SERVICIO WEB API REST SOBRE FRAMEWORK SPRING

NOMBRE: ROGER CAMACHO VALLEJOS

MATERIA: PROGRAMACIONIII

DOCENTE: ING. WILLIAM

CBBA-BOLIVIA

A. Servicios Web

El consorcio W3C (World Wide Web Consortium) define un servicio web como un sistema software diseñado para soportar la interacción máquina-a-máquina, a través de una red, de forma interoperable. Cuenta con una interfaz descrita en un formato procesable por un equipo informático (específicamente en WSDL), a través de la que es posible interactuar con el mismo mediante el intercambio de mensajes SOAP, típicamente transmitidos usando serialización XML sobre HTTP conjuntamente con otros estándares web.

A.1 Tipos de Servicios Web

Remote Procedure Calls (RPC, Llamadas a Procedimientos Remotos): están basados en RPC y presentan una interfaz de llamada a procedimientos remotos y funciones distribuidas. Es una comunicación nodo a nodo entre cliente y servidor, donde el cliente solicita que se ejecute cierto procedimiento o función y el servidor envía la respuesta. Las primeras referencias de servicios web estaban basadas en esta versión, sin embargo, las numerosas problemáticas por el acoplamiento entre los sistemas y el nacimiento de nuevas tecnologías, han quedado a éste casi olvidado.

Service Oriented Architecture (SOA, Arquitectura Orientada a Servicios): es una arquitectura de aplicación en la cual todas las funciones están definidas como servicios independientes con interfaces invocables que pueden ser llamados en secuencias bien definidas para formar los procesos de negocio. Al contrario que los Servicios Web basados en RPC, este estilo es débilmente acoplado, lo cual es preferible ya que se centra en los servicios proporcionados por el documento WSDL, más que en los detalles de implementación con los distintos sistemas. El más relevante sería SOAP (Simple Object Access Protocol).

REST (Representational State Transfer): es un conjunto de principios de arquitectura para describir cualquier interfaz entre sistemas que utilice directamente HTTP para obtener datos o indicar la ejecución de operaciones sobre los datos, en cualquier formato (XML, JSON, etc.) y sin las abstracciones adicionales de los protocolos basados en patrones de intercambio de mensajes, como por ejemplo SOAP. En los siguientes apartados se va a desarrollar el concepto de los servicios web más usados en la actualidad, SOAP y REST.

A.2 Características de SOAP

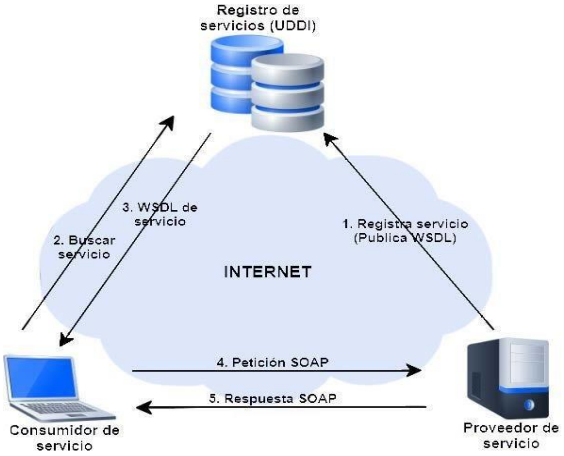
SOAP es un protocolo sobre el que se establece el intercambio de información con el servicio web, es decir, ofrece las directrices básicas a través de las cuales un servicio web se puede construir. Este protocolo está basado en XML y formado por tres partes:

- Envelope: el cual define qué hay en el mensaje y cómo procesarlo.
- Conjunto de reglas de codificación para expresar instancias de tipos de datos.
- La convención para representar llamadas a procedimientos y respuestas.

En la arquitectura de un servicio web que implementa este protocolo también se pueden diferenciar tres partes:

- Proveedor del servicio.
- Solicitante.
- Publicador.

El proveedor de servicios envía al publicador del servicio un fichero WSDL con la definición del servicio web. El solicitante interactúa con el publicador, descubre quién es el proveedor (protocolo WSDL) y contacta con él (protocolo SOAP). El proveedor valida la petición de servicio y envía el dato estructurado en formato XML utilizando el protocolo SOAP. El fichero XML es validado de nuevo por el que pide el servicio utilizando un fichero XSD. Dentro de esta arquitectura tendría cabida definir dos estándares importantes en esta estructuración: WSDL (Web Services Description Language): es el lenguaje de la interfaz pública para los servicios web. Es una descripción basada en XML de los requisitos funcionales necesarios para establecer una comunicación con los servicios web. UDDI (Universal Description, Discovery and Integration): protocolo para publicar la información de los servicios web. Permite comprobar qué servicios web están disponibles.



A.3 Características de REST

REST (Representational State Transfer) es un estilo de arquitectura de software para sistemas hipermedias distribuidos tales como la Web. El término fue introducido en la tesis doctoral de Roy Fielding en 2000, quien es uno de los principales autores de la especificación de HTTP.

REST no es más que una colección de recursos definidos y diseccionados. El término a menudo es utilizado para describir a cualquier interfaz que transmite datos específicos de un domino sobre HTTP sin una capa adicional como hace SOAP. Cabe destacar que es posible diseñar un sistema de acuerdo con la arquitectura 24 Servicio Web API REST sobre el Framework Spring, Hibernate, JSON Web Token y BBDD Oracle 24 propuesta por Fielding sin utilizar HTTP o sin interactuar con la Web. Así como también es posible diseñar una simple interfaz XML+HTTP que no sigue los principios REST, y en cambio seguir un modelo RPC. REST es un estilo de arquitectura basado en estándares como son HTTP, URL, la representación de los recursos: XML/HTML/GIF/JPEG/etc. y los tipos MIME: text/xml, text/html, … La motivación de REST es la de capturar las características de la Web que la han hecho tan exitosa. REST se ha centrado en explotar el éxito de la Web, que no es más que el uso de formatos de mensaje extensibles, estándares y un esquema de direccionamiento global. En particular, el concepto central de la Web es un espacio de URIs unificado. Las URIs

identifican recursos, los cuales son objetos conceptuales. La representación de tales objetos se distribuye por medio de mensajes a través de la Web. Este sistema es extremadamente desacoplado.

