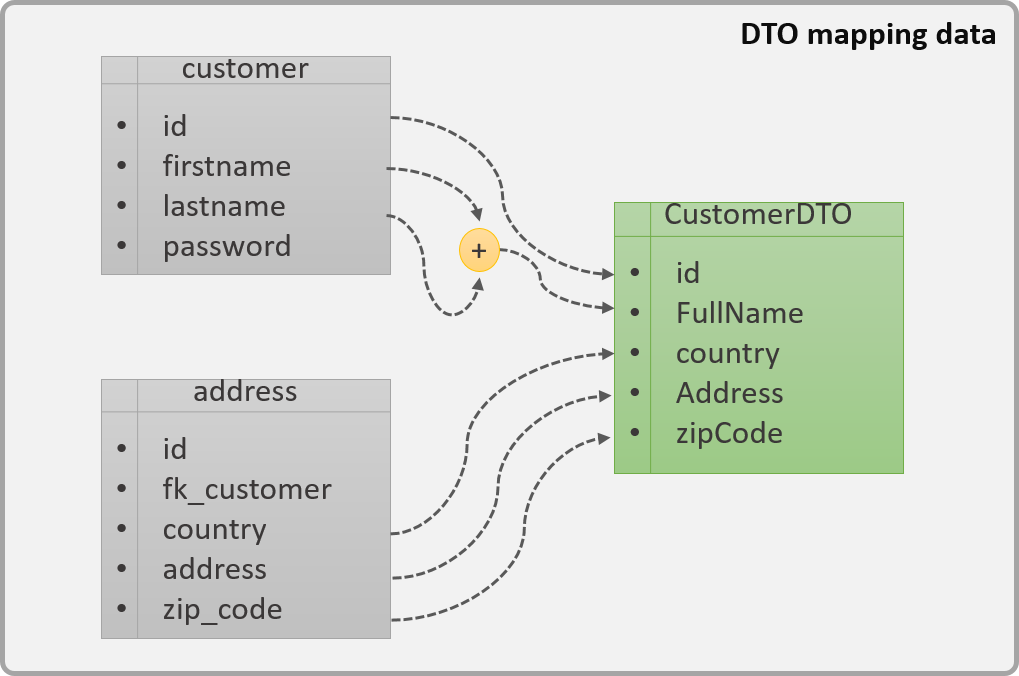
* status(int status): Cree un nuevo ResponseBuilder con el estado proporcionado.
* status(Response.Status status) : Cree un nuevo ResponseBuilder con el estado proporcionado.
* status(Response.StatusType status) : Cree un nuevo ResponseBuilder con el estado proporcionado.

Método ok:

* ok() : Cree un nuevo ResponseBuilder con un estado OK.
* ok(Object Entity) : Cree un nuevo ResponseBuilder que contenga una representación.
* ok(Object entity, MediaType type) : Cree un nuevo ResponseBuilder que contenga una representación.
* ok(Object entity, String type): Cree un nuevo ResponseBuilder que contenga una representación.
* ok(Object entity, Variant variant): Cree un nuevo ResponseBuilder que contenga una representación.

**DTO**

El patrón DTO tiene como finalidad de crear un objeto plano (POJO) con una serie de atributos que puedan ser enviados o recuperados del servidor en **una sola invocación**, de tal forma que un DTO puede contener información de **múltiples fuentes** o tablas y concentrarlas en una única clase simple.



En la imagen anterior podemos apreciar gráficamente como es que un DTO se conforma de una serie de atributos que puede o no, estar conformados por más de una fuente de datos. Para esto, el servidor obtiene la información de las tablas customer y address (izquierda) y realiza un mapping con el DTO (derecha). Adicional, la información puede ser pasada de un lado intacta como es el caso del id , fullName , country , address  y zipCode  o ser una derivada de más de un campo, como es el caso del fullName , el cual es la unión del firstname  y lastname .

Otra de las ventajas no tan claras en la imagen, es que nos permite omitir información que el usuario no requiere, como es el caso de password. No es solo que no lo requiere, sino que además podría ser una gran falla de seguridad está enviando los passwords, es por ello que en el DTO lo omitimos.

