Microservicios con Spring

Victor Herrero Cazurro

Contenidos

Spring Cloud

¿Que son los Microservicios?

Segun Martin Fowler y James Lewis, los microservicios son pequeños servicios autonomos que se comunican con APIs ligeros, tipicamente con APIs REST.

El concepto de autonomo, indica que el microservicio encierra toda la lógica necesaria para cubrir una funcionalidad completa, desde el API que expone que puede hacer hasta el acceso a la base de datos.

Las aplicaciones modernas presentan habitualmente la necesidad de ser accedidas por diversos tipos de clientes, browser, moviles, aplicaciones nativas, para ello deben implementar un API para ser consumidas, además de para ser integradas con otras aplicaciones a traves de servicios web o brokers de mensajeria.

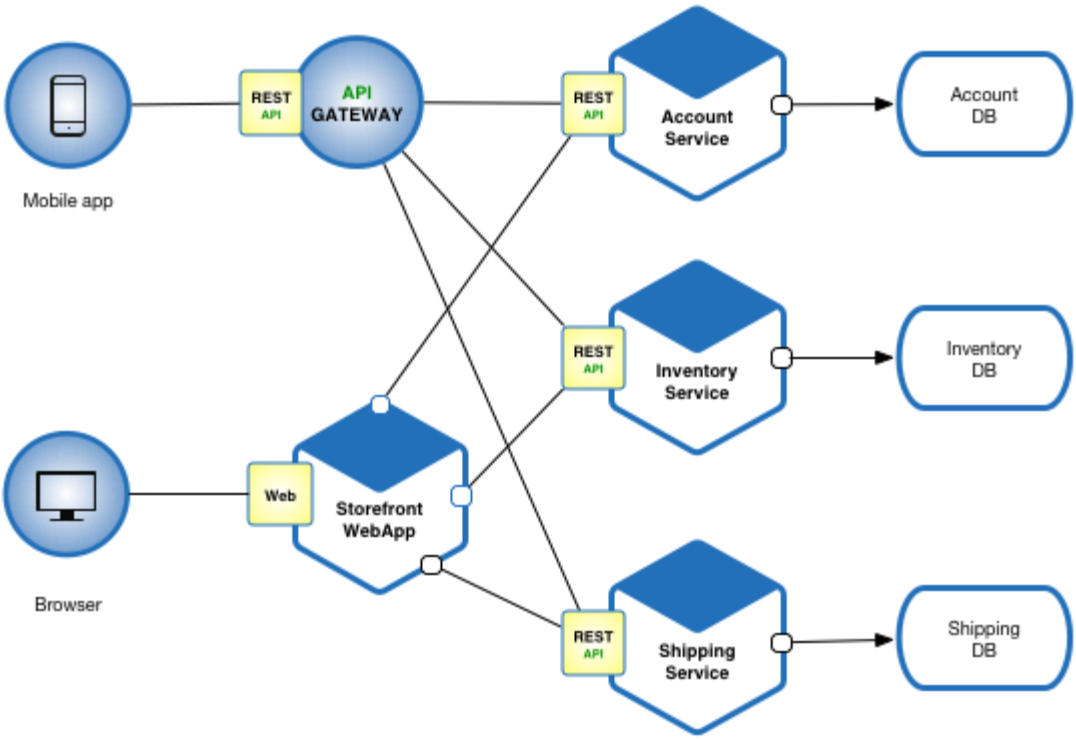
Tambien se busca que los nuevos miembros del equipo puedan ser productivos rapidamente, para lo cual se necesita que la aplicacion sea facilmente comprensible y modificable.

En ocasiones se aplican tecnicas de desarrollo agil, realizando despligue continuo, para lo cual la aplicacion debe poder pasar de desarrollo a producción de forma rapida.

Generalmente las aplicaciones necesitan ser escaladas y tienen criterios de disponiblidad, por lo que se necesitan ejecutar multiples copias de la aplicación.

Y siempre es interesante poder aplicar las nuevas tendencias en frameworks y tecnologias qeu mejoran los desarrollos, por lo que es interesante que los nuevos desarrollos no se vean obligados a seguir tecnologias de los antiguos.

Por estos motivos, será interesante aplicar una arquitectura con bajo acoplamiento y servicios colaborativos, el bajo acoplamiento lo conseguiremos empleando servicios HTTP y protocolos asincronos como AMQP y haciendo que cada microservicio tenga su propia base de datos.



Ventajas de los Microservicios

- Son ligeros, desarrollados con poco codigo.
- Permiten aprovechar de forma mas eficiente los recursos, ya que al ser cada microservicio una aplicación en sí, se pueden aumentar los recursos de dicha funcionalidad sin tener que aumnetar los recursos de otras piezas que quizas no los necesiten, por lo que los recursos son asignados de forma más granular.
- Permiten emplear tecnologias distintas, aprovechando las ventajas puntuales de cada una de ellas, sin que esas ventajas puedan lastrar otros componentes.
- Permiten realizar despliegues más rapidos, ya que evoluciones que se produzcan en un microservicio, al ser independientes no deben afectar a otras piezas del puzle y por tanto según esten terminados se pueden poner en producción, sin tener que esperar a hacer un despliegue de una version completa de toda la aplicación, esto unido a que los microservicios son **pequeños**, hace que el periodo desde que se comienza a desarrollar la mejora hasta la puesta en producción, sea corto y por tanto facilite la aplicación de **Continuous Delivery** (entrega continua).
- Mejoran el comportamiento de las aplicaciones frente a los fallos, ya que estos quedan mas aislados

Desventajas de los Microservicios

- Exigen mayor esfuerzo en el despliegue, control del versionado, copatibilidad entre versiones, actualización, monitorización, ...
- Es mas complejo realizar test de integracion.
- No existe un ambito transaccional entre todos los microservicios que participan.

- Con aplicaciones existentes, es mas complicado de aplicar.

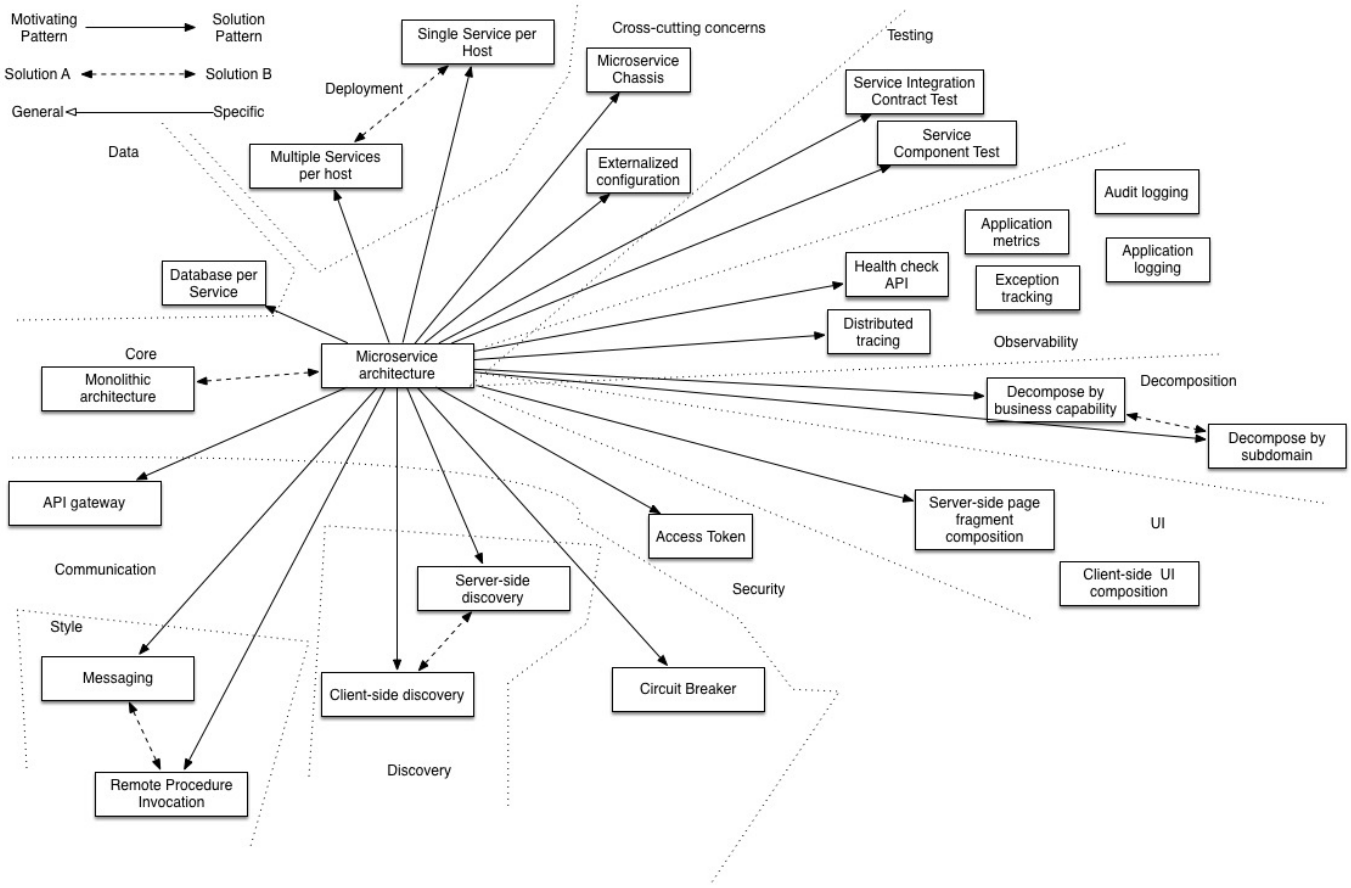
Arquitectura de Microservicios

Cuando se habla de arquitectura de microservicios, se habla de **Cloud** o de arquitectura distribuida.

En esta arquitectura, se pueden producir problemas propios de la arquitectura relacionados con

- Monitorización de la arquitectura,
- Configuración de los microservicios,
- Descubrimiento de microservicos,
- Detección de caida de microservicios,
- Balanceo de carga,
- Seguridad centralizada,
- Logs centralizados, ...

Patrones



Para solventarlos, se suelen emplear patrones, aqui tenemos algunos de los mas importantes agrupados

- **Principio de la responsabilidad unica**: Cada servicio debe tener responsabilidad sobre una sola parte de la funcionalidad total. La responsabilidad debe estar encapsulada en su totalidad por el servicio. Se dice que un **Servicio** debe tener solo una razón para cambiar.
- **Common Closure Principle**: Los componentes que cambian juntos, deben de estar empaquetados juntos, por lo que cada cambio afecta a un solo servicio.

Descomposicion

- **Descomposicion por capacidades de negocio**: Cada funcionalidad del negocio se encapsula en un unico sercivio. Por ejemplo: **Gestor del catalogo de productos**, **Gestor del inventario**, **Gestor de compras**, **Gestor de envios**, ...
- **Descomposicion por acciones**: Descomponer la aplicacion en microservicios por las acciones/casos de uso que ha de implementar. Por ejemplo: **Servicio de Envios**.
- **Descomposicion por recursos**: Descomponer la aplicacion en microservicios por los recursos que manejan. Por ejemplo: **Servicio de Usuario**.

Despliegue

- **Multiples servicios por servidor**: Se aprovechan mas los recursos, en cambio es mas dificil medir los recursos que necesita cada servicio, a parte de posibles conflictos en cuanto a los recursos compartidos.
- **Un Servicio por servidor**: Es mas sencillo redesplegar, gestionar y monitorizar el servicio, no existiendo conflictos con otros servicios, como pega es el no aprovechamiento optimo de los recursos, dado que el contenedor, consume parte de los recursos.
- **Un servicio por VM**: Facilita el despliegue dado que permite definir los requisitos del servicio de forma aislada, sin tener sorpresas, tambien permite gestionar aspectos del despliegue como memoria, CPU, ... El escalado es mas facil, como pega se tiene el tiempo que se tarda en montar la VM.
- **Un servicio por Contenedor**: Se trata de encapsular el despliegue del servicio dentro de un contenedor como Docker, los pros y contras son similares a los de la VM.
- **Serverless deployment**: Se basa en el empleo de una infraestructura de servicios, que abstrae el concepto de entorno de despliegue, basicamente se pone la aplicacion en el servicio y este reserva recursos y la pone en funcionamiento. Algunas son: AWS Lambda, Google Cloud Functions o Azure Functions.
- **Service deployment platform**: Se basa en el empleo de una plataforma de despliegue, que abstrae el servicio, porporcionando mecanismos de alta disponibilidad por nombre de servicio. Algunos ejemplo de plataformas son: Docker Swarm, Kubernetes, Cloud Foundry o AWS Elastic Beanstalk.

Cross cutting concerns (Conceptos Transversales)

- **Chassis**: Establece la conveniencia de usar una herramienta que gestione la configuracion de los aspectos trasversales del desarrollo como son: Externalizar configuraciones como credenciales de acceso o localizacion de los servicios como son Bases de datos o Brokers de mensajeria,

configuracion del framework de Logging, Servicios de monitorizacion del estado de la aplicacion, Servicios de metricas que permiten ajustar el rendimiento, Servicios de traza distribuida.

- **Externalizar configuraciones**: Establece la necesidad de externalizar la configuracion de tal forma que la aplicación pueda obtener su configuracion en la fase de arranque, para independizar su despliegue del entorno, que la aplicación no tenga que cambiar en cada despliegue. Se traduce en un **Servidor de Configuración** que permite centralizar las configuraciones de todos los microservicios que forman el sistema en un único punto, facilitando la gestión y posibilitando cambios en caliente.

Formas de comunicacion

- **Remote Procedure Invocation**: Permite definir una interface comunicacion entre las aplicaciones clientes y los servicios, asi como entre los propios servicios cuando han de colaborar para satisfacer la peticion del cliente. La implementacion mas habitual es REST.
- **Messaging**: Permite la comunicacion asincrona y desacoplada entre los servicios. Algunas implementaciones son Apache Kafka y RabbitMQ.
- **Domain-specific protocol**: Describe el uso de un protocolo especifico para el dominio tratado, como puede ser SMTP para los correos electronicos.

Exponer APIs

- **API Gateway**: Es el punto de entrada para los clientes, este puede simplemente actuar como un enrutador o bien componer una respuesta basada en peticiones a varios servicios.
- **Backend for front-end**: Caso particular del anterior, donde se proporciona un *API Gatway para cada tipo de cliente.
- **Enrutador**: Permite exponer todos los servicios que los clientes necesitan, independientemente del tipo de cliente.
- **Balanceo de carga (LoadBalancing)**: Necesidad de que los clientes puedan elegir cual de las instancias de un mismo servicio al que desean conectarse se va a emplear, todo de forma transparente. Esta funcionalidad se basa en obtener las instancias del Servidor de Registro y Descubrimiento.

Descubrir Servicios

- **Service registry**: Servicio que mantiene las ubicaciones de las distintas instancias de los microservicios y que es consultable.
- **Client-side discovery**: La forma en la que un cliente (API Gateway u otro mecroservicio) obtiene la referencia a un microservicio, es a traves del servicio de registro, donde todas las instancias de todos los microervicios se han de registrar para que el resto los encuentre.
- **Server-side discovery**: El cliente no accede directamente al servicio, sino que lo accede a traves de un enrutador, que consulta al servicio de registro.
- **Self registration**: Capacidad de los servicios para registrarse en el servicio de registro de forma

autonoma, para quitar responsabilidad y complejidad al servicio de registro.

- **3rd party registration**: Cuando el registro lo realiza una herramienta independiente.

Confiabilidad (Reliability)

- **Control de ruptura de comunicación con los servicios (CircuitBreaker)**: Permite controlar que la caida de un microservicio consultado, no provoque la caida de los microservicios que realizan la consulta, proporcionando un resultado estatico para la consulta.