Las características principales de un modelo REST serían las siguientes:

- Escalabilidad. La variedad de sistemas y de clientes crece continuamente, pero cualquiera de ellos puede acceder a través de la Web. Gracias al protocolo HTTP, pueden interactuar con cualquier servidor HTTP sin ninguna configuración especial.

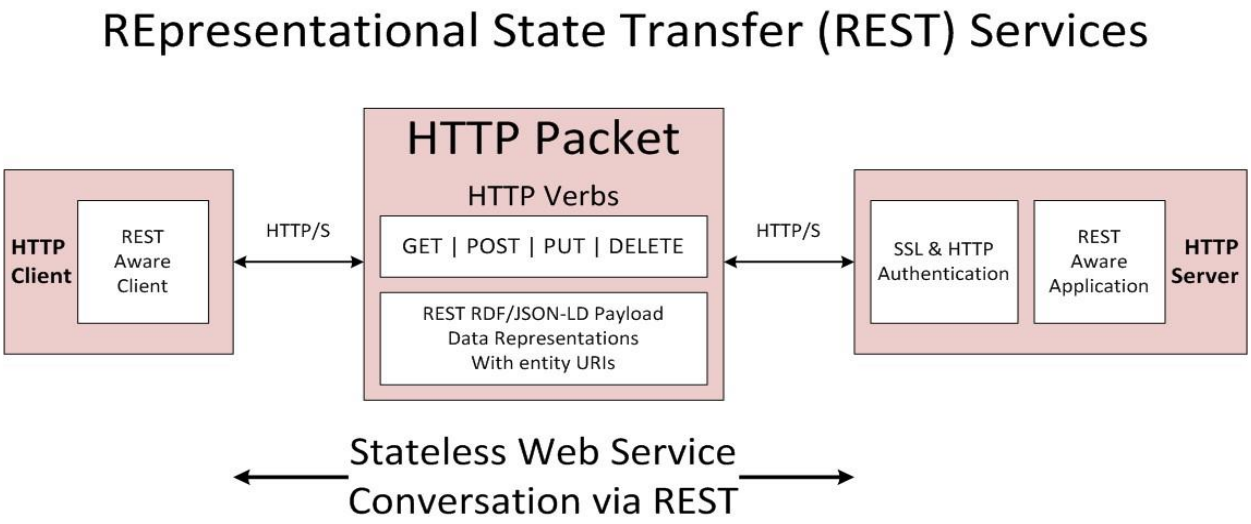
- Independencia. Los clientes y servidores pueden tener puestas en funcionamiento complejas. Diseñar un protocolo que permita este tipo de características resulta muy complicado. HTTP permite la extensibilidad mediante el uso de las cabeceras, a través de las URIs.

- Compatibilidad. En ocasiones existen componentes intermedios que dificultan la comunicación entre sistemas, como pueden ser los firewalls. Las organizaciones protegen sus redes mediante firewalls y cierran casi todos los puertos TCP salvo el 80, el que usan los navegadores web. REST al utilizar HTTP sobre Transmission Control Protocol (TCP) en el puerto de red 80 no resulta bloqueado. Es importante señalar que los servicios web se pueden utilizar sobre cualquier protocolo, sin embargo, TCP es el más común.

- Identificación de recursos. REST utiliza una sintaxis universal como es el uso de URIs. HTTP es un protocolo centrado en URIs, donde los recursos son los objetos lógicos a los que se le envían mensajes.

- Protocolo cliente/servidor sin estado. Cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión.

- Operaciones bien definidas. HTTP en sí define un conjunto pequeño de operaciones, las más importantes son POST, GET, PUT y DELETE.



A.4 Comparativa entre REST y SOAP

De manera general se puede resaltar de SOAP que es fuertemente acoplado, pudiendo ser testado y depurado antes de poner en marcha la aplicación. Mientras que de REST se puede declarar que su potencial recae en su escalabilidad en todo tipo de sistemas y su escaso consumo de recursos debido al limitado número de operaciones y al esquema de direccionamiento unificado.

REST	SOAP
• Las operaciones se definen en los mensajes. • Una dirección única para cada instancia del proceso. • Cada objeto soporta las operaciones estándares definidas. • Componentes débilmente acoplados. • Bajo consumo de recursos. • Las instancias del proceso son creadas explícitamente. • El cliente no necesita información de enrutamiento a partir de la URI inicial. • Los clientes pueden tener una interfaz "listener" (escuchadora) genérica para las notificaciones. • Generalmente fácil de construir y adoptar.	• Las operaciones son definidas como puertos WSDL. • Dirección única para todas las operaciones. • Múltiples instancias del proceso comparten la misma operación. • Componentes fuertemente acoplados. • Fácil (generalmente) de utilizar. • La depuración es posible. • Las operaciones complejas pueden ser escondidas detrás de una fachada. • Envolver APIs existentes es sencillo • Incrementa la privacidad. • Herramientas de desarrollo.

B. Maven

Maven es una herramienta para la gestión y construcción de proyectos java, que se basa en el concepto POM (Proyect Object Model). Con Maven se pueden generar arquetipos, gestionar librerías, compilar, empaquetar, generar documentación…

Antes de la llegada de Maven, un desarrollador dedicaba gran parte de tiempo en comprender la estructura y peculiaridades de un proyecto, analizar que librerías utilizaba el código, donde incluirlas, que dependencias de compilación hacían falta en el mismo. Todo este trabajo de build es reemplazado con la llegada de Maven.

Maven se basa en patrones y estándares y trabaja con arquetipos, los cuales son configurables a través de su fichero protagonista, el pom.xml. Desde el cuál es posible configurar muchos aspectos de nuestro proyecto y del cual se hablará en profundidad más adelante.

B.1 Ciclo de vida

Maven define tres ciclos de build con etapas diferenciadas. Cada una de las etapas es una instrucción Maven, la cual se ejecutaría llamándola con *mvn <instruccion>*. Cuando se ejecuta dicha instrucción, Maven irá verificando cada una de las fases hasta llegar a la del comando utilizado.

Ciclo de vida por defecto

– **Validación (validate):** Validar que el proyecto es correcto.

– **Compilación (compile).**

– **Test (test):** Probar el código fuente usando un framework de pruebas unitarias.

– **Empaquetar (package):** Empaquetar el código compilado y transformarlo en algún formato tipo .jar o .war.

