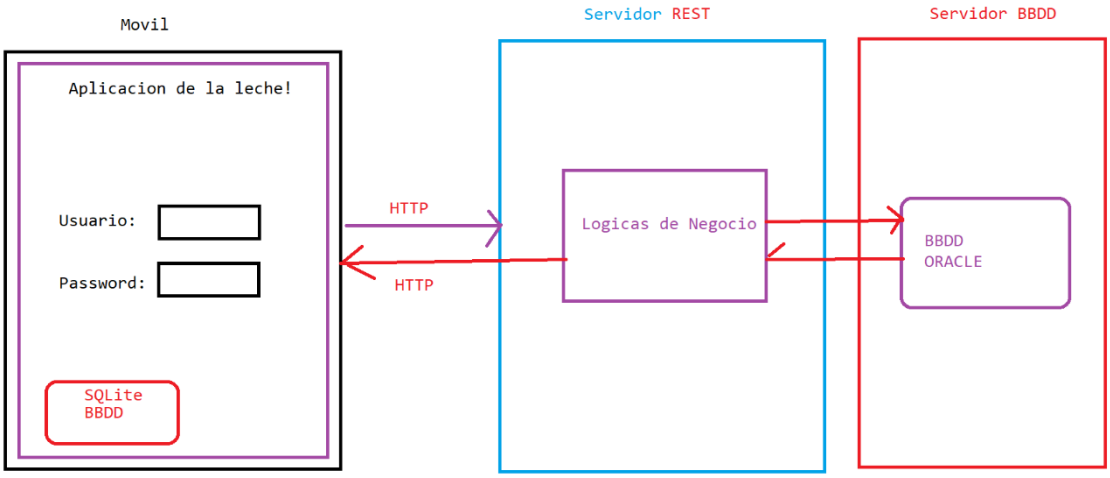
# CLOUD




Cuando se crea una aplicación móvil,  por un lado los dispositivos contienen una base de datos que se usa para guardar información, normalmente suelen ocupar 4MB. Normalmente se dispone de un servidor que se encargar de conectarse con la base de datos. Para realizar la operación de validación, el servidor se divide en diferentes módulos, una de ellas es la lógica de negocio ( es la lógica que tiene que hacer una aplicación) y está se conecta con la base de datos para sacar información.  Se encarga de validar si los datos que se encuentran en la base de datos sean iguales a los que me pase el usuario en la aplicación.

Modelos con servidor:

Ventajas:

-Evitar la sobrecarga de los móviles

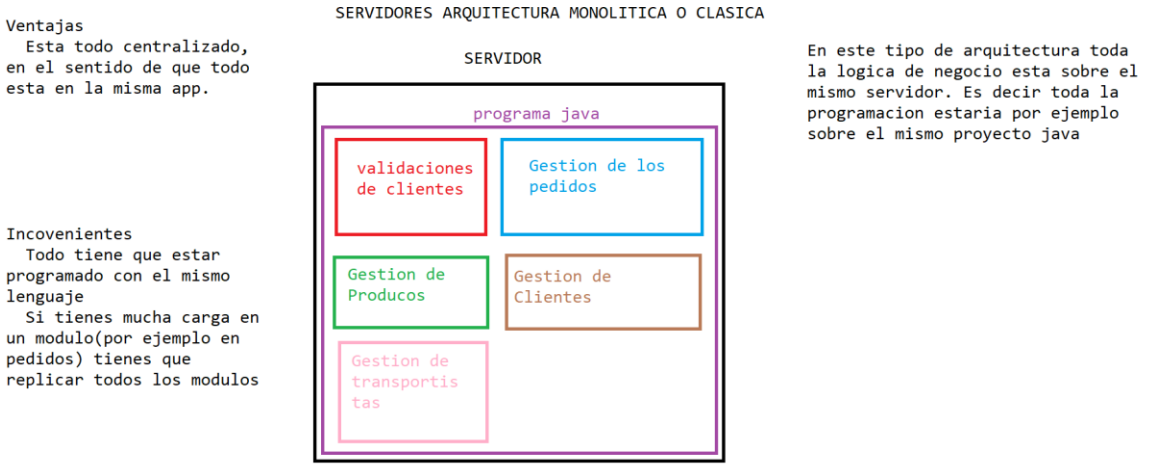
-No se comparte información sensible(passwords)

-No hay que actualizar las apps cada vez que se hagan cambios en los datos

Inconvenientes:

-Sin conexión no hay datos.

