# Arquitectura de software 2

Alumno: Santiago Vietto

Docente: Eduardo Carlos Gaite

DNI: 42654882

Institución: UCC

Año: 2021

# Servicios Web

## General

_ En esta unidad vamos a repasar algunos conceptos en términos generales. Vamos a construir Servicios REST utilizando el protocolo HTTP, especificando nuestras API's y luego vamos a consumir los mismos codificando distintos clientes. Por último, vamos a revisar conceptos generales de web servers, escalabilidad, RPM, escalamiento, Tolerancia a Fallos, Control de errores.

## Protocolo HTTP

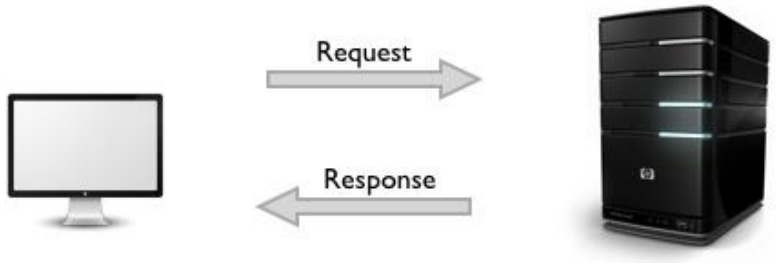
_ El protocolo HTTP es básicamente una arquitectura de comunicación donde tenemos un cliente y un servidor, y en donde hay una estructura de comunicación que es válida y que obviamente tiene que ser entendida tanto del lado del cliente como del servidor, y es este último el que lo genera. Se dice que el protocolo es stateless, es decir, básicamente no tiene estados, sino que tiene una comunicación de ida, un request y un response, y de alguna manera esta es la parte que es importante.

Bases de comunicación: tenemos un cliente que en este caso sería cualquier navegador web, alguna línea de comando, algún restclient o cualquiera que pueda generar alguna petición a un servidor, y bueno obviamente tenemos un servidor. Como clientes vamos a generar algo que se llama request y vamos a estar recibiendo de parte del servidor algo que se llama response, de esta forma hablamos con propiedad sobre lo que estamos mandando y lo que podemos recibir. El puerto por defecto para la comunicación TCP/IP, que realiza el protocolo, es el puerto 80 pero otros puertos pueden ser usados, ahora, esto es cierto, lo que no significa que el puerto 80, y sobre todo en Linux, sea un puerto que habitualmente este abierto o se pueda abrir para poder exponerlo a una IP publica por ejemplo, normalmente todo lo que son servidores web, como un tomcat por ejemplo, no se levantan en el puerto 80, sino que se levantan en el 8080, 8090, 9080, entre otras nomenclaturas, pero luego, a nivel ruteo lo que hacemos es que terminamos indicando que toda aquella petición que venga al puerto 80, la termina resolviendo por ejemplo el puerto 8080 o el que hayamos definido.

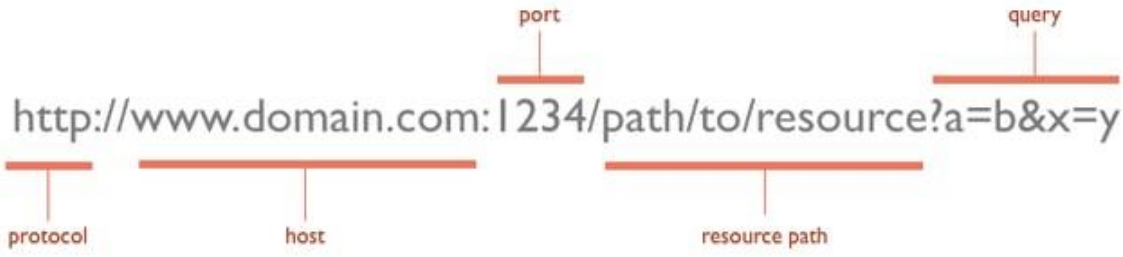
_ Por ejemplo, cuando nosotros vemos en una URL cualquiera, y como no vemos el puerto puesto en la misma, significa que estamos entrando por default al puerto 80, pero eso no quiere decir que por default está abierto ese puerto, sino que seguramente hay un redirect interno en esa URL, en la capa que resuelve esa petición, la cual le indica que, si viene una petición en la 80, lo va a terminar atendiendo un servidor distinto. Es importante saber esto porque quizás después lo que queremos hacer es cambiar la configuración de un servidor para que levante por ejemplo en el puerto 80 y eso a nivel solución no está bien, ya que generalmente se levantan en otros puertos como dijimos el 8080, 8090, etc. Entonces, no es conveniente usar el puerto 80 porque estos por cuestiones de seguridad, en todos los servidores Linux,

vienen siempre cerrados o reservados para algunas cuestiones, por lo que no nos permiten levantar un servidor directamente en este puerto, porque cuando hacemos un redirect a nivel router es un poco más seguro, por eso es que nunca levantamos un servidor que está en producción directamente en el puerto 80, generalmente por cuestiones de seguridad.

_ Tenemos que quedarnos con el concepto de que es un cliente que genera un request, y que vamos a recibir un response, y que básicamente este es un protocolo que es sin estados.




Armado de URLs: vamos a ver cómo se comportan las URL para poder implementar el protocolo. Lo que tenemos primero es el protocolo que es HTTP o HTTPS, en segundo lugar tenemos lo que es el host, que en este caso tenemos un host que es una URL en el que existe un DNS por detrás que me está diciendo cual es la IP de ese host, ya que sino lo que tendríamos acá sería una dirección IP, luego en tercer lugar tendríamos el puerto que normalmente no se ve cuando navegamos por internet ya que el puerto por default es el 80 y cuando ponemos el 80 automáticamente no lo vemos sino que directamente vemos la barra "/", después en cuarto lugar tenemos algo que es el path para llegar al recurso, cuando generamos un servicio, es el path que tenemos para llegar a un recurso, por ejemplo si tenemos una veterinaria y queremos hacer un listado de los clientes, en ese caso podría ser /clientes/1 y así debería devolver el cliente número 1, por ultimo empiezan a venir un par de claves valor, donde en este caso como vemos la "a" es la clave y la "b" es el valor, lo mismo con la "x" y la "y". Básicamente esta es la forma en la que se van a armar las URL para poder resolver como vamos a brindar la información, y esto va a generar lo que se llama un request para poder devolver algo, es decir, esta es la forma en la que vamos a construir una URL para poder hacer una petición directamente al servidor.

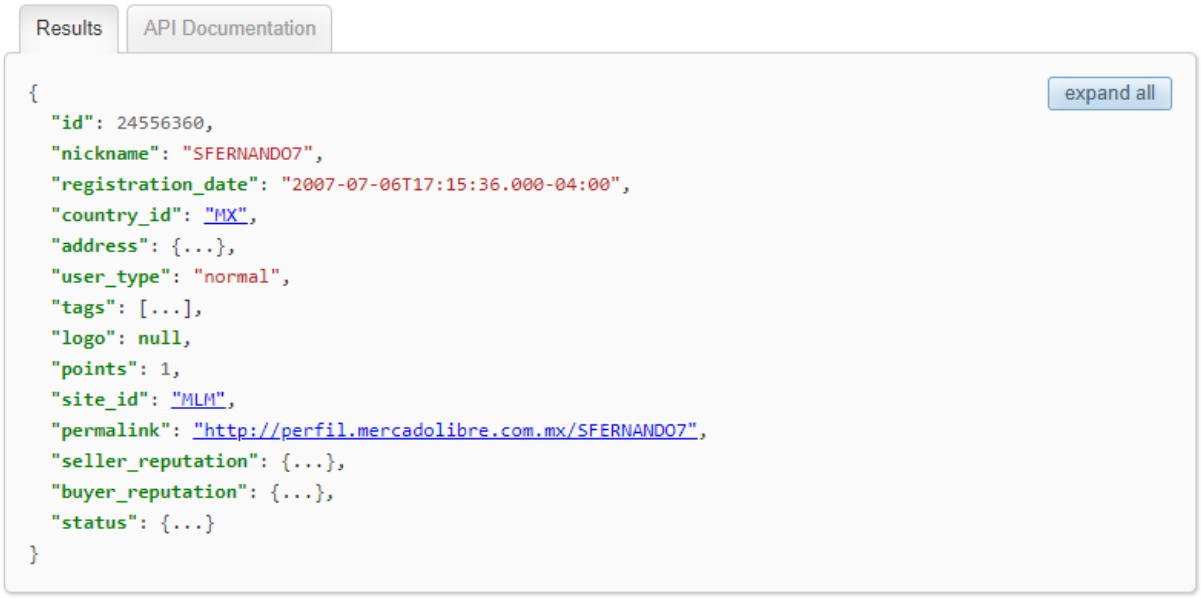



_ Por ejemplo entramos a un api publica de mercado libre para buscar un usuario, y como vemos en primer lugar tenemos el protocolo HTTP que en realidad no lo vemos,

luego tenemos el servidor o la IP, cuya IP es publica, después tenemos el path hasta el recurso que en este caso es /users y siguiente tenemos una forma de pasar un parámetro que en este caso es dentro de la URL, es decir, va como parte de la URL, sino tendría que decir /users?ID=24556360.