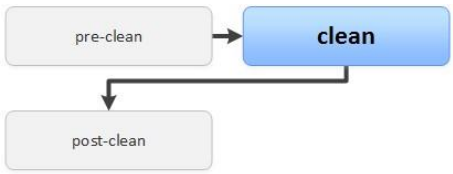
– **Pruebas de integración (integration-test):** Procesar y desplegar el código en algún entorno donde se puedan ejecutar las pruebas de integración.

– **Verificar** que el código empaquetado es válido y cumple los criterios de calidad (**verify**).

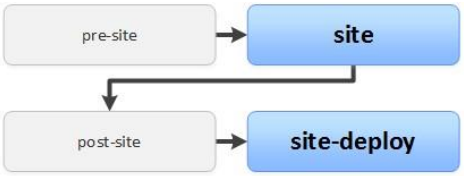
– **Instalar** el código empaquetado en el repositorio local de Maven, para usarlo como dependencia de otros proyectos (**install**).

Desplegar el código a un entorno (**deploy**).

Ciclo de limpieza clean: elimina todos los ficheros generados por construcciones anteriores.



Ciclo de documentación site: genera la página web de documentación del proyecto. **Site-deploy:** despliega la página de documentación en el servidor indicado.



B.2 POM

POM es un modelo de objeto para un proyecto, o como describe la documentación de Maven:

"*El fichero "pom.xml" es el núcleo de configuración de un proyecto Maven. Simplemente es un fichero de configuración, que contiene la mayoría de la información necesaria para construir (build) un proyecto al gusto del desarrollador*"

El pom al ser un fichero xml está compuesto por un conjunto de etiquetas que dan forma al proyecto. La primera de ellas es la etiqueta <**project** >, que engloba todo el pom y en la que se define la arquitectura del xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/mavenv4_0_0.xsd">
```

La primera etiqueta que forma el cuerpo del archivo es la etiqueta <**modelVersion**>. Esta indica la versión del pom, para que funcione con las versiones de Maven 2 y 3 debe tener el valor "4.0.0".

```
<modelVersion>4.0.0</modelVersion>
```

Las siguientes son las llamadas etiquetas básicas, entre las que se puede destacar:

- <**groupId**>: Suele ser el nombre o la web de la organización. Aunque no es necesario que tenga puntos de separación, se recomienda para actúe como paquete de Java.
- <**artifactId**>: El nombre del artefacto.
- <**name**>: El nombre del proyecto.
- <**version**>: La versión de del proyecto. Por defecto se suele usar "1.0.0". Aunque es posible usar la metodología de versiones que más nos guste.
- <**packaging**>: Con ella se indica cómo se desea que sea empaquetado el proyecto cuando Maven lo construya. Si por ejemplo se usa como valor "jar", se nos creará una biblioteca de Java. Otro ejemplo sería definirlo como "war", que sería un empaquetado web para desplegar en un servidor.

En el proyecto que se presenta se hace uso de los siguientes valores:

```
<groupId>org.es.wsus</groupId>
<artifactId>wsus</artifactId>
<name>wsus</name>
<packaging>war</packaging>
<version>1.0.0</version>
```

Siguiendo con la enumeración de las etiquetas, otra muy usada es <**properties**> que engloba un conjunto de otras etiquetas. Lo que englobe esta etiqueta <properties> será el conjunto de variables globales que podrán ser usadas en otras partes del fichero POM. Un ejemplo de gran

utilidad sería formar etiquetas con las versiones de las dependencias que se incorporarán al proyecto, siendo visibles de cara a futuras actualizaciones de las mismas.

```
<properties>
<java-version>1.8</java-version>
<org.springframework-version>4.3.14.RELEASE</org.springframework-version>
<hibernate.version>4.3.6.Final</hibernate.version>
<org.slf4j-version>1.7.5</org.slf4j-version>
<jackson.databind-version>2.5.2</jackson.databind-version>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

En el proyecto se definen etiquetas para las versiones que se utilizarán para Java, el Framework Spring, Hibernate, el log slj4j, Jackson-Databind y la codificación del proyecto.

Con la etiqueta <**repositories**> se indica cuáles serán los repositorios en los que Maven debe buscar las librerías para nuestro proyecto. En este proyecto se ha configurado de la siguiente forma:

```
<repositories>
  <repository>
    <id>central</id>
    <name>Maven Repository Switchboard</name>
    <layout>default</layout>
    <url>http://repo1.maven.org/maven2</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

La etiqueta <**dependencies**> engloba a todas sus etiquetas hijas <**dependency**> en las que se definen los artefactos que quiere que Maven descargue e incorpore a nuestro proyecto, siendo ésta una de las funcionalidades más potentes que nos aporta. A modo de ejemplo:

```
<dependencies>
      <dependency>
          <groupId>com.oracle</groupId>
          <artifactId>ojdbc6</artifactId>
          <version>11.2.0.3</version>
     </dependency>