-Se consume ancho de banda




Dentro del servidor, hay un programa JAVA, en el cual existen unas clases java como por ejemplo las validaciones de clientes, la gestión de pedidos, la gestión de productos, la gestión de clientes, la gestión de transportistas..

Ventajas:

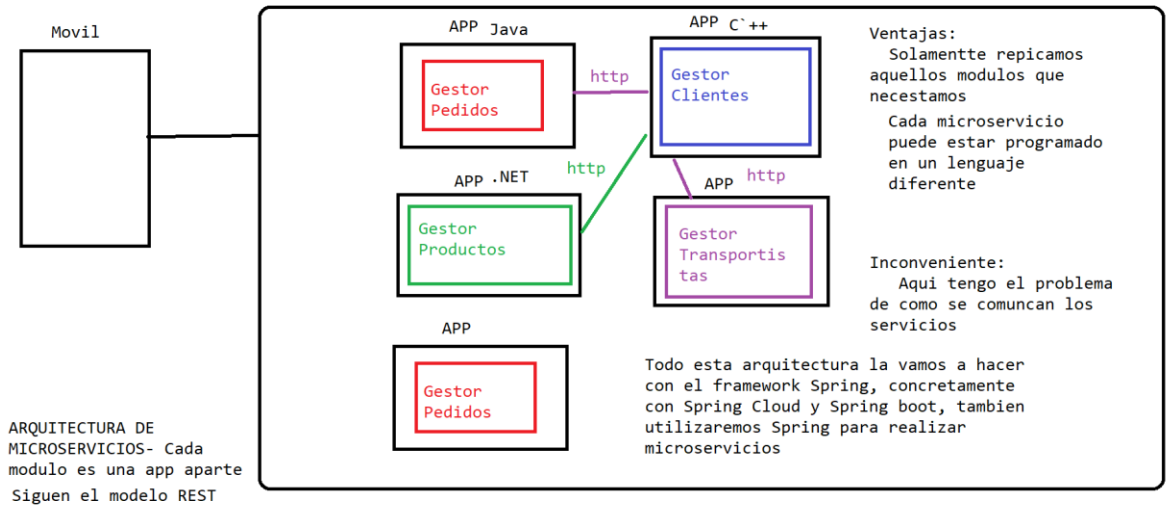
-Está todo centralizado, en el sentido de que si necesitas acceder a algo , todo se encuentra en la misma app.

Inconvenientes:

-Todo tiene que estar programado con el mismo lenguaje.

-Si se tiene mucha carga en un módulo, se tiene que replicar todos los módulos.

## Arquitectura de Microservicios:



En la arquitectura de microservicios, está compuesto por varios servidores que se encargan cada uno de un módulo distinto. Es decir se descompone todo lo que hay en una aplicación en distintos módulos. El cliente del móvil se conectaría a una nube que contiene todos los módulos. Cada módulo puede estar hecho en un lenguaje de programación distinta.

Ventajas:

-Sólo se replica el módulo que sea necesario.

-Cada servicio puede estar programado en un lenguaje diferente. Por el protocolo http es independiente del cliente y del servidor.

Inconvenientes:

-La comunicación de los servicios. Las comunicaciones internas se hacen mediante el protocolo *http*. Esto implica conocerse las direcciones de cada servicio.

# SPRING

Spring es un framework java para realizar aplicaciones de una manera sencilla. Además ayuda a hacer las relaciones que tienen los objetos entre sí, es decir , sus dependencias.

1.¿Qué es Spring?

Spring es un contenedor de objetos java sencillos, POJOs(plain old java object), es un objeto que no tiene herencia ni interfaces y está construido bajo el convenio JavaBean. Un JavaBean es una clase cuyas propiedades son privadas y tiene métodos accesores y modificadores públicos.