Un DTO tiene que cumplir algunas reglas:

* **Solo lectura**: Dado que el objetivo de un DTO es utilizarlo como un objeto de transferencia entre el cliente y el servidor, es importante evitar tener operaciones de negocio o métodos que realicen cálculos sobre los datos, es por ello que solo deberemos de tener los métodos GET y SET de los respectivos atributos del DTO.
* **Serializable**: Es claro que, si los objetos tendrán que viajar por la red, deberán de poder ser serializables, pero no hablamos solamente de la clase en sí, sino que también todos los atributos que contenga el DTO deberán ser fácilmente serializables. Un error clásico en Java es, por ejemplo, crear atributos de tipo Date o Calendar para transmitir la fecha u hora, ya que estos no tienen una forma estándar para serializarse por ejemplo en Webservices o REST.

public class CustomerDTO implements Serializable{

private Long id;

private String FullName;

private String country;

private String Address;

private String zipCode;

/\*\* GET and SET \*/

}

@Path("customers")

public class CustomerService {

@GET

@PathParam("{customerId}")

private Response findCustomer(@PathParam("customerId") Long customerId) {

Customer customer = customerDAO.findCustomerById(customerId); //Entity

Address address = customerDAO.findAddressByCustomer(customerId); //Entity

//Create dto

CustomerDTO dto = new CustomerDTO();

dto.setAddress(address.getAddress());

dto.setCountry(address.getCountry());

dto.setZipCode(address.getZipCode());

dto.setFullName(customer.getFirstname() + " " + customer.getLastname());

dto.setId(customer.getId());

//Return DTO

return Response.ok(dto, MediaType.APPLICATION\_JSON).build();

}

}

El ejemplo que acabamos de ver, corresponde a una implementación de un servicio REST utilizando el API JAX-RS de Java, el cual indica que existe un servicio GET en la url */customers/{customerID}*, donde *{customerId}* corresponde al ID del cliente a buscar. En el servicio, realizamos la consulta del cliente y su dirección en dos pasos, para finalmente, mapear los datos de estas dos entidades en un simple DTO que será retornado.

ModelMapper es una librería Java para copiar o *mapear* propiedades de un tipo de objeto a otro tipo de objeto (Por ejemplo, de un Entity a un DTO), permitiendo copiar también los datos de las referencias a los objetos que contengan. Soporta diferentes convenciones, copiados explícitos, conversiones y proveedores para construir los objetos destino e integraciones con diferentes librerías.

Para usarla, la importo en mi clase y además hago su correspondiente inyección de dependencia

**@Autowired**

**private** ModelMapper modelMapper**;**

Customer customer **=** **new** Customer**(**"Francisco"**,** "Ibáñez"**);**

Address billigAddress **=** **new** Address**(**"c\\ Rue del Percebe, 13"**,** "Madrid"**);**

Order order **=** **new** Order**(**customer**,** billigAddress**);**

para crear el objeto dto mapeado utilizo el método:

map(clase origen, claseorigenDTO.class)

OrderDTO orderDTO **=** modelMapper**.**map**(**order**,** OrderDTO**.**class**);**

Siendo order mi Entity a transformar en DTO

System**.**out**.**printf**(**"Customer First Name: %s%n"**,** orderDTO**.**getCustomerFirstName**());**

¿Cuál es el propósito de la clase generada por el asistente que tiene la notacion @SpringBootApplication?

Lo más destacado aquí es la anotación **@SpringBootApplication** . Esto internamente es una combinación de las siguientes 3 anotaciones. Y en forma de resumen más simplificado realiza:

* **@Configuración**: Necesaria para configuraciones de resorte manual. La adición de esta anotación garantiza que la configuración se puede realizar en una clase Java en lugar de usar un archivo xml separado.
* **@EnableAutoConfiguration**: Spring necesita mucha configuración por hacer. Esta anotación asegura que gran parte de la configuración se realiza automáticamente.
* **@ComponentScan:** Esto le dice a Spring, donde todos deben buscar componentes.

¿Para qué se utiliza el archivo src/main/resources/application.properties?

La configuración es una de las partes más importantes de cualquier aplicación, pues permite que se pueda configurar según el entorno de ejecución o el ambiente en el que se desplegará, por suerte Spring boot cuenta con el archivo application.properties, en el cual podemos guardar todos aquellos valores que pueden cambiar con el tiempo, lo que evita la mala práctica de crear código hardcode o código duro.

En el podemos por ejemplo, establecer la configuración de conexión con la base de datos.