<!-- Jackson -->
     <dependency>
          <groupId>com.fasterxml.jackson.core</groupId>
          <artifactId>jackson-databind</artifactId>
      <version>${jackson.databind-version}</version> </dependency>
<dependencies>
```

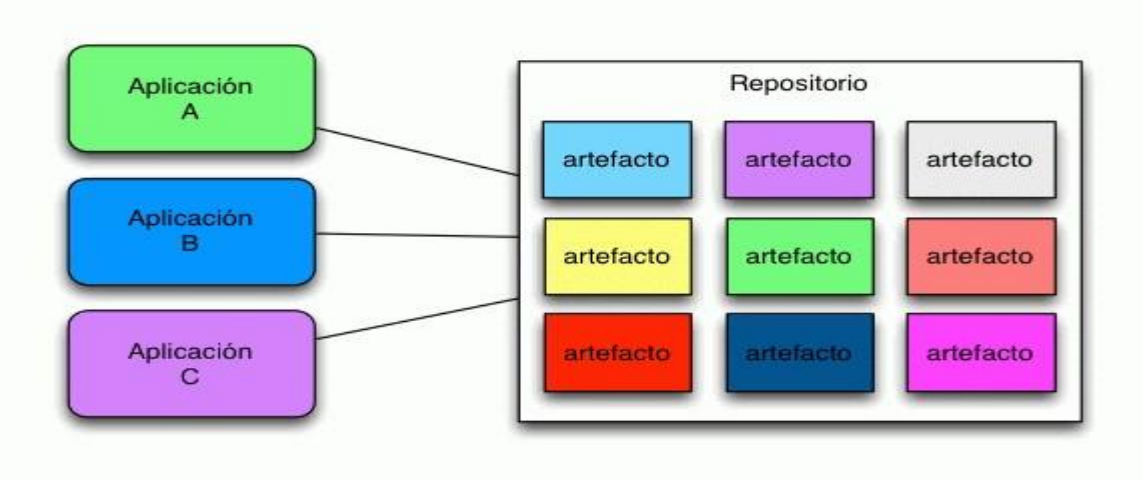
Con las etiquetas <**gropuId**>, <**artifactId**> y <**version**> es posible identificar la biblioteca y la versión. Además, en la segunda de ellas se ve como se hace uso de una de las variables globales que se han definido anteriormente en la etiqueta <**properties**>.

Además de éstas, se puede añadir la etiqueta <**scope**>, que sirve para indicar a Maven cuando se quiere que utilice una biblioteca u otra. Es decir, si se quiere que sólo la añada al proyecto cuando se compile en pruebas(test), sólo en tiempo de ejecución (runtime)… Si no se especifica ningún valor, por defecto Maven la asigna (compile), compila la dependencia para que esté disponible en todos los classpaths.

Otra etiqueta muy usada es <**exclusions**>, a través de la cual se puede indicar a Maven que librería heredada de las que se han incorporado, no se quiere añadir al proyecto. Esto puede ser muy útil a la hora de evitar conflictos entre librerías y da pie a explicar la potencia de Maven para ahorrar el buscar no sólo la librería que se quiere, sino también las que necesita la propia librería para su correcto funcionamiento.

B.3 El repositorio de Maven

Una vez definidas todas las dependencias, Maven las descarga y las guarda todas en un mismo repositorio, generalmente <USER_HOME>/.m2/repository. Aunque se puede cambiar si se desea.



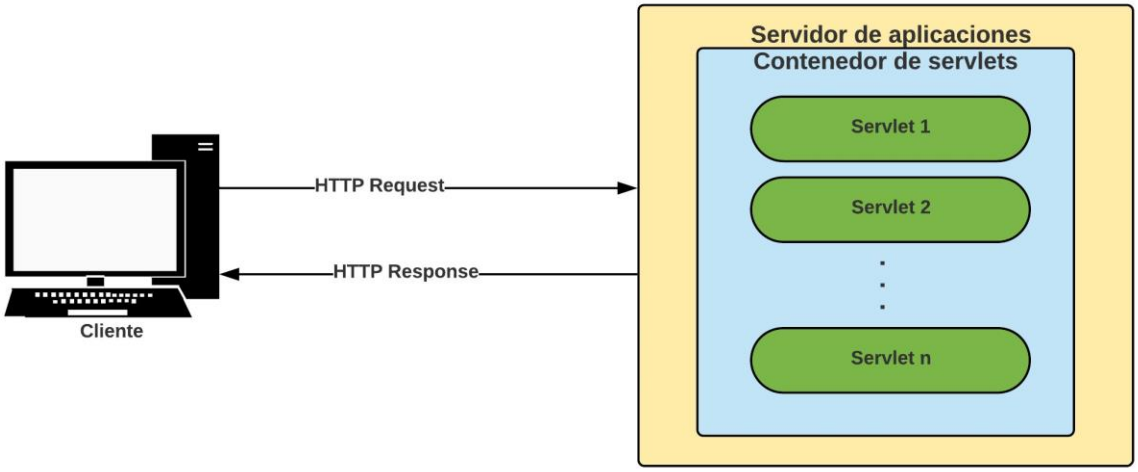
Como se puede ver en la imagen, este repositorio es compartido por todos los proyectos, evitando jars duplicados y guardando las distintas versiones de cada una de las librerías que son necesarias sin que exista ningún conflicto con el uso de las mismas en los distintos proyectos que generen.

C. Spring Framework

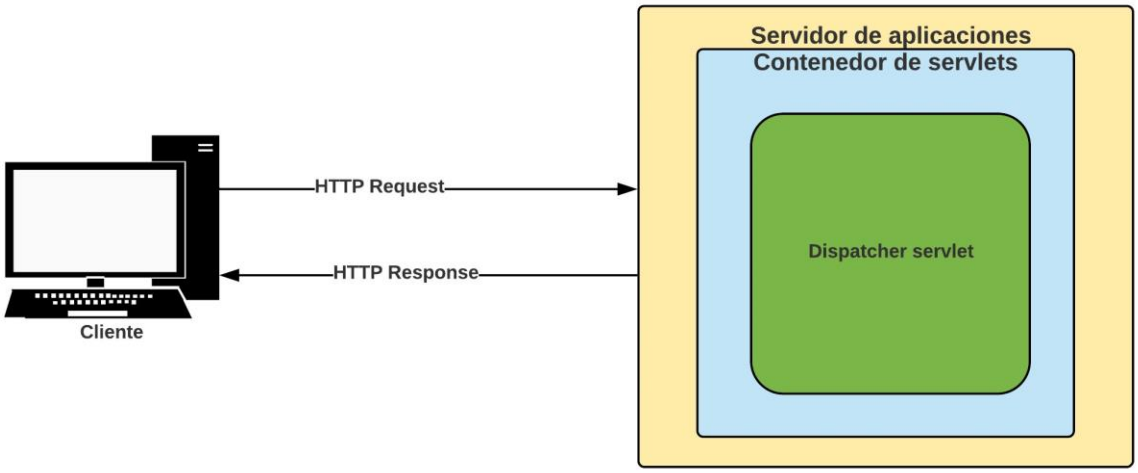
Un Framework son un conjunto de clases y soluciones ya implementadas para su uso con el fin de estandarizar, agilizar y resolver los problemas.

El Framework Spring nos permite desarrollar aplicaciones de manera más eficaz y rápida, ahorrando a su vez muchas líneas de código.

En un modelo de aplicación web habitual, sin hacer uso de algún framework de gestión de las acciones de la web, el cliente envía peticiones HTTP al servidor y éste tendrá configurados distintos servlets para dar una respuesta al cliente. De esta manera, la aplicación estará formada por un número determinado de servlets que desarrollarán las distintas funcionalidades definidas para la misma.



El framework de Spring sin embargo solo necesita tener configurado un servlet principal, llamado normalmente DispatcherServlet, que recibe todas las peticiones HTTP que llegan a la aplicación, es decir se recoge toda la funcionalidad en un único servlet.



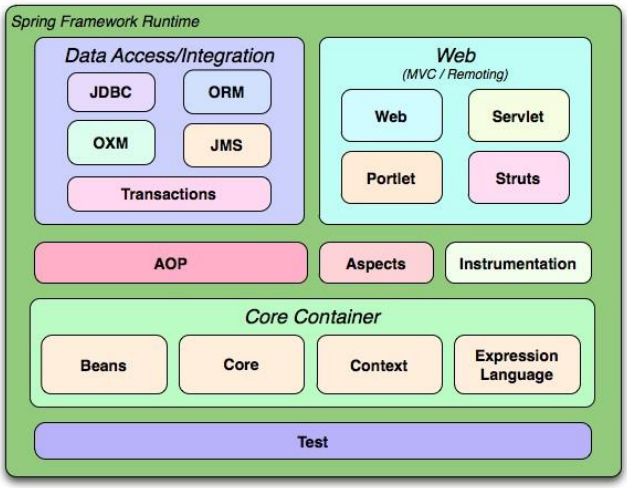