2.¿Cómo usarlo?

Spring implementa dos tipos de conceptos muy arraigados en la programación orientada a objetos:

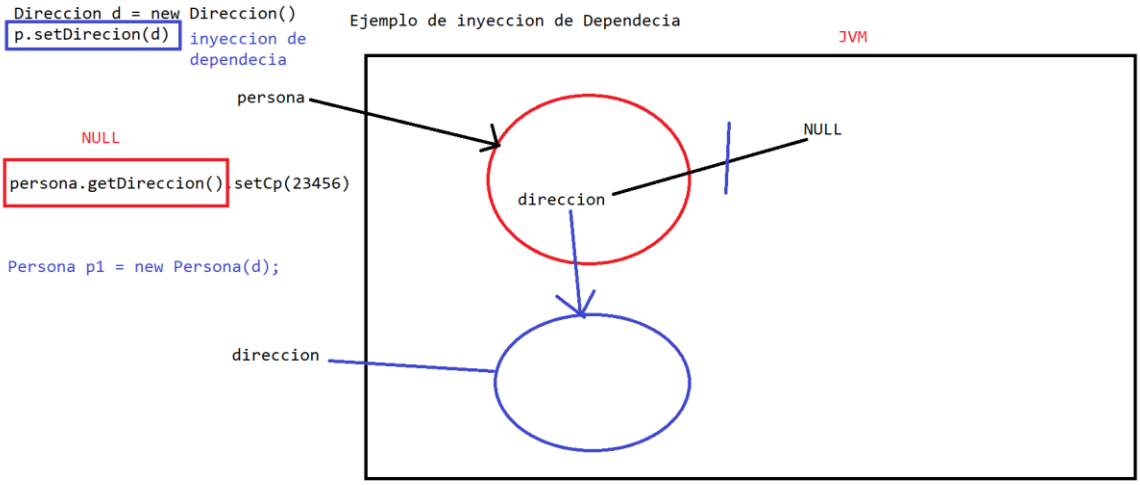
a) IoC(Inversion of Control o Inversión de control)

   Es un paradigma de la programación el cual, muestra que el ciclo de vida de los objetos está gestionado por otra entidad. El ciclo de vida termina cuando pasa el recolector de basura. Cuando el objeto no está referenciado el recolector de basura se lleva al objeto. Por lo que en Spring, no hay de que preocuparse de referenciar objetos para mantener el ciclo de vida ya que éste se encarga de llevarlo por el programador.

b) DI(Dependens Injection o Inyección de dependencias)

Concepto de POO el cual, dice que hay que intentar desacoplar las clases entre sí.

Se hacen acoplamientos de clases cuando se crean objetos dentro de una clase concreta con lo cual se obtiene una dependencia fuerte de una clase con otra, y por ello, los cambios que se hagan en una clase puede afectar al resto de clases.




## @Autowired

Spring @Autowired es una anotación que permite inyectar unas dependencias con otras dentro de Spring,

Funcionamiento:

Cuando se tiene la clase Respository(*@Repository*) , siendo éste el primer componente. Se quiere enlazar este componente desde un Servicio. Se le coloca el Respository referenciado en la clase SevicioRest(*@RestController*) , con la anotación *@Autowired* y por lo tanto quedan enlazados ambos elementos, por si se quiere ejecutar la aplicación mediante Spring Boot, se dispondrá de una URL con los elementos a mostrar.

Y por otro lado, también se utiliza esta anotación a nivel de constructor para un mejor manejo de test. Es decir , manejando un código más limpio y flexible a la hora de usarlo.

## @Component

Con esta anotación se usa para dar de alta a los objetos y además con esta anotación se le indica a Spring que se quiere crear un bean a partir de la clase que se le coloque *@Component*.

A raíz de ésta, heredan otras anotaciones puramente semánticas: *@Controller*, *@Service*, *@Repository*

### @RestController

La configuración será suficiente con crear una clase que use la anotación *@RestController,* Esta anotación sirve para dar de alta un objeto controlador en nuestra app y automáticamente se publicara como un Spring REST Service.