https://api.mercadolibre.com/users/24556360

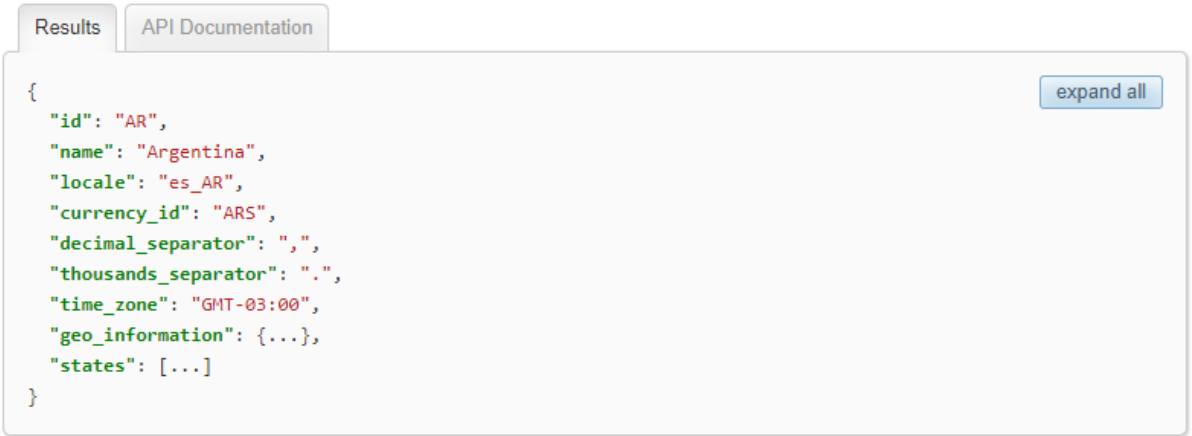
_ A continuación vemos la respuesta de ese api, donde tenemos un GET con datos tanto públicos como no públicos ya que no estamos registrados:






_ Lo mismo podemos hacer con el api para ver países, donde cambiamos el path por countries y le damos como parámetro AR de Argentina, BR de Brasil, etc:

https://api.mercadolibre.com/countries/AR



_ Podemos realizar los mismo con el path en valor de sites donde nos permite ver otra información como los métodos de pago, el valor de cambio de la moneda, etc.

https://api.mercadolibre.com/sites/MCO

_ Entonces esta es la forma en la que estructuramos las URLs y la forma en la que se van estructurando los distintos recursos que no necesariamente viven en un solo servidor. Cada uno de los ya sea sites, ítems, users, countries, son distintas construcciones que están hechas todas en servicios distintos y viven seguramente en servidores distintos, donde hay un orquestador por encima que se encarga de decir quien atiende a cada uno, y si por ejemplo la api de sites se cae por algún motivo, las demás deberían seguir funcionando tranquilamente pudiendo devolver valores, o sea son todas APIs que viven de forma separada, en donde si llamamos a users esta termina llamando y verificando seguramente a todas las APIs que está utilizando generando una relación, es decir, cuando traiga un usuario me va a decir que país, site, etc pertenece y va a hacer esa relación.

_ Para concluir decimos que la forma de armar una URL es con HTTP://, seguido tenemos el dominio o una dirección IP, luego ":" y el puerto, y por último cual es el path del recurso al que queremos llegar. Esta es la forma en la que vamos a construir las distintas llamadas independientemente de que sea un GET, POST, PUT o DELETE, y después dependiendo de cuál sea el verbo que vamos a estar utilizando, es como vamos a estar pasando la información que hace falta para obtener las distintas cuestiones.

Verbos HTTP: los verbos más comunes que usamos son cuatros:

- GET: este se utiliza cada vez que queremos traer información de un recurso, es decir, cada vez que vamos a buscar datos de ese recurso, por ejemplo, si estamos hablando de users, cada vez que voy y busco los datos de un usuario o una lista de usuarios, estamos usando un GET, en donde nos va a traer toda la información que nos hace falta y que le estamos pidiendo al servidor.
- POST: este es cuando vamos a crear un recurso nuevo, por ejemplo, vamos a hacer un POST a la API de users cuando vamos a crear un usuario nuevo.
- PUT: es cuando vamos a hacer un update de un recurso que esta existente, donde por ejemplo si queremos modificar la dirección de un usuario, vamos a realizar un PUT para poder hacer ese cambio.
- DELETE: este es básicamente cuando queremos eliminar o borrar un recurso que es existente.

Códigos de retorno (successful): tenemos que hablar de los códigos de retorno que tiene cada una de las llamadas que habitualmente hacemos. Del número 100 nos olvidamos porque es del protocolo viejo, y nos concentramos en los 200.

- 200: Cuando se devuelve un 200 OK básicamente significa que la consulta que estamos haciendo está bien, se puede procesar y se puede devolver la información. Por ejemplo, en la API de users, corremos el enlace y le damos a

inspeccionar, vamos a la parte de network, en donde si refrescamos la página podemos ver cuál es la URL a la que estamos llamando (Request URL), tenemos también el método por el cual estamos llamando (Request Method) y en la parte de Status Code podemos ver el 200, que quiere decir que esta todo bien. Entonces siempre que hagamos un GET y la respuesta tenga que ser correcta, vamos a ver un 200. Esta respuesta nos sirve porque cuando estamos implementando una API o un servicio, una cosa es que sea una API privada, y otra cosa es que sea publica donde puede estar usándose o puede estar soportada o se va a consumir por un montón de clientes, donde si tenemos que construir un cliente y vamos a consumir una API que no conocemos, y vamos a hacer un GET, va a ser muy probable que nosotros que estamos consumiendo un GET de la API, lo primero que vamos a estar esperando es ver ese código de respuesta que viene para saber que vamos a mostrar o que no vamos a mostrar, entonces si la API no devuelve un 200 y devuelve cualquier número, podemos entender de que hay algo que estamos haciendo mal y podemos mostrar algo erróneo si no interpretamos bien la respuesta del API que nos está enviando. Entonces siempre que sea un OK vamos a devolver un 200.

- 201: cuando vamos a hacer un POST y vamos a generar un recurso nuevo, lo que vamos a devolver siempre es un 201. Lo más común que hacemos siempre es devolver un 200, ahora si estamos haciendo un POST para crear un usuario nuevo por ejemplo, y el que construyo el servicio no nos devuelve el 201 y nos devuelve otro número o código, es muy probable que entendamos que no se está creando bien el servicio, entonces por más que venga un mensaje en el response diciendo que se creó el usuario correctamente, si nosotros que estamos codeando, lo primero que hacemos es entender cuál es el código de response del protocolo y en realidad el que programo se equivocó y en vez de devolvernos un 201 nos devolvió un 200, está mal. Entonces cuando hacemos un POST nos debería devolver un 201, ya que esta respuesta nos dice que lo que creamos se creó bien.

<u>Errores de redirección (redirection):</u> tenemos una gama de códigos de errores que son los 300. Por ejemplo, cuando entramos a una URL y por algún motivo se movió de lugar, entonces se empiezan a devolver los siguientes errores:

- 301: si nos devuelve un 301 significa que directamente se movió permanentemente toda la URL para que la atienda otra URL distinta. Si nosotros mandamos un 301 el navegador no va a intentar ir a la dirección que le estamos diciendo, sino que se va a dar cuenta que hay alguien que nos dijo que la respuesta era un 301, por lo cual directamente tenemos una nueva URL a la que debemos ir, entonces el navegador va a ir directamente al servidor nuevo.
- 303: esto es similar al 301, pero es temporal, por ejemplo, decimos que por ahora estamos redireccionando a un cliente a una URL distinta, y luego lo redireccionaremos a la original. En este caso el navegador si va a seguir

intentando durante un tiempo ir a la URL, porque le estamos diciendo que en algún momento la URL original va a volver a atender.

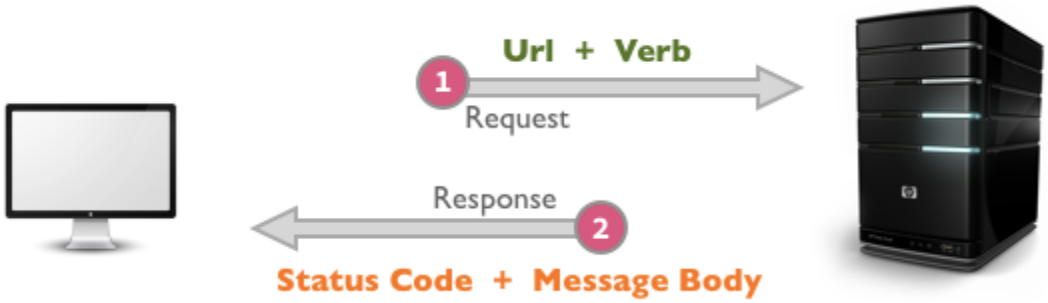
Errores del cliente (client error): tenemos la gama de códigos de errores del lado del cliente que son los 400:

- 400: (bad request) por ejemplo si tenemos que dar de alta un usuario, suponemos que el usuario tiene nombre, apellido y DNI, todos obligatorios, y si en el JSON, es decir, en el body del POST, no le estamos mandando al gun dato en particular o ningún dato directamente, el servidor interpreta que no le mandamos ningún dato o que falta de mandar por ejemplo el nombre, y devuelve un 400 pudiendo especificar en un mensaje que es lo que paso o que es lo que falta.
- 401: se utiliza cuando tenemos algún método que utiliza cierta autorización, y cuando no tenemos esa autorización, lo que devuelve es un unauthorized para poder indicar que es un método que no tiene permisos.
- 404: (not found) este es el más común y es cuando estamos buscando por ejemplo un recurso que es invalido porque no existe, porque no está del lado del server o porque tiene algún error, por ejemplo cuando buscamos in ID que no existe.