Para que esto sea posible el servlet se apoya en un contenedor que se carga en el despliegue de la aplicación y el cual es configurado a través del fichero normalmente nombrado “application-context”. Este fichero es de tipo XML y en él se recogen en forma de bean todos los recursos de la aplicación. Cuando la aplicación sea desplegada, el framework leerá este fichero e instanciará todos aquellos objetos que así hayan sido definidos para su posterior uso en la aplicación.

El framework llevará a cabo la creación y destrucción de las instancias de los objetos en relación a cómo se definan en el contexto. De esta manera se encuentran los siguientes ámbitos de creación de bean:

- **Singleton**: Es el ámbito por defecto de Spring, es decir, si no se especifica el tipo en la creación del bean, Spring lo creará con este ámbito. El contenedor de Spring creará una única instancia compartida de la clase designada por este bean, por lo que siempre que se solicite este bean se estará inyectando el mismo objeto.
- **Prototype**: El contenedor de Spring creará una nueva instancia del objeto descrito por el bean cada vez que se le solicite el mismo. En algunos casos puede ser necesario, pero hay que tener en cuenta que no se debe abusar de este tipo puesto que puede causar una pérdida de rendimiento en la aplicación.
- **Request**: El contenedor de Spring creará una nueva instancia del objeto definido por el bean cada vez que reciba un HTTP request.
- **Session:** El contenedor de Spring creará una nueva instancia del objeto definido por el bean para cada una de las sesiones HTTP y entregará esa misma instancia cada vez que reciba una petición dentro de la misma sesión.

C.1 Módulos

Spring se apoya en diferentes módulos con sus respectivas funcionalidades. Los cuales presentan el siguiente esquema:



Uno de los más importante es el módulo de core, que permite crear e inyectar beans de cara al diseño de inversión del control (IoC) que proporciona Spring. Este diseño consiste en especificar respuestas y acciones deseadas ante sucesos para que algún tipo de entidad o arquitectura externa lo ejecute, sin necesidad de preocuparse de qué manera se lleva a cabo.

Otro módulo importante de cara a la implementación llevada a cabo en este proyecto ha sido el Web, Spring integra en este módulo el Modelo Vista Controlador(MVC) y la definición interna de servlets.

C.2 Creación de contexto y beans

El principal componente del framework Spring es el contenedor de inversión de control (IoC), este contenedor es el encargado de administrar los *beans*, un bean es un objeto Java que es administrado por Spring, es decir, la tarea de instanciar, inicializar y destruir objetos será delegada a este contenedor, el mismo también realiza otras tareas como la inyección de dependencias (DI).

Para poder usar este contenedor se tiene que configurar, la forma tradicional de hacerlo es a través del archivo XML de configuración de Spring, aunque es posible utilizar anotaciones y código Java.

Una vez definido el contexto es posible formar su cuerpo, añadiéndole beans a través de etiquetas y pudiendo éstos ser usados en cualquier parte de la aplicación. El XML se mostraría de la siguiente manera:

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

<beans:bean id="servicioA" class="es.prueba.ServicioA"></beans:bean>

<beans:bean id="servicioB" class="es.prueba.ServicioB"></beans:bean>

</beans:beans>
```