Al crear un servicio REST que nos devuelve una Persona en una URL determinada, concretamente en *https://www.localhost.8080/personas* recordemos que para las URL REST normalmente se utilizan los plurales.

Iguales:

> Ambas anotaciones sirven para mapear URLs con métodos(o recursos) y ambas procesan peticiones HTTP.

Diferencias:

> Cuando se hace un REST el intercambio de información en el body del HTTP se hacía con JSON, en este caso lo que devolverán los métodos serán páginas web

### @ComponentScan

La anotación *@ComponentScan* escanea los packages buscando clases que SpringFramework pueda inyectar.

### @SpringBootAplication

La clase que es la encargada de arrancar nuestra aplicación de Spring, no hace falta desplegarla en un servidor web ya que Spring Boot provee de uno. Y esta clase lleva la anotación *@SpringBootAplication*

## Spring Boot

Fundamentalmente existen tres pasos a realizar . El primero es crear un proyecto Maven/Gradle y descargar las dependencias necesarias. En segundo lugar desarrollar la aplicación y en tercer lugar desplegarla en un servidor. Es decir, el paso dos es una tarea de desarrollo. Los otros pasos están más orientados a infraestructura.

Spring Boot nos ha simplificado toda la operativa a la hora de construir la aplicación prácticamente no hemos tenido que seleccionar dependencias de Spring y no ha hecho falta definir ningún servidor Tomcat en nuestro entorno de desarrollo ya que Spring Boot trae uno integrado.

### Hacer un proyecto SpringBoot:

> botón derecho -> new -> Spring Starter Proyect
>
> 2.1) Elegir nombre del proyecto 2.2) Elegir la versión de java 2.3) Cambiar el nombre del paquete inicial(opcional)

Pulsar Next y elegir los starter que queramos(Spring Web, H2, MySQL), en este caso hemos elegido H2 que es una base de datos

En el caso de que se vaya a construir una aplicación Spring MVC y se leige la dependencia web o Starter Web. Pulsamos generar proyecto y nos descargará un proyecto Maven en formato zip

## Estructura de un proyecto Spring Boot con Maven:

src/main/java -> aquí van todos nuestros fuentes de la aplicación, es decir, los .java

src/main/resources -> aquí van todos nuestro ficheros de recursos de nuestra aplicación, por ejemplo el context.xml, banner.txt

Context.xml

En este fichero se crean todos los objetos que vayan a ser gestionados por el contexto Spring, es decir al aplicar la inversión de control.

Un objeto en Spring (bean) se crea con la etiqueta “bean” y al menos hay que darle 2 atributos, el id que debe ser único y la clase a la que pertenece el objeto que se vaya a crear.

```
<bean id=”persona”  class=”entidad.Persona”></bean>
```

El ciclo de vida de este objeto será gestionado por Spring y por defecto será @Scope(‘singleton’), que significa que el objeto siempre existirá en la aplicación, es decir, será único. El objeto será anónimo con un identificador que será el id. Pero si se quiere inyectar un objeto de manera AUTOMÁTICA, existen dos formas: la primera se coloca el autowired con el valor byType y esto quiere decir que Spring busca un objeto de ese tipo y se lo inyecta.

```
<bean id=”persona”  class=”entidad.Persona” autowired = “byType”></bean>
```

Y el otro se coloca el autowired con valor byName y esto quiere decir que Spring buscará un ID de BEAN, cuyo ID coincida con la propiedad nombre(name).

```
<bean id=”1”  class=”entidad.NombreClasePOJO” autoqired = “byName”></bean>
```

Cuando se crea un objeto en tiempo de compilación y que no vengan creados por lo que hay que usar el alcance *@Scope(‘prototype’*), que éste a diferencia de *@Scope(‘singleton’),* cada vez que se invoque al identificador del objeto, se creará uno nuevo, a imagen y semejanza.