Gestion de Datos

- **Database per Service**: Cada servicio tiene su propia base de datos, que solo es accesible mediante el API que proporciona el servicio, para ello se pueden tener tablas privadas por servicio, esquemas por servicio o incluso base de datos por servicio. Con esta aproximación se consigue un bajo acoplamiento entre servicios, cada servicio puede emplear la tecnologia de persistencia mas adecuada. Existen dos desventajas principalmente, la no posibilidad de trabajar con transacciones distribuidas, que se puede paliar con el patrón **Saga** y la complejidad de implementar consultas con **joins**.
- **Shared database**: Al contrario que la anterior, indica compartir una misma base de datos por todos los microservicios, proporciona posibilidad de realizar transacciones tradicionales y es mas facilmente de operar. Como desventajas estan la necesidad de que los equipos coordinen cambios en el esquema, problemas de rendimiento al acceder tantos microservicios a la misma base de datos y que algunos microservicios no puedan adaptar sus necesidades a esa situación.
- **Patrón Saga**: Permite mantener la consistencia de los datos, que queda afectada por el echo de que cada microservicio tiene su propia base de datos y no es posible aplicar transacciones distribuidas. La idea es que cuando el servicio ve afectado sus datos, envia un evento que permite al resto actualizar sus datos.
- **API Composition**: Describe la creación de un nuevo componente (servicio), que se encarga de consultar a cada servicio propietario de los datos, realizando un join en memoria. Un API Gateway puede realizar esta operacion.
- **CQRS (Command Query Responsibility Segregation)**: Se usa junto con **Event Sourcing**. Describe la separación de las operaciones sobre los datos en dos partes **command** (actualizan el estado) y **query** (consultan el estado), de tal forma que podemos tener distintos motores para las **query** que se mantienen actualizados por la subscripcion a los eventos generados por los **command**.
- **Event sourcing**: Describe como se persiste el estado de una entidad, guardando un listado de eventos que han llevado a la entidad al estado actual, recomoniendo el estado cada vez que es necesario, para mejorar el rendimiento ante entidades que modifican su estado con asiduidad, se realizan snapshot que sirven para recomponer el estado aplicando unicamente los ultimos eventos.
- **Application events**: Describe como se insertan registros en una tabla **events** por cada operacion y un proceso los publica en un **message broker**.
- **Traza de seguimiento de transacciones (Transaction log tailing)**: Describe la publicacion del log generado por cada transaccion como evento para trasladar los cambios al resto de microservicios.

Seguridad

- **Access Token**: Define la autenticación centralizada en el **API Gateway**, ya que es el que interaciona con el cliente, que obtiene un **token** que traslada al resto de microservicios para que estos puedan asegurar que el cliente que realiza la petición tiene permisos para invocarlos.

Observacion

- **Gestión centralizada de logs**: Define la generación de una traza formada por todas las trazas generadas por los microservicios participantes en una petición.
- **Metricas de aplicacion**: Define la creación de servicios que permiten obtener el estado de los microservicios.

Orquestacion vs Coreografia

Se habla de **Orquestación**, cuando una aplicacion, gestiona como invocar a otras aplicaciones, estableciendo los criterios y orden de invocacion.

Se habla de **Coreografía**, cuando una aplicación produce un evento, que hace que otras aplicaciones realicen una acción (Bus de mensajeria).

Servidor de Configuracion

Las aplicaciones que contienen los microservicios se conectarán al servidor de configuracion para obtener configuraciones.

El servidor se conecta a un repositorio **git** de donde saca las configuraciones que expone, lo que permite versionar facilmente dichas configuraciones.

El OSS de Netflix proporciona para esta labor **Archaius**.

Para levantar un servidor de configuración, debemos incluir la dependencia

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Para definir una aplicación como **Servidor de Configuración** basta con realizar dos cosas

- Añadir la anotación **@EnableConfigServer** a la clase **@SpringBootApplication** o **@Configuration**.

```
@EnableConfigServer
@SpringBootApplication
public class ConfigurationApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConfigurationApplication.class, args);
  }
}
```

- Definir en las propiedades de la aplicación, la conexión con el repositorio **git** que alberga las configuraciones.

```
spring.cloud.config.server.git.uri=https://github.com/victorherrerocazurro/config-cloud
spring.cloud.config.server.git.basedir=config
```

NOTE	La uri puede ser hacia un repositorio local.

En el repositorio **git** deberan existir tantos ficheros de **properties** o **yml** como aplicaciones configuradas, siendo el nombre de dichos ficheros, el nombre que se le dé a las aplicaciones

Por ejemplo si hay un microservicio que va a obtener su configuracion del servidor de configuración, configurado en el **application.properties** con el nombre

```
spring.application.name=microservicio
```

o **application.yml**

```
spring:
  application:
    name:microservicio
```

Debera existir en el repositorio git un fichero **microservicio.properties** o **microservicio.yml**.

Las propiedades son expuestas via servicio REST, pudiendose acceder a ellas siguiendo estos patrones

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

Donde

- **application** → será el identificador de la aplicacion **spring.application.name**
- **profile** → será uno de los perfiles definidos, sino se ha definido ninguno, siempre estará **default**
- **label** → será la rama en el repo Git, la por defecto **master**

Seguridad

Las funcionalidades del servidor de configuracion estan securizadas, para que cualquier usuario no pueda cambiar los datos de configuracion de la aplicación.

Para configurar la seguridad, hay que añadir la siguiente dependencia

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Cuando se arranca el servidor, se imprimirá un password en la consola

```
Using default security password: 60bc8f1a-477d-484f-aaf8-da7835c207ab
```

Que sirve como password para el usuario **user**. Si se desea otra configuracion se habra de configrar con Srping Security.

Se puede establecer con la propiedad

```
security:
  user:
     password: mipassword
```

Clientes del Servidor de Configuracion

Una vez definido el **Servidor de Configuración**, los microservicios se conectarán a el para obtener las configuraciones, para poder conectar estos microservicios, se debe añadir las dependencias

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Siempre que se añada dependencias de spring cloud, habra que configurar

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Camden.SR6</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Y configurar a través del fichero **bootstrap.properties** donde encontrar el **Servidor de Configuracion**. Se configura el fichero **bootstrap.properties**, ya que se necesita que los properties sean cargados antes que el resto de configuraciones de la aplicación.

```
spring.application.name=microservicio

spring.cloud.config.enabled=true
spring.cloud.config.uri=http://localhost:8888
```

El puerto 8888 es el puerto por defecto donde se levanta el servidor de configuracion, se puede modificar añadiendo al **application.yml**

```
server:
  port: 8082
```

Dado que el **Servidor de Configuración** estará securizado se debera indicar las credenciales con la sintaxis

```
spring.cloud.config.uri=http://usuario:password@localhost:8888
```

Si se quiere evitar que se arranque el microservicio si hay algun problema al obtener la configuración, se puede definir

```
spring.cloud.config.fail-fast=true
```

Una vez configurado el acceso del microservicio al **Servidor de Configuracion**, habrá que configurar que hacer con las configuraciones recibidas.

```
@RestController
class HolaMundoController {

  @Value("${message:Hello default}")
  private String message;

  @RequestMapping("/")
  public String home() {
    return message;

  }
}
```

En este caso se accede a la propiedad message que se obtendra del servidor de configuración, de no obtenerla su valor será **Hello default**.

Actualizar en caliente las configuraciones

Dado que las configuraciones por defecto son solo cargadas al levantar el contexto, si se desea que los cambios en las configuraciones tengan repercusion inmediata, habrà que realizar configuraciones, en este caso la configuracion necesaria supone añadir la anotacion **@RefreshScope** sobre el componente a refrescar.

```
@RefreshScope
@RestController
class HolaMundoController {

  @Value("${message:Hello default}")
  private String message;

  @RequestMapping("/")
  public String home() {
    return message;

  }
}
```

Una vez preparado el microservicio para aceptar cambios en caliente, basta con hacer el cambio en el repo Git e invocar el servicio de refresco del microservicio del cual ha cambiado su configuracion

```
(POST) http://<usuario>:<password>@localhost:8080/refresh
```

Este servicio de refresco es seguro por lo que habrá que configurar la seguridad en el microservicio

Servidor de Registro y Descubrimiento

Permite gestionar todas las instancias disponibles de los microservicios.

Los microservicios enviaran su estado al servidor Eureka a traves de mensajes **heartbeat**, cuando estos mensajes no sean correctos, el servidor desregistrará la instancia del microservicio.

Los clientes del servidor de registro, buscarán en el las instancias disponibles del microservicio que necesiten.

Es habitual que los propios microservicios, a parte de registrarse en el servidor, sean a su vez clientes para consumir otros micoservicios.

Se incluyen varias implementaciones en Spring Cloud para serrvidor de registro/descubrimiento, Eureka Server, Zookeper, Consul ...

Para configurarlo hay que incluir la dependencia

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

Se precisa configurar algunos aspectos del servicio, para ello en el fichero **application.yml** o **application.properties**

```
server:
  port: 8084 #El 8761 es el puerto para servidor Eureka por defecto

eureka:
  instance:
    hostname: localhost
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
  client:
    registerWithEureka: false
    fetchRegistry: false
```

Para arrancar el servicio Eureka, unicamente es necesario lanzar la siguiente configuración.

```
@SpringBootApplication
@EnableEurekaServer
public class RegistrationServer {

    public static void main(String[] args) {
        SpringApplication.run(RegistrationServer.class, args);
    }
}
```

Registrar Microservicio

Lo primero para poder registrar un microservicio en el servidor de descubrimineto es añadir la dependencia de maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

Para registrar el microservicio habrá que añadir la anotación **@EnableDiscoveryClient**

```
@EnableAutoConfiguration
@EnableDiscoveryClient
@SpringBootApplication
public class GreetingServer {

    public static void main(String[] args) {
        SpringApplication.run(GreetingServer.class, args);
    }
}
```

Y se ha de configurar el nombre de la aplicación con el que se registrará en el servidor de registro Eureka.

```
spring:
  application:
     name: holamundo

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8084/eureka/ # Ha de coincidir con lo definido en el
Eureka Server
```

El tiempo de refresco de las instancias disponibles para los clientes es de por defecto 30 sg, si se desea cambiar, se ha de configurar la siguiente propiedad

```
eureka:
  instance:
    leaseRenewalIntervalInSeconds: 10
```

NOTE

Puede ser interesante lanzar varias instancias del mismo microservicio, para que se registren en el servidor de Descubrimiento, para ello se pueden cambiar las propiedades desde el script de arranque

```
mvn spring-boot:run -Dserver.port=8081
```

Localizacion de Microservicio registrado en Eureka con Ribbon

El cliente empleará el API de **RestTemplate** al que se proxeara con el balanceador de carga **Ribbon**

para poder emplear el servicio de localización de **Eureka** para consumir el servicio.

Se ha de definir un nuevo Bean en el contexto de Spring de tipo **RestTemplate**, al que se ha de anotar con **@LoadBalanced**

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
  return new RestTemplate();
}
```

Una vez proxeado, las peticiones empleando este **RestTemplate**, no se harán sobre el **EndPoint** del servicio, sino sobre el nombre del servicio con el que se registro en **Eureka**.

```
@Autowired
private RestTemplate restTemplate;

public MessageWrapper<Customer> getCustomer(int id) {
    Customer customer = restTemplate.exchange( "http://customer-service/customer/{id}",
HttpMethod.GET, null, new ParameterizedTypeReference<Customer>() { }, id).getBody();
    return new MessageWrapper<>(customer, "server called using eureka with rest template
");
}
```

Si el servicio es seguro, se pueden emplear las herramientas de **RestTemplate** para realizar la autenticación.

```
restTemplate.getInterceptors().add(new BasicAuthorizationInterceptor("user", "mipassword
"));

ResponseEntity<String> respuesta = restTemplate.exchange("http://holamundo", HttpMethod
.GET, null, String.class, new Object[]{});
```

Para que **Ribbon** sea capaz de enlazar la URL que hace referencia al ientificador del servicio en **Eureka** con el servicio real, se debe configurar donde encntrar el servidor **Eureka**

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8084/eureka/
```

Y configurar la aplicación para que pueda consumir el servicio de **Eureka**

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {

  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

Uso de Ribbon sin Eureka

Se puede emplear el balanceador de carga Ribbon, definiendo un pool de servidores donde encontrar el servicio a consultar, no siendo necesario el uso de Eureka.

```
customer-service:
  ribbon:
    eureka:
      enabled: false
    listOfServers: localhost:8090,localhost:8290,localhost:8490
```

Simplificación de Clientes de Microservicios con Feign

Feign abstrae el uso del API de RestTemplate para consultar los microservicios, encapsulandolo todo con la definición de una interface.

Para activar su uso, lo primero será añadir la dependencia con Maven

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

El siguiente paso sera activar el autodescubimiento de las configuraciones de **Feign**, como la anotacion **@FeignClient**, para lo que se ha de incluir la anotacion en la configuracion de la aplicación **@EnableFeignClients**

```java
@SpringBootApplication
@EnableFeignClients
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Luego se definen las interaces con la anotacion @FeignClient

```java
@FeignClient(name="holamundo")
interface HolaMundoCliente {

    @RequestMapping(path = "/", method = RequestMethod.GET)
    public String holaMundo();
}
```

Solo resta asociar el nombre que se ha dado al cliente **Feign** con un sevicio real, para ello en el fichero **application.yml** y gracias a **Ribbon**, se pueden definir el pool de servidores que tienen el servicio a consumir.

```yaml
holamundo:
  ribbon:
    listOfServers: http://localhost:8080
```

Acceso a un servicio seguro

Si al servicio al que hay que acceder es seguro, se pueden realizar configuraciones extras como el usuario y password, haciendo referencia a los **Beans** definidos en una clase de configuracion particular

```
@FeignClient(name="holamundo", configuration = Configuracion.class)
interface HolaMundoCliente {

    @RequestMapping(path = "/", method = RequestMethod.GET)
    public String holaMundo();
}

@Configuration
public class Configuracion {
    @Bean
    public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
         return new BasicAuthRequestInterceptor("user", "mipassword");
    }
}
```

Uso de Eureka

En vez de definir un pool de servidores en el cliente, se puede acceder al servidor **Eureka** facilmente, basta con tener la precaución de emplear en el **name** del Cliente **Feign**, el identificador en **Eureka** del servicio que se ha de consumir.

Añadir la anotacion **@EnableDiscoveryClient** para poder buscar en **Eureka**

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Y configurar la direccion de **Eureka**, no siendo necesario configurar el pool de **Ribbon**

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8084/eureka/
```

Servidor de Enrutado

Permite definir paths y asociarlos a los microservicios de la arquitectura, será por tanto el componente

expuesto de toda la arquitectura.

Spring Cloud proporciona **Zuul** como Servidor de enrutado, que se acopla perfectamente con **Eureka**, permitiendo definir rutas que se enlacen directamente con los microservicios publicados en **Eureka** por su nombre.

Se necesita añadir la dependencia Maven.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

Lo siguientee es activar el Servidor **Zuul**, para lo cual habrá que añadir la anotación **@EnableZuulProxy** a una aplicación Spring Boot.

```
@SpringBootApplication
@EnableZuulProxy
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Solo restarán definir las rutas en el fichero **application.yml**

Estas pueden ser hacia el servicio directamente por su url

```
zuul:
  routes:
    holamundo:
      path: /holamundo/**
      url: http://localhost:8080/
```

Con lo que se consigue que las rutas hacia **zuul** con path **/holamundo/** se redireccionen hacia el servidor **http://localhost:8080/**

NOTE	Se ha de crear una clave nueva para cada enrutado, dado que la propiedad **routes** es un mapa, en este caso la clave es **holamundo**.

O hacia el servidor de descubrimiento **Eureka** por el identificador del servicio en **Eureka**

```
zuul:
  routes:
    holamundo:
      path: /holamundo/**
      #Para mapeo de servicios registrados en Eureka
      serviceId: holamundo
```

Para esto último, habra que añadir la dependencia de Maven

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

Activar el descubrimiento en el proyecto añadiendo la anotación **@EnableDiscoveryClient**

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableZuulProxy
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

E indicar en las propiedades del proyecto, donde se encuentra el servidor **Eureka**

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8084/eureka/
```

Seguridad

En el caso de enrutar hacia servicios seguros, se puede configrar **Zuul** para que siendo el que reciba los token de seguridad, los propague a los servicios a los que enruta, esta configuración por defecto viene desacivada dado que los servicios a los que redirecciona o tienen porque ser de la misma arquitectua y en ese caso, no sería seguro.

```
zuul:
    routes:
        holamundo:
            path: /holamundo/**
            #Para mapeo de las url directas a un servicio
            url: http://localhost:8080/

            #No se incluye ninguna cabecera como sensible, ya que todas las definidas
como sensibles, no se propagan
            sensitive-headers:
            custom-sensitive-headers: true
    #Se evita añadir las cabeceras de seguridad a la lista de sensibles.
    ignore-security-headers: false
```

Circuit Breaker

La idea de este componente es la de evitar fallos en cascadaa, es decir que falle un componente, no por error propio del componente, sino porque falle otro componente de la aquitectura al que se invoca.

Para ello Spring Cloud integra **Hystrix**.

Para emplearlo, se ha de añadr la dependencia Maven

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

La idea de este framework, es proxear la llamada del cliente del servicio sensible a caerse proporcionando una via de ejecución alterantiva **fallback**, para así evitar el error en la invocacion.

Para ello se ha de anotar el método que haga la petición al cliente con **@HystrixCommand** indicando el método de **fallback**

```
@RestController
class HolaMundoClienteController {

  @Autowired
    private HolaMundoCliente holaMundoCliente;

  @HystrixCommand(fallbackMethod="fallbackHome")
  @RequestMapping("/")
  public String home() {
    return holaMundoCliente.holaMundo() + " con Feign";

  }

  public String fallbackHome() {
        return "Hubo un problema con un servicio";
    }
}
```

El método de **Fallback** debera retornar el mismo tipo de dato que el método proxeado.