Como se puede ver, se hace referencia a la clase y se le aporta un id para acceder a ella:

```
ServicioA servicioA = (ServicioA) contexto.getBean("servicioA");

System.out.println(servicioA.mensaje());

ServicioA servicioB = (ServicioB) contexto.getBean("servicioB");

System.out.println(servicioB.mensaje());
```

Sin embargo, se tiene el problema de que, si se quiere declarar un gran número de componentes, el fichero xml se puede hacer excesivamente pesado en su manejo y lectura. Para solucionar esto, Spring da la posibilidad de usar anotaciones en las clases, de manera que se puede obtener el mismo resultado, pero de manera más cómoda. Para ejemplificar el caso anterior con anotaciones se tiene que usar la etiqueta @Service.

```java
@Service
public class ServicioA {

      public String mensaje(){  return "Hola ServicioA";
     }

}
```

Una vez hecho esto, se debe indicar al xml que cargue todas las anotaciones creadas. Para ello se usan dos etiquetas especiales:

- **<context:annotation-config />** : con la que se le informa al framework que se van a usar anotaciones en el código.
- **<context:component-scan />** : se le indica al framework la ruta de paquetes que debe escanear en busca de clases para crear sus beans.

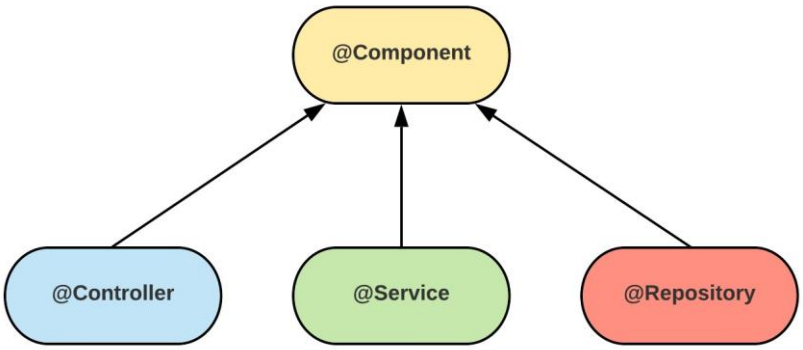
```xml
<context:annotation-config/>

<context:component-scan base-package="es.prueba" />
```

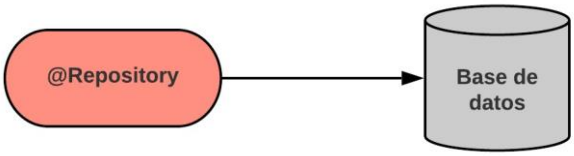
C.3 Spring estereotipos y anotaciones

Spring ha definido multitud de etiquetas y anotaciones con el fin de categorizar componentes dentro del código y asignarles un cometido. **Estereotipos**

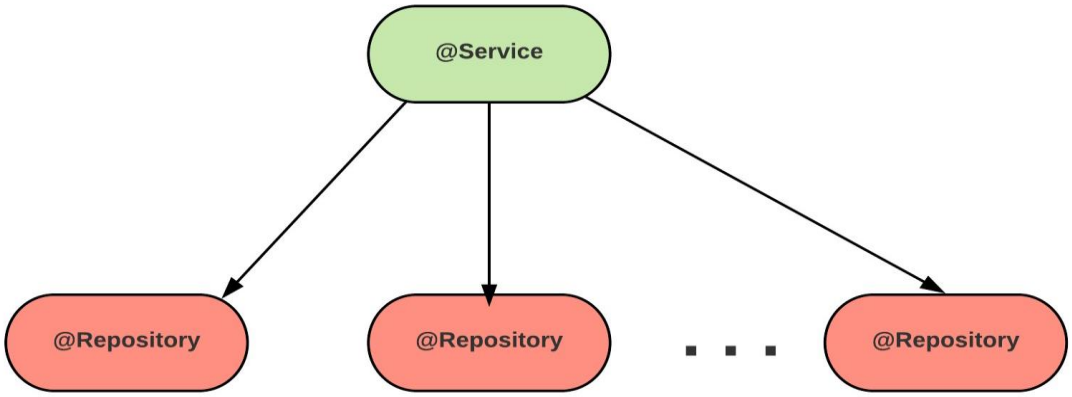
@Component: es el estereotipo principal, indica que la clase anotada es un componente o bean de Spring.



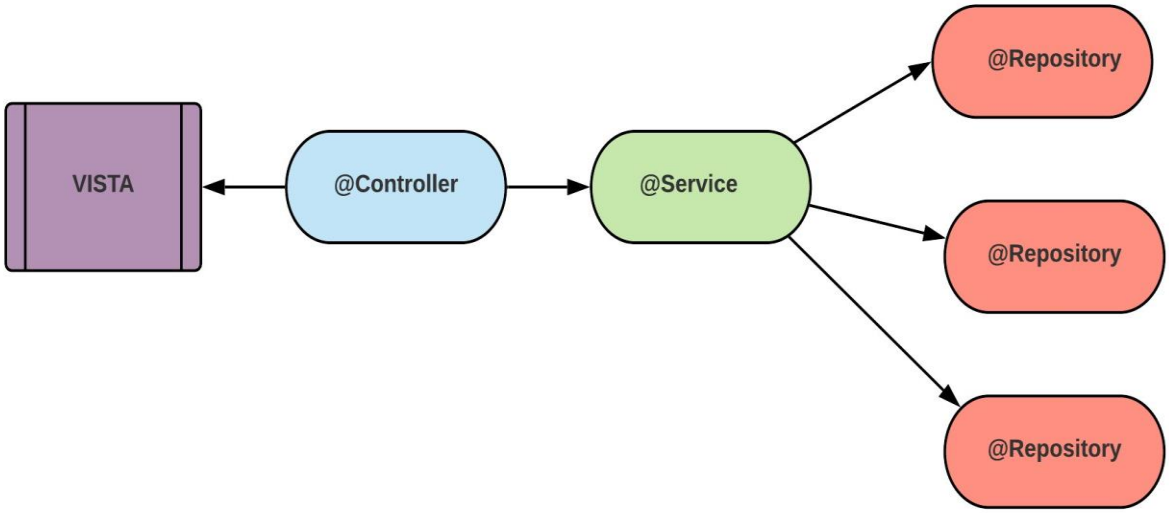
@Repository: Es el estereotipo que tiene como función dar de alta un bean para que implemente el patrón repositorio, que es el encargado de almacenar datos en una base de datos o repositorio de información que se necesite. Al marcar el bean con esta anotación, Spring aporta servicios transversales como conversión de tipos de excepciones.



@Service: Este estereotipo se encarga de gestionar las operaciones de negocio más importantes a nivel de la aplicación y aglutina llamadas a varios repositorios de forma simultánea. Su tarea fundamental es la de agregador.



@Controller: El último de los estereotipos, es el que realiza las tareas de controlador y gestión de la comunicación entre el usuario y el aplicativo. Para ello se apoya habitualmente en algún motor de plantillas o librería de etiquetas que facilitan la creación de páginas.



Anotaciones

Una de las anotaciones más importantes de Spring es **@Autowired**, con ella es posible inyectar un componente como puede ser un servicio o un bean en la clase que se desea hacer uso de él. Por ejemplo:

Se declara una interfaz:

```
public interface AlumnosSvc {

     AlumnosJson buscarAlumno(String dni) throws AppException;
    List<AlumnosJson> buscarAlumnos(List<String> listaDnis) throws AppException;

}
```

Se implementan los métodos de la interfaz:

```
@Transactional @Service
public class AlumnosSvcImpl implements AlumnosSvc {


private final Logger logger = LoggerFactory.getLogger(AlumnosSvcImpl.class);

     /**
* Obtener Alumno por dni de BBDD
      */
     public AlumnosJson buscarAlumno(String dni) throws AppException{
          //IMPLEMENTACIÓN
     }

     /**
* Obtener Alumnos por dni de BBDD
      */
 public List<AlumnosJson> buscarAlumnos(List<String> listaDnis) throws
AppException{
          //IMPLEMENTACIÓN


}

}
```

Se inyecta la interfaz en otra clase con “@Autowired”:

```
@Service("restService")
public class RestServiceImpl implements RestService {



      @Autowired(required = true)
      private AlumnosSvc alumnosSvc;
```