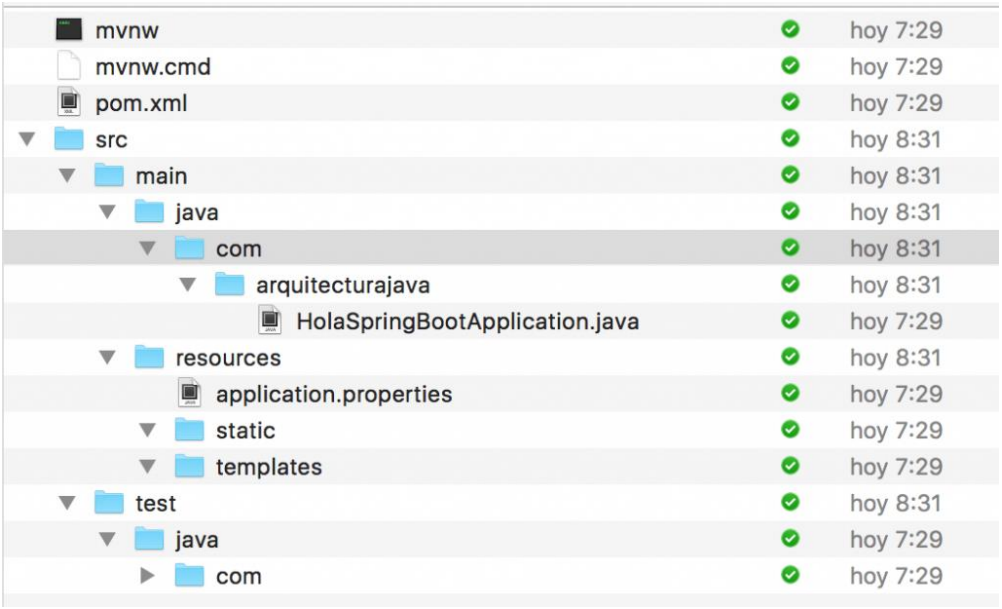
Spring no va a buscar las anotaciones previamente nombradas, sino que sólo va a buscar los beans dados de alta en el fichero de context.xml, si se quiere que Spring lo busque así habría que decírselo mediante la siguiente anotación

```
<context:component-scan base-package=’modelo’></context:component-scan>
```

**OJO!** En esta carpeta esta el fichero más importante de una aplicación Spring Boot que es el *"aplication.properties"*, aquí iría todo el tema de configuración

src/test/java -> aquí van todos los fuentes de pruebas de nuestra aplicación como por ejemplo JUNIT

Maven Dependencies -> aquí van todas las dependencias de nuestro proyecto




### Editar los starters del proyecto:

Botón derecho sobre el proyecto -> Spring -> Edit Starters

## Spring Boot JPA Data:

El primer paso es generar un proyecto de Spring Boot que soporte JPA en este caso es sumamente sencillo ya que solo necesitamos un starter adicional, JPA DATA. Habrá que incluir en fichero pom.xml está dependencia para poder usar ese starter.

<dependency>

<groupId>or.springframework.boot</groupId>

<artifactId> org.springframework.boot spring-boot-starter-data-jpa</ artifactId>

</dependency>

Una vez que se tenga configurado el proyecto de arranque, el primer paso es generar una entidad que pueda ser persistida o seleccionada por JPA. Es decir un POJO, una clase con atributos privados, sus métodos accesores y modificadores, el constructor(sin parámetros y con parámetros) y el toString. Colocando las correspondientes anotaciones *@Entity* la entidad de la clase POJO, @Id al atributo que se coloque como identificador.

La anotación *@GeneratedValue*, permite que la base de datos autogenere el id de manera incremental. Para referirse a una columna en concreto, se coloca la anotación *@Column* y se le añade el parámetro (unique = true), quiere decir que esa columna es única.