NOTE	No deberan aplicarse las anotaciones sobre los controladores, dado que los proxys entran en conflicto

Para activar estas anotaciones se ha de añadir **@EnableCircuitBreaker**.

```
@SpringBootApplication
@EnableCircuitBreaker
public class Application {

  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

Monitorizacion: Hystrix Dashboard

Se ha de crear un nuevo servicio con la dependencia de Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
```

Y activar el servicio de monitorizacion con la anotacion **@EnableHystrixDashboard**

```
@SpringBootApplication
@EnableHystrixDashboard
public class Application {

  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

Se accederá al panel de monitorizacion en la ruta **http://<host>:<port>/hystrix** y allí se indicará la url del servicio a monitorizar **http://<host>:<port>/hystrix.stream**

Para que la aplicación configurada con **Hystrix** proporcione información a traves del servicio **hystrix.stream**, se ha de añadir a dicha aplicación **Actuator**, con Maven.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Monitorizacion: Turbine

Se puede añadir un servicio de monitorización de varios servicios a la vez, llamado **Turbine**, para ello se ha de añadir la dependencia

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-netflix-turbine</artifactId>
</dependency>
```

Configuracion Distribuida en Bus de Mensajeria

Se trata de emplear un bus de mensajeria para trasladar el evento de refresco a todos los nodos del los microservicios que emplean una configuracion distribuida.

Se precisa por tanto de un bus de mensajeria, en este caso Spring Cloud apuesta por implementaciones **AMQP** frente a otras alternativas como podrian ser **JMS**. Y mas concretamente **RabbitMQ**.

Para instalar RabbitMQ, se necesita instalar Erlang a parte de RabbitMQ

La configuracion por defecto de **RabbitMQ** es escuchar por el puerto 5672

Servidor

Se necesitará incluir las siguientes dependencias en el servidor de configuracion

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-monitor</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

Y la siguiente configuracion de la ubicacion de RabbitMQ

application.yml

```
server:
  port: 8180

spring:
  cloud:
   bus:
     enabled: true #Habilitamos la publicacion en el bus

   #Indicamos donde esta el repositorio con las configuraciones
   config:
      server:
        git:
          uri: https://github.com/victorherrerocazurro/RepositorioConfiguraciones

#Se necesita conocer donde esta rabbitMQ para enviar los eventos de cambio de
#propiedades
  rabbitmq:
    host: localhost
    port: 5672
```

A partir de este punto el servidor aceptará el refresco de las propiedades a traves del bus, empleando el servicio /**monitor**, al cual podrán acceder los repositorio Git a taves de **Hook**

```
curl -v -X POST "http://localhost:8100/monitor" -H "Content-Type: application/json" -H
"X-Event-Key: repo:push" -H "X-Hook-UUID: webhook-uuid" -d '{"push": {"changes": []} }'
```

Cliente

Se necesitará incluir la siguiente dependencia en los clientes que servidor de configuración

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

Como bus por defecto se emplea **RabitMQ**, al que habrá que configurar las siguientes propiedades

application.properties

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
```

OAuth2

Caracteristicas

- Emplea Https en vez de la Criptografia de OAuth 1.
- Permite flujos con aplicaciones nativas y no solo con Navegadores.
- Posibilidad de asignacion de Privilegios a los Token basados en la caracteristica SCOPE (LEER, ESCRIBIR, ...)
- Caducidad de los Token, **Refresh Token**.

Que se quiere conseguir

- Diferenciar cuando el acceso a los datos privados los hace el dueño de los datos y cuando lo hace una aplicación en nombre del usuario.
- Permitir al usuario dar distintos privilegios de acceso a las distintas aplicaciones que acceden en nombre del usuario a sus datos privados.
- Poder revocar los privilegios concedidos.

Actores

- **Resource Owner** - Usuario propietario de los **datos seguros**.
- **Resource Server** - Servidor que alberga los **datos seguros**, el servidor que realiza la autenticación por OAuth puede ser uno de ellos, por ejemplo permitiendo el acceso a los datos del perfil del usuario.
- **Authorization Server** - Plataforma OAuth, controla que clientes pueden acceder a los **datos**

seguros.

- **Client** - Aplicacion que quiere acceder a los **datos seguros** de un usuario, residentes en otra aplicación y cuyo usuario (propietario) es usuario de las dos aplicaciones tanto del cliente como del servidor de recursos.

Caracteristicas

El dueño del **dato seguro**, es usuario del **Authorization Server**, donde describe su login y posiblemente otros datos seguros.

La aplicacion (cliente), se registra en el **Authorization Server**, para delegar en él, el proceso de Login.

El usuario del **Authorization Server**, otorga permisos a la aplicacion ciente para acceder a determinados **datos seguros** a través de los SCOPE, esta asignación de SCOPEs es revocable.

La aplicación cliente solicitará una serie de SCOPEs, pero el usuario le otorgará los que el decida, de no otorgarle todos los SCOPE, la aplicación cliente puede no funcionar correctamente.

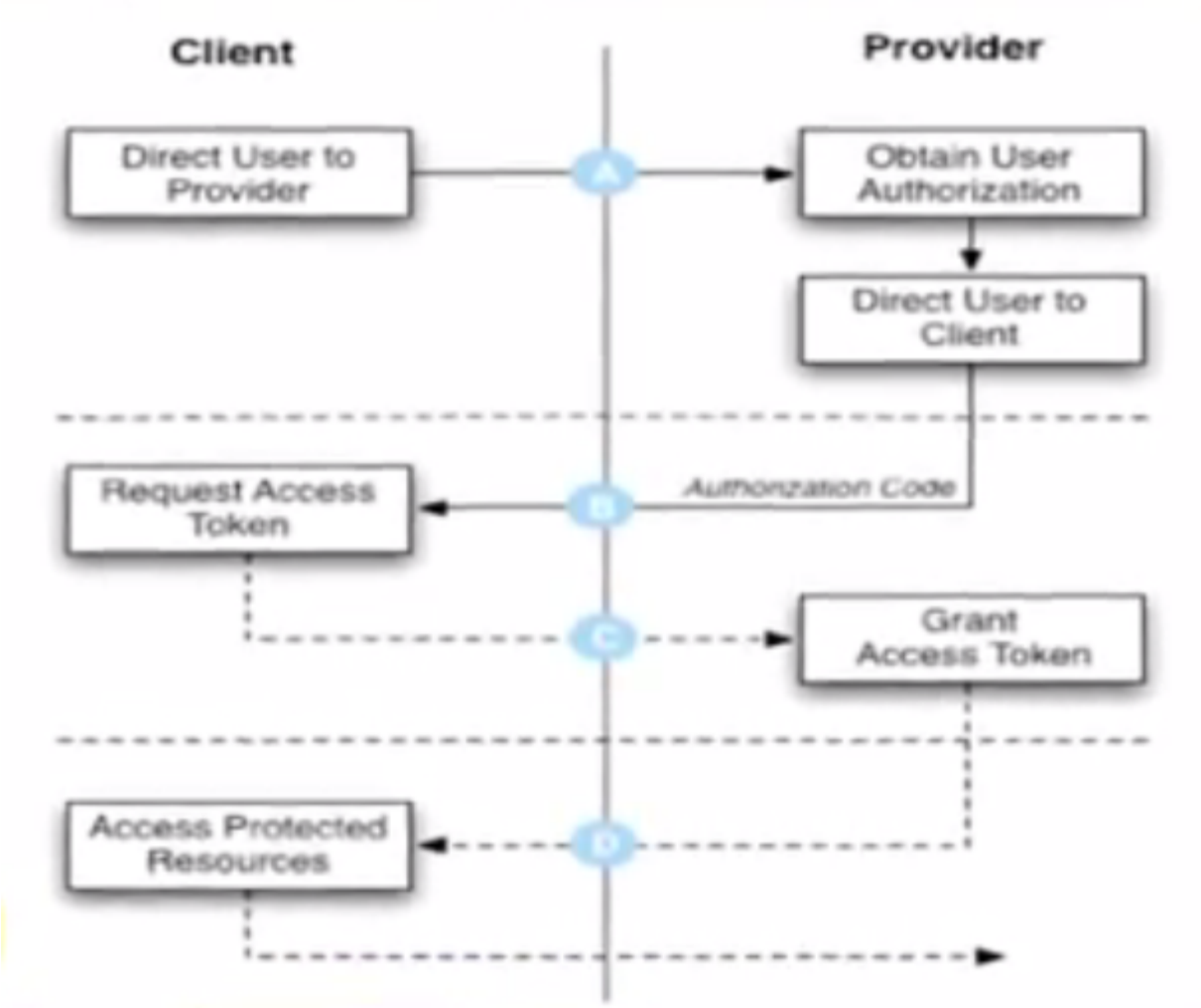
Los SCOPE, son algo similar a los permisos (READ, WRITE, ….), que son definidos por el servidor de recursos. Se definen de forma anologa a los ROLES asociados a las URLs.

Grants

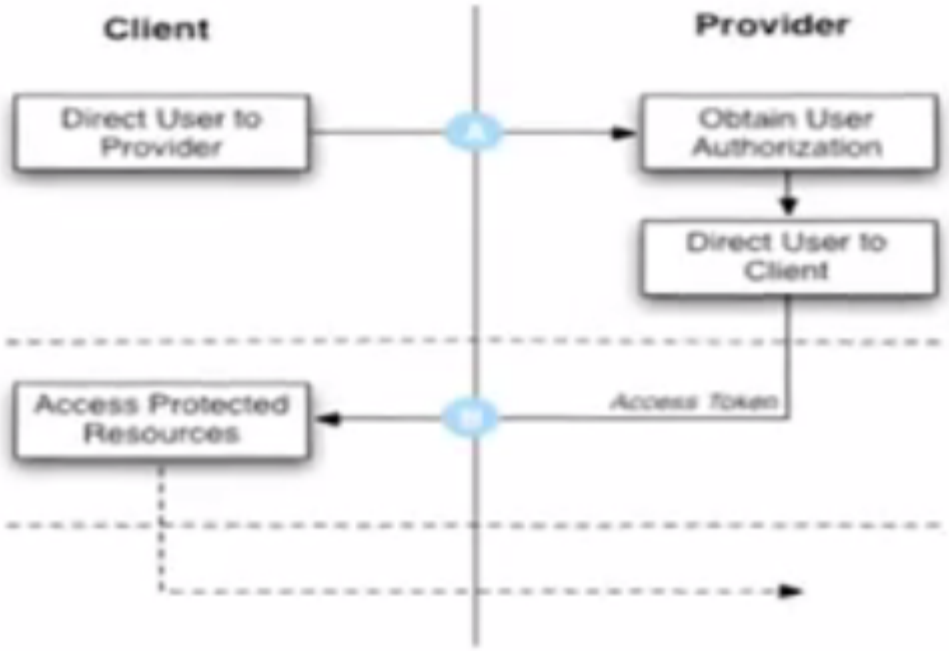
Describen el dialogo que se produce entre los distintos actores en distintos escenarios.

En **Oauth2** se describen 5 formas por las cuales los clientes pueden obtener el **AccessToken**

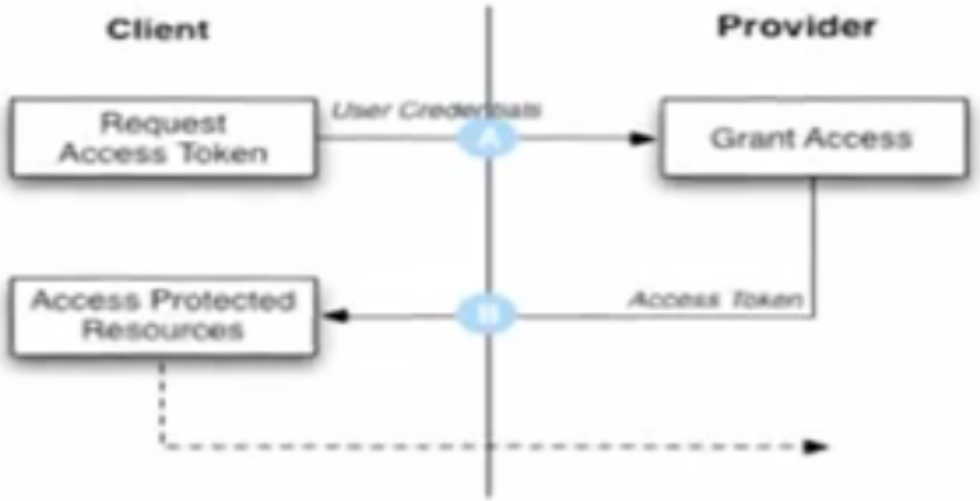
- **Authorization Code Grant** → Es el más habitual en las aplicaciones web.



- **Implicit Grant** → Aplicaciones Javascript, que reducen el dialogo.



- **Resource Owner Credential Grant** o **Password Grant** → Para aplicaciones muy confiables, dado que se les da el user/password, es el tipico caso de las aplicaciones en movilidad.



- **Client Credential Grant** → Empleada en comunicaciones entre aplicaciones, donde que el usuario epecifique los permisos no es necesario.

- **Refresh token grant** → Permite obtener un AccessToken nuevo cuando el antiguo caduca.

Authorization Code Grant

1- Se define el **servidor a autorizacion**.

2- Se define la aplicación **cliente**, que se ha de dar de alta en el **servidor de autorizacion**, obteniendo un **Id** y un **Secret**, que debe almacenar porque estos datos se los tendrá que enviar posteriormente al **servidor de autorizacion**.

3- El usuario se registra en el **servidor a autorizacion**.

4- El usuario accede a la aplicacion **cliente** y está lo redirecciona al **servidor de autorizacion**, enviando a dicho servidor los siguientes datos

- **response_type** → Indica que es lo que espera la aplicacion cliente del proceso de login del usuario en el **servidor de autorizacion**. En este caso se indica que se quiere un **autorization code**, para ello se indica el valor **code**
- **client_id** → con el identificador de la aplicacion cliente en el **servidor de autorizacion**
- **redirect_uri** → con la url a la que el **servidor de autorizacion** deberá redireccionar al usuario una vez terminado el login y aprovación de scopes.
- **scope** → listado de scopes separados por comas que la aplicacion cliente requiere al usuario.
- **state** → con el **CSRFToken**

Si el usuario aprueba los **scopes**, el **servidor de autorizacion** lo redirecciona al **redirect_uri** o a uno especificado por la aplicacion cliente de forma generica, enviando

- **code** → con el **authorization code**

- **state** → con el mismo **CSRFToken** que se recibio

5- Cuando la **aplicacion cliente** recibe la petición, realiza una peticion al **Servidor de Autenticacion** enviando

- **grant_type** → Indica en tipo de Grant a emplear, en este caso con valor **authorization_code**
- **client_id** → con el identificador de la aplicacion cliente en el **servidor de autorizacion**
- **client_secret** → con la clave secreta asociada al cliente
- **redirect_uri** → con la misma uri que paso por parametros al **servidor de autorizacion** cuando redirecciono al usuario
- **code** → con el **authorization code** que obtuvo de la anterior redireccion.

Y el **servidor de autorizacion** le retorna un JSON con

- **token_type** → con valor **Bearer**
- **expires_in** → con un entero que indica cuando expira el Token
- **access_token** → con el **accessToken**
- **refresh_token** → con un **refreshToken** que podrá ser empleado para obtener un nuevo **accessToken** cuando el original haya expirado

6- Por último se produce el acceso a los **Datos privados**, ya que la aplicacion **cliente** tiene el **accessToken**, le pide con este Token los datos privados al **Servidor de Recursos**, el cual se los da, dado que internamente es capaz de validarlo contra el **servidor de autorizacion**.

Implicit grant

Similar al anterior, pero dado que es el empleado por lo browser y estos no son capaces de asegurar guardar un **secret** de la aplicacion **cliente**, el flujo se simplifica, dado que en la redireccion del usuario hacia el **servidor de autorizacion** se obtendrá el **accesstoken**, no existen los **authorization code** y ni tampoco **refresh_token** dado que el browser no lo podria almacenar de forma segura.

1- Se define el **servidor a autorizacion**.

2- Se define la aplicación **cliente**, que se ha de dar de alta en el **servidor de autorizacion**, obteniendo un **Id**, que debe almacenar porque este dato se lo tendrá que enviar posteriormente al **servidor de autorizacion**.

3- El usuario se registra en el **servidor a autorizacion**.

4- El usuario accede a la aplicacion **cliente** y está lo redirecciona al **servidor de autorizacion**, enviando a dicho servidor los siguientes datos

- **response_type** con el valor **token**
- **client_id** con el id de la aplicacion cliente

- **redirect_uri** con la url a la que el **servidor de autorizacion** deberá redireccionar al usuario una vez terminado el login y aprovación de scopes.
- **scope** listado de scopes separados por comas que la aplicacion cliente requiere al usuario.
- **state** con el **CSRFToken**

Si el usuario aprueba los **scopes**, el **servidor de autorizacion** lo redirecciona a **redirect_uri** enviando

- **token_type** con el valor **Bearer**
- **expires_in** con un entero que indica cuando expira el Token
- **access_token** con el **accessToken**
- **state** con el mismo **CSRFToken** que se recibio

Resource owner credentials grant

Es una forma de obtener el **accessToken** en la que hay una completa confianza en la aplicación **cliente**, dado que se le dan **login/password**.

1- Se define el **servidor a autorizacion**.

2- Se define la aplicación **cliente**, que se ha de dar de alta en el **servidor de autorizacion**, obteniendo un **Id**, que debe almacenar porque este dato se lo tendrá que enviar posteriormente al **servidor de autorizacion**.

3- El usuario se registra en el **servidor a autorizacion**.

4- El usuario accede a la aplicacion **cliente** y está le pregunta cual es su **login/password**

5- La aplicacion cliente realiza una peticion al **servidor de autorizacion**, enviando a dicho servidor los siguientes datos

- **grant_type** con el valor **password**
- **client_id** con el identificador de la aplicacion cliente en el **servidor de autorizacion**
- **client_secret** con la clave secreta asociada al cliente
- **scope** listado de scopes separados por comas que la aplicacion cliente requiere al usuario.
- **username** el login obtenido del usuario
- **password** la contraseña obtenida del usuario

El servidor de autorizacion responderá un JSON con los datos

- **token_type** con valor **Bearer**
- **expires_in** con un entero que indica cuando expira el Token
- **access_token** con el **accessToken**

- **refresh_token** con un **refreshToken** que podrá ser empleado para obtener un nuevo **accessToken** cuando el original haya expirado

Client credentials grant

Es el mas simple de los **Grant** de OAuth2

1- Se define el **servidor a autorizacion**.

2- Se define la aplicación **cliente**, que se ha de dar de alta en el **servidor de autorizacion**, obteniendo un **Id** y un +secret*, que debe almacenar porque este dato se lo tendrá que enviar posteriormente al **servidor de autorizacion**.

3- La aplicacion **cliente** realiza una peticion POST al **servidor de autorizacion**, enviando a dicho servidor los siguientes datos

- **grant_type** con el valor **client_credentials**
- **client_id** con el identificador de la aplicacion cliente en el **servidor de autorizacion**
- **client_secret** con la clave secreta asociada al cliente
- **scope** listado de scopes separados por comas que la aplicacion cliente requiere al usuario.

El servidor de autorizacion responderá un JSON con los datos

- **token_type** con valor **Bearer**
- **expires_in** con un entero que indica cuando expira el Token
- **access_token** con el **accessToken**

Refresh token grant

Se emplea cuando se puede regenerar el **accessToken** sin la intervencion del usuario

1- La aplicacion **cliente** realiza una peticion POST al **servidor de autorizacion**, enviando a dicho servidor los siguientes datos

- **grant_type** con el valor **refresh_token**
- **refresh_token** con el **refreshToken** que se obtuvo con el ahora caducado **accessToken**
- **client_id** con el identificador de la aplicacion cliente en el **servidor de autorizacion**
- **client_secret** con la clave secreta asociada al cliente
- **scope** listado de scopes separados por comas que la aplicacion cliente requiere al usuario.

El servidor de autorizacion responderá un JSON con los datos

- **token_type** con valor **Bearer**
- **expires_in** con un entero que indica cuando expira el Token

- **access_token** con el **accessToken**
- **refresh_token** con un **refreshToken** que podrá ser empleado para obtener un nuevo **accessToken** cuando el original haya expirado

La configuracion a establecer se divide en tres,

- Definir como las aplicaciones clientes pueden acceder a los endpoints que permiten la gestion del **AccessToken**.
  - Obtencion de clave publica del certificado con el que se firma la información que provee el servidor, la cual se configura con el método **tokenKeyAccess** y responde a la url **/oauth/token_key**
  - Verificacion del **AccessToken** con el método **checkTokenAccess** y responde a la url **/oauth/check_token**