Errores del servidor (server error): estos son errores del lado del servidor y son códigos de la gama de los 500:

- 501: (Not implemented) esto es cuando hay un método que no está implementado, donde por ejemplo tratamos de hacer un POST cuando en realidad el método no soporta un POST y soporta solamente un GET.
- 503: (Service Unavailable) es cuando el servicio este caído, y lo devuelve el servidor porque está habiendo algún problema ahí justamente

Request y response: entonces como vemos en el gráfico, tenemos un request que viene del lado del cliente hacia el servidor y un response que viene del servidor hacia el cliente, donde básicamente lo que tenemos es la forma de como lo haríamos. Por un lado, en el request tenemos la URL más el verbo, esto es recibido por el servidor, es interpretado y va a saber cuál es el método que va a implementar para poder hacer algo contra la aplicación, y lo que va a devolver este servidor es un response en donde lo primero que tiene el response es el status code que puede estar dentro de la gama de los 200, 300, 400 o 500, junto con un mensaje que viene en el body que es básicamente un mensaje extra en formato JSON de que si esta todo bien o información para que alguien pueda procesar.



_ Por ejemplo si estamos buscando una lista de clientes, voy a hacer un GET a clientes sin ningún parámetro porque quiero que vengan todos los cliente que están la base de datos, el servidor me va a devolver un 200 de OK y en el body me va a devolver un JSON que represente un array con todos los clientes que tenemos. Si en la URL hacemos un POST a por ejemplo dirección, y en el message body del request pasamos los datos que estamos dando de alta como por ejemplo la calle, código postal, etc, a todo esto lo va a agarrar el servidor, va a generar un nuevo recurso haciendo un INSERT en la base de datos, y en el response va a devolver un 201, porque estamos creando un recurso nuevo mas un message body que queda a la decisión del desarrollador, muchas veces es un mensaje que dice "el recurso se generó correctamente" o se retorna el mismo recurso que creamos pero se le agrega el ID de ese recurso que antes no existía en la base de datos y ahora sí, entonces con el status code y el ID podemos saber cual es el recurso que se creó.

_ Entonces vimos que en este protocolo tenemos un cliente y un servidor, tenemos la forma de comunicación de estos a través de un request y un response. Vimos que este protocolo no maneja estados, es decir, no es que vamos a tener una comunicación y en función de un estado va a hacer una cosa o va a hacer otra, sino que simplemente es un request que se envía y un response que llega y ya la nueva petición que queramos hacer tiene que contemplar todo lo necesario según la aplicación. Hasta el momento vimos cómo se compone una URL para hacer un GET. Incluimos además cuales son los cuatro verbos que se pueden utilizar. Siempre que sea un GET, se lo puede probar con un navegador web, es por eso que todas las APIs que vimos eran GET y por eso los podemos usar por el navegador, porque justamente el navegador termina haciendo un GET, cuando nosotros hacemos una búsqueda en Google, lo que hacemos es un GET a esos servicios de búsqueda de Google que devuelven toda la información que hace falta para que visualicemos lo que vemos en una página de búsqueda. Siempre que sea un POST o un PUT, como habitualmente tenemos que pasar algunos valores dentro del body de un mensaje, vamos a necesitar otras herramientas como un restclient o un curl (línea de comando para Linux).

Headers: tenemos que tener en cuenta que los headers se pueden mandar dentro de los request o como de los response, en donde estos son formas de pasar información que se puede leer tanto del lado del cliente como del servidor. Hay algunos headers que son generales como:

- Cache-control: es un header general que puede ser que vaya o con el request cuando hacemos la petición, o con el response cuando el servidor nos envía la respuesta que necesitamos. Cache-control acepta una serie de valores por los cuales le estamos indicamos al cliente que a ese mismo valor queremos que quede cacheado en el cliente y que cuando tenga que hacer una petición y la misma este dentro del valor seteado, no venga una nueva petición directamente al servidor sino que automáticamente el cliente vea la respuesta que le pasamos antes, porque tiene un valor en el cual nosotros, como dueños del servicio, estimamos que esa respuesta es válida durante un periodo de

tiempo determinado. Acá hay mucha inteligencia por parte del desarrollador o el equipo que trabaja en el servicio de entender cuál es un valor aceptable para que no lo vuelva a pedir en los próximos por ejemplo 10 minutos, 60 segundos o 24 horas. Es una forma de implementar una estrategia de cache muy simple que está del lado del protocolo.
Por ejemplo, si ejecutamos en consola lo siguiente, nos da como resultado un JSON con todos los datos que devuelve esa API y podemos ver los Headers.
curl https://api.mercadolibre.com/sites/MCO -v

Response Headers: particularmente nos interesa un header en la parte del response que es el siguiente:

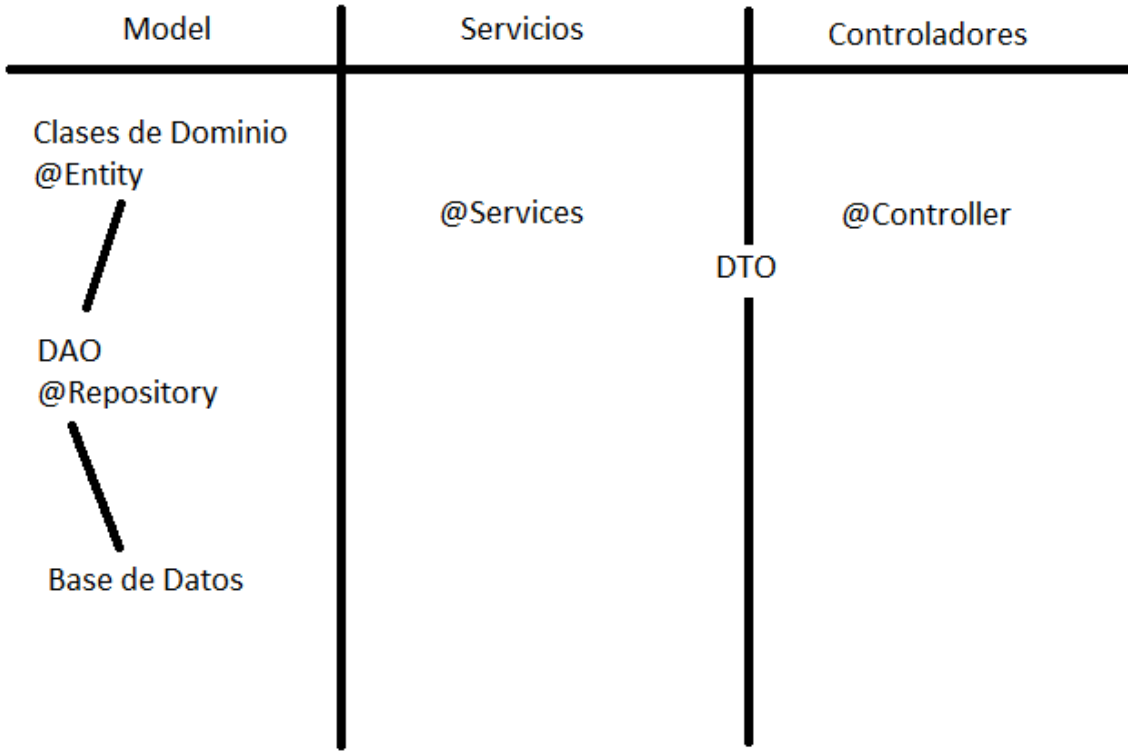
- Location: cuando decimos que alguien puso una redirección, en este Location seteamos cuál es la nueva URL a la cual tiene que apuntar, y entonces el navegador interpreta que se le devuelve un 301, se fija en el Location y directamente redirecciona a la página que tiene que entrar.

## Modelo de negocios

_ Vamos a estar desarrollando esto en capas, y pensamos como haríamos la capa de negocio con Java utilizando un ORM, y como estructuraríamos acá las distintas capas. Muy a grandes rasgos nosotros tenemos 3 capas, donde justamente nos enfocamos en la capa del modelo y en donde vamos a estar escribiendo básicamente clases que forman parte del dominio del problema, y algunas clases que son las que nos van a permitir interactuar contra alguna base de datos. Si nosotros tuviéramos una base de datos cualquiera como MySQL, lo que vamos a estar construyendo acá son clases del dominio, donde estas clases del dominio terminan de alguna manera hablando con un patrón de diseño que se llama DAO (Data Access Object), que es el que de alguna manera terminamos usando en la clase del dominio Hibernate o JPA (tipos de ORM) para poder hacer todo el mapeo y esto termina hablando con el DAO, y el DAO termina ejecutando lo que va a hacer la base de datos. A continuación, vemos que tenemos las capas de servicios y controladores, no ponemos la capa de vista porque no vamos a implementar una vista, sino que vamos a consumir los distintos servicios ya sea a través de una utilidad como un curl o POST, y si es un GET, para obtener un navegador para poder entrar.

