



9- Programación Web - progra de java en window que probablemente ni te sirva asique pasa de largo

Análisis matematico (UTN Facultad Regional Mendoza)

CURSO DE PROGRAMACIÓN FULL STACK

PROGRAMACIÓN WEB CON JAVA



This document is available free of charge on



Downloaded by Edgar Andrada (edgar.andrada@gmail.com)

GUÍA DE PROGRAMACIÓN WEB

FUNDAMENTOS WEB

El éxito de la web se basa en dos factores fundamentales: el protocolo HTTP y el lenguaje HTML. El primero permite una implementación sencilla de un sistema de comunicaciones que permite enviar cualquier archivo de forma fácil, simplificando el funcionamiento del servidor y posibilitando que servidores poco potentes atiendan cientos o miles de peticiones y reduzcan de este modo los costes de despliegue. El segundo, el lenguaje HTML, proporciona un mecanismo sencillo y muy eficiente de creación de páginas enlazadas.

EL PROTOCOLO HTTP

El protocolo HTTP (Hypertext Transfer Protocol) es el protocolo principal de la World Wide Web. Es un protocolo simple, orientado a conexión y sin estado. Está orientado a conexión porque emplea para su funcionamiento un protocolo de comunicaciones (TCP, o Transport Control Protocol) de modo conectado, que establece un canal de comunicaciones entre el cliente y el servidor, por el cual pasan los bytes que constituyen los datos de la transferencia, en contraposición a los protocolos denominados de datagrama (o no orientados a conexión) que dividen la serie de datos en pequeños paquetes (o datagramas) antes de enviarlos, pudiendo llegar por diversas vías del servidor al cliente.

Explicado de manera más simple, cuando escribes una dirección web en tu navegador y se abre la página que deseas, es porque tu navegador se ha comunicado con el servidor web por HTTP. Dicho de otra manera, el protocolo HTTP es el código o lenguaje en el que el navegador le comunica al servidor qué página quiere visualizar.

HTTPS

Existe una variante de HTTP denominada HTTPS (S significa "secure", o "seguro") que utiliza el protocolo de seguridad SSL (o "Secure Socket Layer") para cifrar y autenticar el tráfico de datos, muy utilizada por los servidores web orientados al comercio electrónico o por aquellos que albergan información de tipo personal o confidencial.

¿CÓMO FUNCIONA HTTP?

De forma esquemática, el funcionamiento de HTTP es como sigue: el cliente establece una conexión TCP con el servidor, hacia el puerto por defecto para el protocolo HTTP (o el indicado expresamente en la conexión), envía una orden HTTP de solicitud de un recurso (añadiendo algunas cabeceras con información) y, utilizando la misma conexión, el servidor responde enviando los datos solicitados y, además, añadiendo algunas cabeceras con información.

La manera más fácil de explicar cómo funciona HTTP es describiendo cómo se abre una página web:

1. En la barra de direcciones del navegador, el usuario teclea **example.com**.
2. El navegador envía esa *solicitud*, es decir, la **petición HTTP**, al servidor web que administre el dominio example.com. Normalmente, la solicitud del cliente dice algo así como "Envíame este archivo", pero también puede ser simplemente "¿Tienes este archivo?".
3. El servidor web recibe la *solicitud HTTP*, busca el archivo en cuestión (en nuestro ejemplo, la página de inicio de example.com, que corresponde al archivo **index.html**) y el servidor envía una *respuesta*. En primer lugar envía una cabecera o **header**. Esta cabecera le comunica al cliente, mediante un *código de estado*, el resultado de la búsqueda.
4. Si se ha encontrado el archivo solicitado y el cliente ha solicitado recibirlo (y no solo saber si existe), el servidor envía, tras el header, el **message body** o cuerpo del mensaje, es decir, el contenido solicitado: en nuestro ejemplo, el **archivo index.html**.
5. El navegador recibe el archivo y lo abre en forma de página web.

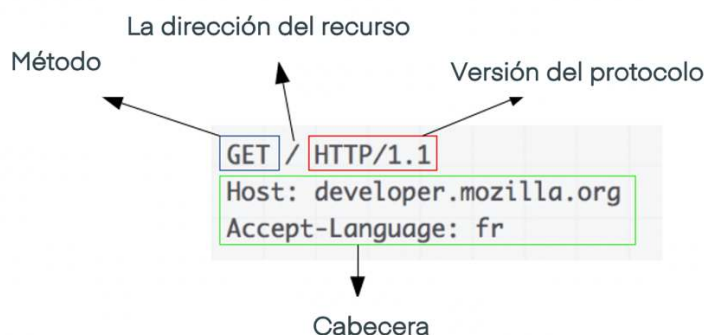
MENSAJES HTTP

Como vimos el servidor recibe un mensaje que es la **petición HTTP** del usuario y después envía una **respuesta HTTP** al cliente/navegador, en base a la petición.

Existen dos tipos de mensajes HTTP: peticiones y respuestas, cada uno sigue su propio formato.

Peticiones

Una petición de HTTP se ve de la siguiente manera:



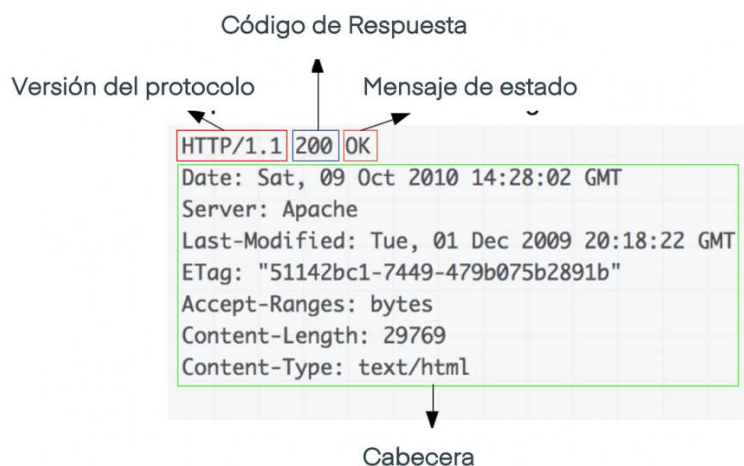
Una petición de HTTP, está formado por los siguientes campos:

- Un **método HTTP**, normalmente pueden ser un verbo, como: **GET**, **POST** o un nombre como: **OPTIONS** (en-US) o **HEAD** (en-US), que defina la operación que el cliente quiera realizar. El objetivo de un cliente, suele ser una petición de recursos, usando **GET**, o presentar un valor de un formulario HTML, usando **POST**, aunque en otras ocasiones puede hacer otros tipos de peticiones.

- La **dirección del recurso pedido**; la URL del recurso, sin los elementos obvios por el contexto, como pueden ser: sin el protocolo (`http://`), el dominio (aquí `developer.mozilla.org`), o el puerto TCP (aquí el 80).
- La **versión del protocolo HTTP**.
- **Cabeceras HTTP**, opcionales, que pueden aportar información adicional a los servidores.

Respuestas

Un ejemplo de repuesta:



Las respuestas están formadas por los siguientes campos:

- La **versión del protocolo HTTP** que están usando.
- Un **código de respuesta**, indicando si la petición ha sido exitosa, o no, y debido a qué.
- Un **mensaje de estado**, una breve descripción del código de estado.
- **Cabeceras HTTP**, como las de las peticiones.

MÉTODOS DE PETICIÓN

En la web, los clientes, como un navegador, por ejemplo, se comunican con los distintos servidores web con ayuda del protocolo HTTP, el cual regula cómo ha de formular sus peticiones el cliente y cómo ha de responder el servidor. El protocolo HTTP emplea varios métodos de petición diferentes.

GET

GET es la madre de todas las peticiones de HTTP. Este método de petición existía ya en los inicios de la *world wide web* y se utiliza para **solicitar un recurso**, como un archivo HTML, **del servidor web**. Esto podría ser cuando un usuario clickea un link para ir a una pagina concreta.

Cuando escribes la dirección URL `www.ejemplo.com/test.html` en tu navegador, este se conecta con el servidor web y le envía una petición GET:

GET /test.html

El servidor enviaría el archivo *test.htm* como respuesta.

PARTES DE UNA URL

A la petición GET puede añadirse **más información**, con la intención de que el servidor web también la procese. Estos llamados parámetros de URL se adjuntan a la dirección URL, la URL puede estar compuesta de varias partes, y las vamos a ver a continuación:

Ruta (Path)

Es lo que viene **después de la barra /**.

Normalmente indica páginas y subpáginas que podemos encontrar en un sitio web.

`www.ejemplo.com/otraPagina.html`

Parámetro (Query String)

Es lo que viene **después del signo de interrogación ?**. Todos los parámetros se componen de un nombre y un valor: "Nombre=Valor".

En una URL puede haber varios parámetros. Y cuando es el caso, éstos se separan con el símbolo de **ampersand &**.

Los parámetros pueden indicar diferentes cosas. A veces tienen que ver con una **búsqueda en el sitio**, a veces son parámetros de campañas publicitarias, etc.

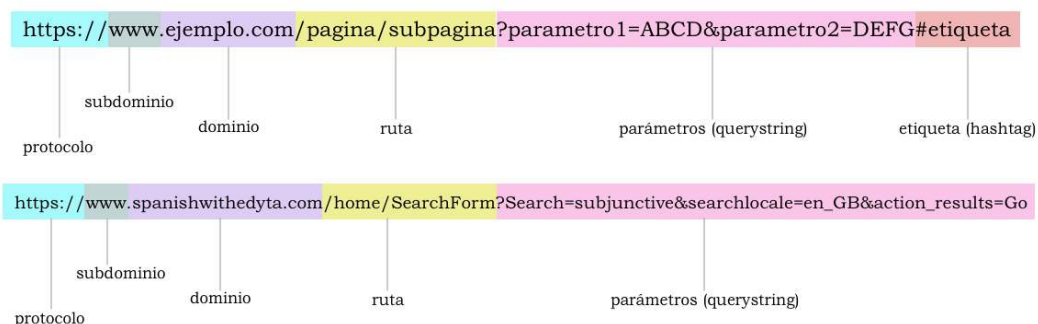
Veámoslo con este ejemplo: para buscar ciertas ofertas en la página web de una empresa de software, en la petición GET se indicará "Windows" como plataforma y "Office" como categoría:

GET /search?platform=Windows&category=office

Etiqueta

Las etiquetas en una URL aparecen **después del hashtag #**.

Su función, entre otras cosas, consiste en permitir hacer scroll hasta un elemento en concreto. Por ejemplo, si mandamos a alguien una URL que contenga una etiqueta, ésta le dirigirá a la parte exacta de la página en cuestión. Veamos una URL completa:



POST

Cuando se tienen que enviar al servidor web paquetes grandes de datos, como imágenes o datos de formulario de carácter privado por ejemplo. El método GET se queda corto, porque todos los datos que se transmiten se escriben abiertos en la barra de direcciones del navegador.

En estos casos, se recurre al método POST. Este método no escribe el parámetro del URL en la dirección URL, sino que lo adjunta al encabezado HTTP.

Las peticiones POST suelen emplearse con **formularios digitales**. Abajo encontrarás el ejemplo de un formulario que recoge un nombre y una dirección de correo electrónico y lo envía al servidor por medio de POST:

```
<html>
<body>
<form action="/prueba " method="post">
Name: <input type="text" name="name"><br>
E-mail: <input type="text" name="email"><br>
<button type="submit">
</form>
</body>
</html>
```

¿CUÁNDO USAR UNO U OTRO?