```
@Override
public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws Exception
{
    oauthServer
            .tokenKeyAccess("permitAll()")
            .checkTokenAccess("isAuthenticated()");
}
```

- Definir como el servidor OAuth gestiona los **AccessToken**, tanto para el almacenamiento, la transformación (encriptacion) y el acceso. Para esta configuración se precisa de la configuración de otros Beans
- **AuthenticationManager**, que será provisto por inyeccion por el contexto de Spring.
- **JwtAccessTokenConverter**, que será el encargado de definir como el servidor gestiona la encriptacion de los Token.
- **JwtTokenStore**, que será el encargado de definir como se almacenan los Token.

```
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception
{
    endpoints
            .authenticationManager(authenticationManager)
            .tokenStore(tokenStore())
            .accessTokenConverter(tokenEnhancer());
}

@Value("${config.oauth2.privateKey}")
private String privateKey;

@Value("${config.oauth2.publicKey}")
private String publicKey;

@Autowired
private AuthenticationManager authenticationManager;

@Bean
public JwtAccessTokenConverter tokenEnhancer() {
    log.info("Initializing JWT with public key:\n" + publicKey);
    JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
    converter.setSigningKey(privateKey);
    converter.setVerifierKey(publicKey);
    return converter;
}

@Bean
public JwtTokenStore tokenStore() {
    return new JwtTokenStore(tokenEnhancer());
}
```

Para esta configuracion se está empleando JWT (Json Web Token) para el almacen de los Token y su encriptado, para poder emplear este API, es necesario incluir la dependencia

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
</dependency>
```

Y en el caso del ejemplo definir una clave publica/privada como propiedades o tambien sería posible el uso de un certificado

```
config:
    oauth2:
        # openssl genrsa -out jwt.pem 2048
        # openssl rsa -in jwt.pem
        privateKey: |
            -----BEGIN RSA PRIVATE KEY-----

MIICXQIBAAKBgQDNQZKqTlO/+2b4ZdhqGJzGBDltb5PZmBz1ALN2YLvt341pH6i5

mO1V9cX5Ty1LM70fKfnIoYUP4KCE33dPnC7LkUwE/myh1zM6m8cbL5cYFPyP099t

hbVxzJkjHWqywvQih/qOOjliomKbM9pxG8Z1dB26hL9dSAZuA8xExjlPmQIDAQAB

AoGAImnYGU3ApPOVtBf/TOqLfne+2SZX96eVU06myDY3zA4rO3DfbR7CzCLE6qPn

yDAIiW0UQBs0oBDdWOnOqz5YaePZu/yrLyj6KM6Q2e9ywRDtDh3ywrSfGpjdSvvo

aeL1WesBWsgWv1vFKKvES7ILFLUxKwyCRC2Lgh7aI9GGZfECQQD84m98Yrehhin3

fZuRaBNIu348Ci7ZFZmrvyxAIxrV4jBjpACW0RM2BvF5oYM2gOJqIfBOVjmPwUro

bYEFcHRvAkEAz8jsfmxsZVwh3Y/Y47BzhKIC5FLaads541jNjVWfrPirljyCy1n4

sg3WQH2IEyap3WTP84+csCtsfNfyK7fQdwJBAJNRyobY74cupJYkW5OK4OkXKQQL

Hp2iosJV/Y5jpQeC3JO/gARcSmfIBbbI66q9zKjtmpPYUXI4tc3PtUEY8QsCQQCc

xySyC0sKe6bNzyC+Q8AVvkxiTKWiI5idEr8duhJd589H72Zc2wkMB+a2CEGo+Y5H

jy5cvuph/pG/7Qw7sljnAkAy/feClt1mUEiAcWrHRwcQ71AoA0+21yC9VkqPNrn3
            w7OEg8gBqPjRlXBNb00QieNeGGSkXOoU6gFschR22Dzy
            -----END RSA PRIVATE KEY-----

        # openssl rsa -in jwt.pem -pubout
        publicKey: |
            -----BEGIN PUBLIC KEY-----

MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDNQZKqTlO/+2b4ZdhqGJzGBDlt

b5PZmBz1ALN2YLvt341pH6i5mO1V9cX5Ty1LM70fKfnIoYUP4KCE33dPnC7LkUwE

/myh1zM6m8cbL5cYFPyP099thbVxzJkjHWqywvQih/qOOjliomKbM9pxG8Z1dB26
            hL9dSAZuA8xExjlPmQIDAQAB
            -----END PUBLIC KEY-----
```

Además de la configuracion de OAuth, se ha de configurar la propia del acceso de los usuarios al

NOTE	Recordad que hay dos tipos de login en el servidor de OAuth, el de los usuarios y el de las aplicaciones cliente.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ConfiguracionSeguridadWeb extends WebSecurityConfigurerAdapter {

    private static final Logger log = LoggerFactory.getLogger(ConfiguracionSeguridadWeb
.class);

    @Override
    @Autowired
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        log.info("Defining inMemoryAuthentication (2 users)");
        auth.inMemoryAuthentication()

                .withUser("user").password("password").roles("USER")

                .and()

                .withUser("admin").password("password").roles("USER", "ADMIN");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.formLogin()

                .and()

                .httpBasic().disable()
                .anonymous().disable()
                .authorizeRequests().anyRequest().authenticated();
    }
}
```

Implementacion Aplicacion Cliente OAuth con Spring

Se ha de añadir la misma dependencia que en el servidor.

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

El primer paso será definir la aplicación como Cliente OAuth

```
@Configuration
@EnableOAuth2Client
public class ConfiguracionOauth {}
```

El siguiente paso será definir un **Bean** de tipo **OAuth2RestOperations** que será el encargado de realizar las peticiones al **Servidor OAuth**

```
@Bean
public OAuth2RestOperations restTemplate(OAuth2ClientContext oauth2ClientContext) {
    return new OAuth2RestTemplate(resource(), oauth2ClientContext);
}

private OAuth2ProtectedResourceDetails resource() {
    AuthorizationCodeResourceDetails resource = new AuthorizationCodeResourceDetails();
    resource.setClientId(clientID);
    resource.setClientSecret(clientSecret);
    resource.setAccessTokenUri(accessTokenUri);
    resource.setUserAuthorizationUri(userAuthorizationUri);
    resource.setScope(Arrays.asList("read"));

    return resource;
}
```

Que será el objeto a emplear cuando la aplicación Cliente desee acceder a información privada

```
@RestController
public class UserController {

    @Autowired
    private OAuth2RestOperations restTemplate;

    @Value("${config.oauth2.resourceURI}")
    private String resourceURI;

    @RequestMapping("/")
    public JsonNode home() {
        return restTemplate.getForObject(resourceURI, JsonNode.class);
    }

}
```

Implementacion Servidor de Recursos OAuth con Spring

Como en el Servidor de OAuth, son necesarias las dependencias

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

Spring Boot

Introduccion

Framework orientado a la construcción/configuración de proyectos de la familia Spring basado en **Convention-Over-Configuration**, por lo que minimiza la cantidad de código de configuración de las aplicaciones.

Afecta principalmente a dos aspectos de los proyectos

- **Configuracion de dependencias**: Proporcionado por **Starters** Aunque sigue empleando Maven o Gradle para configurar las dependencias del proyecto, abstrae de las versiones de los APIs y lo que es más importante de las versiones compatibles de unos APIs con otros, dado que proporciona un conjunto de librerias que ya están probadas trabajado juntas.

- **Configuracion de los APIs**: Cada API de Spring que se incluye, ya tendrá una preconfiguración por defecto, la cual si se desea se podrá cambiar, además de incluir elementos tan comunes en los desarrollos como un contenedor de servlets embebido ya configurado, estas preconfiguraciones se establecen simplemente por el hecho de que la libreria esté en el classpath, como un Datasource de una base de datos, JDBCTemplate, Java Persistence API (JPA), Thymeleaf templates, Spring Security o Spring MVC.

Además proporciona otras herramientas como

- La consola Spring Boot CLI
- Actuator

Instalación de Spring Boot CLI

Permite la creación de aplicaciones Spring, de forma poco convencional, centrandose unicamente en el código, la consola se encarga de resolver dependencias y configurar el entorno de ejecución.

Emplea scripts de Groovy.

Para descargar la distribución pinchar aquí

Descomprimir y añadir a la variable entorno PATH la ruta **$SPRING_BOOT_CLI_HOME/bin**

Se puede acceder a la consola en modo ayuda (completion), con lo que se obtiene ayuda para escribir los comandos con TAB, para ello se introduce

```
> spring shell
```

Una vez en la consola se puede acceder a varios comandos uno de ellos es el de la ayuda general **help**

```
Spring-CLI# help
```

O la ayuda de alguno de los comandos

```
Spring-CLI# help init
```

Con Spring Boot se puede creear un poryecto MVC tan rapido como definir la siguiente clase Groovy **HelloController.groovy**

```
@RestController
class HelloController {

    @RequestMapping("/")
    def hello() {
        return "Hello World"
    }

}
```

Y ejecutar desde la consola Spring Boot CLI

```
> spring run HelloController.groovy
```

La consola se encarga de resolver las dependencias, de compilar y de establecer las configuraciones por defecto para una aplicacion Web MVC, en el web.xml, ... por lo que una vez ejecutado el comando de la consola, al abrir el navegador con la url http://localhost:8080 se accede a la aplicación.

Si se dispone de mas de un fichero **groovy**, se puede lanzar todos los que se quiera con el comando

```
> spring run *.groovy
```

El directorio sobre el que se ejecuta el comando es considerado el root del classpath, por lo que si se añade un fichero **application.properties**, este permite configurar el proyecto.

Si se quiere añadir motores de plantillas, se deberá incluir la dependencia, lo cual se puede hacer con **Grab**, por ejemplo para añadir **Thymeleaf**

```
@Grab(group='org.springframework.boot', module='spring-boot-starter-thymeleaf', version=
'1.5.7.RELEASE')

@Controller
class Application {
    @RequestMapping("/")
    public String greeting() {
        return "greeting"
    }
}
```

Y definir las plantillas en la carpeta **templates**, en este caso **templates/greeting.html**

Si se desea contenido estatico, este se debe poner en la carpeta **resources** o **static**

Creación e implementación de una aplicación

Lo primero a resolver al crear una aplicación son las dependencias, para ellos Spring Boot ofrece el siguiente mecanismo basando en la herencia del POM.

```
<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/xsd/maven-4.0.0.xsd">

    ...

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.2.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    ...

</project>
```

De no poderse establecer dicha herencia, por heredar de otro proyecto, se ofrece la posibilidad de añadir la siguiente dependedencia.

```
<project>

    ...

    <dependencyManagement>
         <dependencies>
            <dependency>
                <!-- Import dependency management from Spring Boot -->
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>1.4.2.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    ...