_ En el modelo, las clases del dominio tienen una notación que es @Entity porque son entidades, las clases DAO tiene una notación que es @Repository para poder identificarlos. Por un lado, lo que va a suceder en la capa del servidor es que vamos a construir clases Java con una notación en Hibernate que es @Servicies al igual que las otras que mencionamos, y hacemos lo mismo en la capa controladores donde se las clases se denominan @Controller. Lo que va a suceder es que entre la capa de modelo y la de servicio, los objetos que van a viajar si o si son objetos de las clases de dominio, pero entre la capa de servicio y la capa de controladores, los objetos que vamos a estar haciendo viajar se llaman DTO (Data Transfer Object), porque la idea de que cuando programamos en capas, el objeto del modelo no tiene por qué llegar como objeto o

lista de objetos directamente al controlador, sino que entre el servicio y el controlador se comunican con el DTO que en definitiva es lo que se termina convirtiendo, en este caso como estamos implementando una estrategia en la que vamos a tener objetos JSON por lo que vamos a estar devolviendo de nuestros servicios todos objetos JSON, la idea es que ese DTO de alguna manera represente esos objetos JSON que o vamos a recibir o vamos a estar devolviendo en uno de los cuatro verbos que son GET, POST, DELETE y PUT,  o bien vamos a devolver uno de los DTO que vamos a estar recibiendo para hacer un POST cuando queremos dar de alta una nueva entidad en mi servicio o cuando queremos hacer una actualización de datos ese JSON que recibimos en el controlador se va a transformar en un DTO y es básicamente lo que le va a estar llegando al servicio, en donde el servicio es prácticamente el que se encarga de estructurar todo. Todas las clases que construyamos en el controller son simples delegan prácticamente toda la operación a la capa de servicios que es incluso la que termina haciendo transacciones directamente con la base de datos en el caso de tener que registrar operaciones atómicas.




_ Todas las capas que vemos son de construcción, entonces entre el controlador y el servicio, siempre la forma en la que se pasa información es a través de un objeto DTO, y entre el servicio y el modelo, lo que se termina hablando es de entidades de negocio. Lo que no tiene que aparecer nunca en el controlador es que llegue una entidad de negocio o una lista de entidades de negocio.

## Capa del modelo

IDE Eclipse: se debe instalar la versión Eclipse IDE for Enterprise Java Developers. Eclipse utiliza una Máquina Virtual de Java (JVM), es probable que ya este instalada, en caso que no sea así se debe descarga.

_ Nosotros vamos a utilizar una utilidad que se llama Maven, que nos permite no únicamente crear proyectos de un determinado arquetipo y una determinada estructura de arquetipo, donde nos permite crear un proyecto que va a ser justamente para poder hacer una aplicación web directamente preconfigurada, sino que ademas nos permite hacer uso y manejo de un montón de librerías. Como sabemos, cuando codeamos en algún momento incluimos un montón de librerías y no necesariamente codeamos esas librerías, es decir, para casos como por ejemplo el acceso a datos, si la aplicación tiene que disparar un mail, generar un código QR, etc. En Java las librerías tienen una extensión que es .jar y usamos justamente Maven también para todo el ciclo de vida del proyecto para generar y compilar el proyecto, cuando hacemos iteraciones continuas seguramente usaríamos Maven para hacer los build, también se podría configurar para correr todos los test, y así Maven ayuda en distintas fases del desarrollo de la iniciativa, donde lo primero que vamos a hacer es construir un proyecto que tenga las características de Maven para después trabajar más cómodos.

_ En el IDE Eclipse creamos un nuevo Maven Proyect. Maven nos da un montón de arquetipos para distintas soluciones como por ejemplo para crear un proyecto que ya viene preconfigurado con distintas cuestiones. Maven-archetype-webapp es el arquetipo que nos va a permitir generar un proyecto de las características que necesitamos para armar un proyecto web donde vamos a estar codeando el servicio que queramos crear. Cuando en los datos nos pide el grupo colocamos la URL de la que se va a estar viendo la aplicación escribiéndola de atrás hacia delante como por ejemplo edu.ucc.arqsoft2, y luego le colocamos el nombre al proyecto que estamos generando. Una vez creado el proyecto podemos observar que tiene un archivo llamado pom.xml, que es el indicador que hace referencia que el proyecto que construimos es un proyecto Maven. En ese archivo XML tenemos básicamente definidas todas las librerías que nuestra aplicación usa, donde podemos agregar en la sección de “dependencies” todas las librerías que necesitemos y Maven guarda en una sola carpeta en común todas las librerías que agreguemos. La carpeta src/main/java es donde básicamente vamos a empezar a escribir todo el código que deberíamos escribir en nuestra aplicación, y las clases que agreguemos se les tiene que dar una cierta estructura mediante paquetes por lo que creamos paquetes para las clases de modelos, servicios, controladores, los DTO y los DAO para el acceso a datos, y de esta forma estructuramos la aplicación para el caso de que si alguien que trabaja con nosotros tiene que corregir errores, sepa rápidamente donde se encuentra cada cosa.

_ Recordamos que definimos a un ORM como si fuese un traductor que termina hablando entre un lenguaje que es el lenguaje que entiende la base de datos y el otro lenguaje orientado a objetos que es el de la aplicación nuestra, entonces hace una conversión entre los dos lenguajes. Lo que podemos hacer para sacar ese traductor del

medio es que existen bases de datos que son orientadas a objeto, como por ejemplo DB4O (Data Base 4 (for) Objects), entonces si tenemos una base de datos que conoce la orientación a objetos y tenemos nuestra aplicación que conoce el lenguaje de objetos, ya no nos hace falta el traductor en el medio.

_ Siempre que descarguemos un código fuente o una carpeta con todo el código, cuando lo abramos, si este tiene un archivo que es pom.xml, significa que si o si es una aplicación que está escrita en Java y que está usando Maven.

_ Las clases del modelo de negocio (dominio) que vamos a estar codeando son por ejemplo factura, detalle de factura, remito, proveedor, usuario, película, genero de película, socio, alquiler, transacciones, pagos, etc, es decir, dependiendo del dominio son los distintos objetos que pensamos cuando diseñamos el diagrama de clases. En este caso analizamos un video club, donde en las clases del modelo tenemos alquiler, genero, película y socio, donde la idea es realizar un mapeo para que este mapeo, ya sea utilizando Hibernate o JPA, se pueda construir la base de datos y a partir de ahí poder hacer todas las query que vayamos a hacer a través del DTO.

_ Analizando por ejemplo la clase película, vemos que tiene sus atributos privados como cualquier clase normal, también tienen el tipo de dato, y después tienen todos sus setters y getters. Entonces una clase del modelo solamente tiene propiedades y getters y setters para esas mismas propiedades. A la clase película así como también a las otras se le genero una extensión que representa una relación de herencia porque siempre cuando empezamos a escribir las clases aparece la cuestión de saber cuál es la clave primaria y la foránea cuando hacemos relaciones entre una clase u otra, donde la clave primaria nunca debería ser ni un solo campo ni una combinación de campos, usar el DNI como clave primaria no es una buena opción ya que es poco performante y además se repite, por otro lado la combinación que no se repite es sexo (masculino y femenino), tipo de documento y numero de documento, por lo que tendríamos que hacer una clave foránea y esto demora el proceso, por ende cada clase tiene que tener un identificador numérico que la represente después en la base de datos y que le dé la condición de unicidad. Esto no significa que podamos definir por ejemplo dentro de la clase socio, que a pesar de tener un ID o un identificador único, numérica, autogenerada y demás, poder definir condiciones de unicidad, donde por ejemplo definir que la propiedad email no se puede repetir y es única, entonces esto nos da la seguridad de que nunca vamos a poder insertar por ejemplo un socio que tenga el mismo email que otro, pero todo esto es dependiendo de si el modelo de negocio nos especifica eso.

_ Entonces estamos de acuerdo con que es mucho más performante para una base de datos tener un campo identificador numérico, por ende, en todas las clases deberíamos tener un identificador numérico. Como todas las clases tienen un ID, vemos que siempre es el mismo campo ID, por lo que en el paquete common se generó una clase abstracta llamada GenericObject que simplemente lo que tiene es que define un solo campo ID de tipo long con su setter y getter. Recordamos que una clase abstracta significa que nunca vamos a poder implementar los objetos de la

misma, diferente de una interfaz donde definimos métodos y comportamientos en la que después tenemos una clase que termina implementando esa interfaz y termina definiendo ese comportamiento, en el caso de una clase abstracta podemos ya definir el comportamiento que queramos y podemos heredar ese comportamiento tal cual está definido en esa clase o podemos sobre escribir el comportamiento sobre esa misma clase. Entonces en Java tenemos por un lado el concepto de interfaces y el concepto de clase abstracta. Volviendo a la clase GenericObject decimos que nunca vamos poder hacer un new de esa clase para crear un objeto de la misma porque es una clase abstracta, a diferencia de las otras clases socio, película, género y alquiler.