El método **POST** es aconsejable cuando el usuario debe enviar datos o archivos al servidor, como, por ejemplo, cuando se rellenan formularios o se suben fotos.

El método **GET** es adecuado para la personalización de páginas web: el usuario puede guardar búsquedas, configuraciones de filtros y ordenaciones de listas junto al URL como marcadores, de manera que en su próxima visita la página web se mostrará según sus preferencias.

A modo de resumen:

GET – Utilizado para obtener un recurso del servidor, identificado por una url. Para la configuración de páginas web (filtros, ordenación, búsquedas, links, etc.).

POST – Utilizado para la transferencia de información y datos al servidor. Puede utilizarse para enviar parámetros y su longitud es ilimitada.

CÓDIGOS DE RESPUESTA

Al iniciar el navegador (llamado cliente en este caso) se realiza una petición al servidor web, quien responde, a su vez, con un código de estado HTTP en forma de cadena de tres dígitos.

Con este mensaje, el servidor web le indica al navegador si su solicitud ha sido procesada correctamente, si ha ocurrido un error o si se necesita una autenticación. Como consecuencia, el código de estado HTTP se convierte en una parte esencial en la transmisión de mensajes de respuesta por parte del servidor web, que es insertado automáticamente en su encabezado. Por lo general, los usuarios se encuentran con páginas en formato HTML en vez de códigos de estado HTTP, cuando el servidor web no puede o no tiene permitido procesar la solicitud del cliente o no es posible realizar la transmisión de datos.

TIPOS DE RESPUESTA DE LOS CÓDIGOS DE ESTADO HTTP

En principio, los códigos de estado HTTP se dividen en cinco categorías diferentes, identificadas a su vez, por el primer dígito del código. Por ejemplo, el código de estado HTTP 200 forma parte del tipo de respuesta 2xx, el código 404 del tipo de respuesta 4xx. Esta clasificación se deriva principalmente de la importancia y la función de los códigos de estado, divididos principalmente en 5 tipos:

Códigos de estado 1xx – Información: Cuando se envía un código de estado HTTP 1xx, el servidor le notifica al cliente que la petición actual aún continúa. Esta clase reúne y proporciona información sobre el procesamiento y envío de una solicitud.

Códigos de estado 2xx – Éxito: Los códigos que comienzan con un 2 informan sobre una operación exitosa. Cuando se reciben este tipo de respuestas quiere decir que la solicitud del cliente fue recibida, comprendida y aceptada. Por lo general, el usuario solo percibe la web solicitada.

Códigos de estado 3xx – Redirecciones: Aquellos códigos que comienzan con 3 indican que la solicitud ha sido recibida por el servidor. Sin embargo, para asegurar un procesamiento exitoso es necesario que el *cliente* tome una acción adicional. Este tipo de códigos aparecen principalmente cuando hay redirecciones.

Códigos de estado 4xx – Errores del cliente: Cuando aparece un código 4xx quiere decir que se ha presentado un error de cliente. Esto quiere decir que el servidor ha recibido la solicitud, pero esta no se puede llevar a cabo. Una de las principales causas de este tipo de respuestas son las solicitudes defectuosas. Los usuarios de Internet son informados de este error por medio de una página HTML generada automáticamente.

Códigos de estado 5xx – Errores del servidor: El servidor indica un error propio cuando usa un código 5xx. Este tipo de respuestas indican que la solicitud correspondiente está temporalmente deshabilitada o es imposible de llevar a cabo. De nuevo, se generará automáticamente una página en formato HTML.

CÓDIGOS DE ESTADO HTTP MÁS IMPORTANTES

Los únicos códigos visibles para los visitantes son principalmente los códigos de error del cliente, como el 404 (Not Found), o de error del servidor como el 503 (Service Unavailable), ya que estos siempre se muestran automáticamente como páginas en formato HTML.

Pero ahora que vamos a trabajar sobre la creación de estas paginas web, va a ver códigos que nos van a informar de cosas a arreglar dentro de nuestro programa.

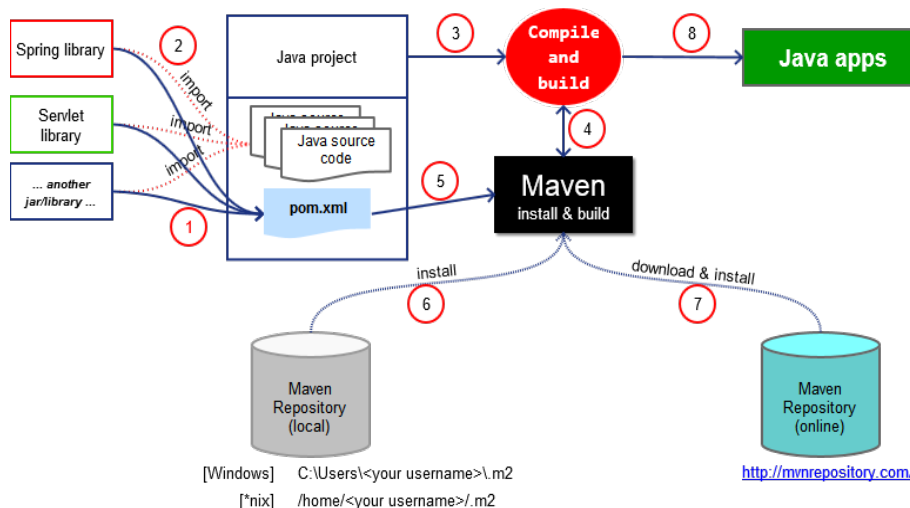
A continuación, presentamos una pequeña selección de los códigos de respuesta más comunes:

- 200 – OK, petición procesada correctamente.
- 301 – Indica al browser que visite otra dirección.
- 403 – Acceso prohibido, por falta de permisos.
- 404 – No encontrado, cuando el documento no existe.
- 500 – Error interno en el servidor.

En el siguiente link puedes ver todos los códigos de estado: [Códigos de Estado](#)

MAVEN

Maven es una herramienta de software para la gestión y construcción de proyectos Java. Utiliza un Project Object Model (POM) para describir el proyecto de software a construir, sus **dependencias** de otros módulos y componentes externos, y el orden de construcción de los elementos. El modelo de configuración es simple y está basado en un formato XML (pom.xml). Además, Maven tiene objetivos predefinidos para realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado. La siguiente figura ilustra los pasos que lleva a cabo esta herramienta desde la importación de librerías hasta la generación de la aplicación Java.



DEPENDENCIAS

Uno de los puntos fuertes de Maven son las dependencias. En nuestro proyecto podemos decirle a Maven que necesitamos un jar (por ejemplo, *log4j* o el **conector de MySQL**) y maven es capaz de ir a internet, buscar esos jar y bajárselos automáticamente. Es más, si alguno de esos jar necesitara otros jar para funcionar, maven "tira del hilo" y va bajándose todos los jar que sean necesarios. Vamos a ver todo esto con un poco de detalle. Las dependencias se recopilan en el archivo **pom.xml**, dentro de una etiqueta **<dependencies>**.

Cuando ejecuta una compilación o ejecutamos un proyecto Maven, estas dependencias se resuelven y luego se cargan desde el repositorio local. Si no están presentes allí, Maven los descargará de un repositorio remoto y los almacenará en el repositorio local. También se le permite instalar manualmente las dependencias.

AÑADIR DEPENDENCIAS EN NUESTRO PROYECTO

Para indicarle a maven que necesitamos un jar determinado, debemos editar el fichero *pom.xml* que tenemos en el directorio raíz de nuestro proyecto. En el *pom.xml* generado por defecto por maven veremos un trozo como el siguiente:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Esta es una dependencia que pone maven automáticamente cuando creamos nuestro proyecto. Presupone que vamos a usar *junit* y en concreto, la versión 3.8.1. Si nosotros queremos añadir más dependencias, debemos poner más trozos como este. Por ejemplo, si añadimos la dependencia del conector de *mysql* versión 5.1.12, debemos poner lo siguiente:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.12</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

SPRING FRAMEWORK

Spring es un framework alternativo al stack de tecnologías estándar en aplicaciones JavaEE. Spring popularizó ideas como la inyección de dependencias o el uso de objetos convencionales (POJOs) como objetos de negocio.

Spring es el framework más popular para el desarrollo de aplicaciones empresariales en Java, para crear código de alto rendimiento, liviano y reutilizable. Su finalidad es estandarizar, agilizar, manejar y resolver los problemas que puedan ir surgiendo en el trayecto de la programación.

¿QUÉ ES UN FRAMEWORK?

Un **framework** es un entorno de trabajo que tiene como **objetivo facilitar la labor de programación** ofreciendo una serie de **características y funciones** que aceleran el proceso, reducen los errores, favorecen el trabajo colaborativo y consiguen obtener un producto de mayor calidad.

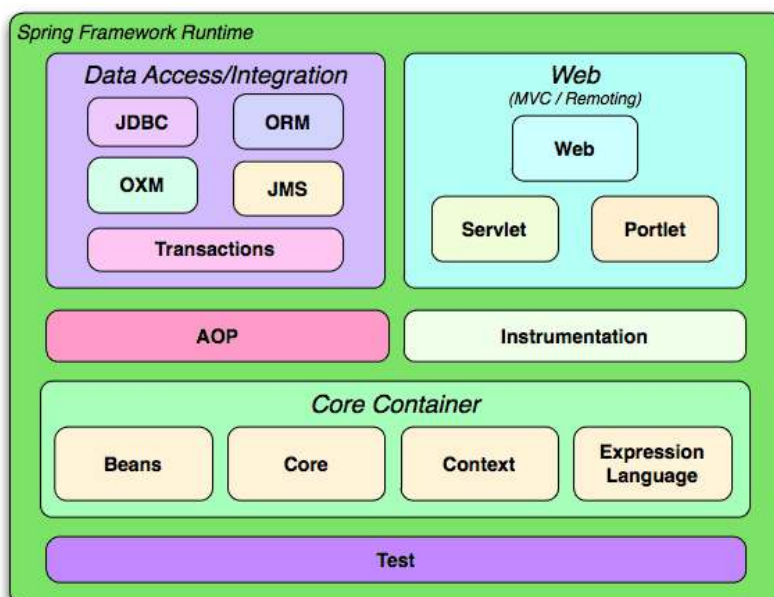
Los framework **ofrecen una estructura para el desarrollo** y no tienen que estar sujetos a un único lenguaje de programación, aunque es habitual encontrar en el mercado, distintos frameworks específicos para un lenguaje concreto.

SPRING FUNCIONALIDADES

Spring, ofrece como elemento clave la inyección de dependencias a nuestro proyecto pero existen otras funcionalidades también muy útiles:

- **Core container:** proporciona inyección de dependencias e inversión de control.
- **Web:** nos permite crear controladores Web, tanto de vistas **MVC** como aplicaciones REST. Esto facilita en gran medida la programación basada en **MVC (Modelo Vista Controlador)**
- **Acceso a datos:** abstracciones sobre JDBC, ORMs como Hibernate, sistemas OXM (Object XML Mappers), JSM y transacciones.
- **Instrumentación:** proporciona soporte para la instrumentación de clases.
- **Pruebas de código:** contiene un framework de testing, con soporte para JUnit y TestNG y todo lo necesario para probar los mecanismos de Spring.

Estos módulos son opcionales, por lo que podemos utilizar los que necesitemos.



SPRING MVC

Antes de pasar a ver la inyección de dependencias, veremos otra funcionalidad, que es el Spring MVC.

Spring Web MVC es un sub-proyecto Spring que está dirigido a facilitar y optimizar el proceso creación de aplicaciones web utilizando el patrón **Modelo Vista Controlador**.