</project>
```

Esta dependencia permite a Spring Boot hacer el trabajo sucio para manejar el ciclo de vida de un proyecto Spring normal, pero normalmente se precisarán otras dependencias, para esto Spring Boot ofrece los **Starters**, por ejemplo esta seria la dependencia para un proyecto Web MVC

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

Una vez solventadas las dependencias, habrá que configurar el proyecto, ya hemos mencionado que la configuración quedará muy reducida, en este caso unicamnete necesitamos definir una clase anotada con **@SpringBootApplication**

```
@SpringBootApplication
public class HolaMundoApplication {
    ...
}
```

Esta anotacion en realidad es la suma de otras tres:

- @Configuration → Se designa a la clase como un posible origen de definiciones de Bean.

- @ComponentScan → Se indica que se buscarán otras clases con anotaciones que definan componentes de Spring como @Controller
- @EnableAutoConfiguration → Es la que incluye toda la configuración por defecto para los distintos APIs seleccionados.

Con esto ya se tendría el proyecto preparado para incluir unicamente el código de aplicación necesario, por ejemplo un Controller de Spring MVC

```
@Controller
public class HolaMundoController {
    @RequestMapping("/")
    @ResponseBody
    public String holaMundo() {
        return "Hola Mundo!!!!!";
    }
}
```

Una vez finalizada la aplicación, se podría ejecutar de varias formas

- Como jar autoejecutable, para lo que habrá que definir un método **Main** que invoque **SpringApplication.run()**

```
@SpringBootApplication
public class HolaMundoApplication {
    public static void main(String[] args) {
        SpringApplication.run(HolaMundoApplication.class, args);
    }
}
```

Y posteriormente ejecutandolo con

- Una tarea de Maven

```
mvn spring-boot:run
```

- Una tarea de Gradle

```
gradle bootRun
```

- O como jar autoejecutable, generando primero el jar

Con Maven

```
mvn package
```

O Gradle

```
gradle build
```

Y ejecutando desde la linea de comandos

```
java -jar HolaMundo-0.0.1-SNAPSHOT.jar
```

- O desplegando como WAR en un contenedor web, para lo cual hay que añadir el plugin de WAR
  - En Maven, con cambiar el package bastará

```
<packaging>war</packaging>
```

- En Gradle alicando el plugin de WAR y cambiando la configuracion JAR por la WAR

```
apply plugin: 'war'

war {
    baseName = 'HolaMundo'
    version = '0.0.1-SNAPSHOT'
}
```

En estos casos, dado que no se ha generado el **web.xml**, es necesario realizar dicha inicialización, para ello Srping Boot ofrece la clase **org.springframework.boot.web.support.SpringBootServletInitializer**

```
public class HolaMundoServletInitializer extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return builder.sources(HolaMundoApplication.class);
    }
}
```

Uso de plantillas

Los proyectos **Spring Boot Web** vienen configurados para emplear plantillas, basta con añadir el starter del motor deseado y definir las plantillas en la carpeta **src/main/resources/templates**.

Algunos de los motores a emplear son Thymeleaf, freemaker, velocity, jsp, ...

Thymeleaf

Motor de plantillas que se basa en la instrumentalización de **html** con atributos obtenidos del esquema **th**

```
<html xmlns:th="http://www.thymeleaf.org"></html>
```

Para añadir esta caracteristica al proyecto, se añade la dependencia de Maven

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Por defecto cualquier **String** retornado por un **Controlador** será considerado el nombre de un **html** instrumentalizado con **thymeleaf** que se ha de encontrar en la carpeta **/src/main/resources/templates**

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="ISO-8859-1"></meta>
    <title>Insert title here</title>
</head>
<body>
    <span th:text="${mensaje}"></span>
    <span th:text="#"></span>
</body>
</html>
```

NOTE No es necesario indicar el espacio de nombres en el html

JSP

Para poder emplear **JSP** en lugar de **Thymeleaf**, hay dos opciones, la primera es definir el proyecto de Spring Boot como War en el pom.xml, definiendo la siguiente configuracion en el contexto de Spring

```
@SpringBootApplication
public class SampleWebJspApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(SampleWebJspApplication.class);
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(SampleWebJspApplication.class, args);
    }

}
```

y las siguientes propiedades en el fichero **application.properties**

```
spring.mvc.view.prefix: /WEB-INF/views/
spring.mvc.view.suffix: .jsp
```

NOTE El directorio desde donde creacerá **WEB-INF**, sera **src/main/webapp**

La segunda opcion, será mantener el tipo de proyecto como Jar y añadir las siguientes dependencias al **pom.xml**

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
</dependency>
```

Por último indicar donde encontrar los ficheros mediante las siguientes propiedades en el fichero **application.properties**

```
spring.mvc.view.prefix: /WEB-INF/views/
spring.mvc.view.suffix: .jsp
```

NOTE El directorio desde donde creacerá **WEB-INF**, sera **src/main/resources/META-INF/resources/**

Recursos estaticos

Si se desean publicar recursos estaticos (html, js, css, ...), se pueden incluir en los proyectos en las rutas:

- **src/main/resources/META-INF/resources**
- **src/main/resources/resources**
- **src/main/resources/static**
- **src/main/resources/public**

Siendo el descrito el orden de inspeccion.

Webjars

Desde hace algun tiempo se encuentran disponibles como dependencias de Maven las distribuciones de algunos frameworks javascript bajo el groupid **org.webjars**, pudiendo añadir dichas dependencias a los proyectos para poder gestionar con herramientas de construccion como Maven o Gradle tambien las versiones de los frameworks javascript.

Estos artefactos tienen incluido los ficheros js, en la carpeta **/META-INF/resources/webjars/<artifactId>/<version>**, con lo que las dependencias hacia los ficheros javascript de los framework añadidos con Maven será **webjars/<artifactId>/<version>/<artifactId>.min.js**

```
<html>
<head>
    <script src="webjars/jquery/2.0.3/jquery.min.js"></script>
...
```

Recolección de métricas

El API de Actuator, permite recoger información del contexto de Spring en ejecución, como

- Qué beans se han configurado en el contexto de Spring.
- Qué configuraciones automáticas se han establecido con Spring Boot.
- Qué variables de entorno, propiedades del sistema, argumentos de la línea de comandos están disponibles para la aplicación.
- Estado actual de los subprocesos
- Rastreo de solicitudes HTTP recientes gestionadas por la aplicación
- Métricas relacionadas con el uso de memoria, recolección de basura, solicitudes web, y uso de fuentes de datos.

Estas metricas se exponen via Web o via shell.

Para activarlo, es necesario incluir una dependencia

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Y a partir de ahi, bajo la aplicación desplegada, se encuentran los path con la info, que retornan JSON

- Estado

http://localhost:8080/ListadoDeTareas/health

- Mapeos de URL

http://localhost:8080/ListadoDeTareas/mappings

- Descarga de estado de la memoria de la JVM

http://localhost:8080/ListadoDeTareas/heapdump

- Beans de la aplicacion

http://localhost:8080/ListadoDeTareas/beans

Se pueden configurar las funcionalidades para que sean privadas, modificando la propiedad **sensitive** del endpoint

```
endpoints:
  info:
    sensitive: true
```

Si son privadas, se necesitará configurar **Spring Security** para definir el origen de la autenticacion.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Una vez añadido, se han de configurar las siguientes propiedades

```
security.user.name=admin

security.user.password=secret

security.user.role=SUPERUSER

management.security.role=SUPERUSER
```

Para desactivar la seguridad

```
management.security.enabled=false
```

Endpoint Custom

Se pueden añadir nuevos EndPoints a la aplicación para que muestren algún tipo de información, para ello basta definir un Bean de Spring que extienda la clase **AbstractEndPoint**

```
@Component
public class ListEndPoints extends AbstractEndpoint<List<Endpoint>> {

    private List<Endpoint> endpoints;

    @Autowired
    public ListEndPoints(List<Endpoint> endpoints) {
        super("listEndpoints");
        this.endpoints = endpoints;
    }

    public List<Endpoint> invoke() {
        return this.endpoints;
    }
}
```

TIP	Solo esta implementacion, puede dar error, por encontrar valores en los Bean a Null, y el parser de Jackson no aceptarlo, para solventarlo, se puede definir en el application.properties la propiedad **spring.jackson.serialization.FAIL_ON_EMPTY_BEANS** a **false**

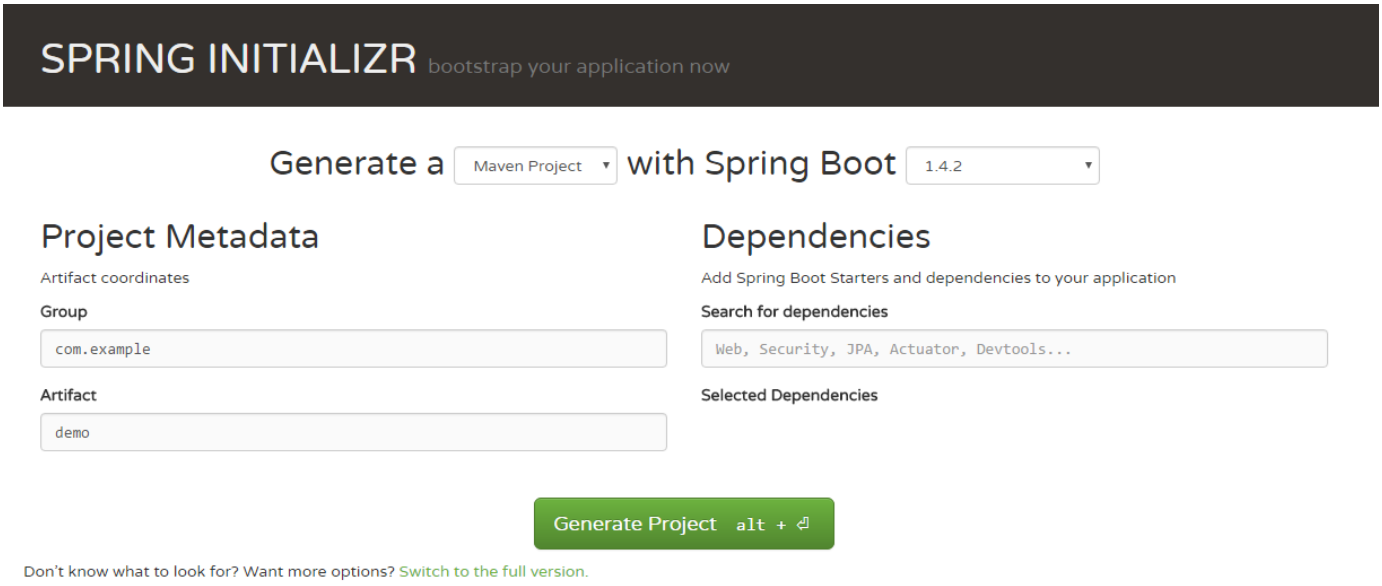
Uso de Java con start.spring.io

Es uno de los modos de emplear el API de **Spring Initializr**, al que tambien se tiene acceso desde

- Spring Tool Suite
- IntelliJ IDEA
- Spring Boot CLI

Es una herramienta que permite crear estructuras de proyectos de forma rapida, a través de plantillas.

Desde la pagina start.spring.io se puede generar una plantilla de proyecto.



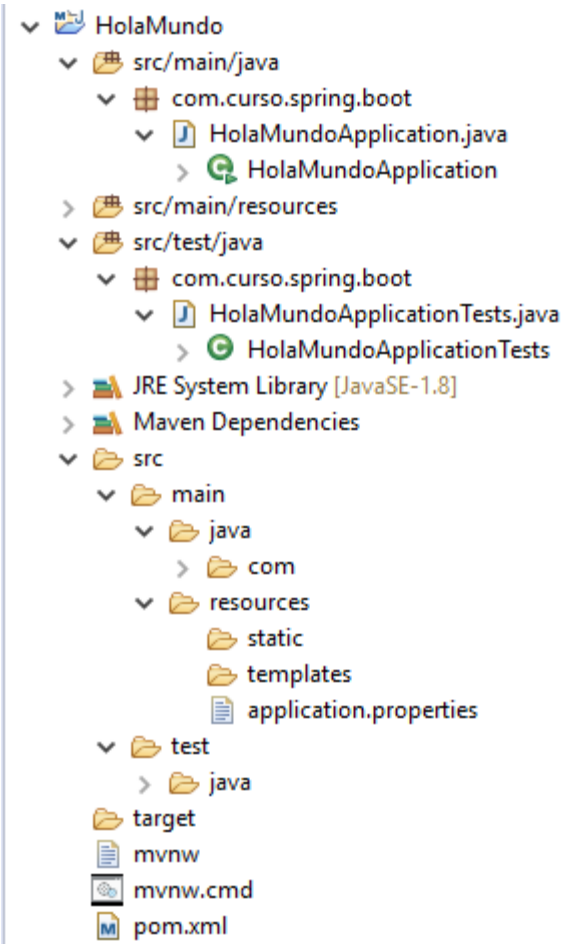
Lo que se ha de proporcionar es

- Tipo de proyecto (Maven o Gradle)
- Versión de Spring Boot
- GroupId
- ArtifactId
- Dependencias

Existe una vista avanzada donde se pueden indicar otros parametros como

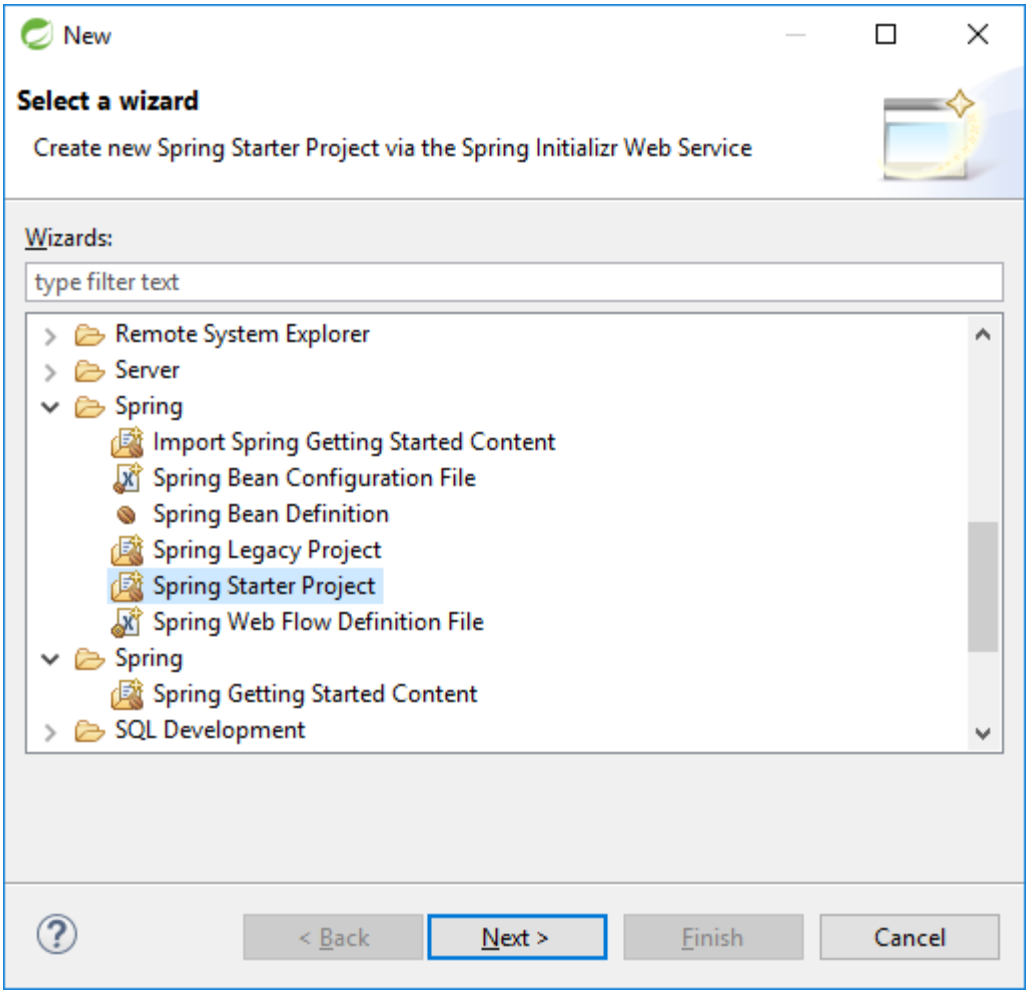
- Versión de java
- El tipo de packaging
- El lenguaje del proyecto
- Seleccion mas detallada de las dependencias

La estructura del proyecto con dependencia web generado será

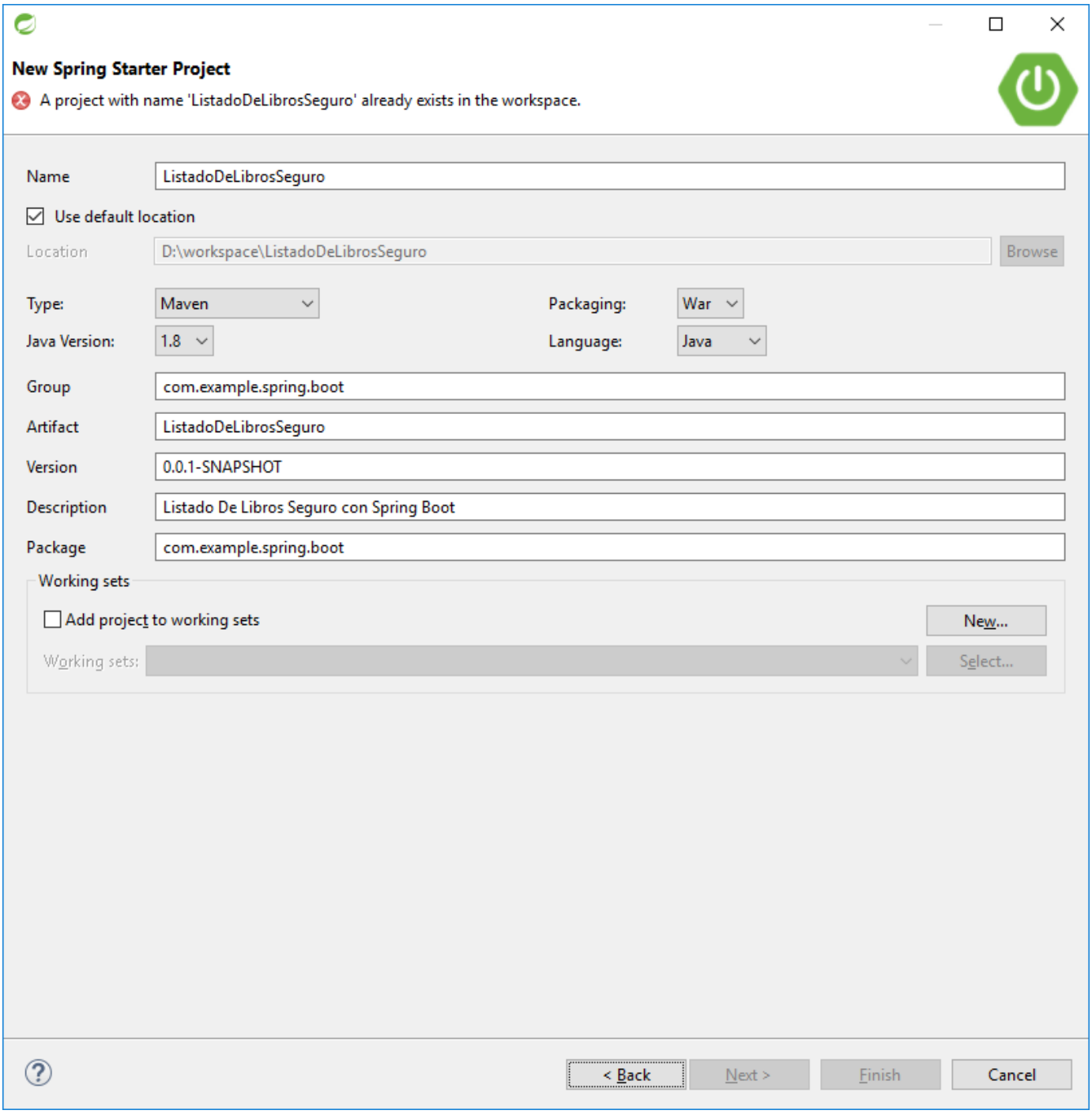


En esta estructura, cabe destacar el directorio **static**, destinado a contener cualquier recurso estatico de una aplicación web.

Desde Spring Tools Suite, se puede acceder a esta misma funcionalidad desde **New** > **Other** > **Spring** > **Spring Starter Project**, es necesario tener internet, ya que STS se conecta a **start.spring.io**



Una vez seleccionada la opcion, se muestra un ofrmulario similar al de la web



Y desde Spring CLI con el comando **init** tambien, un ejemplo de comando seria