_ Analizando la clase socio, vemos que como atributos tiene nombre, apellido, DNI, vemos también la extensión del GenericObject por lo tanto también tiene un ID, tiene un mail y además tiene una colección de alquileres, sumando aparte sus getters y setters. En el caso de la clase alquiler, tiene asociado una película, un socio, y por otro lado tiene una fecha de alquiler.

Enumeración: vemos que las clases dicen en el logo que las representan tienen como un letra E pequeña sobre ellas, donde básicamente representa una enumeración, y esto es un montón de valores como si fueran constantes en un lenguaje de programación en donde la definimos como si fuera la clase y a su vez define un lista o numeración de cuestiones que por algún motivo no queremos guardar en la base de datos entonces lo tenemos definido como una enumeración. En este caso lo podemos ver reflejado en la clase genero donde tenemos listado constantes que en este caso son los tipos de género (terror, comedia, drama, etc), también por dar otros ejemplos podemos decir que constantes son tipo de documento (documento único, pasaporte, libreta cívica, etc), operaciones en una cuenta de facturación (crédito, débito, efectivo, etc), genero de persona (masculino, femenino, no definido, etc). Aparecen en la clase genero del modelo porque si bien no es una clase que hayamos definido como alquiler socio o película que aparecen rápidamente cuando uno hace el diseño del dominio, pero básicamente para Java no es una clase de implementación, sino que tiene que ver con algo que sucede y que ayuda a construir un buen modelo de datos. La enumeración no tiene ninguna notación, como @Entity o @Table porque no es nada que se vaya a generar en la base de datos, ya que es algo que existe en el diagrama de clases, pero no existe en el diagrama de entidad relación que se crea en la base de datos.

_ Analizando nuevamente la clase película vemos que como atributos tienen un título, sinopsis, un año y tiene un género que hace referencia el género enumerado que construimos. Entonces hasta acá solo armamos un modelo de tres clases básicas.

Set: en Java hay muchas formas de usar hash set y hash map y demás, pero a nivel listas hay dos conceptos que son muy similares que son el set y el list, ambas interfaces. En list si uno agrega N elementos del 1 al 5 y vuelve a agregar el 1, si le hacemos un size a ese list nos dirá que es 6, y en el caso del set el size nos dirá que es 5 ya que el 1 está agregado al principio por eso no lo agrega al final, por ende, el set no tiene elementos repetidos. Nosotros en el caso del ejemplo usamos un set en la parte

de los objetos de negocio, que en este caso es el alquiler, donde estos tienen un identificador y van a venir de la base de datos, por eso no vamos a tener dos alquileres repetidos.

Anotaciones: la forma en la que vamos a trabajar el ORM (Hibernate o JPA) es mediante un concepto de Java que se llama anotaciones, haciendo referencia al @Entity. Básicamente lo que hace el compilador cuando empieza a ejecutar lo que tenga que ejecutar, recorre todas las clases y cuando se encuentra con un @Entity, internamente el compilador entiende que si es un @Entity entonces significa que va a estar persistiendo en una base de datos, por lo que en la siguiente anotación que encuentra como @Table quiere decir que le estamos poniendo un nombre a la tabla que queremos que se construya, por ejemplo a ese nombre se lo ponemos todo en mayúscula porque no necesariamente el nombre de la clase tiene que ser el mismo nombre de la tabla, pero por una cuestión de practicidad, si estamos empezando a construir una base de datos desde cero y una aplicación desde cero, es conveniente de que el nombre de la clase sea el mismo nombre de la tabla. Por otro lado, dependiendo del tipo de columna que tenemos, podríamos tener distintas anotaciones que pueden estar como así también no estar, como por ejemplo para definir a un atributo usamos @NotNull para indicar que no puede ser nulo, @Size para el tamaño, @Column le decimos el nombre de la columna, con @Unique podemos decir que sea único, etc, y para el caso de los atributos con enumeraciones, les damos un nombre con @Column pero en vez de tener un @Size u otra cosa, le indicamos que es una enumeración con @Enumerated y lo que queremos que guarde es el ORDINAL de la enumeración lo que significa que cuando le saquemos una foto a la base de datos cuando existan las películas ya creadas, el ORDINAL hace referencia al orden en el que aparecen las constantes de a clase con las enumeraciones, es decir, a los valores que va a poder adoptar el atributo donde la posición inicial es 0, pero también podemos decir que sea un STRING y a la tabla la va a crear como si fuese todas las constantes juntas (TERROR, COMEDIA, DRAMA, DOCUMENTAL). La enumeración no tiene ninguna notación, porque no es algo que se vaya a generar en la base de datos, sino que es algo que existe en el diagrama de clases pero no existe en el diagrama de entidad relación que se va a crear en la base de datos. Entonces la clase película está guardando en una columna de su propia tabla, el valor de esa numeración.

_ Volviendo a la clase GenericObject, indicamos con @ID que esto va a ser el identificador univoco que va a tener en la base de datos esta clase, luego con @GeneratedValue e IDENTITY decimos que la clave es autoincremental. Con @MappedSuperclass indicamos que esta clase esta mapeada pero que en realidad la termina implementando una super clase, es decir, las clases que terminan implementando ese GenericObject, por lo que este le pertenece a cualquier clase que lo herede como alquiler, socio y película. Esta clase no tiene @Entity porque no existe una tabla que se llame GenericObject, sino que es un valor que va a estar mapeado por las clases hijas.

_ La clase película tiene una relación uno a muchos con el alquiler, ya que una película puede estar varias veces alquilada a lo largo de su historia, pero el alquiler tiene solo una película. Por ende, aplicamos @OneToMany, donde tenemos una colección de películas (Set), en donde al cargar la película número 1 cargamos sus datos, y se va a cargar todas las veces que esa película se alquiló en la colección de alquileres. Entonces, en el set de alquileres, que es la colección de alquileres, se estarían guardando las películas.

_ Seria anti productivo que al momento de buscar una película, la busquemos en la base de datos y también busque toda la información de todos los alquileres históricos que esa película tuvo, entonces esto seria anti productivo porque son recursos que consumimos contra la base de datos y es muy probable que no terminemos usando todos los alquileres de la película. Entonces con fetch = type.EAGER, en la consulta que hagamos a la base de datos, ya va a traer cargado todos los alquileres que tuvo esa película, y si le colocamos type.LAZY, lo que sucede es que la película va a venir solamente cargada con todos sus atributos (genero, anio, sinopsis, titulo), y cuando necesite consultar los alquileres que tuvo esa película ahí va a ir a la base de datos a buscar esa película y traer toda la información que necesite. Si en todo nuestro modelo de objetos definimos una estrategia EAGER lo que va a pasar es que seguramente en algún momento vamos a entrar en una referencia circular, porque por ejemplo cuando vaya a buscar el alquiler, el alquiler tiene una porque es una relación oneToMany, por ende si la película también es EAGER, al momento de buscar el alquiler va a buscar la película, y de esa película va a cargar el mismo alquiler y así se sucesivamente se repite pudiendo dar un error de mapeo por una referencia cíclica. Por eso es mejor trabajar con LAZY hasta que el modelo de datos diga lo contrario.

_ La relación de la clase alquiler con la clase película seria al revés, es decir muchos a uno, y lo indicamos con @ManyToOne, y la diferencia con la anterior es que acá indicamos mediante @JoinColumn la columna por la cual haremos un join cuando hagamos las consultas, que en ese caso es el ID de película y con el que se va a referenciar en la base de datos. También podemos ver que tiene una relación ManyToOne o con la clase socio.

_ En la clase socio tenemos un mapeo de una relación oneToMany con la clase alquiler, donde vemos que este va a tener una colección de alquileres.

_ Tanto el @NotNull como el @Size y como también otros, que podrían no estar y funcionaria todo igual, lo que van a permitir es realizar ciertas validaciones del lado de la aplicación antes de intentar ir a la base de datos. Por ejemplo si queremos crear un nuevo socio, seteamos el nombre, el apellido , el DNI, y supongamos que le seteamos un apellido que tenga 300 caracteres teniendo un @Size de 250, a nivel Java nos lo va a permitir porque el tipo de dato es un string por lo que no tendríamos problema de setear una cadena del tamaño que queramos, pero cuando intentemos hacer el INSERT va a chequear todos las anotaciones, y antes de generar la conexión con la base de datos e intentar ir a la misma, lo que nos va a indicar es que no podemos insertar ese nuevo socio porque nos estaríamos excediendo en la cantidad máxima de

caracteres que se definió. De la otra forma, si la base de datos está bien diseñada con un varchar de tantos caracteres, lo que va a hacer es generar una conexión, intentar hacer el INSERT en la base de datos y el que devuelve el error es la base de datos misma. Entonces conceptualmente es lo mismo, porque no se va a estar insertando en la base de datos un apellido que sea mayor a 250 por ejemplo, pero a nivel recursos es mucho mejor no generar la conexión con la base de datos, es decir, no intentar insertar en la base de datos y mucho antes que Hibernate o JPA haga la validación, por ejemplo, si mandamos null en un campo que tiene definido @NotNull, la base de datos nos va a responder que no podemos hacer esto, y lo que va a pasar es que antes de ejecutar la conexión con la base de datos, nos va a saltar el error.