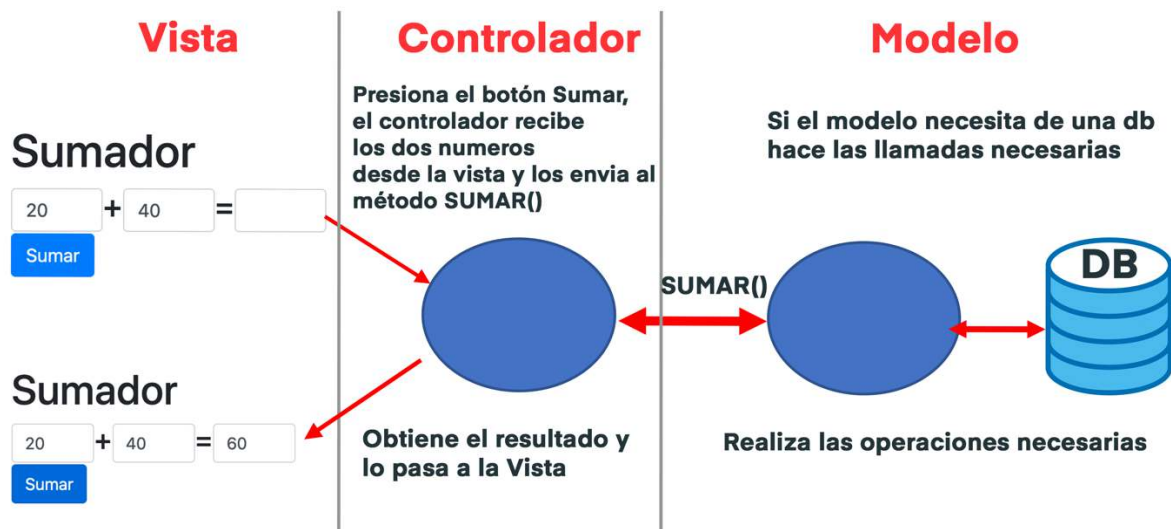
¿QUE ES EL PATRÓN DE DISEÑO MVC?

MVC es un **patrón de diseño** que se estructura mediante tres componentes: *modelo, vista y controlador*. Este patrón tiene como principio que cada uno de los componentes esté separado en diferentes objetos, esto significa que los componentes no se pueden combinar dentro de una misma clase. Sirve para clasificar la información, la lógica del sistema y la interfaz que se le presenta al usuario.

Modelo: Esta capa representa todo lo que tiene que ver con el acceso a datos: guardar, actualizar, obtener datos, además todo el código de la lógica del negocio, básicamente son las clases Java y parte de la lógica de negocio. No contiene ninguna lógica que describa como presentar los datos a un usuario.

Vista: este componente presenta los datos del modelo al usuario. La vista sabe cómo acceder a los datos del modelo, pero no sabe que significa esta información o que puede hacer el usuario para manipularla.

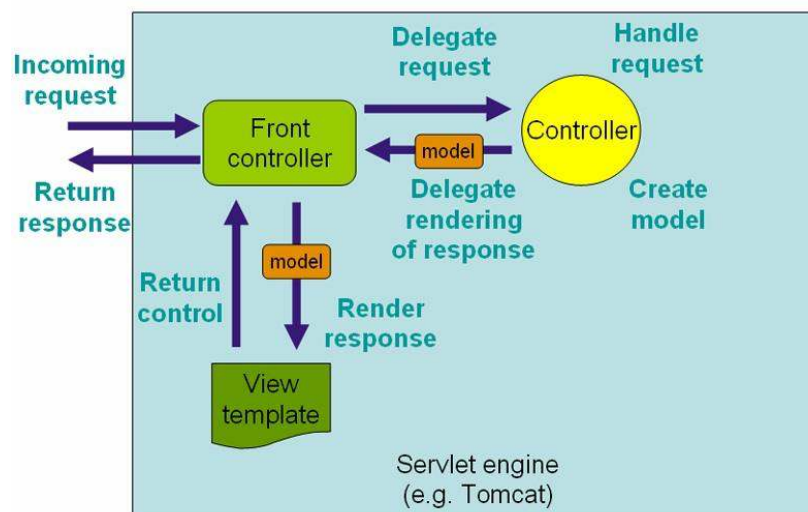
Controlador: este componente se encarga de gestionar las instrucciones que se reciben, atenderlas y procesarlas. El controlador es el encargado de conectar el modelo con las vistas, funciona como un puente entre la vista y el modelo, el controlador recibe eventos generados por el usuario desde las vistas y se encarga de direccionar al modelo la petición, recibir los resultados y entregarlos a la vista para que pueda mostrarlos.



PROCESAMIENTO DE UNA PETICIÓN EN SPRING MVC

Spring MVC se basa en este patrón de diseño para el manejo de las peticiones http y sus respuestas.

A continuación, se describe el flujo de procesamiento típico para una petición HTTP en Spring MVC. Spring es una implementación del patrón de diseño "front controller".



- Todas las peticiones HTTP se canalizan a través del *front controller*. En casi todos los frameworks MVC que siguen este patrón, el *front controller* no es más que un servlet cuya implementación es propia del framework. En el caso de Spring, la clase `DispatcherServlet`.
- El *front controller* averigua, normalmente a partir de la URL, a qué *Controller* hay que llamar para servir la petición. Para esto se usa un `HandlerMapping`.
- Se llama al *Controller*, que ejecuta la lógica de negocio, obtiene los resultados y los devuelve al servlet, encapsulados en un objeto del tipo `Model`. Además, se devolverá el nombre lógico de la vista a mostrar (normalmente devolviendo un `String`, como en JSF).

- Un `ViewResolver` se encarga de averiguar el nombre físico de la vista que se corresponde con el nombre lógico del paso anterior.
- Finalmente, el *front controller* (el `DispatcherServlet`) redirige la petición hacia la vista, que muestra los resultados de la operación realizada.

INYECCIÓN DE DEPENDENCIAS

La inyección de dependencias es quizás la característica más destacable del core de Spring Framework, que consiste en que en lugar de que cada clase tenga que instanciar los objetos que necesite, **sea Spring el que inyecte esos objetos**, lo que quiere decir que es Spring el que crea los objetos y cuando una clase necesite usarlos se le pasan (como cuando le pasas un parámetro a un método).

La DI (*Dependency Injector o Inyector de Dependencias*) consiste en que en lugar de que sean las clases las encargadas de crear (instanciar) los objetos que van a usar (sus atributos), los objetos se inyectan mediante los métodos setters o mediante el constructor en el momento en el que se cree la clase y cuando se quiera usar la clase en cuestión ya estará lista, en cambio sin usar DI la clase necesita crear los objetos que necesita cada vez que se use.

En Spring hay un Contenedor DI que es el encargado de inyectar a cada objeto los objetos que necesita (de los que depende) según se le indique ya sea en un archivo de configuración XML o mediante anotaciones.

Spring a estas clases que van a ser inyectadas por el contenedor, las llama **Spring Beans**.

¿QUE ES UN BEAN?

Un Bean es una clase de Java que debe cumplir los siguientes requisitos:

- Tener todos sus atributos privados (`private`).
- Tener métodos `set()` y `get()` públicos de los atributos privados que nos interese.
- Tener un constructor público por defecto.

A diferencia de los Bean convencionales que representan una clase, la particularidad de los Beans de Spring es que son objetos creados y manejados por el contenedor Spring.

CONTENEDOR SPRING

En Spring hay un Contenedor DI que es el encargado de inyectar a cada objeto los objetos que necesita (de los que depende) según se le indique ya sea en un archivo de configuración XML o mediante anotaciones. En el caso de Spring ese objeto es el contenedor IoC el cual es provisto por los módulos `spring-core` y `spring-beans`.

Spring se basa en el principio de **Inversión de Control (IoC)** o **Patrón Hollywood** («No nos llames, nosotros le llamaremos») consiste en:

- Un Contenedor que maneja objetos por vos, este contenedor es un archivo XML. Este archivo se llama **application-context.xml**.
- El contenedor generalmente controla la creación de estos objetos. Por decirlo de alguna manera, el contenedor hace los “new” de las clases java para que no los realices vos.
- El contenedor resuelve dependencias entre los objetos que contiene.

Un ejemplo típico para ver su utilidad es el de una clase que necesita una conexión a base de datos, sin DI si varios usuarios necesitan usar esta clase se tendrán que crear múltiples conexiones a la base de datos con la consiguiente posible pérdida de rendimiento, pero usando la inyección de dependencia, las dependencias de la clase (sus atributos), son instanciados una única vez cuando se despliega la aplicación y se comparten por todas las instancias de modo que una única conexión a base de datos es compartida por múltiples peticiones.

TIPOS DE INYECCIÓN DE DEPENDENCIAS

Las variantes de DI soportadas por el contenedor IoC de Spring son constructor, setter y con el uso de anotaciones que nos evitaría tener que escribir nada en el xml.

1. Constructor

En este caso el contenedor se encarga de invocar el constructor de la clase pasando los argumentos como dependencias. Es la recomendada para la mayoría de los casos, puedes leer más detalles en la [documentación](#) de Spring.

El siguiente ejemplo muestra una clase que solo puede inyectarse en dependencia con inyección de constructor:

```
public class ListadorPelículasServicio {
// la clase ListadorPelículasServicio tiene una dependencia de
BuscadorPelículaServicio

    private final BuscadorPelículaServicio buscadorPelículaServicio;

// Un constructor para que el contenedor de Spring pueda inyectar
BuscadorPelícula

    public ListadorPelículasServicio(BuscadorPelículaServicio
buscadorPelículaServicio) {

        this.buscadorPelículaServicio = buscadorPelículaServicio;

    }

// la lógica de negocio que usa el buscadorPelículaServicio se omite...
}
```

De esta manera la clase ListadorPelículasServicio, puede usar todos los atributos y métodos de la clase BuscadorPelículaServicio, sin la necesidad de usar el new para instanciar la clase cada vez que queramos usar sus métodos o sus atributos en un método de ListadorPelículasServicio.