```
Spring-CLI# init --build maven --groupId com.ejemplo.spring.boot.web --version 1.0 --java
-version 1.8 --dependencies web --name HolaMundo HolaMundo
```

Que genera la estructura anterior dentro de la carpeta **HolaMundo**

Se puede obtener ayuda sobre los parametros con el comando

```
Spring-CLI# init --list
```

Starters

Son dependencias ya preparadas por Spring, para dotar del conjunto de librerias necesarias para obtener un funcionalidad sin que existan conflictos entre las versiones de las distintas librerias.

Se pueden conocer las dependencias reales con las siguientes tareas

- Maven

```
mvn dependency:tree
```

- Gradle

```
gradle dependencies
```

De necesitarse, se pueden sobrescribir las versiones o incluso excluir librerias, de las que nos proporcionan los **Starter**

- Maven

```
<dependency>
    <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <exclusion>
                <groupId>com.fasterxml.jackson.core</groupId>
            </exclusion>
    </exclusions>
</dependency>
```

- Gradle

```
compile("org.springframework.boot:spring-boot-starter-web") {
    exclude group: 'com.fasterxml.jackson.core'
}
```

Soporte a propiedades

Spring Boot permite configurar unas 300 propiedades, aqui una lista de ellas.

Se pueden configurar los proyectos de Spring Boot unicamnete modificando propiedades, estas se pueden definir en

- Argumentos de la linea de comandos

```
java -jar app-0.0.1-SNAPSHOT.jar --spring.main.show-banner=false
```

- JNDI

```
java:comp/env/spring.main.show-banner=false
```

- Propiedades del Sistema Java

```
java -jar app-0.0.1-SNAPSHOT.jar -Dspring.main.show-banner=false
```

- Variables de entorno del SO

```
SET SPRING_MAIN_SHOW_BANNER=false;
```

- Un fichero **application.properties**

```
spring.main.show-banner=false
```

- Un fichero **application.yml**

```
spring:
    main:
        show-banner: false
```

Las listas en formato **YAML** tiene la siguiente sintaxis

```
security:
  user:
    role:
      - SUPERUSER
      - USER
```

De existir varias de las siguientes, el orden de preferencia es el del listado, por lo que la mas prioritaria es la linea de comandos.

Los ficheros **application.properties** y **application.yml** pueden situarse en varios lugares

- En un directorio **config** hijo del directorio desde donde se ejecuta la aplicación.
- En el directorio desde donde se ejecuta la aplicación.
- En un paquete **config** del proyecto
- En la raiz del classpath.

Siendo el orden de preferencia el del listado, si aparecieran los dos ficheros, el **.propertes** y el **.yml**, tiene prioridad el properties.

Algunas de las propiedades que se pueden definir son:

- **spring.main.show-banner** → Mostrar el banner de spring en el log (por defecto true).
- **spring.thymeleaf.cache** → Deshabilitar la cache del generador de plantillas thymeleaf
- **spring.freemarker.cache** → Deshabilitar la cache del generador de plantillas freemarker
- **spring.groovy.template.cache**→ Deshabilitar la cache de plantillas generadas con groovy
- **spring.velocity.cache**→ Deshabilitar la cache del generador de plantillas velocity
- **spring.profiles.active** → Perfil activado en la ejecución

TIP	La cache de las plantillas, se emplea en producción para mejorar el rendimiento, pero se debe desactivar en desarrollo ya que sino se ha de parar el servidor cada vez que se haga un cambio en las plantillas.

Configuracion del Servidor

- **server.port** → Puerto del Contenedor Web donde se exponen los recursos (por defecto 8080, para ssl 8443).
- **server.contextPath** → Permite definir el primer nivel del path de las url para el acceso a la aplicacion (Ej: /resource).
- **server.ssl.key-store** → Ubicación del fichero de certificado (Ej: file:///path/to/mykeys.jks).
- **server.ssl.key-store-password**→ Contraseña del almacen.

- **server.ssl.key-password** → Contraseña del certificado.

TIP

Para generar un certificado, se puede emplear la herramienta keytool* que incluye la jdk

```
keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA
```

Configuracion del Logger

- **logging.level.root**→ Nivel del log para el log principal (Ej: WARN)
- **logging.level.<paquete>**→ Nivel del log para un log particular (Ej: logging.level.org.springframework.security: DEBUG)
- **logging.path** → Ubicacion del fichero de log (Ej: /var/logs/)
- **logging.file** → Nombre del fichero de log (Ej: miApp.log)

Configuracion del Datasource

- **spring.datasource.url** → Cadena de conexión con el origen de datos por defecto de la auto-configuración (Ej: jdbc:mysql://localhost/test)
- **spring.datasource.username** → Nombre de usuario para conectar al origen de datos por defecto de la auto-configuración (Ej: dbuser)
- **spring.datasource.password** → Password del usuario que se conecta al origen de datos por defecto de la auto-configuración (Ej: dbpass)
- **spring.datasource.driver-class-name**→ Driver a emplear para conectra con el origen de datos por defecto de la auto-configuración (Ej: com.mysql.jdbc.Driver)
- **spring.datasource.jndi-name** → Nombre JNDI del datasorce que se quiere emplear como origen de datos por defecto de la auto-configuración.
- **spring.datasource.name**→ El nombre del origen de datos
- **spring.datasource.initialize**→ Whether or not to populate using data.sql (default:true)
- **spring.datasource.schema** → The name of a schema (DDL) script resource
- **spring.datasource.data**→ The name of a data (DML) script resource
- **spring.datasource.sql-script-encoding**→ The character set for reading SQL scripts
- **spring.datasource.platform**→ The platform to use when reading the schema resource (for example, "schema-{platform}.sql")
- **spring.datasource.continue-on-error**→ Whether or not to continue if initialization fails (default: false)
- **spring.datasource.separator**→ The separator in the SQL scripts (default: ;)
- **spring.datasource.max-active**→ Maximum active connections (default: 100)

- **spring.datasource.max-idle** → Maximum idle connections (default: 8)
- **spring.datasource.min-idle** → Minimum idle connections (default: 8)
- **spring.datasource.initial-size** → The initial size of the connection pool (default: 10)
- **spring.datasource.validation-query** → A query to execute to verify the connection
- **spring.datasource.test-on-borrow** → Whether or not to test a connection as it's borrowed from the pool (default: false)
- **spring.datasource.test-on-return** → Whether or not to test a connection as it's returned to the pool (default: false)
- **spring.datasource.test-while-idle** → Whether or not to test a connection while it is idle (default: false)
- **spring.datasource.max-wait** → The maximum time (in milliseconds) that the pool will wait when no connections are available before failing (default: 30000)
- **spring.datasource.jmx-enabled** → Whether or not the data source is managed by JMX (default: false)

TIP	Solo se puede configurar un unico datasource por auto-configuración, para definir otro, se ha de definir el bean correspondiente

Custom Properties

Se puede definir nuevas propiedades y emplearlas en la aplicación dentro de los Bean.

- Para ello se ha de definir, dentro de un Bean de Spring, un atributo de clase que refleje la propiedad y su método de SET

```
private String prefijo;
public void setPrefijo(String prefijo) {
    this.prefijo = prefijo;
}
```

- Para las propiedades con nombre compuesto, se ha de configurar el prefijo con la anotación **@ConfigurationProperties** a nivel de clase

```
@Controller
@RequestMapping("/")
@ConfigurationProperties(prefix="saludo")
public class HolaMundoController {}
```

- Ya solo falta definir el valor de la propiedad en **application.properties** o en **application.yml**

```yaml
saludo:
  prefijo: Hola
```

TIP	Para que la funcionalidad de properties funcione, se debe añadir **@EnableConfigurationProperties**, pero con Spring Boot no es necesario, ya que está incluido por defecto.

Otra opcion para emplear propiedades, es el uso de la anotacion **@Value** en cualquier propiedad de un bean de spring, que permite leer la propiedad si esta existe o asignar un valor por defecto en caso que no exista.

```java
@Value("${message:Hello default}")
private String message;
```

Profiles

Se pueden anotar **@Bean** con **@Profile**, para que dicho Bean sea solo añadido al contexto de Spring cuando el profile indicado esté activo.

```java
@Bean
@Profile("production")
public DataSource dataSource() {
    DataSource ds = new DataSource();
    ds.setDriverClassName("org.mysql.Driver");
    ds.setUrl("jdbc:mysql://localhost:5432/test");
    ds.setUsername("admin");
    ds.setPassword("admin");
    return ds;
}
```

Tambien se puede definir un conjunto de propiedades que solo se empleen si un perfil esta acivo, para ello, se ha de crear un nuevo fichero **application-{profile}.properties**.

En el caso de los ficheros de YAML, solo se define un fichero, el **application.yml**, y en el se definen todos los perfiles, separados por **---**

```yaml
---
spring:
    profiles: production
    datasource:
        url: jdbc:mysql://localhost:5432/test
        username: admin
        password: admin
        jpa:
            database-platform: org.hibernate.dialect.MySQLDialect
```

Para activar un **Profile**, se emplea la propiedad **spring.profiles.active**, la cual puede establecerse como:

- Variable de entorno

```
SET SPRING_PROFILES_ACTIVE=production;
```

- Con un parametro de inicio

```
java -jar applicacion-0.0.1-SNAPSHOT.jar --spring.profiles.active=production
```

TIP De definirse mas de un perfil activo, se indicaran con un listado separado por comas

JPA

Al añadir el starter de JPA, por defecto Spring Boot va a localizar todos los Bean detro del paquete y subpaquetes donde se encuentra la clase anotada con **@SpringBootApplication** en busca de interfaces Repositorio, que extiendan la interface **JpaRepository**, de no encontrarse la interface que define el repositorio dentro del paquete o subapaquetes, se puede referenciar con **@EnableJpaRepositories**

Cuando se emplea JPA con Hibernate como implementación, esté último tiene la posibilidad de configurar su comportamiento con respecto al schema de base de datos, pudiendo indicarle que lo cree, que lo actualice, que lo borre, que lo valide... esto se consigue con la propiedad **hibernate.ddl-auto**

```yaml
spring:
    jpa:
        hibernate:
            ddl-auto: validate
```

NOTE

Los posibles valores para esta propiedad son:

- none → This is the default for MySQL, no change to the database structure.
- update → Hibernate changes the database according to the given Entity structures.
- create → Creates the database every time, but don't drop it when close.
- create-drop → Por defecto para H2. the database then drops it when the SessionFactory closes.

Habrá que añadir al classpath, con dependencias de Maven, el driver de la base de datos a emplear, Spring Boot detectará el driver añadido y conectará con una base de datos por defecto.

La dependencia para MySQL será

```
 <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

NOTE

Las versiones de algunas depedencias no es necesario que se indiquen en Spring Boot, ya que vienen predefinidas en el **parent**

Para configurar un nuevo origen de datos, se indican las siguientes propiedades

```
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:mysql://localhost:3306/db_example
spring.datasource.username=springuser
spring.datasource.password=ThePassword
```

Errores

Por defecto Spring Boot proporciona una pagina para represenar los errores que se producen en las aplicaciones llamada **whitelabel**, para sustituirla por una personalizada, basta con definir alguno de los siguientes componentes

- Cuanlquier Bena que implemente **View** con Id **error**, que será resuelto por **BeanNameViewResolver**.
- Plantilla **Thymeleaf** llamada **error.html** si **Thymeleaf** esta configurado.
- Plantilla **FreeMarker** llamada **error.ftl** si **FreeMarker** esta configurado.
- Plantilla **Velocity** llamada **error.vm** si **Velocity** esta configurado.
- Plantilla **JSP** llamada **error.jsp** si se emplean vistas JSP.

Dentro de la vista, se puede acceder a la siguiente información relativa al error

- **timestamp** → La hora a la que ha ocurrido el error
- **status** → El código HTTP
- **error** → La causa del error
- **exception** → El nombre de la clase de la excepción.
- **message** → El mensaje del error
- **errors** → Los errores si hay mas de uno
- **trace** → La traza del error
- **path** → La URL a la que se accedia cuando se produjo el error.

Seguridad de las aplicaciones

Para añadir Spring security a un proyecto, habrá que añadir

- En Maven

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

- En Gradle

```
compile("org.springframework.boot:spring-boot-starter-security")
```

Al añadir Spring Security al Classpath, automaticamente Spring Boot, hace que la aplicación sea segura, nada es accesible.

Se creará un usuario por defecto **user** cuyo password e generará cada vez que se arranque la aplicación y se pintará en el log

```
Using default security password: ce9dadfa-4397-4a69-9fc7-af87e0580a10
```

Evidentemente esto es configurable, dado que cada aplicación, tendrá sus condiciones de seguridad, para establecer la configuración se puede añadir una nueva clase de configuración, anotada con **@Configuration** y además para que permita configurar la seguridad, debe estar anotada con **@EnableWebSecurity** y extender de **WebSecurityConfigurerAdapter**

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private ReaderRepository readerRepository;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/").access("hasRole('READER')")
            .antMatchers("/**").permitAll()
            .and()
            .formLogin()
            .loginPage("/login")
            .failureUrl("/login?error=true");
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .userDetailsService(new UserDetailsService() {
                @Override
                public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
                    return readerRepository.findOne(username);
                }
            });
    }
}
```

En esta clase, se puede configurar tanto los requisitos para acceder a recursos via web (autorización), como la vía de obtener los usuarios validos de la aplicación (autenticación), como otras configuraciones propias de la seguridad, como SSL, la pagina de login, ...

Soporte Mensajeria JMS

Para emplear JMS de nuevo Spring Boot, proporciona un starter, en este caso para varias tecnologias: ActiveMQ, Artemis y HornetQ

Para añadir por ejemplo ActiveMQ, se añadirá a dependencia Maven

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-activemq</artifactId>
    </dependency>
</dependencies>
```

Una vez Spring Boot encuentra en el classpath el jar de **ActiveMQ**, creará los objetos necesarios para conectarse al recurso JMS **Topic/Queue**, simplemente hará falta configurar las siguientes propiedades para inicar donde se encuentra el endpoint de **ActiveMQ**

- **spring.activemq.broker-url** → (Ej: tcp:/localhost:61616)
- **spring.activemq.user** → (EJ: admin)
- **spring.activemq.password** → (Ej: admin)

NOTE

Se espera que este configurado un endpoint de ActiveMQ, se puede descargar la distribución de aquí.

Para arrancarlo se ha de ejecutar el comando **/bin/activemq start** que levanta el servicio en local con los puertos **8161** para la consola administrativa y **61616** para la comunicacion de con los clientes.

El usuario y password por defecto son **admin/admin**

Como es habitual en las aplicaciones **Boot**, no será necesario añadir la anotacion **@EnableJms** a la clase de aplicación, ya que estará contemplada con **@SpringBootApplication**.