Paquete common DAO: en este paquete tenemos las clases DAO, donde en la clase GenericDao, vemos el concepto de Generics (E), donde básicamente indicamos que es una interfaz donde definimos todos los métodos o comportamientos que puede tener cualquier clase que implemente esa interfaz, entre esas tenemos un insert(E Entity) donde ese entity va a ir tomando los distintos valores de las clases que pueda implementar como alquiler, película o socio. Tenemos por otro lado una GenericDaoImp abstracta que implementa la interfaz de la clase GenericDao original, y para que implemente, la clase abstracta tiene que tener si o si todos los métodos implementados de la clase original. En la parte del DAO y en la clase abstracta tienen la anotación @Repository que permite saber de qué esto es un repositorio para poder acceder a los datos. En la clase abstracta básicamente tenemos toda la configuración de como nosotros mediante un EntityManager implementa todos los métodos.

Paquete service DAO: en este tenemos una interfaz SocioDao que extiende de GenericDao siendo socio, en donde automáticamente SocioDao tiene todos los métodos que están en la clase abstracta de GenericDao, ya que es una interfaz por lo tanto extiende a otra interfaz. Y en la implementación tenemos el tag @Repository, pero extiende a GenericDaoImp, es decir, la abstracta, e implementa la misma clase SocioDao. Con esto ya podemos usar a socio para hacer absolutamente todas las distintas operaciones con una base de datos. Podemos hacer esto mismo con la clase alquiler y película.

_ Para poder configurar y correr el test, en la carpeta src/test/resource lo que deberíamos hacer es cambiar el archivo test-persistence.xml para asegurarnos de que tengamos la base de datos con el nombre que queramos, el usuario y contraseña. En esta carpeta además tenemos un archivo que se llama import.sql donde se insertan todas las consultas SQL que se realicen. Cada vez que corremos el test vuelve a construir las tablas de la base de datos, inserta lo que estaba cargado con el import.sql, y va a tratar de validar de lo que trajo sea justamente lo que tiene la clase test y así va a pasar.

## Repaso modelo de negocios

_ Antes, nosotros nos concentramos sobre todo en la parte del modelo si se quiere, en donde vimos las clases del dominio con la notación @Entity, también creamos el concepto de lo que es un repositorio usando un patrón de diseño que se llama DAO (Data Access Object) y es el que tiene de alguna manera la conexión con la base de datos, luego construimos un par de clases de dominio, hablamos de enumeración en java, de herencia y del concepto de la implementación de una interfaz en java ademas de extender el comportamiento de una clase abstracta.

_ Lo que vamos a construir ahora es la capa siguiente, la que está en el medio, que es la capa de servicio donde si se quiere hay más lógica desde el negocio. También construiremos un controlador, donde los controladores van a ser los encargados de recibir todas las peticiones que puedan venir de nuestros clientes del pseudo servicio web que estamos construyendo y que es bastante chico pero que sirve a los fines de empezar.

_ Entre la capa de servicios y el modelo, lo que se va a estar pasando acá son las clases de entidad o las clases de dominio que habíamos generado, por ejemplo, película, socio, etc. Pero entre servicios y controladores los objetos que vamos a tener que construir para hablar entre estas dos capas se llamas DTO (Data Transfer Object), en principios son muy parecidos a las clases de dominio, donde tienen solamente propiedades, Getters y Setters, y de alguna manera es lo que vamos a estar recibiendo cuando queramos hacer un POST o PUT, por ejemplo, y es lo que vamos a estar devolviendo hacia el servicio cuando queramos devolver un objeto de cierta característica. Esto quiere decir que nunca en una capa de controlador debería aparecer una clase de dominio o una colección de clases de dominio y nunca debería aparecer en el dominio un DTO para la construcción. Entonces toda la transformación entre DTOs y clases del dominio y viceversa, se termina construyendo en la capa de servicio.

_ Decimos que la capa de servicio es la que más lógica tiene ya que, si nos imaginamos que tenemos un servicio que nos va a permitir de alguna manera generar una orden de compra o un detalle de una factura junto con su encabezado, entonces en la capa controlador vamos a tener un método POST que nos va a permitir registrar facturas básicamente y va a recibir un objeto DTO que en realidad es un JSON que llega y tiene un formato de factura donde esta tiene datos en el encabezado (tipo, fecha, quien la emite, destinatario, teléfono) y también la factura también tiene muchas líneas, en donde cada una es un ítem de la misma, por ejemplo nombre de articulo, código de artículo, cantidad de unidades, precio unitario, etc. Entonces si tuviéramos que registrar eso, básicamente el controlador recibe ese objeto DTO factura donde tiene una colección de cada ítem de la misma, y el DTO debería pasar todo esto al servicio, en donde este lo que tiene que hacer es empezar a entender que es todo lo que tiene que registrar, porque básicamente tiene que registrar el encabezado de la factura y una vez que lo registra tiene que iterar por cada uno de los ítems que tiene esa factura y debería ir registrando cada uno de esos ítems asociados a ese encabezado que se

acaba de registrar, y después por cada una de las líneas que inserta debería hacer un descuento del stock del artículo que está comprando y de esta forma se termina concretando de alguna manera la transacción. En el servicio podíamos manejar toda la transaccionalidad de las distintas operaciones que pueden ser atómicas o no, y en este caso el registro de una factura, es decir, el registro del encabezado y de cada uno de los ítems y después la actualización del stock tienen que ser atómico, y como no se pueden cortar en el medio tenemos que garantizar que ante algún fallo en la base de datos o alguna caída deberíamos hacer un commit de esa transacción.

Patrón: es un problema que ya se ha presentado de distintas maneras y se encontró una solución particular para esos problemas y así aplicar esa misma solución siempre que el problema en cuestión sea adaptable a ese patrón.

Patrón encabezado factura: este es un patrón de diseño que básicamente resuelve la problemática de cuando tenemos una factura que tiene un encabezado o un remito, o una orden de nota de venta, de crédito. Entonces el patrón factura detalle, básicamente cada vez que tengamos una factura, nos dice cuáles son las clases que tener que tener y cuáles son las relaciones que se generan entre los mismos junto con los atributos que va a tener cada una de las distintas clases que forman parte de ese patrón.

Explicación de archivos, continuación: en el primer archivo service tenemos un modelo donde teníamos básicamente tres entidades que se persistían en la base de datos, y género que se refería a una enumeración. Por ejemplo, si abrimos la entidad Alquiler, lo que vimos fue la notación que usaba Sprint para poder entender que esto iba a persistir en una base de datos (@Entity), y después todas las anotaciones asociadas a JPA o al ORM en realidad que termina mapeando contra una tabla en la base de datos, después vemos la forma en la que se va a relacionar con las distintas clases. Vimos que esta clase y las otras se estaban extendiendo de un objeto que creamos que se llamaba GenericObject que era una clase abstracta por lo que no podríamos instanciar nunca una clase de este tipo sino que íbamos a poder utilizar este comportamiento que básicamente lo que tenía era un identificador para la clase porque ya sabemos que en cada uno de los objetos que vamos a estar persistiendo en la base de datos necesitamos tener un ID que va a ser el identificador autoincrementar en la base de datos.

_ Vimos que la enumeración no tiene nada, ya que la enumeración no puede extender tampoco de una clase abstracta porque es como si fueran constantes. En la clase película estábamos haciendo uso de como podíamos trabajar una notación de este tipo, por ejemplo, EnumType.ORDINAL significa que tenemos una numeración de algún tipo como por ejemplo género.

_ Vimos que cuando las columnas son de un tipo en particular, como por ejemplo el String, el hecho de poder anotarlo con @Size o @NotNull, lo que nos va a permitir a nosotros es por ejemplo si creamos una nueva película y le seteamos un campo String en sinopsis que es mayor a los 2000 caracteres, cuando queramos hacer un save de esa

película, nos va a permitir de que el error salte antes de intentar generar la conexión con la base de datos y de intentar hacer el INSERT, y sea la misma base de datos la que nos informe que no podemos hacer el save porque el campo es mayor a 2000 caracteres.

_ También vimos el DAO, que es el patrón que nos permite acceder de alguna manera a la base de datos. La creamos en la carpeta common DAO con dos archivos, una interfaz y una clase abstracta que básicamente encapsula todo el proceso de conectarse a la base de datos y sacar toda la información. El EntityManager es el que de alguna manera conoce cada una de las distintas entidades como se están mapeando gracias a las anotaciones como @Entity entre otras, y sabe cómo implementar un INSERT cuando recibe una entidad de cada uno de los tipos que vamos generando, como alquiler, película o socio. Ademas vimos el concepto de interfaz en java que es ni más ni menos que una definición de métodos que alguien tiene que implementar y que en este caso. Nosotros podemos hacer distintas implementaciones de la misma interfaz, por ejemplo una en la que en vez de ir contra una base de datos, podríamos a grabar cada una de las distintas entidades, o vamos persistir, buscar o hacer todas las implementaciones en file system, entre otras cosas.