Nuestro application-context.xml se vería así, dentro tendremos los beans que queremos inyectar a nuestras clases:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="buscadorPeliculaServicio" class="com.pkg.applicationcontext.BuscadorPeliculaServicio">

  </bean>

</beans>
```

2. Setter

Aquí el contenedor asigna las dependencias usando los métodos setter de los atributos. Recomendada para dependencias opcionales.

// En vez de tener un constructor para que el contenedor de Spring pueda inyectar BuscadorPelicula, tendríamos un método Setter

```
public setBuscadorPelicuaServicio(BuscadorPeliculaServicio
buscadorPeliculaServicio) {
    this.buscadorPeliculaServicio = buscadorPeliculaServicio;
}
```

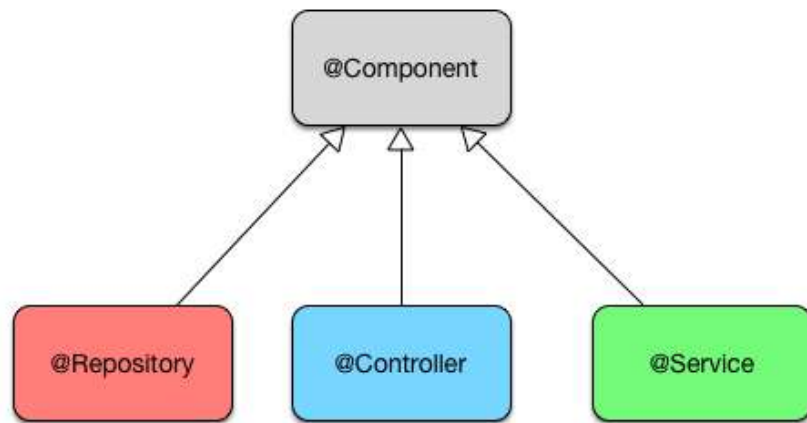
ANOTACIONES

De las dos formas anteriores era necesario indicar en el archivo de configuración que beans podían ser inyectados en otros y sobre los que se querían inyectar.

Ahora, cuando una clase está anotada con uno de las siguientes anotaciones, Spring las registrará automáticamente en el application-context. Esto hace que la clase esté disponible para la inyección de dependencias en otras clases y esto se vuelve vital para construir nuestras aplicaciones. Estas anotaciones se las conoce como **Spring Stereotypes** y se pueden encontrar en el paquete **org.springframework.stereotype**.

Spring Stereotypes

@Component: Es el estereotipo general y permite anotar un bean para que Spring lo considere uno de sus objetos. Un bean es un componente hecho en software que se puede reutilizar y que puede ser manipulado visualmente por una herramienta de programación en lenguaje Java. Sustituye la declaración del bean en el xml.



@Repository: Es el estereotipo que se encarga de dar de alta un bean para que implemente el patrón repositorio. Esta anotación se utiliza en clases Java que acceden directamente a la base de datos. Los repositorios tendrán los métodos para ingresar, editar, eliminar, etc objetos en la base de datos, también, tendrá consultas a la base de datos. La anotación **@Repository** funciona como un marcador para cualquier clase que cumpla la función de repositorio u Objeto de acceso a datos.

@Repository

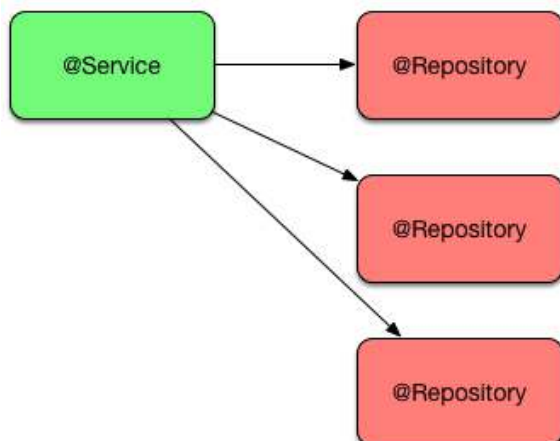
```
public class Repositorio{}
```



@Service: Este estereotipo se encarga de gestionar las operaciones de negocio más importantes a nivel de la aplicación. Usualmente operaciones CRUD, trabajará con los repositorios para enviar los resultados de las operaciones de negocio a la base de datos y persistir los objetos.

@Service

```
public class Servicio{}
```

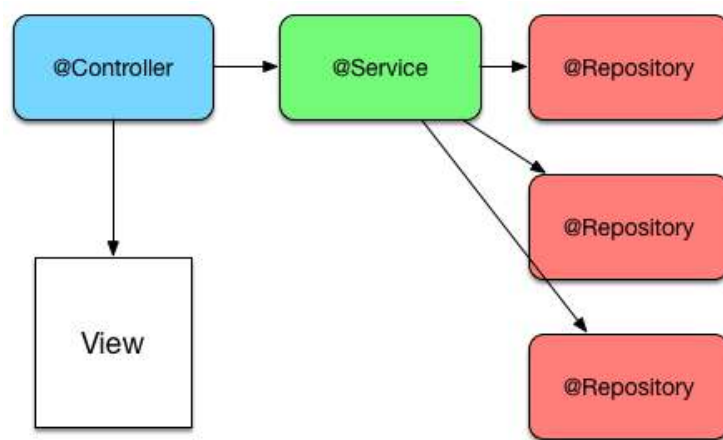


@Controller: El último de los estereotipos que es el que realiza las tareas de controlador y gestión de la comunicación entre el usuario y la aplicación. Para ello se apoya habitualmente en algún motor de plantillas o librería de etiquetas que facilitan la creación de páginas. Donde se realiza la asignación de solicitudes desde la página de presentación, es decir, la capa de presentación (o Interface) no va a ningún otro archivo, va directamente a la clase **@Controller** y comprueba la ruta solicitada en la anotación **@RequestMapping** escrita antes de las llamadas al método si es necesario.

@Controller

```
public class Controlador{}
```

Esta es una clase de controlador simple que contiene métodos para manejar peticiones HTTP para diferentes URLs.



@Autowired Esta anotación le indica a Spring dónde debe ocurrir una inyección. Si se lo coloca en un método, por ejemplo: `setMovie`, entiende (por el prefijo que establece la anotación **@Autowired**) que se necesita inyectar un bean. Spring busca un bean de tipo `Movie` y, si lo encuentra, lo inyecta a este método. Sustituye la declaración de los atributos del bean en el xml. **@Autowired** se emplea para generar la inyección de dependencia de un tipo de Objeto que pertenece a una clase con la **@Component**(**@Controller**, **@Service**, **@Repository**)

@Autowired

```
private final PeliculaServicio peliculaServicio;
```

@Qualifier(«nombreBean»): es una de las anotaciones más prácticas de Spring cuando se quiere añadir versatilidad a como se realiza un **@Autowired** en los componentes. Sirve para indicar que clase es la que se debe inyectar. Con esta anotación podemos indicar el id del bean que se quiere inyectar, esta anotación se usa cuando el atributo que vamos a inyectar es una interfaz de la que hay varias implementaciones y entonces será mediante esta anotación con la que le diremos cual es la clase que queremos inyectar.

SPRING MVC ANOTACIONES

También existen otras anotaciones que nos ayudarán con el manejo del patrón de diseño Spring MVC. Nos darán facilidades para la comunicación entre las vistas, el controlador y los modelos.

@Controller: esta anotación se repite en este apartado, ya que nos da la posibilidad de marcar a una clase como un controlador. Esta anotación se utiliza para crear una clase como controlador web, que puede manejar las solicitudes del cliente y enviar una respuesta al cliente.

@RequestMapping: La clase Controller contiene varios métodos para manejar diferentes peticiones HTTP, pero ¿cómo asigna Spring una petición en particular a un método del controlador en particular? Bueno, eso se hace con la ayuda de la anotación **@RequestMapping**. Es una anotación que se especifica sobre un método del controlador.

Proporciona el mapeo entre la **ruta de la petición** y el **método del controlador**. También admite alguna opción avanzada que se puede usar para especificar métodos de controlador separados para diferentes tipos de petición en el mismo URI como puede especificar un método para manejar una petición GET y otro para manejar la petición POST.

```
@Controller
public class Controlador{

    @RequestMapping("/")
    public String hola(){
        return "Hola Spring MVC";
    }
}
```

En este ejemplo, la página de inicio se asignará a este método de controlador. Entonces, cualquier petición sobre la ruta localhost:8080"/", irá a este método que devolverá "Hola Spring MVC".

@GetMapping: esta anotación se utiliza para asignar solicitudes HTTP GET a métodos de controlador específicos. **@GetMapping** es una anotación compuesta que actúa como un acceso directo para **@RequestMapping** (method = RequestMethod.GET).

```
@Controller
public class Controlador{

    @GetMapping("/")
    public String hola(){
        return "Hola Spring MVC";
    }
}
```

@PostMapping: esta anotación se utiliza para asignar solicitudes HTTP POST a métodos de controlador específicos. **@PostMapping** es una anotación compuesta que actúa como un acceso directo para **@RequestMapping** (method = RequestMethod.POST).

```
@Controller
public class Controlador{

@PostMapping("/guardar")
    public String guardarUsuario(){
        return "Usuario Guardado ";
    }
}
```

@RequestParam: esta es otra anotación Spring MVC útil que se usa para vincular los parámetros de una petición HTTP a los argumentos de un método controlador. Por ejemplo, si envía parámetros de un formulario junto con URL para guardar un usuario, el método puede obtenerlos como argumentos propios.

```
@GetMapping("/libro"){
public void mostrarDetalleLibro(@RequestParam("ISBN") String ISBN){
    System.out.println(ISBN);
}}
```

Si accedes a tu aplicación web que proporciona detalles del libro con un parámetro de consulta(query string) como el siguiente:

http://localhost:8080/libro?ISBN=900848893

Entonces se llamará al método del controlador porque está vinculado a la URL **"/libro"** y el **parámetro de consulta ISBN** se usará para completar el **argumento del método** con el mismo nombre **"ISBN"** dentro del método **mostrarDetalleLibro()**.

De esa manera podemos obtener en nuestro controlador un dato que viaja a través de una URL, que va a venir de una petición HTTP.

@PathVariable: esta es otra anotación que se utiliza para recuperar datos de la URL. A diferencia de la anotación **@RequestParam** que se usa para extraer parámetros de consulta, esta anotación permite al controlador manejar una petición HTTP con URLs parametrizadas, estas serían URLs que tiene parámetros como parte de su ruta, por ejemplo:

http://localhost:8080/libro/900848893

Entonces para poder acceder a este detalle que se encuentra en la ruta de la URL, usaríamos la anotación **@PathVariable** de la siguiente manera:

```
@GetMapping("/libro/{ISBN}") {
public void mostrarDetalleLibro(@PathVariable("ISBN") String ISBN){
    System.out.println(ISBN);
}
}
```

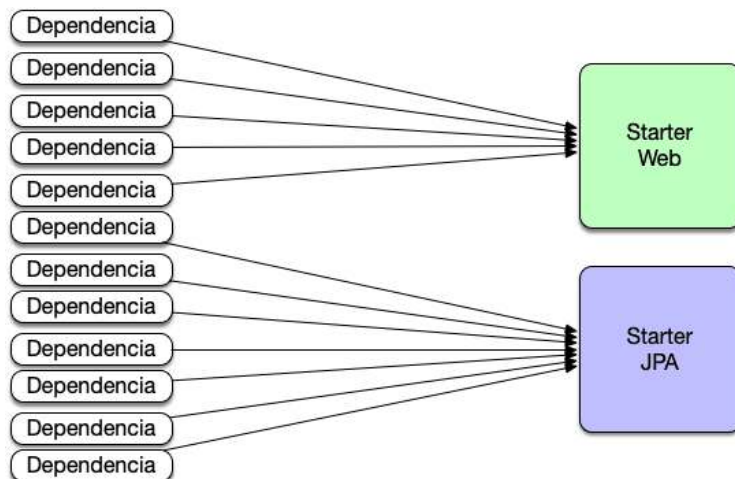
La variable Path o variable de ruta se representa entre llaves como **{ISBN}** en nuestra ruta de petición, lo que significa que la parte después de **/libro** se extrae y se completa en el **ISBN** del argumento del método, que está anotado con **@PathVariable**.

SPRING BOOT

Spring Boot es una de las tecnologías dentro del mundo de Spring de las que más se está hablando últimamente. **¿Qué es y cómo funciona Spring Boot?** . Para entender el concepto primero debemos reflexionar sobre cómo construimos aplicaciones con Spring Framework

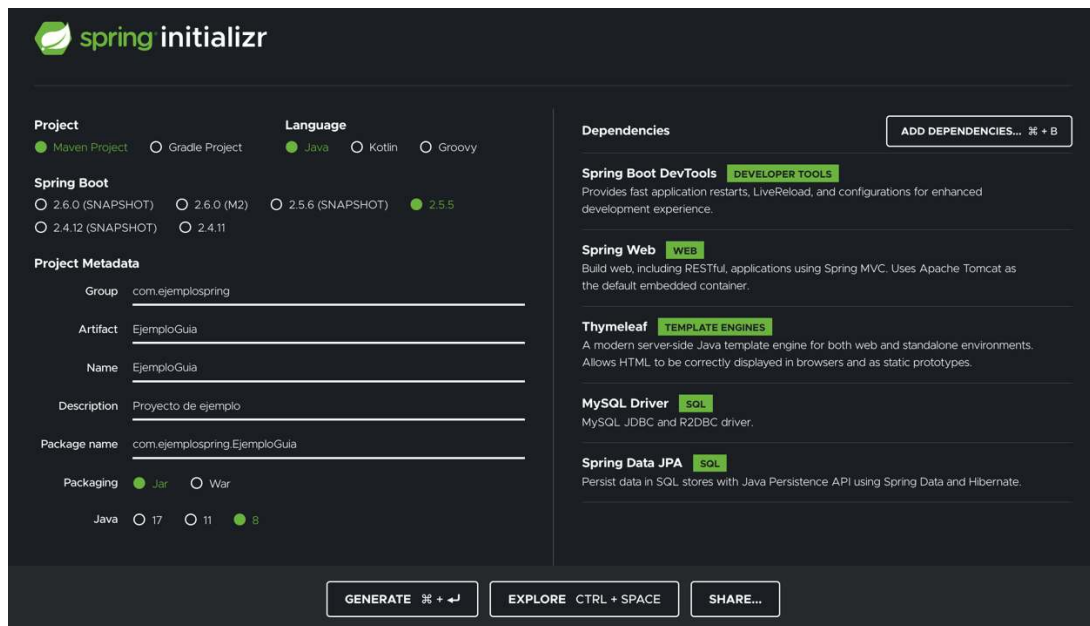


Fundamentalmente existen tres pasos a realizar. El primero es crear un proyecto Maven/Gradle y descargar las dependencias necesarias. En segundo lugar desarrollamos la aplicación y en tercer lugar la desplegamos en un servidor. Si nos ponemos a pensar un poco a detalle en el tema, **únicamente el paso dos es una tarea de desarrollo**. Los otros pasos están más orientados a infraestructura. No deberíamos tener que estar eligiendo continuamente las dependencias y el servidor de despliegue.



SPRING INITIALIZER

SpringBoot nace con la intención de simplificar los pasos 1 y 3 y que nos podamos centrar en el desarrollo de nuestra aplicación. ¿Cómo funciona?. El enfoque es sencillo y lo entenderemos realizando un ejemplo. Para ello nos vamos a conectarnos al asistente de Boot que se denomina Spring Initializer.



The image shows the Spring Initializr web interface. It has a dark theme with green accents. The 'Project' section on the left has 'Maven Project' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '2.5.5' selected. The 'Project Metadata' section has fields for Group (com.ejemplospring), Artifact (EjemploGuia), Name (EjemploGuia), Description (Proyecto de ejemplo), and Package name (com.ejemplospring.EjemploGuia). The 'Packaging' section has 'Jar' selected. The 'Dependencies' section on the right lists several dependencies: Spring Boot DevTools, Spring Web, Thymeleaf, MySQL Driver, and Spring Data JPA. At the bottom, there are buttons for 'GENERATE', 'EXPLORE', and 'SHARE'.

En este caso voy a construir una aplicación **Spring MVC** y elijo la dependencia **web** o **Spring Web**. Pulsamos generar proyecto y nos descargará un proyecto Maven en formato zip. Lo descomprimos y lo abrimos en nuestro IDE, cuando lo vayamos a compilar, Maven se encargará de descargar todas la dependencias y sumarlas a nuestro proyecto.

Una vez que se termine de descargar nuestro proyecto Maven, se convertirá en proyecto Spring para poder trabajar, dentro encontraremos la clase **EjemploGuiaApplication**, se verá de la siguiente manera:

```

1. package com.ejemplospring;
2.
3. import org.springframework.boot.SpringApplication;
4. import org.springframework.boot.autoconfigure.SpringBootApplication;
5.
6. @SpringBootApplication
7. public class EjemploGuiaApplication{
8.
9.     public static void main(String[] args) {
10.         SpringApplication.run(EjemploGuiaApplication.class, args);
11.     }
12. }
```

Esta clase es la encargada de arrancar nuestra aplicación de Spring a diferencia de un enfoque clásico no hace falta desplegarla en un servidor web ya que Spring Boot provee de uno. Vamos a modificarla y añadir una anotación.

SERVIDOR LOCAL

Levantar un servidor o tener un servidor a nuestra disposición no es algo fácil, ni barato. Por suerte Spring Boot nos deja, a través de **Apache Tomcat** y la clase que vimos previamente levantar un servidor local.

Tomcat nos permite hacer una conexión por red a si mismo, o escuchar a la espera de conexiones entrantes que se vayan a originar en el mismo dispositivo.

Se usa para desarrollo y pruebas: normalmente como desarrollador montas un servidor web (apache) y este escucha en el puerto 8080. Entonces lo que hace el desarrollador para probar las páginas web que esta creando, o las aplicaciones web, o los APIs o servicios, es apuntar su navegador a <http://localhost/> o <http://localhost:8080/> para hacer sus pruebas, cuando el puerto 80 se usa no se requiere especificar, solo cuando es un puerto diferente se tiene que poner con : después del nombre

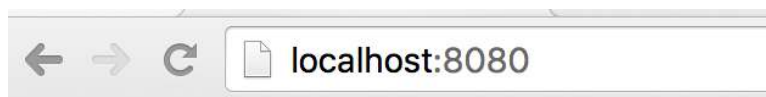
Por lo que si tenemos la siguiente clase:

@Controller

```
public class ControladorHola {  
@GetMapping("/")  
    Public String home() {  
        return "holaMundo";  
    }  
}
```

El controlador recibe la petición GET de HTTP con el GetMapping y usando el **return**, retorna como respuesta HTTP una pagina HTML como String, con el nombre holaMundo, que dentro tiene un **<p>HolaMundo</p>**, también por eso ponemos el método como String.

Entendido esto, vamos a nuestra clase EjemploGuiaApplication y corremos nuestro proyecto, se nos va a levantar un servidor local, por lo que si vamos a **localhost:8080/**, nos encontraremos con la siguiente página:



HolaMundo

Programación en Capas

La programación por capas es un estilo de programación en el que el objetivo primordial es la separación de la lógica de negocios de la lógica de diseño.

La programación por capas es una técnica de ingeniería de software propia de la programación por objetos, éstos se organizan principalmente en las siguientes capas:

Capa de Interfaz

Esta capa resuelve la presentación de datos al usuario. Esta capa se encarga de "dibujar" las pantallas de la aplicación al usuario, y tomar los eventos que el cliente genere (por ejemplo, el hacer click en un botón).

Capa de Comunicación

En esta capa están los controladores y es capa responsable de mediar entre la interfaz de usuario y las capas inferiores. En esta capa contiene el dispatcher encargado de enrutar las peticiones, así como los controladores de acceso a los servicios web.

Capa de Servicios

Esta capa resuelve la lógica de la aplicación. Contiene los algoritmos, validaciones y coordinación necesaria para resolver la problemática. Los elementos fundamentales de esta capa son los objetos de dominio. Estos objetos representan los objetos principales del negocio. La lógica para manipular los objetos que representan los datos se encuentra en los llamados objetos de negocio (Service Object).

Capa de Acceso a Datos (Repositorios)

Esta capa resuelve el acceso a datos, abstrayendo a su capa superior de la complejidad del acceso e interacción con los diferentes orígenes de datos. Esta capa se encarga de proveer un API simple de usar, orientado al negocio, sin exponer complejidades propias de un repositorio de datos.

En esta capa se resuelven:

- cualquier acceso a la base de datos
- cualquier acceso a filesystem
- cualquier acceso a otros sistemas
- cualquier acceso a un repositorio de datos en cualquier forma.

THYMELEAF

Thymeleaf es un motor de plantillas, es decir, **es una tecnología que nos va a permitir definir una plantilla** y, conjuntamente con un modelo de datos, obtener un nuevo documento, sobre todo en entornos web. Para saber más sobre Thymelaf, recomendamos meterse en su documentación: [Thymelaf](#)

¿QUE ES EXACTAMENTE UN MOTOR DE PLANTILLAS?

El motor de plantillas (utilizado específicamente aquí para el desarrollo web) se genera para separar la interfaz de usuario(Vistas), de los datos comerciales(Modelos), puede generar documentos en un formato específico y el motor de plantillas para el sitio web generará un estándar Documento HTML.

Las plantillas, o más exactamente los motores de plantillas (templates engines) leen un fichero de texto, que contiene la presentación ya preparada en HTML, e inserta en él la información dinámica que le ordena el Controlador, la parte que une la vista con la información.

Veamos un ejemplo para ver las posibilidades de las plantillas, que no acaban, ni mucho menos, en la web. La sintaxis a utilizar depende del motor de plantillas utilizado, pero todos son muy similares. Los motores de plantillas suelen tener un pequeño lenguaje de script que permite generar código dinámico, como listas o cierto comportamiento condicional, pero esto también depende del lenguaje.

Este lenguaje de script es absolutamente mínimo, lo justo para posibilitar ese comportamiento dinámico:

```
<html>
<body>
Hola ${nombre}
</body>
</html>
```

Esta claro que este ejemplo, que es una pequeña variación del famoso "Hola Mundo", es bastante simple. Lo que está sucediendo, al procesar este fichero, el motor de plantillas lo recorrerá, analizará y sustituirá esa *"etiqueta clave"* `${nombre}` por el texto que le hallamos indicado, el nombre del visitante, por ejemplo, de forma que tengamos una presentación personalizada.

Básicamente, el motor de plantillas se encarga de recibir, una variable de tipo String llamada nombre, que se la va a enviar el controlador a la vista y la hará parte del HTML, haciéndolo dinámico. Por lo que los diferentes usuarios verán diferentes resultados.

VENTAJAS THYMELEAF

Permite realizar tareas que se conocen como **natural templating**. Es decir, como está basada en añadir atributos y etiquetas, sobre todo HTML, va a permitir que nuestras plantillas se puedan renderizar en local, y esa misma plantilla después utilizarla también para que sea procesada dentro del motor de plantillas. Por lo cual **las tareas de diseño y programación se pueden llevar conjuntamente**.

TIPOS DE EXPRESIONES

Permite trabajar con varios tipos de expresiones:

Expresiones variables: Son quizás las más utilizadas, como por ejemplo `${...}`

Expresiones de selección: Son expresiones que nos permiten reducir la longitud de la expresión si prefijamos un objeto mediante una expresión variable, como por ejemplo `*{...}`

Expresiones de enlace: Nos permiten crear URL que pueden tener parámetros o variables, como por ejemplo `@{...}`

EXPRESIONES VARIABLES

Algunos ejemplos de expresiones variables son:

`${sesión.usuario.nombre}` : Podemos usar la notación de puntos para acceder a las propiedades de un objeto.

`` : Uno de los atributos que podemos usar es `th:text` con diferentes etiquetas HTML, para poder mostrar, por ejemplo, el nombre del autor de un libro. También podemos navegar entre objetos.

`<td th:text="{myObject.myMethod()}">` :También podemos llamar a métodos definidos en nuestros propios objetos, lo vamos a poder hacer desde las plantillas.

ATRIBUTOS BÁSICOS

Los atributos básicos más conocidos con los que nos podemos encontrar son:

TH:TEXT

th:text: Permite reemplazar el texto de la etiqueta por el valor de la expresión que le demos.