Consumidores

Para definir un Bean que consuma los mensajes del servicio JMS, se emplea la anotación **@JmsListener**

```
@Component
public class Receiver {
    @JmsListener(destination = "mailbox")
    public void receiveMessage(Email email) {
        System.out.println("Received <" + email + ">");
    }
}
```

NOTE Debera existir un **Queue** o **Topic** denominado **mailbox**

Productores

Para definir un Bean que envie mensajes se empleará un **Bean** creado por Spring de tipo **JmsTemplate**, empleando las funcionalidades **send** o **convertAndSend**.

```
@Component
public class MyBean {
    private JmsTemplate jmsTemplate;
    @Autowired
    public MyBean(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }
}
```

Se puede redefinir la factoria de contenedores

```
@Bean
public JmsListenerContainerFactory<?> myFactory(ConnectionFactory connectionFactory,

DefaultJmsListenerContainerFactoryConfigurer configurer) {
    DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory(
);
    // This provides all boot's default to this factory, including the message converter
    configurer.configure(factory, connectionFactory);
    // You could still override some of Boot's default if necessary.
    return factory;
}
```

Indicandolo posteriormente en los listener

```
@Component
public class Receiver {
    @JmsListener(destination = "mailbox", containerFactory = "myFactory")
    public void receiveMessage(Email email) {
        System.out.println("Received <" + email + ">");
    }
}
```

Soporte Mensajeria AMQP

AMQP es una especificacion de mensajeria asincrona, donde todos los bytes transmitidos son especificados, por lo que se pueden crear implementaciones en distintas plataformas y lenguajes.

La principal diferencia con JMS, es que mientras que en JMS los productores pueden publicar mensajes sobre: * **Queue** (un consumidor) * y **Topics** (n consumidores)

En AMQP solo hay **Queue** (un receptor), pero se incluye una capa por encima de estas **Queue**, los **Exchange**, que es donde se publican los mensajes por parte de los productores y estos **Exchange**, tienen la capacidad de publicar los mensajes que les llegan en una sola **Queue** o en varias, emulando los dos comportamientos de JMS.

Para trabajar con AMQP, se necesita un servidor de AMQP, como **RabbitMQ**, para instalarlo se necesita instalar Erlang a parte de RabbitMQ

Además de instalar **Erlang**, habra que definir a variable de entorno **ERLANG_HOME**.

La configuracion por defecto de **RabbitMQ** es escuchar por el puerto 5672

Receiver

Una vez instalado el Bus, se necesitará un proyecto que defina los **Receiver**, para ello se ha de incluir la dependencia

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Y añadir un **Bean** al contexto de Spring que haga de **Receiver**, no tiene porque implementar ningun API.

```
@Component
public class Receiver {
    public void receiveMessage(String message) {
        System.out.println("Received <" + message + ">");
    }
}
```

Este **Bean**, se ha de registrar como **Receiver** AMQP, para ello lo primero es indicar que método se ha de ejecutar, para ello se emplea la clase **MessageListenerAdapter**.

```
@Bean
public MessageListenerAdapter listenerAdapter(Receiver receiver) {
    return new MessageListenerAdapter(receiver, "receiveMessage");
}
```

En segundo lugar, hay que definir los objetos que definen la estructura del Broker

- Queue
- Exchange
- Binding

```
final static String queueName = "spring-boot";

@Bean
public Queue queue() {
    return new Queue(queueName, false);
}

@Bean
public TopicExchange exchange() {
    return new TopicExchange("spring-boot-exchange");
}

@Bean
public Binding binding(Queue queue, TopicExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).with(queueName);
}
```

Y por último asociar el **Receiver**, con la estructura del **Broker** a traves de un **ConnectionFactory**, que creará Spring gracias a los Beans anteriormente configurados

```
@Bean
public SimpleMessageListenerContainer container(ConnectionFactory connectionFactory,
MessageListenerAdapter listenerAdapter) {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
    container.setConnectionFactory(connectionFactory);
    container.setQueueNames(queueName);
    container.setMessageListener(listenerAdapter);
    return container;
}
```

Producer

Para producir nuevos mensajes, se emplea la clase **RabbitTemplate**, que permite enviar mensajes con métodos **convertAndSend**

```
rabbitTemplate.convertAndSend(ConfiguracionAMQP.queueName, "Hello from RabbitMQ!");
```

Este objeto, será creado por el contexto de Spring con los Bean que

Testing

Para realizar pruebas en las aplicaciones Spring Boot, se ha de añadir el starter de test

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

Y crear clases de Test, anotadas con

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SmokeTest {
}
```

La anotacion **@SpringBootTest**, crea el contexto de Spring empleando las clase aplicación de Spring Boot, aquella anotada con **@SpringBootApplication**, por lo que el contexto para las pruebas estará compuesto por los mismos beans que el de la aplicación.

Dado que el test tiene el mismo contexto que la aplicación, se puede obtener cualquier bean definido en el contexto de Spring para realizar el test,con la anotacion **@Autowired**

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SmokeTest {
    @Autowired
    private HomeController controller;
}
```

NOTE Una caracteritisca de estos Test, es que cachean el contexto de Spring a lo largo de la ejecucion de todos los metodos de testing, aunque esto se puede controlar con **@DirtiesContext**

NOTE

El Starter de Test, incluye la libreria **AssertJ**, que permite definir predicados para las aserciones, basandose en los métodos estaticos de la clase **org.assertj.core.api.Assertions**. Estos predicados son más legibles que los que se pueden definir con la libreria **JUnit** y los métodos estaticos de la clase **org.junit.Assert**

```
assertThat(this.restTemplate.getForObject("http://localhost:8080/", String
.class)).contains("Hello World");

assertThat(controller).isNotNull();
```

Testing Web

Para testear la capa Web de una aplicacion Spring Boot, se ha de configurar la propiedad **webEnvironment** de la anotacion **@SpringBootTest**, pudiendo tener los siguientes valores

- **WebEnvironment.DEFINED_PORT** → Para emplear la configuracion definida en el contexto de Spring
- **WebEnvironment.NONE** →
- **WebEnvironment.MOCK** →
- **WebEnvironment.RANDOM_PORT** → Genera un puerto de escucha de forma aleatoria, pudiendo obtener el puerto generado por inyeccion dentro de los Test

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HttpRequestTest {

    @LocalServerPort
    private int port;
}
```

Una vez configurada la propiedad **webEnvironment**, en el contexto de Spring se dispone de un Bean de tipo **TestRestTemplate**, que funciona de forma similar a **RestTemplate**.

```
@Autowired
private TestRestTemplate restTemplate;
```

Las pruebas anteriores, serían pruebas de humo o de integración total.

Se pueden definir Mocks de Beans ya existentes en el contexto, empleando dichos Mock en lugar de los Beans del contexto

```
@MockBean
private GreetingService service;

@Before
public void setup() {
    when(service.greet()).thenReturn("Hello Mock");
}
```

Esto puede tener sentido en momentos puntuales, dado que se siguen creando los Beans del contexto, no siendo empleados alguno de ellos, por lo que no es muy recomendable.

Pruebas de Integración

Se pueden realizar pruebas de integración parcial Mockeando el servidor, es decir sin levantar el servidor, para ello se ha de emplear la anotacion **@AutoConfigureMockMvc**, que define un nuevo Bean de Spring de tipo **MockMvc**, que permite realizar los accesos a la capa web.

```
@AutoConfigureMockMvc
public class ApplicationTest {

    @Autowired
    private MockMvc mockMvc;
}
```

El objeto **mockMvc** permite realizar acciones definiendolas con el método **perform()**, siendo las posibles acciones las definidas en la clase **org.springframework.test.web.servlet.request.MockMvcRequestBuilders**, que equivalen a los **METHOD HTTP**.

```
this.mockMvc.perform(get("/"));
```

Una vez realizada la acción, se puede acceder a los resultados de la acción para validarlos, para ello se emplean los **org.springframework.test.web.servlet.result.MockMvcResultHandlers**, que dan acceso al status, el contenido, las cookies, las cabeceras, ... de la respuesta de la teorica petición realizada.

```
this.mockMvc.perform(get("/"))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(content().string(containsString("Hello World")));
```

Pruebas Unitarias

Tambien se pueden realizar pruebas unitarias sobre la capa de controladores, de nuevo mockeando el servidor, y ademas mockeando la capa de servicios de la que dependen los controladores, para ello se evita arrancar todo el contexto de spring, centrandose en crear solo los Bean a inspecciona, los controladores y **Mocks** de la capa de Servicio, para ello se ha de definir la anotación **@WebMvcTest** en lugar de la tupla **@SpringBootTest** y **@AutoConfigureMockMvc**

```
@RunWith(SpringRunner.class)
@WebMvcTest
public class MockMvcTest {
}
```

La anotacion **@WebMvcTest** permite indicar que controladores se quieren instanciar para las pruebas, de no indicarse, se instanciarán todos los que se encuentren dentro del paquete base de la aplicación.

La capa de servicio se puede mockear empleando la anotacion **@MockBean**, que crea un objeto Mock de **Mockito**, el cual habrá que configurar.

```
@RunWith(SpringRunner.class)
@WebMvcTest
public class MockMvcTest {
    @MockBean
    private GreetingService service;

    @Before
    public void setup() {
        when(service.greet()).thenReturn("Hello Mock");
    }
}
```

Como en el caso anterior, se define un bean de tipo **MockMvc**.

Pruebas Integracion parcial: Capa de Persistenca con BD

Para la configuracion del contexto de ejecución de los repositorios de **JPA Data**, se tiene la anotacion **@DataJpaTest**

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class ExampleRepositoryTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository repository;

    @Test
    public void testExample() throws Exception {
        this.entityManager.persist(new User("sboot", "1234"));
        User user = this.repository.findByUsername("sboot");
        assertThat(user.getUsername()).isEqualTo("sboot");
        assertThat(user.getVin()).isEqualTo("1234");
    }

}
```

Rest

Los servicios REST son servicios basados en recursos, montados sobre HTTP, donde se da significado al Method HTTP.

La palabra REST viene de

- **Representacion**: Permite representar los recursos en multiples formatos, aunque el mas habitual es JSON.
- **Estado**: Se centra en el estado del recurso y no en las operaciones que se pueden realizar con el.
- **Transferencia**: Transfiere los recursos al cliente.

Los significados que se dan a los Method HTTP son:

- **POST**: Permite crear un nuevo recurso.
- **GET**: Permite leer/obtener un recurso existente.
- **PUT** o **PATCH**: Permiten actualizar un recurso existente.
- **DELETE**: Permite borrar un recurso.

Spring MVC, ofrece una anotacion **@RestController**, que auna las anotaciones **@Controller** y **@ResponseBody**, esta ultima empleada para representar la respuesta directamente con los objetos retornados por los métodos de controlador.

```
@RestController
@RequestMapping(path="/personas")
public class ServicioRestPersonaControlador {

    @RequestMapping(path="/{id}", method= RequestMethod.GET, produces=MediaType
.APPLICATION_JSON_VALUE)
    public Persona getPersona(@PathVariable("id") int id){
        return new Persona(1, "victor", "herrero", 37, "M", 1.85);
    }
}
```

De esta representación se encargan los **HttpMessageConverter**.

Personalizar el Mapping de la entidad

En transformaciones a XML o JSON, de querer personalizar el Mapping de la entidad retornada, se puede hacer empleando las anotaciones de JAXB, como son **@XmlRootElement**, **@XmlElement** o **@XmlAttribute**.

Estado de la petición

Cuando se habla de servicios REST, es importante ofrecer el estado de la petición al cliente, para ello se emplea el codigo de estado de HTTP.

Para incluir este codigo en las respuestas, se puede encapsular las entidades retornadas con **ResponseEntity**, el cual es capaz de representar tambien el codigo de estado con las constantes de **HttpStatus**

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<Spittle> spittleById(@PathVariable long id) {
    Spittle spittle = spittleRepository.findOne(id);
    HttpStatus status = spittle != null ? HttpStatus.OK : HttpStatus.NOT_FOUND;
    return new ResponseEntity<Spittle>(spittle, status);
}
```

Localización del recurso

En la creación del recurso, peticion POST, se ha de retornar en la cabera **location** de la respuesta la Url para acceder al recurso que se acaba de generar, siendo estas cabeceras retornadas gracias de nuevo al objeto **ResponseEntity**

```
HttpHeaders headers = new HttpHeaders();
URI locationUri = URI.create("http://localhost:8080/spittr/spittles/" + spittle.getId());
headers.setLocation(locationUri);
ResponseEntity<Spittle> responseEntity = new ResponseEntity<Spittle>(spittle, headers,
HttpStatus.CREATED)
```

Cliente se servicios con RestTemplate

Las operaciones que se pueden realizar con RestTemplate son

- **Delete** → Realiza una petición DELETE HTTP en un recurso en una URL especificada

```
public void deleteSpittle(long id) {
    RestTemplate rest = new RestTemplate();
    rest.delete(URI.create("http://localhost:8080/spittr-api/spittles/" + id));
}
```

- **Exchange** → Ejecuta un método HTTP especificado contra una URL, devolviendo un ResponseEntity que contiene un objeto mapeado del cuerpo de respuesta
- **Execute** → Ejecuta un método HTTP especificado contra una URL, devolviendo un objeto mapeado en el cuerpo de la respuesta.
- **GetForEntity** → Envía una solicitud HTTP GET, devolviendo un ResponseEntity que contiene un objeto mapeado del cuerpo de respuesta

```
public Spittle fetchSpittle(long id) {
    RestTemplate rest = new RestTemplate();
    ResponseEntity<Spittle> response = rest.getForEntity("http://localhost:8080/spittr-
api/spittles/{id}", Spittle.class, id);
    if(response.getStatusCode() == HttpStatus.NOT_MODIFIED) {
        throw new NotModifiedException();
    }
        return response.getBody();
    }
}
```

- **GetForObject** → Envía una solicitud HTTP GET, devolviendo un objeto asignado desde un cuerpo de respuesta

```
public Spittle[] fetchFacebookProfile(String id) {
    Map<String, String> urlVariables = new HashMap<String, String>();
    urlVariables.put("id", id);
    RestTemplate rest = new RestTemplate();
    return rest.getForObject("http://graph.facebook.com/{spitter}", Profile.class,
urlVariables);
}
```

- **HeadForHeaders** → Envía una solicitud HTTP HEAD, devolviendo los encabezados HTTP para los URL de recursos
- **OptionsForAllow** → Envía una solicitud HTTP OPTIONS, devolviendo el encabezado Allow URL especificada
- **PostForEntity** → Envia datos en el cuerpo de una URL, devolviendo una ResponseEntity que contiene un objeto en el cuerpo de respuesta

```
    RestTemplate rest = new RestTemplate();
    ResponseEntity<Spitter> response = rest.postForEntity("http://localhost:8080/spittr-
api/spitters", spitter, Spitter.class);
    Spitter spitter = response.getBody();
    URI url = response.getHeaders().getLocation();
}
```

- **PostForLocation** → POSTA datos en una URL, devolviendo la URL del recurso recién creado

```
public String postSpitter(Spitter spitter) {
    RestTemplate rest = new RestTemplate();
    return rest.postForLocation("http://localhost:8080/spittr-api/spitters", spitter)
.toString();
}
```

- **PostForObject** → POSTA datos en una URL, devolviendo un objeto mapeado de la respuesta cuerpo

```
public Spitter postSpitterForObject(Spitter spitter) {
    RestTemplate rest = new RestTemplate();
    return rest.postForObject("http://localhost:8080/spittr-api/spitters", spitter,
Spitter.class);
}
```

- **Put** → PUT pone los datos del recurso en la URL especificada