_ Otra cosa que hicimos fue generar todos los DAO que no tienen métodos porque extienden de ese GenericDao que implementamos, o sea que por defecto al haber implementado todo esto hace que ya tengamos todos los métodos de INSERT, DELETE, GET ALL, etc.

## Capa de servicio

_ Generamos tres paquetes nuevos que son service, DTO y controller que son justamente las dos capas y el DTO donde pasamos del controlador al servicio. En el archivo de service entendemos que el servicio tiene una notación que es @Service y con esto Sprint cuando levante se va a dar cuenta que esto es un servicio y va a generar esta clase y la va a dejar lista. En este caso tenemos PeliculaService, pero deberíamos tener de todas las demás entidades como SocioService y AlquilerService donde cada una de ellas se va a encargar de trabajar con el DAO que les corresponde. Entonces PeliculaService utiliza el DAO de película, porque tiene que saber cómo se va a insertar, como se va a buscar una película, entre otras cosas. El primer método que vamos a implementar es ver si podemos buscar una película por ID.

_ El servicio lo que hace es ir contra la base de datos, va a recibir algo desde el controlador para poder decidir qué es lo que tiene que hacer, va a ir contra los DAO, va a obtener la información, va a generar un DTO Y va a devolver al controlador un DTO. Este es básicamente el flujo o secuencia de lo que tiene que hacer. Es decir, del controlador va a al servidor ya sea con un DTO o con un parámetro, desde el servicio va a ir hasta el DAO para poder obtener las entidades del negocio, va a devolver ya sea una colección de entidades o una entidad, va a generar un DTO y lo va a devolver al controlador.

_ En el archivo de service, PeliculaService, en este caso vamos a buscar una película por su ID, por lo cual sabemos que el Id de las películas es de tipo Long (definido en el GenericObject), en el servicio recibimos un DTO o una clase primitiva de java (long, int double), nunca podemos recibir una clase del dominio. Luego usa el DAO de película, hace un load que era el método que se encargaba de ir a la base de datos y buscar ese identificador y obtiene un objeto película, porque el DAO de película devuelve justamente los objetos. Entonces, va y busca la película en la base de datos, la obtenemos, y a partir de acá nos va a generar un DTO, previamente creado en la carpeta DTO, en donde el archivo PeliculaResponseDTO va a tener como atributos un nombre y un género, en donde el género no es una enumeración sino que es un string porque queremos que nos muestre en un listado una película que diga su nombre y su género, y además tiene getters y setters. Conceptualmente los DTO tienen propiedades y getters y setters. Entonces volviendo al PeliculaService, a ese DTO que creamos le seteamos lo que obtuvimos de la base de datos, el título de la película y el género, que en el caso de este último que era una enumeración, lo transformamos para que sea un String. Y por último lo que retorna este método es un DTO.

_ Hasta acá se cumple que desde el controlador va a pasar al servicio o un DTO o puede pasar un tipo primitivo de java. El servicio va a agarrar ese DTO o primitivo, va a hacer algo con eso y va a hacer la consulta contra el DAO, luego el servicio va a recibir objetos del modelo, y una vez recibido el servicio va a construir un DTO y los va a transformar en los datos que queremos mostrar en el controlador y lo va a devolver a quien lo haya llamado.

_ Estos métodos del servicio son los que deberían tener toda la lógica de negocio. Entonces hasta acá construimos un DTO y un servicio, pero ahora tenemos que construir un controlador, que es lo que termina llamando justamente a este servicio.

## Capa de controlador

_ En la carpeta de main service creamos una carpeta controller con el archivo en este caso de película, que sería PeliculaController. La notación es intuitiva @Controller. Luego tenemos el @RequestMapping que es el mapping que va a tener a nivel de clase, es decir, que para acceder a cada uno de los métodos que escribimos a continuación, siempre tenemos que acceder a /película. Continuando con el archivo, el primer método que escribimos se llama lookupPeliculaById que con las distintas anotaciones le decimos que este método es un GET, y este nos sirve para obtener información con la URL, nos dice también que lo que producimos es un objeto JSON y la forma de acceder a este método va a ser /película/id, es decir, que para acceder a este método deberíamos poner https://localhost:8080/service/pelicula/id_number, donde id_nombre seria 1, 2, 3, etc, es decir, los distinto identificadores. Entonces cuando venga un ID tal, lo vamos a recibir. El controlador es muy sencillo, ya que recibe un ID, puede hacer alguna validación pequeña, como por ejemplo que no reciba un número negativo o que no reciba un 0 o null, y después le debería decir al servicio que se encargue él de resolver lo que tenga que resolver, y que después devuelva un

DTO o alguna primitiva de Java. Con @ResponseBody, ResponseEntity lo que hace es nos abstrae de poder tener varias cosas, como un objeto, adentro.

_ Resumiendo, si en el navegador ponemos por ejemplo, https://localhost:8080/service/pelicula/1, el ID en el parámetro lookupPeliculaById tomaría el valor 1, ya que le indicamos en el @RequestMapping como /1, y debería poder pasar al servicio para hacer la búsqueda en la base de datos para poder obtener la película con el ID numero 1, para luego poder transformarlo en un DTO, para que el PeliculaResponseDto se reciba como un DTO, y en la web lo que veríamos es un JSON que tendría como datos nombre y género.

_ Si suponemos que las películas tienen una colección escritores (2 o 3 por ejemplo), y nosotros queremos saber la lista de escritores de una película en particular, entonces para obtener la lista de los escritores, en @RequestMapping podríamos hacer "/{id}/escritores", en donde este método debería buscar la película con tal ID y sacar la colección de escritores para transformarlos en una colección de escritores DTO y debería poder devolverlos al controlador. Y aparte, también queremos otro método en la que aparte de obtener la lista de escritores, queremos obtener escritores por ID de esa película, entonces en @RequestMapping podríamos hacer "/{id}/escritores/{idEscritor}/", y en lookupPeliculaById(@PathVariable("id") Long id, (@PathVariable("idEscritor") Long idEscritor), y en este caso recibimos los dos parámetros, el de la película y el de escritor.

_ Si queremos un método para obtener todas las películas, por ejemplo getPeliculas() no necesitamos ningún parámetro, por lo tanto en @RequestMapping tampoco necesitaríamos parámetros por lo que puede valer "/". Ahora vamos a construir el método en el archivo PeliculaService, en donde para devolver una lista hacemos un List de peliculasResponseDto, porque no podemos pasar un List de películas directamente al controlador porque estaríamos pasando la clase del modelo. Entonces lo que tendiéramos que hacer es con películasDao crear un método que obtenga todas las películas de la base de datos (el Service le pregunta al DAO para que le devuelva una lista de películas), una vez que obtenemos esa lista, tenemos que devolver al controlador una lista de películas DTO, por eso generamos un List vacio de películas DTOS, en donde la implementación de List en Java es un ArrayList. Iteramos en todas las películas que vinieron de la base de datos, y por cada una de esas películas, se genera un DTO y lo agrega a la lista de arreglos DTOS, y en donde asi podemos setearle los datos que queramos a cada una. Por ultimo se devuelve el List DTOS, en donde si todo anduvo bien y tenemos 3 películas por ejemplo, lo que vamos a tener es una colección o lista de PeliculaResponseDto donde cada una va a tener seteada su nombre y su genero que van a ser sacados de lo que se obtuvo en la base de datos. Volviendo al archivo PeliculaController, agregamos lo que esperamos recibir, es decir, una lista de PeliculaResponseDto que se llama DTOS con el método que recibe que es getPeliculas() y después retorna.

Validar ID: creamos en el controller la siguiente función en donde dentro de la misma tendría la lógica de validación.
https://stackoverflow.com/questions/22335521/creating-an-isvalid-method

```
private boolean isValidID(Long id){
Return true; }
```

_ Luego en por ejemplo en la función de getPeliculasById (en el controlador) agregamos:

```
If (this.isValidID(id)){
        PeliculaResponseDto dto = PeliculaService.getPeliculaById(ID);
return new ResponseEntity<Objetc>(dto, HttpStatus.ok);
}else{
return new ResponseEntity<Objetc>(dto, HttpStatus.BAD_REQUEST);
}
```

Método POST: creamos un método POST para películas en PeliculaController que recibe y produce un JSON y va a devolver un CREATED. En este caso el mapeo de la URL que funciona es @RequestMapping("/pelicula") para poder hacer el POST. El método se llama registrar y devuelve un DTO, pero lo que el método recibe por parámetro (en el @RequestBody) es un PeliculaRequestDto. En el archivo PeliculaService creamos el método registrar, en donde recibe PeliculaRequestDto que va a ser un JSON que son todos los datos que vamos a estar pasando, es decir los datos para dar de alta una película, por ende se genera una película nueva, le setea los cuatro datos que le hacen falta a la película y los datos los obtiene del DTO, la única diferencia es el genero ya que es una enumeración. Luego hace un inserta de la peliculaDao, y después arma un response que es el que va a terminar devolviendo y le setea los valores para devolverlo como corresponde con el return.