En el apartado de *'application.properties'* , se configuran los parámetros contra la base de datos. En él se debe de rellenar las propiedades clásicas con una estructura determinada haciendo referencia a un datasource y al propio hibernate y su dialecto para conectarse a la base de datos

-src/main/resources -> *'application.properties'*

spring.datasource.url=jdbc:mysql://localhost:8889/springjpa

spring.datasource.username=root

spring.datasource.password=root

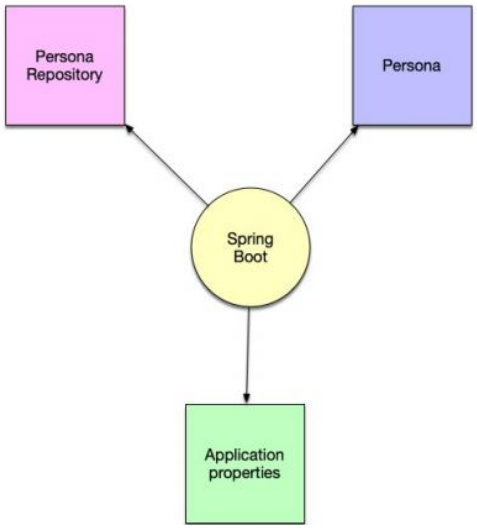
spring.datasource.driver.class=com.mysql.jdbc.Driver

spring.jpa.properties.hibernate.dialect =org.hibernate.dialect.MySQL5Dialect

## Manejo de Repositorios :

Realizado este paso recordemos que Spring nos permite de forma transparente crear interfaces de repositorios apoyados en Spring Data que nos automatiza las operaciones básicas.

public interface PersonaRepository extends CrudRepository < Persona, String > { }



Por último para comprobar cuál es el contenido del programa principal que invoca a este repositorio y nos selecciona los datos de la base de datos. Simplemente inyectamos el repositorio y ejecutamos Spring Boot desde línea de consola, donde los  registros de la base de datos se imprimen.

## Crear un proyecto Spring JPA-DATA

Los pasos para la creación de un proyecto Spring JPA Data:

### Crear un proyecto SpringBoot :

Elegir los siguientes Starters a) Spring MVC b) Spring JPA Data c) H2 ( porque quiero base de datos en memoria, pero podemos elegir MySql) d) Thymeleaf, que sirve para hacer paginas HTML dinámicas

Empezamos con la implementación del proyecto.

Este proyecto va a ser un ejemplo sencillo de una validación de usuario y password a través de una página web. El usuario nos enviará un user y un password y tendremos que validarlo en BBDD.

### Creamos el Modelo:

Esta parte es de las más importantes de la app. Engloba la capa de negocio, la capa DAO y la capa de entidades.

### Crear las entidades: Crear la entidad, la clase POJO.

### Crear la capa de persistencia – DAOs:

Como se está usando JPA Data se debe de crear una interfaz.

El concepto de DAO representa una clase que sirve de conexión de nuestra app con el modelo de datos(bbdd, ficheros).

Para crear el DAO tenemos que crear una INTERFACE que extienda de JpaRepository parametrizar dos valores

1- Persona, que sería la entidad que queremos mapear

2- El tipo de clave primaria de la entidad que estamos mapeando

Lo último que tenemos que hacer, como es una app Spring, es dar de alta este objeto en el contexto de Spring.

### Crear la capa de negocio:

Esta capa recibirá datos de la capa controladora y usará la capa de persistencia para obtener la información que se necesite para hacer la lógica de negocio

### Crear la capa de la vista:

Para la vista se usa páginas HTML, que se pueden crear en 2 carpetas :

src/main/resources/templates o src/main/resources/static

**HTML:**

GET - Se envían los datos(parámetros) como parte de la URL de petición

POST - Se envían los datos(parámetros) que son pertenecientes al BODY del mensaje HTTP (Ej:User, Password, Tarjeta de crédito)

Crea un método que mapee una URL por POST y lo que se devuelve es el nombre de una página web.

En la primera carpeta se pone aquellas páginas que sean dinámica, es decir, aquellas páginas web que su información mostrada dependa de algún otro parámetros. Por ejemplo, la página web de una tienda depende de quien se conecte

En la segunda carpeta se pone aquellas páginas que sean estáticas, es decir, nunca cambian. Por ejemplo, como una página de LOGIN.

Recoger los parámetros que envía el cliente, y se hace por los parámetros de entrada de este método, anotados con @RequestParam("NOMBREDEL_PARAMETRO")

'Form action', es la URL que se va a buscar en el Controlador para poder acceder a la app.

'method', es el método o verbo http que se va a usar

<form action="Logged" method="post/get/put/delete">

'Input type', es el parámetro que se le envía al formulario.