```
<p th:text="{saludo}">saludo</p>
```

TH:EACH

th:each: Nos va a permitir repetir tantas veces como se indique o iterar sobre los elementos de una colección.

```
<li th:each="libro : {libros}"
th:text="{libro.titulo}">El Quijote</li>
```

La plantilla recibe la colección libros, y crea una variable llamada libro que va a ser en algún momento todos los elementos de nuestra colección, al igual que el for each de Java. Después, usamos la variable libro para acceder solo al título y al th:text para mostrar en el HTML el título en cuestión.

TH:VALUE

En la guía de HTML cuando estudiamos los input, aprendimos que a los input en algunos casos puede resultarnos interesante asignar un valor definido al campo en cuestión.

Este valor inicial del campo podía ser expresado mediante el atributo **value**. Thymeleaf nos ofrece hacer esto pero de manera dinámica, con el atributo **th:value**, para darle a los input valores iniciales distintos, dependiendo de que envíe el controlador.

```
<input type="text" name="instituto" th:value="{nombreInstituto}">
```

Esto nos haría pensar que es el mismo atributo que th:text, pero no, ya que th:text nos permite darle un valor a cualquier etiqueta de texto, mientras que th:value solo sirve para las etiquetas input.

TH:IF

A veces, vamos a necesitar que un **fragmento de nuestra plantilla** solo aparezca cuando se cumple una **determinada condición**.

Los atributos **th:if** y **th:unless**, nos permiten mostrar un elemento de HTML dependiendo del resultado de una determinada condición.

```
<td>
    <span th:if="${profesor.sexo == 'F'}">Femenino</span>
    <span th:unless="${profesor.sexo == 'M'}">Masculino</span>
</td>
```

Si el valor de profesor.sexo es igual a F, entonces el elemento span va a mostrar la palabra **Femenino**.

En cambio, si el valor es M, entonces el elemento muestra la palabra **Masculino**.

TH:Href

Sirve para construir URLs que podemos utilizar en cualquier tipo de contexto.

Podríamos utilizarlas para hacer enlaces para URLs que sean absolutas o relativas al propio contexto de la aplicación, al servidor, al documento, etc.

Estos son unos ejemplos:

```
<a th:href="@{order/details}">...</a>
<a th:href="@{.../documents/report}">...</a>
<a th:href="@{http://www.micom.es/index}">...</a>
```

MODEL MAP

Ya vimos como, gracias a Thymelaf podemos recibir del controlador una variable y mostrarla en nuestro HTML, pero, ¿como hacemos para pasar esa variable desde nuestro controlador a nuestro HTML?

Para resolver este problema vamos a utilizar el objeto **ModelMap**, este objeto es parte del paquete **org.springframework.ui.ModelMap**.

El objeto ModelMap tiene el método **addAttribute** que nos permite enviar variables nuestro HTML, la ventaja del objeto ModelMap, es que también nos permite enviar Colecciones a nuestro HTML.

El método **addAttribute(String variable, Objeto nombreObjeto)**, recibe dos argumentos, una es variable de tipo String, que va ser el identificador que le vamos a poner al objeto o colección, que es el identificador con el que va a viajar al HTML y que va a tener que coincidir con la variable de Thymeleaf, y la otra es el objeto de Java que queremos mandar al HTML .

Pongamos un ejemplo, supongamos que tenemos la siguiente etiqueta en HTML con Thymeleaf, en el **th:text**, decimos que va a recibir una variable llamada nombre:

```
<p>Hola<span th:text="${nombre}"></p>
```

En el controlador tendremos el siguiente método:

```
@Controller
public class ControladorHola {
    @GetMapping("/")
    String home(ModelMap model) {
        String nombre = "Fernando"
        model.addAttribute("nombre", nombre)
        return "paginaHTML";
    }
}
```

Como podemos ver en el controlador, recibimos como argumento un ModelMap, esto es para que podemos recibir cualquier modelo que venga de la petición y para que podamos enviar los modelos que queramos como respuesta de x peticiones.

Usando model.addAttribute(), pasamos dos cosas, uno el identificador con el que el objeto va a viajar a la vista, que es **"nombre"**, recordemos que tiene que coincidir con la variable de Thymeleaf, y dos pasamos el objeto en sí.

Entonces cuando se llame a este controlador en localhost:8080/, se enviará el modelo "nombre", lo recibirá la vista, gracias a Thymeleaf y mostrará el nombre Fernando.

PREGUNTAS DE APRENDIZAJE

- 1) ¿Qué significa el acrónimo HTTP?
 - a) HyperText Translation Protocol
 - b) HyperText Transport Protocol
 - c) HyperText Transfer Protocol
 - d) Ninguna de las anteriores

- 2) ¿Cual de estos SI es un método de petición?
 - a) REMOVE
 - b) UPDATE
 - c) REFRESH
 - d) GET

- 3) ¿En una URL, después del signo de interrogación van los?
 - a) Parámetros
 - b) Etiquetas
 - c) Rutas
 - d) Ninguna de las anteriores

- 4) Usualmente usamos el método POST en, ¿qué etiqueta HTML?
 - a)
 - b) <form>
 - c) <table>
 - d) <input>

- 5) ¿Qué indica el código HTTP 200?
 - a) OK, petición procesada correctamente.
 - b) Indica al browser que visite otra dirección.
 - c) Acceso prohibido, por falta de permisos.
 - d) No encontrado, cuando el documento no existe.

- 6) ¿Qué indica el código HTTP 500?
 - a) Indica al browser que visite otra dirección.
 - b) Acceso prohibido, por falta de permisos.
 - c) No encontrado, cuando el documento no existe.
 - d) Error interno en el servidor.

7) Maven:

- a) Es una herramienta para formatear código
- b) Es una herramienta para automatizar tareas
- c) Es un IDE para construir aplicaciones web
- d) Ninguna de las anteriores

8) ¿Cuál es el nombre del archivo donde se encuentran las dependencias de Maven?

- a) Application-context.xml
- b) Dependencies.xml
- c) Maven.xml
- d) Pom.xml

9) Spring Framework es:

- a) Un framework para el desarrollo de aplicaciones PHP
- b) Un framework para el desarrollo de aplicaciones .net
- c) Un framework para el desarrollo de aplicaciones Java
- d) Todas las anteriores

10) ¿Spring Framework utiliza qué patrón de diseño?

- a) Patrón DAO
- b) Patrón DTO
- c) Patrón MVC
- d) Patrón EVC

11) ¿Qué significa el acrónimo MVC?

- a) Modelo Vistas Controlar
- b) Modelo Vistas Controlador
- c) Mostrar Vistas Controlador
- d) Modelo Ver Controlador

12) ¿Cuál de estas NO es una anotación de Spring?

- a) @Component
- b) @Repository
- c) @Table
- d) @Autowired

13) ¿Qué anotación usamos para marcar una clase como Controlador?

- a) @Service
- b) @Controller
- c) @Repository
- d) @Component

14) ¿Qué anotación usamos para marcar una clase como Servicio?

- a) @Service
- b) @Controller
- c) @Repository
- d) @Component

15) ¿Qué anotación usamos para marcar una clase como Repositorio?

- a) @Service
- b) @Controller
- c) @Repository
- d) @Component

16) ¿Qué anotación usamos para inyectar una clase?

- a) @Qualifier
- b) @Autowired
- c) @Repository
- d) @Component

17) ¿Qué anotación usamos para marcar que un método de una clase controlador, va a recibir peticiones GET?

- a) @PostMapping
- b) @GetMapping
- c) @MappingGet
- d) @GetPetition

18) La etiqueta RequestParam nos permite obtener datos del URL, pero, ¿de que parte?

- a) Parámetros
- b) Etiqueta
- c) Ruta
- d) Ninguna de las anteriores

19) ¿Qué es Thymeleaf?

- a) Un motor de vistas
- b) Un lenguaje de programación basado en Java
- c) Un motor de plantillas
- d) Ninguna de las anteriores

20) ¿Cuál de estos NO es un atributo de Thymeleaf?

- a) th:text
- b) th:each
- c) th:if
- d) th:else

EJERCICIOS DE APRENDIZAJE

Para la realización de este trabajo práctico **se recomienda ver todos los videos de Spring**, de esta manera sabemos todos lo que tenemos que hacer, antes de empezar a hacerlo. Además, podrán encontrar en el siguiente link un **GitHub con ejemplos** para descargar de Spring: [GitHubSpring](#)

VER VIDEOS:

- A) [Fundamentos Web](#)
- B) [Configuración Spring](#)

1. Sistema de Guardado de una Librería Web

El objetivo de este ejercicio consiste en el desarrollo de un sistema web de guardado de libros en JAVA utilizando una base de datos MySQL, JPA Repository para persistir objetos y Spring Boot como framework de desarrollo web.

Creación de la Base de Datos MySQL

Crear el esquema sobre el cual operará el sistema de reservas de libros. Para esto, en el IDE de base de datos que esté utilizando (por ejemplo, Workbench) ejecute la sentencia:

```
CREATE DATABASE libreria;
```

Paquetes del Proyecto

Los paquetes que se deben utilizar para el proyecto se deben estructurar de la siguiente manera:

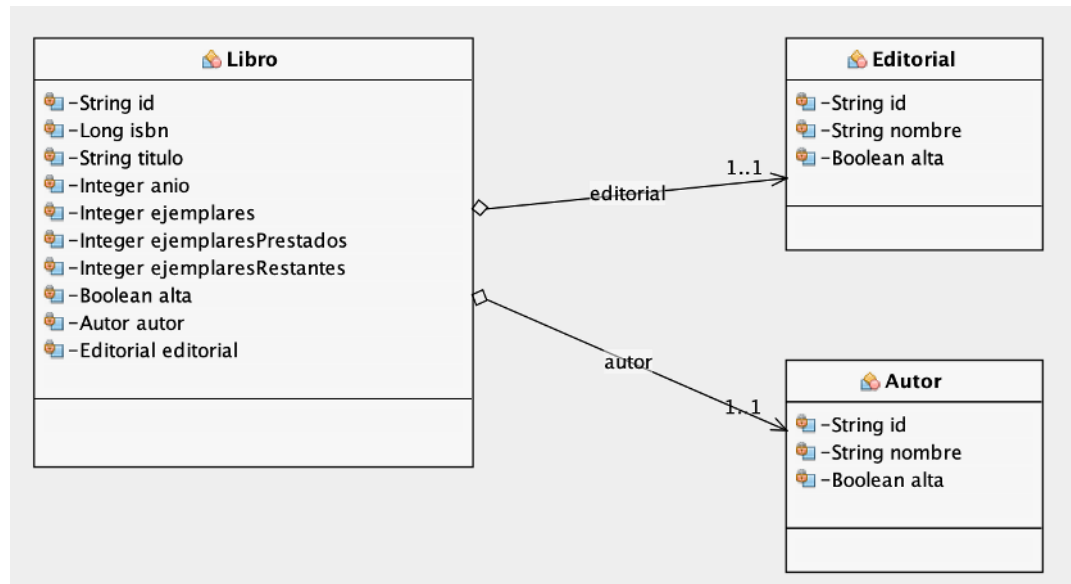
- **vistas:** en este paquete se almacenarán aquellas clases que se utilizarán como vistas con el usuario.
- **controladores:** en este paquete se almacenarán aquellas clases que se utilizarán para mediar entre la vista con el usuario y las capas inferiores.
- **servicios:** en este paquete se almacenarán aquellas clases que llevarán adelante lógica del negocio.
- **repositorios:** en este paquete se crearán los repositorios que servirán como interfaces entre el modelo de objetos y la base de datos relacional.
- **entidades:** en este paquete se almacenarán aquellas clases que es necesario persistir en la base de datos.



Capa de Datos

Entidades y Repositorios

Crear el siguiente modelo de entidades y los repositorios correspondientes para este modelo:



Spring utiliza una anotación para identificar aquellas clases que serán entidades y repositorios. Todas las entidades deben estar marcadas con la anotación `@Entity` y los repositorios con la anotación `@Repository`, los repositorios heredarán la interfaz `JpaRepository`, que nos dará todos los métodos para persistir, editar, eliminar, etc.

Entidad Libro

La entidad libro modela los libros que están disponibles en la biblioteca para ser prestados. En esta entidad, el atributo "ejemplares" contiene la cantidad total de ejemplares de ese libro, mientras que el atributo "prestados" contiene cuántos de esos ejemplares se encuentran prestados en este momento y el atributo "restantes" tiene cuando libros nos quedan para prestar. El repositorio que persiste a esta entidad (LibroRepositorio) debe contener los métodos necesarios para guardar/actualizar libros en la base de datos, realizar consultas o dar de baja según corresponda.

Entidad Autor

La entidad autor modela los autores de libros. El repositorio que persiste a esta entidad debe contener todos los métodos necesarios para guardar en la base de datos, realizar consultas y eliminar o dar de baja según corresponda. El repositorio que persiste a esta entidad (AutorRepositorio) debe contener los métodos necesarios para guardar/actualizar un cliente en la base de datos, realizar consultas o dar de baja según corresponda.

Entidad Editorial

La entidad editorial modela las editoriales que publican libros. El repositorio que persiste a esta entidad (EditorialRepositorio) debe contener los métodos necesarios para guardar/actualizar una editorial en la base de datos, realizar consultas o dar de baja según corresponda.

VER VIDEOS:

- A. Capa de Servicios
- B. Modelo Vista Controlador

Capa de Servicios

Spring utiliza una anotación para identificar aquellas clases que serán servicios. Todos los servicios deben estar marcados con la anotación @Service.

LibroServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar libros (consulta, creación, modificación y dar de baja).

AutorServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar autores (consulta, creación, modificación y dar de baja).

EditorialServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar editoriales (consulta, creación, modificación y dar de baja).

Capa de Comunicación

Spring utiliza una anotación para identificar aquellas clases que serán controladores. Todos los controladores deben estar marcadas con la anotación @Controller. Algunos de los controladores a desarrollar son los siguientes.

LibroController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista del usuario diseñada para la gestión de libros (guardar/modificar libro, listar libros, dar de baja).

AutorController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista que gestiona los autores (guardar/actualizar, listar autores, dar de baja).

EditorialController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con las vistas que gestiona editoriales (guardar/modificar, listar editoriales, dar de baja).

Capa de Vistas

Esta capa tiene la responsabilidad de llevar adelante las funcionalidades necesarias para interactuar con el usuario. Las vistas para este proyecto tienen que estar desarrolladas en HTML5 y se debe utilizar la biblioteca Thymeleaf y CSS para implementar las plantillas. Además, se debe utilizar el framework de Bootstrap para los componentes.

Se deben diseñar y crear todas las vistas web necesarias para llevar a cabo las siguientes funcionalidades:

- Administrar Autores: cargar datos de un autor, modificar datos, listar autores y dar de baja
- Administrar Editoriales: cargar los datos de una editorial, modificar los datos, listar editoriales y dar de baja.
- Administrar Libros: cargar datos de un nuevo libro, modificar datos, listar libros, y dar de baja.

A continuación, se muestran algunos ejemplos para el módulo de Administración de Autores.

a) Listar Autores

Nombre	Editar Autor	Eliminar Autor
Jorde Luis Borges	Editar	Dar de Baja
Julio Cortazar	Editar	Dar de Baja
Alfonsina Storni	Editar	Dar de Baja
Virginia Woolf	Editar	Dar de Baja
Isabel Allende	Editar	Dar de Baja

b) Cargar Autor

Administración de Autores

Utilice este módulo para administrar la base de datos de autores de libro.

Nombre:

Apellido:

[Volver](#)[Guardar](#)

EJERCICIOS EXTRAS

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, puedes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por último, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

1. Sistema de Reservas: Librería Web

El objetivo de este ejercicio consiste en, utilizando las clases del ejercicio/proyecto anterior, el desarrollo de un sistema web de reserva de libros en JAVA.

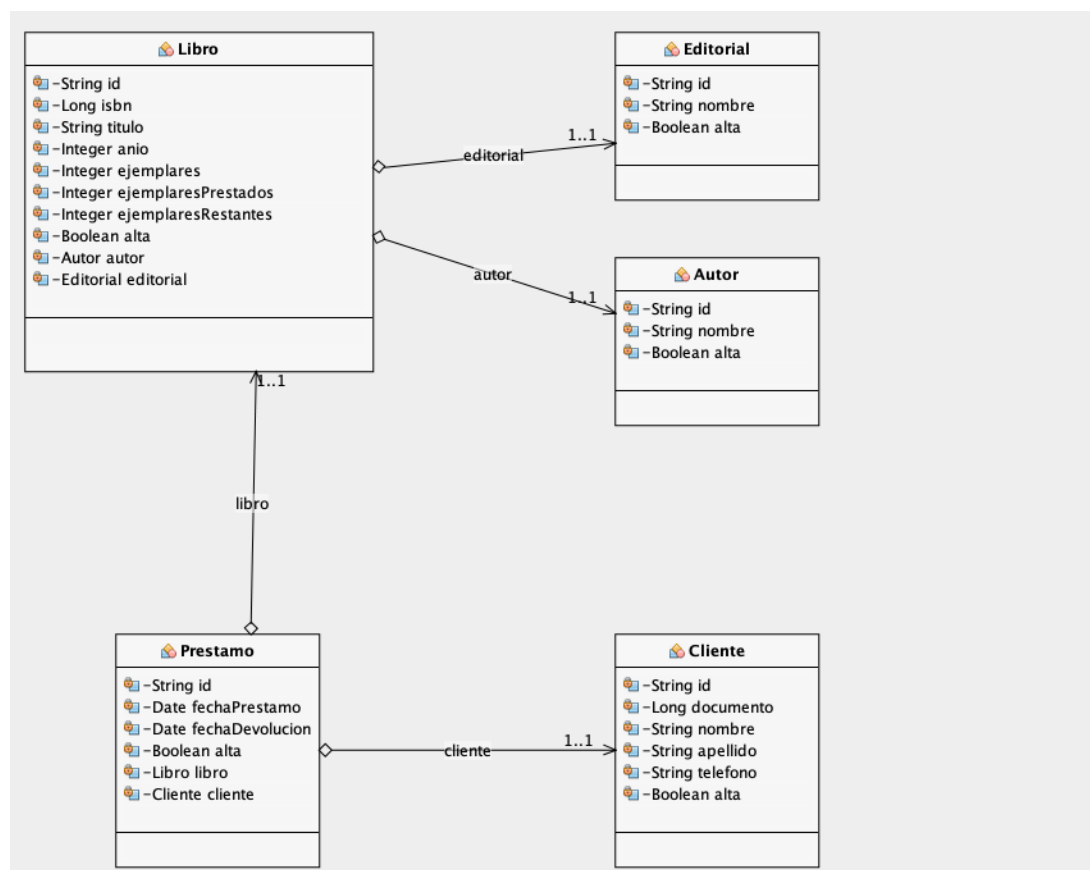
Creación de la Base de Datos MySQL

Usaremos la misma base de datos y se van a crear las tablas que nos faltan.

Capa de Datos

Entidades y Repositorios

Deberemos sumar las Entidades Cliente y Préstamo para que nos quede el siguiente modelo:



Spring utiliza una anotación para identificar aquellas clases que serán entidades y repositorios. Todas las entidades deben estar marcadas con la anotación `@Entity` y los repositorios con la anotación `@Repository`.

Entidad Cliente

La entidad cliente modela los clientes (a quienes se les presta libros) de la biblioteca. Se almacenan los datos personales y de contacto de ese cliente. El repositorio que persiste a esta entidad (`ClienteRepositorio`) debe contener los métodos necesarios para guardar/actualizar un cliente en la base de datos, realizar consultas o dar de baja según corresponda.

Entidad Préstamo

La entidad préstamo modela los datos de un préstamo de un libro. Esta entidad registra la fecha en la que se efectuó el préstamo y la fecha en la que fue devuelto el libro, al devolver el libro este préstamo queda dado de baja. Esta entidad también registra el libro que se llevaron en dicho préstamo y quien fue el cliente al cual se lo prestaron.

El repositorio que persiste a esta entidad (`PréstamoRepositorio`) debe contener los métodos necesarios para registrar un préstamo en la base de datos, realizar consultas y realizar devoluciones, etc.

Capa de Servicios

Spring utiliza una anotación para identificar aquellas clases que serán servicios. Todos los servicios deben estar marcados con la anotación `@Service`. Los servicios que faltan en nuestro proyecto son:

ClienteServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar clientes (consulta, creación, modificación y dar de baja).

PréstamoServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar préstamos (consulta, préstamo, modificación y dar de baja).

Capa de Comunicación

Spring utiliza una anotación para identificar aquellas clases que serán controladores. Todos los controladores deben estar marcados con la anotación `@Controller`. Algunos de los controladores a desarrollar son los siguientes. Los controladores que faltan en nuestro proyecto son:

ClienteController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista que gestiona clientes (guardar/modificar, listar clientes, eliminación).

PrestamoController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vistas o portal para gestionar préstamos (guardar/modificar, listar préstamos realizados, devolución, eliminación).

Capa de Vistas

En la capa de vistas vamos a tener que diseñar y crear todas las vistas web necesarias para llevar a cabo las siguientes funcionalidades:

- **Administrar clientes:** cargar datos de los clientes que desean pedir prestado un libro, modificar datos de los clientes, realizar listados y eliminar clientes.
- **Realizar Préstamos:** cargar los datos de un préstamo. Tener en cuenta que para realizar un préstamo se deben incluir los libros a prestar y el cliente asociado. Modificar un préstamo (por ejemplo, renovar la fecha de devolución), listar préstamos realizados, etc.

2. Sistema de Estancias en el Extranjero Web

El objetivo de este ejercicio consiste en el desarrollo de un sistema web para una pequeña empresa que se dedica a organizar estancias en el extranjero dentro de una familia. El sistema debe registrar la reserva de casas por parte de los clientes que desean realizar alguna estancia. El objetivo es el desarrollo de un sistema web de reservas de casas para realizar estancias en el exterior, utilizando el lenguaje JAVA, una base de datos MySQL, el framework de persistencia JPA y Spring Boot como framework de desarrollo web.

Creación de la Base de Datos MySQL

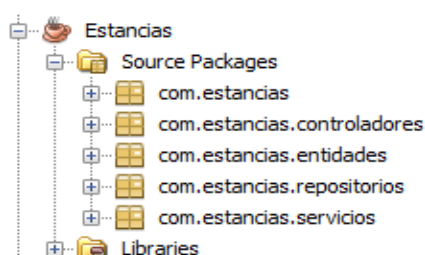
Crear el esquema sobre el cual operará el sistema de reservas de casas. Para esto, en el IDE de base de datos que esté utilizando (por ejemplo, Workbench) se debe ejecutar la sentencia:

```
CREATE DATABASE estancias;
```

De esta manera se creará una base de datos vacía llamada estancias.

Paquetes del Proyecto

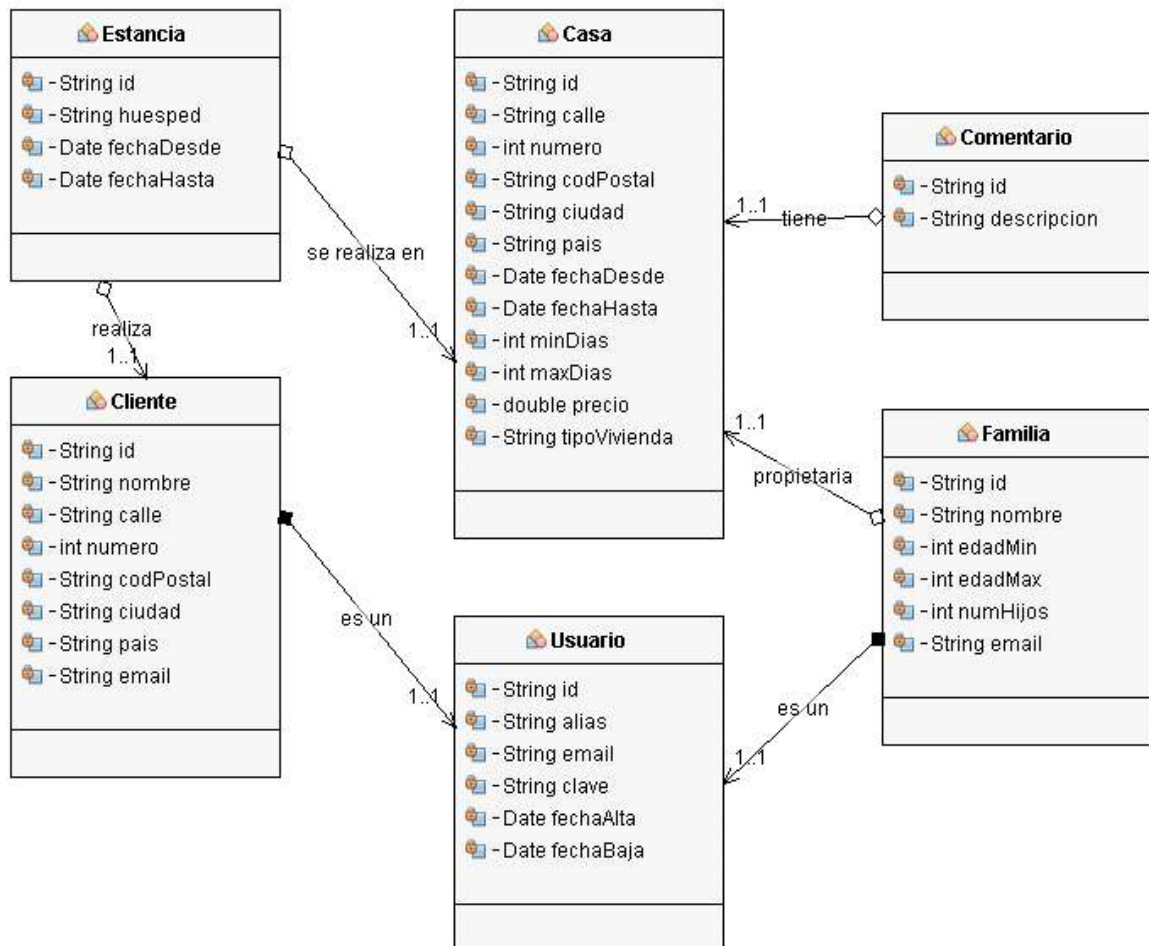
En este proyecto se debe utilizar la misma estructura de capas que en el ejercicio anterior.



Capa de Datos

Entidades y Repositorios

Crear el siguiente modelo de entidades y agregar los repositorios correspondientes. Todas las entidades deben estar marcadas con la anotación `@Entity` y los repositorios con la anotación `@Repository`.



Entidad Usuario

La entidad usuario modela los datos de un usuario que accede al sistema para registrarse como familia y ofrecer una habitación de su casa para estancias, o bien, como un cliente que necesita realizar una reserva. De cada usuario se debe registrar el nombre de usuario (alias), el correo electrónico, el password y la fecha de alta. El repositorio que persiste a esta entidad (UsuarioRepositorio) debe contener los métodos necesarios para registrar el usuario en la base de datos, realizar consultas y eliminar.

Entidad Familia

La entidad familia modela las familias que habitan en diferentes países y que ofrecen alguna de las habitaciones de su hogar para acoger a algún chico (por un módico precio). De cada una de estas familias se conoce el nombre, la edad mínima y máxima de sus hijos, número de hijos y correo electrónico. El repositorio que persiste a esta entidad (FamiliaRepositorio) debe contener los métodos necesarios para guardar/actualizar los datos de las familias en la base de datos, realizar consultas y eliminar o dar de baja según corresponda.

Entidad Casa

La entidad casa modela los datos de las casas donde las familias ofrecen alguna habitación. De cada una de las casas se almacena la dirección (calle, número, código postal, ciudad y país), el periodo de disponibilidad de la casa (fecha_desde, fecha_hasta), la cantidad de días mínimo de estancia y la cantidad máxima de días, el precio de la habitación por día y el tipo de vivienda. El repositorio que persiste a esta entidad (CasaRepositorio) debe contener los métodos necesarios para guardar/actualizar los datos de una vivienda, realizar consultas y eliminar.

Entidad Cliente

La entidad cliente modela información de los clientes que desean mandar a sus hijos a alguna de las casas de las familias. Esta entidad es modelada por el nombre del cliente, dirección (calle, número, código postal, ciudad y país) y su correo electrónico. El repositorio que persiste a esta entidad (ClienteRepositorio) debe contener los métodos necesarios para guardar/actualizar los datos de un cliente, realizar consultas y eliminar.

Entidad Reserva

La entidad reserva modela los datos de las reservas y estancias realizadas por alguno de los clientes. Cada estancia o reserva la realiza un cliente, y además, el cliente puede reservar varias habitaciones al mismo tiempo (por ejemplo para varios de sus hijos), para un periodo determinado (fecha_llegada, fecha_salida). El repositorio que persiste a esta entidad (ReservaRepositorio) debe contener los métodos necesarios para realizar una reserva, actualizar los datos (por ejemplo, fecha de la reserva), realizar consultas de las reservas realizadas para una determinada vivienda y eliminar reserva.

Entidad Comentario

La entidad comentario permite almacenar información brindada por los clientes sobre las casas en las que ya han estado. El repositorio que persiste a esta entidad (ComentarioRepositorio) debe contener los métodos necesarios para guardar los comentarios que realizan los clientes sobre una determinada una vivienda.

Capa de Servicios

Utiliza la anotación @Service para identificar aquellas clases que serán servicios. Todos los servicios deben estar marcados con esta anotación.

UsuarioServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar usuarios (alta de usuario, consultas, y baja o eliminación).

FamiliaServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar familias (creación, consulta, modificación y eliminación).

CasaServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar las casas (creación, consulta, modificación y eliminación).

ClienteServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar clientes (creación, consulta, modificación y eliminación).

ReservaServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para realizar las reservas de viviendas (reservar, consultar reservas realizadas, modificación y eliminación).

Capa de Comunicación

Spring utiliza una anotación para identificar aquellas clases que serán controladores. Todos los controladores deben estar marcadas con la anotación `@Controller`. Algunos de los controladores a desarrollar son los siguientes.

UsuarioController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista del usuario diseñada para la gestión de usuarios (dar de alta un usuario, cambiar clave, listar usuarios registrados, dar de baja un usuario).

FamiliaController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista del usuario diseñada para la gestión de familias (guardar/modificar datos de la familia, listar familias, eliminar).

CasaController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista del usuario diseñada para la gestión de viviendas (guardar/modificar datos de la casa, listar viviendas, eliminar).

ClienteController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista del usuario diseñada para la gestión de clientes (guardar/modificar, listar clientes, eliminar).

ReservaController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista o portal para gestionar reservas de estancias (guardar/modificar, listar estancias reservadas/realizadas, eliminación).

Capa de Vista

Esta capa tiene la responsabilidad de llevar adelante las funcionalidades necesarias para interactuar con el usuario. Las vistas para este proyecto tienen que estar desarrolladas en HTML5 y se debe utilizar la biblioteca Thymeleaf y CSS para implementar las plantillas.

Se deben diseñar y crear todas las vistas web necesarias para llevar a cabo las siguientes funcionalidades:

- Administrar usuarios: registrar nuevos usuarios en el sistema, cambiar clave, listar usuarios, dar de baja o eliminar.
- Administrar familias: cargar datos de una familia, modificar datos, consultar familias, eliminar familias que no ofrecen más sus viviendas para estancias. Las familias deben darse de alta una vez que hayan sido registradas como usuario.

- Administrar casas: cargar los datos de una vivienda y asociar a la familia correspondiente, modificar los datos (por ejemplo, fechas en las cuales se encuentra disponible), listar casas (país, el periodo de disponibilidad, cantidad de días mínima y máxima de estancia, el precio de la habitación por día, el tipo de vivienda y el nombre del propietario). Se debe eliminar los datos de una casa cada vez que se da de baja la familia propietaria.
- Administrar clientes: cargar datos de los clientes que desean reservar una habitación para realizar una estancia, modificar datos de los clientes, realizar consultas y eliminar clientes. Al igual que las familias, un cliente puede darse de alta una vez que se haya creado el usuario correspondiente.
- Realizar Reservas: El portal principal debería permitir que una persona que ingresa a la web pueda consultar las viviendas que se encuentran disponibles para realizar estancias (validando fechas disponibles). Una vez que el usuario encuentra una vivienda que se adecua a sus preferencias y quiere realizar una reserva, recién en ese momento se solicita que se registre como usuario y luego se procede a realizar la reserva.
 - o Cuando el usuario se registra se le debe dar la opción de elegir si se quiere registrar como familia que ofrece una vivienda o como cliente. Dependiendo de la opción elegida se pide que complete los datos correspondientes (familia o cliente) para continuar con su registro.
 - o Si el usuario ya se encuentra registrado entonces debe realizar el login para poder continuar.
 - o Se debe tener en cuenta que para realizar una reserva la vivienda no debe estar ya reservada por otro cliente.
 - o Se debe permitir que un cliente modifique su reserva, por ejemplo: cambiar las fechas, o la elimine en caso de no poder realizarla.
 - o Se deben poder listar las reservas realizadas por parte de los clientes.

BIBLIOGRAFÍA

- <https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/protocolo-http/>
- <https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/http-request/>
- <https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/una-mirada-a-los-codigos-de-estado-http-mas-comunes/>
- <https://edytapukocz.com/url-partes-ejemplos-facil/>
- <https://howtodoinjava.com/maven/maven-dependency-management/>
- http://chuwiki.chuidiang.org/index.php?title=Dependencias_con_maven
- <https://www.arquitecturajava.com/que-es-spring-framework/>
- <https://programandointentandolo.com/2013/05/inyeccion-de-dependencias-en-spring.html>
- <https://proitcsolution.com.ve/inyeccion-de-dependencias-spring/>
- <https://www.java67.com/2019/04/top-10-spring-mvc-and-rest-annotations-examples-java.html>
- <https://www.danvega.dev/blog/2017/03/27/spring-stereotype-annotations/>
- <https://www.arquitecturajava.com/spring-stereotypes/>
- <https://www.arquitecturajava.com/que-es-spring-boot/>
- <https://openwebinars.net/blog/que-es-thymeleaf/>
-