## Conexiones

Archivo web: para indicar que esto es un proyecto web, tenemos que buscar un archivo que se llama web.xml, en donde este archivo, en la estructura que arma Maven está bajo el directorio src/main/webapp/WEB-INF, en donde una vez que se crea el proyecto se crea vacio y hay que empezar a configurarlo. Cuando queremos levantar un servidor web, cuando tomcat empieza a levantar, lo primero que busca es el web.xml y empieza a leer línea por línea el archivo y empieza a ejecutar todas las instrucciones que le decimos que tiene que ejecutar justamente, entonces encuentra el archivo, lo abre y línea por línea empieza a encontrar las cosas que tiene para poder levantar el servidor. Por ende, si nos fijamos en el archivo, lo primero que busca son los archivos applicationContext.xml y infrestructure.xml, donde estos dos están en src/main/resources dentro de la carpeta spring, ya que como estos archivos son .xml no son .java, no están compilados, por lo que están fuera de la carpeta main. La sección de servlet, es la forma en la que le decimos quien va a estar escuchando, es

decir, quien va a ser la clase que va a estar escuchando todo, en este caso es la clase que es un Spring, y lo que recibe esta clase es otro archivo de configuración que es spring-mvc-context.xml.

- Archivo applicationContext: básicamente acá indicamos que tiene que ir a buscar justamente en cada una de las distintas carpetas donde hay software, donde primero busca el modelo, luego el DAO y después el service. Con el component-scan significa que va a buscar todas las clases que estén dentro de la carpeta en este caso edu.ucc.arqSoft, pero lo que busca específicamente son todas las anotaciones que hacíamos como @Entity que poníamos en el modelo, el @Repository que poníamos en el DAO, el @Service que poníamos en el servicio. Un error muy común que se comete es cuando creamos un nuevo servicio y nos olvidamos de poner el @Service, y lo que va a pasar es que en el controller nunca vamos a poder hacer uso de las clases service, porque no está definida como tal. De forma técnica lo que hace el Spring es, mediante una tabla hash, crea la clase y la guarda a la misma ya creada en la tabla, en donde a esa clase la crea con el mismo nombre por ejemplo PeliculaService pero con la primera letra en minúscula, siendo peliculaService y asi con las clases DAO, etc. Por último, con la anotación @Autowired llamamos a las clases y las usa, por ejemplo PeliculaService y le asigna de nombre peliculaService, y por ende no hacemos un new de la clase, es decir, se crea como una única instancia creada de la clase.
- Archivo infrestructure: este nos va a servir para configurar la base de datos. Creamos un archivo persistence.xml y en este lo que hacemos es que estamos usando un archivo plano para guardar la contraseñas en la base de datos y demás, y eso cuando está en producción no está bien, es decir, no es bueno subir a GitHub la configuración de la base de datos con usuario que es de lectura y escritura. Pero desde el inicio definimos un Factory para manejar toda la parte de Hibernate y la parte importante está dentro del archivo persistence.xml.
- Archivo persistence: lo que tenemos acá son ni más ni menos que las clases del modelo que mapeamos (película, socio y alquiler), la base de datos, la conexión, usuario y password, entre otras configuraciones. En este archivo tenemos que modificar como se llama nuestro usuario y contraseña en la base de datos, usualmente se le pone root a ambos, y el nombre de la base de datos, que como vemos en el archivo que sale como mysql://localhost:3306/UCC, el nombre es UCC.
- Archivo spring-mvc-context: en este decimos que busque en la carpeta controller, todos los archivos controladores (@Controller) que hayamos definido y con eso va a entender que cada vez que haya una petición, tiene que fijarse en cada uno de los controladores como son los mapping (“/…”) , y Spring lo que hace es tener anotado todos los métodos en un hash y entonces cuando venga por ejemplo un método GET al /película va a traer tal clase, y de esta forma rápidamente sabe quién resuelve cada cosa, al estar dentro de una tabla

hash, y cuando viene una URL que Spring no sabe cómo resolver nos informa directamente que no puede resolverlo.

Tomcat 5: es un servidor web que debemos instalar de la versión 8 en adelante y se nos descarga en un directorio cualquiera, donde una vez instalado vamos en Eclipse a Windows->preferences->servers y dentro de este tenemos un monton de opciones, pero entramos a Runtime Enviroment y acá tenemos para agregar distintos servidores, en donde es muy probable que este vacio y sin nada y elegimos el que queremos instalar y nos pregunta en donde esta instalado, lo buscamos y listo, asi tenemos el servidor instalado. Un ejemplo puede ser el tomcat 5 (Apache Tomcat v8.5).

_ Para poder correrlo hacemos click derecho en el proyecto, luego Run As->Run On Server-Manualy, agregamos el que tenemos configurado, le ponemos un nombre al servidor, le damos a next y va a deployar el proyecto. En la pestaña consola podemos ver cómo está levantando el tomcat. Una vez que corre el servidor, nos abre un navegador en donde podemos hacer la consulta que queramos y se nos devolverá el JSON, por ejemplo https://localhost:8080/service/pelicula/all. Para hacer un POST a una URL por consola hacemos el siguiente comando, este tiene los mismos campos del request:

- curl -X POST http://localhost:8080/service/pelicula -H 'Content-Type: application/json' -d '{"titulo":"TITULO", "sinopsis":"SINOPSIS", "anio":"2020", "genero":"COMEDIA"}'

_ Como vemos, son los mismos campos o nombres que tiene el objeto RequestDTO. Entonces Java internamente lo que va a hacer es agarrar este JSON, va a crear una instancia del RequestDTO y va a hacer un set a título, a sinopsis, a anio y a género.

_ Ahora se repite todo el procedimiento de crear los archivos Controller, Service, ResponseDto, RequestDto para las clases socio y alquiler, con sus métodos y funcionalidades respectivamente. Por ejemplo si hacemos un POST a socio:

- curl -X POST http://localhost:8080/service/socio -H 'Content-Type: application/json' -d '{"nombre":"Santiago", "apellido":"Vietto", "dni": "42654882", "email":"santiagovietto5@hotmail.com"}'

_ Podríamos tener una validación que avise si uno de los parámetros ingresados no es correcto, por ejemplo en el caso de querer mandar un nombre pero si es /service/película debería dar error o bad request. También una excepción para el caso de no mandar uno de los parámetros requeridos, por ejemplo que al insertar un socio, no se mande el dni.

MySQL: comando en consola:

- mysql -u ucc -p
- show data bases;
- use UCC;
- show tables;

Extra:

```
egaite@ARCBAegaite:~$ cd /utils/web-servers/
egaite@ARCBAegaite:/utils/web-servers$ ls -l
total 10184
drwxrwxr-x 9 egaite egaite     4096 oct  5  2020 apache-tomcat-8.5.58
-rw-rw-r-- 1 egaite egaite 10420837 oct  5  2020 apache-tomcat-8.5.58.tar.gz
egaite@ARCBAegaite:/utils/web-servers$ cd apache-tomcat-8.5.58/
egaite@ARCBAegaite:/utils/web-servers/apache-tomcat-8.5.58$ ls -l
total 144
drwxr-x--- 2 egaite egaite  4096 oct  5  2020 bin
-rw-r----- 1 egaite egaite 19318 sep 10  2020 BUILDING.txt
drwx------ 2 egaite egaite  4096 sep 10  2020 conf
-rw-r----- 1 egaite egaite  5408 sep 10  2020 CONTRIBUTING.md
drwxr-x--- 2 egaite egaite  4096 oct  5  2020 lib
-rw-r----- 1 egaite egaite 57011 sep 10  2020 LICENSE
drwxr-x--- 2 egaite egaite  4096 sep 10  2020 logs
-rw-r----- 1 egaite egaite  1726 sep 10  2020 NOTICE
-rw-r----- 1 egaite egaite  3257 sep 10  2020 README.md
-rw-r----- 1 egaite egaite  7136 sep 10  2020 RELEASE-NOTES
-rw-r----- 1 egaite egaite 16262 sep 10  2020 RUNNING.txt
drwxr-x--- 2 egaite egaite  4096 oct  5  2020 temp
drwxr-x--- 7 egaite egaite  4096 sep 10  2020 webapps
drwxr-x--- 2 egaite egaite  4096 sep 10  2020 work
egaite@ARCBAegaite:/utils/web-servers/apache-tomcat-8.5.58$ cd webapps/
egaite@ARCBAegaite:/utils/web-servers/apache-tomcat-8.5.58/webapps$ ls -l
total 20
drwxr-x--- 15 egaite egaite 4096 oct  5  2020 docs
drwxr-x---  6 egaite egaite 4096 oct  5  2020 examples
drwxr-x---  5 egaite egaite 4096 oct  5  2020 host-manager
drwxr-x---  6 egaite egaite 4096 oct  5  2020 manager
drwxr-x---  3 egaite egaite 4096 oct  5  2020 ROOT
egaite@ARCBAegaite:/utils/web-servers/apache-tomcat-8.5.58/webapps$
```

```java
class Database{
    public static void main(String[] args) throws Exception{
        Connection con=null;
        try{
            Class.forName("com.mysql.jdbc.Driver");
            con=DriverManager.getConnection("jdbc:mysql://localhost/","
            System.out.println("Xampp Mysql Connected..");
            Statement stat=con.createStatement();
            stat.executeUpdate("CREATE DATABASE PROGRAMS");
            System.out.println("database created...");
        }catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
```