<input type="text"/>

Es posible que al crear una página web te la ponga en la ruta src/main/webapp, se puede quedar ahí o se puede mover

### Crear la capa controladora:

Esta capa será la encargada de mapear URLs y de recibir la información del cliente. Una vez recibía usara la capa del modelo para decir que vista tenemos que mostrar.

### Crear algún usuario de pruebas:

En src/main/java/com/mvc/Application creamos algún usuario de pruebas

- src/main/java/com/mvc/Application ->

## HTTP

Dentro del http tenemos dos partes: Los https petición y los de respuesta

El texto http se puede conocer como un texto plano que se intercambia entre servidor y el cliente.

Esto solo ocurre en la versión http de 1.0 en las siguientes versiones el código que se intercambia es de tipo binario

En el *http* de petición podemos diferenciar 3 partes La primera línea que sería la línea de petición, las líneas de cabecera y por último podría ir el body es una parte opcional. La línea de petición se divide en el método , luego el recurso al que estamos accediendo y por último la versión del protocolo siempre separadas por un espacio y una contra barra. Después se encuentra las cabeceras a continuación se describe una línea en blanco y por último el body ,en el que se muestra la información que estamos intercambiando y además es invisible al usuario.

Dentro del protocolo de respuesta tenemos una estructura muy similar lo único que cambia es la línea inicial del *http* de petición. La estructura básica de una línea de respuesta está formado por primero va el protocolo el estado, Luego el código de respuesta y por último el mensaje del estado. El verbo representa la acción que se va a realizar con el recurso que se está utilizando

Los 100 representan la información que nos devuelve al servidor , A familia de los 200 nos devuelven resultados exitosos, los 300 representan redirecciones cuando estás solicitando un recurso y se ha movido por cualquier razón, a continuación está la familia de los 400 que es cuando se nos devuelve un error del cliente es decir que éste devuelve una respuesta que no cumple los requisitos como por ejemplo 404 Not found(no se encuentra el recurso que se está solicitando) Y por último se encuentra la familia de los 500 Que son errores del servidor que esto significa que el programador algo ha hecho mal.

| Code | Type |
|---|---|
| 1XX | Informational |
| 2XX | Success |
| 3XX | Redirection |
| 4XX | Client Error |
| 5XX | Server Error |

No se pueden tener dos recursos con el mismo nombre

REST es un protocolo de comunicación basado en *http*

Los 5 principios básicos de REST son :

1.Url como recurso. Recurso asociado a una URL

Una buena URL tiene como características las siguientes:

- Fácil de leer/escribir
- Describen el recurso
- Utilizan nombres en lugar de verbos
- Todos los niveles(paths) son recursos. https://www.simpsons.com/marge

  (marge – path)
- Asigna un <id> a tus recursos
- filtra recursos mediante ¿query-params

  https://www.simpsons.com/Ned

  (Ned – identificador primario(clave primaria), estos van dentro de la url)

  https://www.simpsons.com/characters?surname=Flanders

  (para acceder a elementos que no son identificadores primarios, se utilizan los filtros)

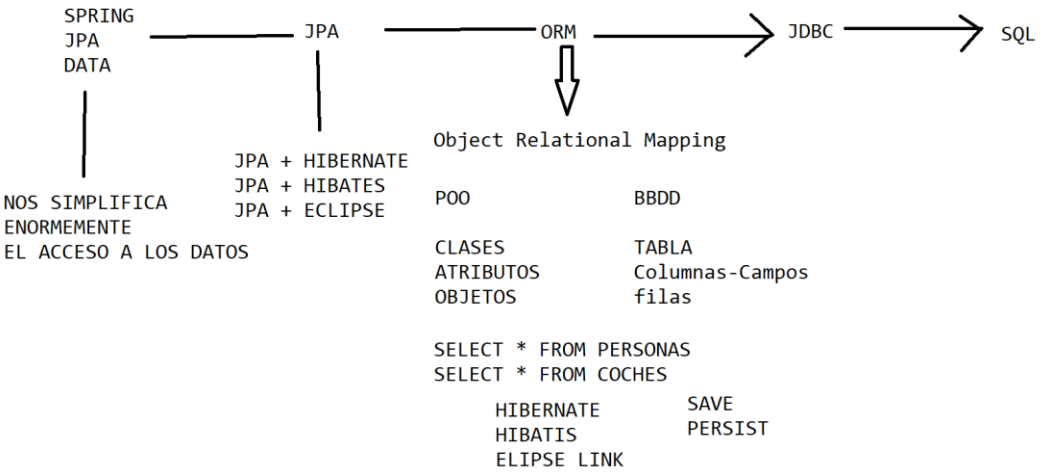
2. Métodos *http*: GET, POST, PUT, DELETE.