Una vez inyectado el recurso en la clase, es posible hacer uso de sus propiedades y métodos:

```
alumnos = alumnosSvc.buscarAlumnos(listaDnis);
```

Si dentro de la interfaz en lugar de métodos se tienen declarados servicios mediante la etiqueta @Service, es posible especificar cuál de ellos se quiere inyectar con la etiqueta **@Qualifier**

```
public interface AlumnosSvc {

@Service("obtenerDNI")
public class ObtenerDNI implements AlumnosSvc {
}

@Service("ingresarAlumno")
public class IngresarAlumno implements AlumnosSvc {
}



}
```

Si se quiere inyectar ObtenerDNI:

```
@Service("restService")
public class RestServiceImpl implements RestService {



     @Autowired(required = true)
      @Qualifier("obtenerDNI")
      private AlumnosSvc alumnosSvc;
```

Otra posibilidad sería realizarlo a través de la anotación **@Resource:**

```
@Service("restService")
public class RestServiceImpl implements RestService {

      @Resource("obtenerDNI")
      private AlumnosSvc alumnosSvc;
```

@RestController. Con esta etiqueta Spring facilita mucho el escenario de un API REST.
En primer lugar, se debe configurar un servlet de escucha en el fichero web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/jav
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

Y añadir esta etiqueta en el fichero que actuará como controlador del servicio:

```
@RestController
public class RestController {

      @RequestMapping(value="uri") public
      ObjetoRespuesta metodoController {
        //…
      return objeto;
      }
}
```

Otra manera de formar un API REST a través de Spring sería con el uso del estereotipo **@Controller**, pero si se usa éste, es necesario añadir a cada uno de los métodos que se definen en el controlador la etiqueta **@ResponseBody**:

```
@Controller
public class RestController {

     @RequestMapping(value="uri")
      @ResponseBody
      public ObjetoRespuesta metodoController {
        //…
      return objeto;
      }
}
```

C.4 Spring Boot

Spring Boot facilita la creación de aplicaciones independientes basadas en Spring de grado de producción que puede "simplemente ejecutar".

Tomamos una visión obstinada de la plataforma Spring y las bibliotecas de terceros para que pueda comenzar con un mínimo de alboroto. La mayoría de las aplicaciones Spring Boot necesitan una configuración mínima de Spring.

Caracteristicas

- Cree aplicaciones Spring independientes
- Incruste Tomcat, Jetty o Undertow directamente (no es necesario implementar archivos WAR)
- Proporcione dependencias "iniciales" obstinadas para simplificar su configuración de compilación
- Configure automáticamente las bibliotecas de Spring y de terceros siempre que sea posible
- Proporcione funciones listas para producción, como métricas, comprobaciones de estado y configuración externa.
- Absolutamente ninguna generación de código y ningún requisito para la configuración XML

D. JPA (Java Persistence API)

El desarrollo que se realiza en una aplicación de negocio JAVA es orientado a objetos, sin embargo, las bases de datos relacionales almacenan la información en forma de tabla con sus respectivas filas y columnas. Para realizar una correlación entre ambos se necesita de una interfaz que permita guardar la información que se maneja dentro de la aplicación JAVA en el modelo relacional de base de datos.

JPA es una abstracción de JDBC que permite comunicar ambos modelos de manera sencilla, realizando la conversión entre objetos y tablas. Esta conversión se llama ORM (Mapeo Relacional de Objetos) y puede configurarse mediante un xml o anotaciones. Toda esta relación es transparente el desarrollador y para ello se deben crear los objetos de una manera singular, a éstos se les llama entidades.

JPA es una interfaz común y generalmente ésta es implementada por frameworks de persistencia, en este proyecto se ha elegido Hibernate. De esta manera será Hibernate el que realice el trabajo, pero siempre funcionando bajo la API de JPA.

D.1 Anotaciones principales

Cuando se habla de una entidad, se hace referencia en cualquier caso a un objeto POJO (Plain Old Java Object), es decir un objeto sencillo que no extiende de otro ni implementa funcionalidad fuera del mismo.

Una entidad se caracteriza por ser una clase de primer nivel, no ser final, proporcionar un constructor e implementar la interfaz java.io.Serializable.

Cualquier entidad JPA presenta numerosas etiquetas que la caracterizan, las necesarias para formar una entidad sencilla serían:

- **@Entity**: informa al proveedor de persistencia que cada instancia de esta clase es una entidad.
- **@Table:** permite configurar el nombre de la tabla que se está mapeando con dicha entidad. Para ello se hace uso del atributo name.
- **@Colum:** acompañará a cada uno de los atributos de la entidad y permitirá configurar el nombre de la columna a la que dicho atributo hace referencia dentro del sistema relacional. Esta etiqueta posee numerosos atributos, donde se puede indicar al framework propiedades que debe tener en cuenta para la columna.
- **@Id:** acompañará aquel atributo que permita diferenciar las distintas entidades de manera que cada una sea única, normalmente acompaña al que se asocie a la clave primaria de la tabla que se esté mapeando.
- **@JoinColumn:** permite configurar aquel atributo que contiene una clave foránea a otra tabla.
- **@OneToMany:** esta etiqueta irá acompañando a la anterior en el caso de que el atributo de la entidad puede referenciar a más de un atributo de la tabla a la que hace referencia. A la hora de obtener estos valores en una consulta, se pueden diferenciar dos tipos:
    - **EAGER:** se puede definir como un tipo de lectura temprana, es decir, en la consulta del objeto se obtienen todos los valores de las entidades que están relacionadas con la entidad. Esto es desaconsejable en el caso de asociaciones en las que se puedan ver involucrados muchos objetos, ya que puede suponer un problema de rendimiento para la aplicación.
    - **LAZY:** se define como lectura demorada, permite obtener un objeto de base de datos sin los valores de la relación a la que hace referencia el atributo. Es muy útil de cara al rendimiento de la aplicación ya que evita obtener ciertas propiedades que puedan no ser necesarias en la creación del objeto. Lo que se inicializaría es una asociación, de manera que, si se quiere acceder a la información en algún momento, se obtendría.

E. Hibernate

Hibernate es un framework de persistencia de software libre que implementa la gestión de la API de persistencia de Java. Este framework actúa de manera transparente al usuario y le aporta escalabilidad, eficiencia y facilidades a la hora de hacer consultas en la base de datos.