```
public void updateSpittle(Spittle spittle) throws SpitterException {
    RestTemplate rest = new RestTemplate();
    String url = "http://localhost:8080/spittr-api/spittles/" + spittle.getId();
    rest.put(URI.create(url), spittle);
}
```

Spring Security

La seguridad en Spring:

- Basada en otorgar el acceso.
- Jerárquica y perimetral. Aplica niveles.
- Transportable.

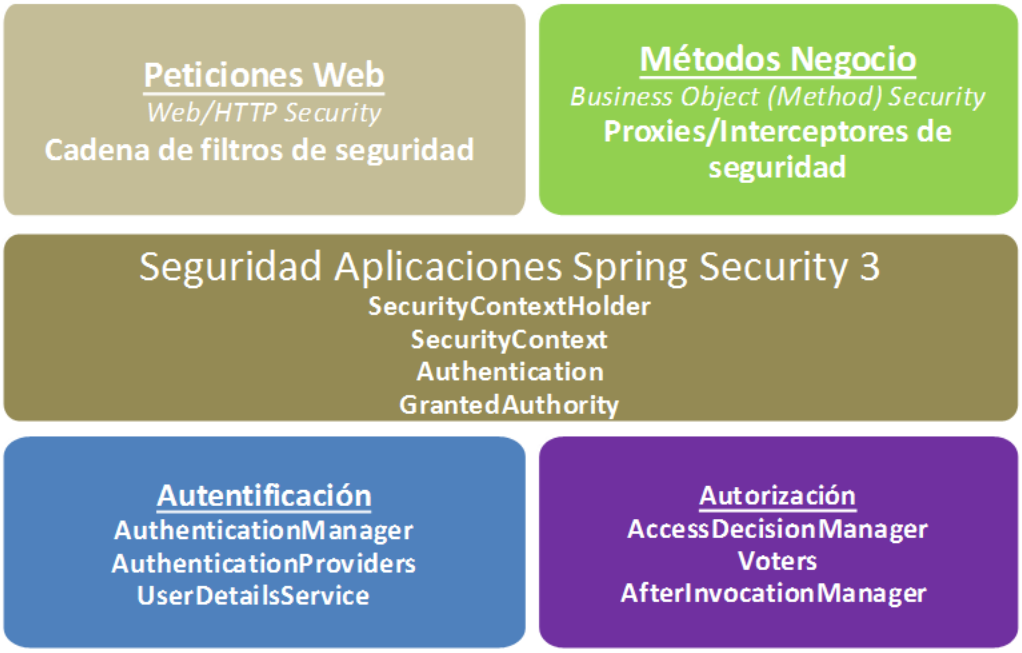
NOTE	Perimetral = estas dentro o no. Jerarquica = se pueden aplicar niveles de acceso a los contenidos.

La seguridad en JEE:

- Basada en restricciones.
- Perimetral.
- Dificil migrar.

NOTE	Restricciones = Todo es accesible hasta que se restringe el acceso. No es estandar dentro de los contenedores JEE, no es facil migrar.

Arquitectura



NOTE

Acceder a la app de fbi.war sin seguridad, se ve que accede hasta el fondo, a todas las funcionalidades sin restriccion.

localhost:8081/fbi → "mostrar expedientes" → "clasificar" → "desclasificar" → "mostrar"

Dependencias con Maven

Se han de añadir las siguientes dependencias al proyecto.

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-taglibs</artifactId>
    <version>3.2.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>3.2.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-crypto</artifactId>
    <version>3.2.4.RELEASE</version>
</dependency>
```

Filtro de seguridad

La seguridad con Spring, se basa en una clasede Spring **FilterChainProxy**, este filtro no será más que un Bean de Spring.

```
<bean id="springSecurityFilterChain" class=
"org.springframework.security.web.FilterChainProxy">
```

NOTE	El Bean de Spring de tipo **FilterChainProxy**, no es necesario definirlo directamente, ya que es una de las configuraciones por defecto que se añaden al activar la seguridad en Spring.

Sobre este Bean, delegará un Filtro Web especial, el **DelegatingFilterProxy**, que como su nombre indica, delega en el contexto de Spring lo que intercepta.

Con XML

```
<filter>
    <display-name>springSecurityFilterChain</display-name>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Con Java Config

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        // ...
        servletContext.addFilter("springSecurityFilterChain",
            new DelegatingFilterProxy("springSecurityFilterChain"))
                .addMappingForUrlPatterns(null, false, "/*");
        // ...
    }
}
```

Contexto de Seguridad

Se ha de añadir la anotación **@EnableWebSecurity** a una clase con **@Configuration** para que se genere el objeto **WebSecurityConfigurer**, que tiene la configuración por defecto de Spring Security.

Está configuración puede ser sobreescrita haciendo a su vez extender la clase **@Configuration** de **WebSecurityConfigurerAdapter**.

```
@Configuration
@EnableWebSecurity
public class ConfiguracionSpringSecurity extends WebSecurityConfigurerAdapter {

}
```

AuthenticationManagerBuilder

Permite definir de donde se han de obtener los usuarios y roles que se emplearán en la aplicación, para ello sobrescribir el método correspondiente de la clase **WebSecurityConfigurerAdapter**.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication().withUser("Fernando").password("1234").roles("AGENTE");
    auth.inMemoryAuthentication().withUser("Mulder").password("fox").roles(
"AGENTE_ESPECIAL");
    auth.inMemoryAuthentication().withUser("Scully").password("dana").roles(
"AGENTE_ESPECIAL");
    auth.inMemoryAuthentication().withUser("Skinner").password("walter").roles("DIRECTOR
");
}
```

NOTE	El objeto **AuthenticationManagerBuilder**, tiene un método **jdbcAuthentication()** que permite definir la conexion contra una base de datos para obtener los usuarios y roles.

Proteccion de recursos

Permite configurar las seguridad Web sobre las peticiones http.

Patrón de recursos protegido y rol que puede acceder.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/**").access("hasAnyRole('AGENTE_ESPECIAL','DIRECTOR')");
}
```

NOTE En este caso, esta limitado el aceso a cualquier recurso a aquellos usuarios que tienen como ROL = ROLE_AGENTE_ESPECIAL, además de que el proceso de Login, se realiza con formulario.

Para excluir recursos de la protección.

```
http
    .authorizeRequests()
        .antMatchers("/paginas/*").permitAll()
        .antMatchers("/css/*").permitAll()
        .antMatchers("/imagenes/*").permitAll();
```

Login

Para definir un proceso de Login a través de formulario con una pagina de login personalizada.

```
http
    .formLogin()
        .loginPage("/paginas/nuestro-login.jsp")
        .failureUrl("/paginas/nuestro-login.jsp?login_error");
```

NOTE Dado que cuando se intenta acceder a un recurso y este está asegurado, se nos redirigirá a la página de login, para conocer a que recurso se queria realmente acceder, Spring crea una **Session** temporal donde almacena la URL.

Logout

Para activar el logout indicando la url para realizar el logout, definir la pagina a la que se redirecciona una vez realizado el logout y el nombre de las cookies que se han de borrar en el proceso.

```
http
    .logout()
        .logoutUrl("/logout")                                                    2
        .invalidateHttpSession(true)
        .logoutSuccessUrl("/paginas/desconectado.jsp")
        .deleteCookies("JSESSIONID");
```

Por tanto para realizar el logout, basta con invocar la url **/logout**

```
<a href="<c:url value='/logout'/>">desconectar</a>
```

CSRF

El CSRF (Cross-site request forgery) o control de accesos desde sitio externos, permite mediante la adición de una huella aleatoria en las transacciones entre servidor y cliente, controlar que no se acceda al servidor desde otro sitio que no sea la propia aplicación.

Consiste en añadir un campo oculto para guardar el token en todos los formularios que utilicen el método POST:

```
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
```

Este Token, tambien se puede añadir con la libreria de etiquetas de Spring security:

```
<sec:csrfInput />
```

Este control se puede desactivar.

```
    http
        .csrf().disable();
```

UserDetailService

Permite personalizar la forma en la que se realiza la autenticación.

Spring proporciona implementaciones de referencia como **InMemoryUserDetailsManager** o **JdbcUserDetailsManager**

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService());
}

public UserDetailsService userDetailsService(){
    Properties usuarios = new Properties();
    usuarios.put("Fernando","1234,ROLE_AGENTE,enabled");
    usuarios.put("Mulder"  ,"fox,ROLE_AGENTE_ESPECIAL,enabled");
    usuarios.put("Scully"  ,"dana,ROLE_AGENTE_ESPECIAL,enabled");
    usuarios.put("Skinner" ,"walter,ROLE_DIRECTOR,enabled");

    return new InMemoryUserDetailsManager(usuarios);
}
```

Encriptación

Se trata de posibilitar el medio para que se almacenen las contraseñas encriptadas, pero que se sigan pudiendo resolver, para ello se ha de definir un **Encoder** y asociarlo al **AuthenticationManagerBuilder**.

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    PasswordEncoder encoder = new BCryptPasswordEncoder();
    auth.userDetailsService(userDetailsService()).passwordEncoder(encoder);
}

public UserDetailsService userDetailsService(){
    Properties usuarios = new Properties();
    usuarios.put("Fernando",
"$2a$10$SMPYtil7Hs2.cV7nrMjrM.dRAkuoLdYM8NdVrF.GeHfs/MrzcQ/zi,ROLE_AGENTE,enabled");
    usuarios.put("Mulder"  ,
"$2a$10$M2JRRHUHTfv4uMR4NWmCLebk1r9DyWSwCMZmuq4LKbImOkfhGFAIa,ROLE_AGENTE_ESPECIAL,enable
d");
    usuarios.put("Scully"  ,
"$2a$10$cbF5xp0grCOGcI6jZvPhA.asgmILATW1hNbM2MEqGJEFnRhhQd3ba,ROLE_AGENTE_ESPECIAL,enable
d");
    usuarios.put("Skinner" ,
"$2a$10$ZFtPIULMcxPe3r/5VunbVujMD7Lw8hkqAmJlxmK5Y1TK3L1bf8ULG,ROLE_DIRECTOR,enabled");

    return new InMemoryUserDetailsManager(usuarios);
}
```

Adicionalmente se puede realizar dicha configuración aprovechando una caracteristicas de las clases

anotadas con **@Configuration**, donde se ejecutan todos los métodos anotados con **@Autowired**, recibiendo la inyección de los parametros definidos.

```
@Bean
public PasswordEncoder passwordEncoder(){
    PasswordEncoder encoder = new BCryptPasswordEncoder();
    return encoder;
}

@Autowired
public void configureGlobalSecurity(AuthenticationManagerBuilder auth, PasswordEncoder
pe) throws Exception {
    auth.userDetailsService(userDetailsService()).passwordEncoder(pe);
}

public UserDetailsService userDetailsService(){
    Properties usuarios = new Properties();
    usuarios.put("Fernando",
"$2a$10$SMPYtil7Hs2.cV7nrMjrM.dRAkuoLdYM8NdVrF.GeHfs/MrzcQ/zi,ROLE_AGENTE,enabled");
    usuarios.put("Mulder"  ,
"$2a$10$M2JRRHUHTfv4uMR4NWmCLebk1r9DyWSwCMZmuq4LKbImOkfhGFAIa,ROLE_AGENTE_ESPECIAL,enable
d");
    usuarios.put("Scully"  ,
"$2a$10$cbF5xp0grCOGcI6jZvPhA.asgmILATW1hNbM2MEqGJEFnRhhQd3ba,ROLE_AGENTE_ESPECIAL,enable
d");
    usuarios.put("Skinner" ,
"$2a$10$ZFtPIULMcxPe3r/5VunbVujMD7Lw8hkqAmJlxmK5Y1TK3L1bf8ULG,ROLE_DIRECTOR,enabled");

    return new InMemoryUserDetailsManager(usuarios);
}
```

NOTE	Para la configuración anterior, no será necesario definir el método **configure(AuthenticationManagerBuilder auth)**, ya que puede ser sustituido por este otro.

Remember Me

Funcionalidad que permite incluir una **Cookie** para que la aplicación no pida al usuario que realice el proceso de **login**, recordandolo.

```
http
    .rememberMe()
        .rememberMeParameter("remember-me-param")
        .rememberMeCookieName("my-remember-me")
        .tokenValiditySeconds(86400);
```

Seguridad en la capa transporte - HTTPS

Se puede indicar a Spring que peticiones necesitan de un canal seguro (https), así como establecer redirecciones automaticas entre puertos.

```
http
    .requiresChannel()
        .anyRequest().requiresSecure()
.and()
    .portMapper()
        .http(8080).mapsTo(8443);
```

Para activar HTTPS, se ha de realizar configuraciones en el servidor, en el caso de un tomcat, se haria algo así.

En el fichero de configuración **server.xml** *copiar*

```
<Connector SSLEnabled="true" acceptCount="100"
    connectionTimeout="20000" executor="tomcatThreadPool"
    keyAlias="tcserver" keystoreFile="${catalina.base}/conf/tcserver.keystore"
    keystorePass="changeme"
    maxKeepAliveRequests="15" port="8443" protocol="HTTP/1.1"
    redirectPort="8443" scheme="https" secure="true"/>
```

Sesiones concurrentes

En el directorio **${catalina.base}/conf** *copiar un fichero de claves* **tcserver.keystore** *con los datos definidos en la configuración anterior.*

Se puede controlar la concurrencia de sesiones, permitiendo controlar que varios navegadores accedan con el mismo usuario. El procedimiento crea un contador, que cuando cumple con el numero especificado, no deja crear nuevas conexiones.

```
http
    .sessionManagement()
        .maximumSessions(1)
        .maxSessionsPreventsLogin(true);
```

Tiene un problema cuando se cierra el navegador y no se da a desconectar, ya que no se ejecuta la actualización del contador, luego cuando se llegue al máximo ya solo se podría acceder desde los navegadores que consiguieron el acceso.

Para solventar este problema, se dipone de un Listener que se encarga de está eventualidad.

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        servletContext.addListener(new HttpSessionEventPublisher());
    }
}
```

SessionFixation

Es un efecto que se puede producir en las aplicaciones web, que consiste en la sobrescritura de los roles en el objeto sesión.

La idea basicamente es que al realizarse el proceso de **login** sobre una sesión ya creada (repetir el login), sino se crea de nuevo un objeto session con su correspondiente identificador de sesión (JSESSIONID), es decir se recicla el objeto sesión, se puede producir que existan dos usuarios con la misma sesión, pero con los datos de roles del segundo usuario, que puede tener mas permisos.

Spring por defecto lo contempla.

```
http
    .sessionManagement()
        .sessionFixation()
        .migrateSession();
```

A mayores, se puede proporcionar una migración de los datos de la sesión antigua a la nueva, para poder seguir manteniendo información aunque se produzca el cambio de usuario.

Libreria de etiquetas

Para añadir las etiquetas propias de la seguridad se ha de incluir la cabecerá

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags"%>
```

Permite el acceso al objeto de authenticación para

1. Controlar que partes de la **View** se van a renderizar

```
<sec:authorize access="hasRole('ROLE_DIRECTOR')">
    <a href="<c:url value='/expedientesx/clasificar?id=${expediente.id}'/>">
clasificar</a>
    <a href="<c:url value='/expedientesx/desclasificar?id=${expediente.id}'/>"
>desclasificar</a>
</sec:authorize>
```

NOTE Este tipo de seguridad, unicamente protege que en el uso normal de la aplicación un usuario pueda ver un enlace o información, a la que no tiene acceso, para evitar errores de acceso, pero no protege la funcionalidad (negocio) en si.

1. Mostrar información del usuario

```
<sec:authentication property="principal.username" />
```

Expresiones SpEL

Para crear las expresiones que se han ido empleando, el lenguaje de expresiones de Spring (SpEL), ofrece una serie de comandos

- hasRole(role)
- hasAnyRole([role1,role2])
- permitAll
- denyAll
- isAnonymous()
- isAuthenticated()

Seguridad de métodos

Para activar la seguridad de los métodos, se ha de incluir la anotación **@EnableGlobalMethodSecurity** en la clase **@Configuration**

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity
public class ConfiguracionSpringSecurity extends WebSecurityConfigurerAdapter {}
```

Tambien es recomendable definir una página de error cuando se produzcan intentos de acceso no autorizados

```
http
    .exceptionHandling()
        .accessDeniedPage("/paginas/acceso-denegado.jsp");
```

Spring Security, es compatible con anotaciones propias y de la especificación java JSR-250. Las propias se dividen en dos grupos

- **@Secured** que se activa con **securedEnabled**
- **prepost** que se activa con **prePostEnabled**

Para activar las de JSR-250 habrá que activar **jsr250Enabled**.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true, jsr250Enabled=true, prePostEnabled =
true)
public class ConfiguracionSpringSecurity extends WebSecurityConfigurerAdapter {}
```

Las anotaciones de Spring son

- **@Secured**

```
@Secured("ROLE_AGENTE_ESPECIAL,ROLE_DIRECTOR")
void clasificar(Expediente expediente);
```

- **@PreAuthorize**

```
@PreAuthorize("hasRole('ROLE_DIRECTOR') or #expediente.investigador ==
authentication.name")
void desclasificar(Expediente expediente);
```

- **@PostAuthorize**

```
@PostAuthorize("hasRole('ROLE_DIRECTOR') or returnObject.investigador ==
authentication.name")
Expediente mostrar(Long id);
```

- **@PreFilter**
- **@PostFilter**

```
@PostFilter("(hasRole('ROLE_AGENTE') and not filterObject.clasificado) " +
    "or (hasAnyRole('ROLE_AGENTE_ESPECIAL','ROLE_DIRECTOR') and not
filterObject.informe.contains(principal.username))")
List<Expediente> listarTodos();
```

Las anotaciones de JSR-250 son

- **@RolesAllowed**

```
@RolesAllowed("ROLE_AGENTE_ESPECIAL,ROLE_DIRECTOR")
void desclasificar(Expediente expediente);
```

- PermitAll
- DenyAll
- RunAs