3. Múltiples estados de representación que puede enviar los recursos por HTML ,por Json..

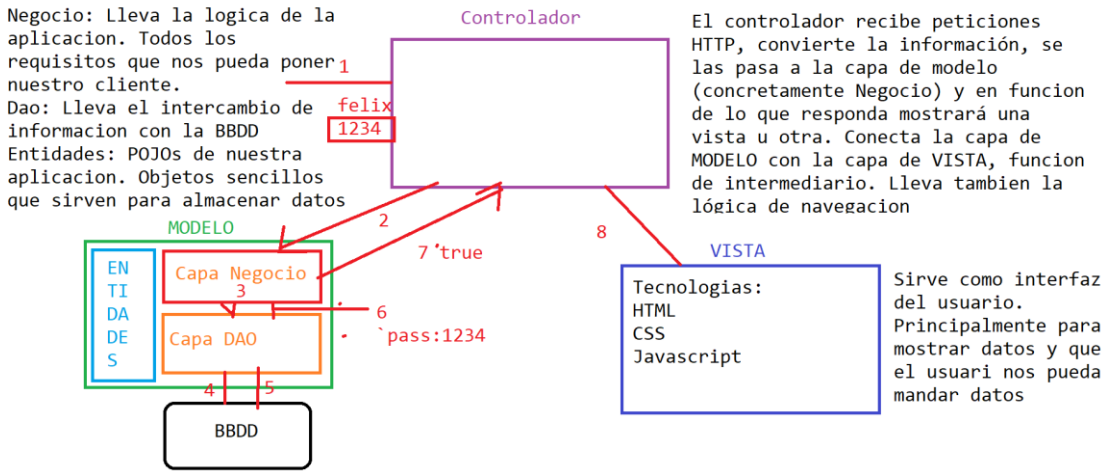
4. Códigos de respuestas

5. El uso de las cabeceras.


ORM : Object Relational Mapping(Mapeo relacional de objetos). Mapea un modelo de datos de POO a tablas BBDD. Existen diferentes tipos de ORM: Hibernate, Hibatis, Eclipse Link…



# MVC




1.Llega, mediante el protocolo http el nombre y el password, al controlador.

2.Manda una petición a la capa Negocio, es decir, manda la información recogida. En la capa Negocio existe un método que se llama validarUsuario(). La capa de negocio tiene la lógica, por lo que tiene que comprobar si los datos pasados por el controlador coinciden con los ya almacenados en la base de datos.

3.La capa Negocio se comunica con la capa DAO.

4. La capa DAO solicita a la base de datos la información reciente pasada por la capa Negocio.

5. Por lo que la base de datos responde a la capa DAO, depende los valores que haya introducido el cliente.

6. La capa DAO devuelve los valores del nombre y del password que ya estaban almacenados a la capa Negocio.

7. La capa Negocio responde al Controlador con un valor booleano = True, que eso quiere decir que los datos están bien introducidos.

8. Por último el controlador decide que vista muestra. Es decir si le llega un valor = True puede mostrar un valor HTML: inicio.html, en cambio el valor = False le puede mostrar error.html.

Thymeleaf: sirve para hacer páginas HTML dinámicas. Es un motor de creación de páginas web dinámicas, que permite escribir páginas web en función de los datos que envíe el Controlador.

${} , se coloca entre llaves el nombre del atributo que se ha cargado en el Controlador dentro del objeto Model.

<p th:text= “Hola ${nombre_usuario}></p>