En lo referente a la flexibilidad y escalabilidad, hibernate está diseñado para adaptarse a cualquier esquema y modelo de datos, además ofrece la relación inversa, pudiendo así obtener un modelo de tablas relacional a partir de la definición de entidades en Java.

Hibernate presenta un sistema de doble caché, el segundo es totalmente configurable. Estos sistemas se definen como nivel 1 y nivel 2.

El First Level Caché es el que presenta automáticamente Hibernate cuando dentro de una transacción se interactúa con la base de datos, en éste caso se mantienen en memoria los objetos que fueron cargados y si más adelante en el flujo del proceso se vuelven a necesitar van a ser retornados desde la cache, ahorrando accesos sobre la base de datos. Se puede considerar como una caché de corta duración ya que es válido solamente entre el begin y el commit de una transacción, de forma aislada a las demás.

El Second Level Cache permite ser configurado de cara a la mejora del rendimiento. La diferencia fundamental es que éste tipo de caché es válida para todas las transacciones y puede persistir en memoria durante todo el tiempo en que el aplicativo esté online, se puede considerar como una caché global.

Hibernate genera las consultas SQL y libera al desarrollador del manejo más primario de las mimas, además posee un lenguaje de consulta propio denominado HQL (Hibernate Query Language) y de una API para construir consultas conocida como Criteria.

E.1 Criteria

Criteria es una API creada por hibernate pensada específicamente para facilitar las consultas a base de datos. La principal ventaja que tiene es que la forma de construir las querys de base de datos es absolutamente orientada a objetos. Esto es de gran utilidad de cara a dar soporte a un formulario de búsqueda que presente multitud de campos, ya que nos evitaría líneas de comprobación para cada uno de los campos, simplificando mucho el código de las consultas.

La mecánica es sencilla, se crea una instancia de Criteria para un objeto en concreto y haciendo uso de métodos de esta instancia se da forma a las restricciones de búsqueda en torno a ese objeto.

```
Criteria crit = sess.createCriteria(Cat.class);
```

Para limitar el número de resultados:

```
crit.setMaxResults(50);
```

Añadir criterios individuales:

```
crit.add( Restrictions.like("name", "Fritz%") )
crit.add( Restrictions.between("weight", minWeight, maxWeight) )
```

Para disyunciones:

```
crit.add( Restrictions.or(
      Restrictions.eq( "age", new Integer(0) ),
      Restrictions.isNull("age")
  ) )
```

Ordenar resultados:

```
crit.addOrder( Order.asc("name") )
```

Con esto se puede conseguir cualquier consulta básica, pero el abanico de posibilidades es mucho más extenso pudiendo incluso introducir en ellas código sql, crear proyecciones o formar subconsultas. La siguiente ilustración presenta un ejemplo de consulta en el servicio web que se presenta.

```
public List<Titulaciones> obtenerTitulacionesPorCentro(List<String> centros) throws AppException{
    List<Titulaciones> hs = null;

    try{
        Criteria criteria = sessionFactory.getCurrentSession().createCriteria(Titulaciones.class);
        criteria.createAlias("centros", "c");
        Disjunction matchDisjunction = Restrictions.disjunction();
        for (String centro : centros) {
            matchDisjunction.add(Restrictions.ilike("c.centNmNombre", centro, MatchMode.ANYWHERE));
        }
        criteria.add(matchDisjunction);
        hs = (List<Titulaciones>) criteria.list();
    }catch(Exception e){
        logger.error(e.getMessage(), e);
        throw new AppException(AppException.ERROR_10, "Ha ocurrido un error al obtener las titulaciones.",e);
    }

    return hs;
}
```

F. JWT (JSON Web Token)

La autenticación basada en token es una referencia en el desarrollo de aplicaciones web, ya que presenta algunas ventajas respecto a la autenticación más común, en la que se guardan en sesión los datos del usuario.

En la autenticación con token, el usuario se identifica bien con un usuario/contraseña o mediante una única clave y la aplicación web le devuelve una especie de firma cifrada que el usuario usará en las cabeceras de cada una de las peticiones HTTP.

Esta información no se tiene que almacenar en la parte del servidor, como en el caso de la autenticación a través de sesión, sino que se guarda en la parte del cliente y es la aplicación la que comprobará si es válida en cada una de las peticiones.

A su vez, con ello se gana en escalabilidad, pudiendo usar cualquier tecnología (Web, Android, iOS…) para hacer uso de la aplicación, sin diferenciar entre ellas en el servidor.

El token que envía como respuesta la aplicación tiene un tiempo de vida el cuál se debe configurar acorde a lo que interese.

El JWT está formado por tres cadenas separadas por punto:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9.eyJ
zdWIiOiIiLCJleHAiOjE1MzU5NjcwMTgsImlhdCI
6MTUzNTg4MDYxOH0.aaavOmuvUE2AMbkzMRef7Gx
6i9IEgcYnY3XNPZXhNZGhj7L3EFjJWgyTB4j4phi
tRY_AGbXo1N4teshYTAn8QA
```

Header

Es la primera parte del token, está formada por el tipo de token y el algoritmo de codificación utilizado:

```
{
"typ": "JWT",
 "alg": "HS512"
}
```

Payload

Está compuesto por atributos llamados Claims, existen:

- **iss**: especifica la tarea para la que se va a usar el token.
- **sub**: presenta información del usuario.
- **aud**: indica para que se emite el token. Es útil en caso de que la aplicación tenga varios servicios que se quieren distinguir.
- **iat**: indica la fecha en la que el token fue creado.
- **exp**: indica el tiempo de expiración del token, se calcula a partir del iat.
- **nbf**: indica el tiempo en el que el token no será válido hasta que no transcurra.
- **jti**: identificador único del token. Es utilizado en aplicaciones con diferentes proveedores.

Los más comunes son **sub**, **iat** y **exp**. Para el proyecto en cuestión, en el que no se hace login por usuario:

```
{
  "sub": "",
  "exp": 1535967018,
  "iat": 1535880618
}
```

También admite campos personalizados, por ejemplo:

```
{
  "sub": "",
  "exp": 1535967018,
  "iat": 1535880618,
  "rol": 1,
  "seg": "alta"
}
```

Signature

La firma es la última de las tres partes, está formada por la información del Header y el Payload codificada en Base64, más una clave secreta que se configura en la propia aplicación.

```
HMACSHA512(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```