Spring v.6
Spring Boot v.3

Enlaces

- Spring:
  - https://spring.io/projects
- Spring Core
  - https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.htm
- Spring Data
  - https://docs.spring.io/spring-data/jpa/docs/2.1.5.RELEASE/reference/html/
- Spring MVC
  - https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html
- Spring HATEOAS
  - https://docs.spring.io/spring-hateoas/docs/0.25.1.RELEASE/reference/html
- Spring Data REST
  - https://docs.spring.io/spring-data/rest/docs/3.1.5.RELEASE/reference/html/
- Ejemplos:
  - https://github.com/spring-projects/spring-data-examples
  - https://github.com/spring-projects/spring-data-rest-webmvc
  - https://github.com/spring-projects/spring-hateoas-examples
  - https://github.com/spring-projects/spring-integration-samples

http://spring.io

SPRING CON SPRING BOOT

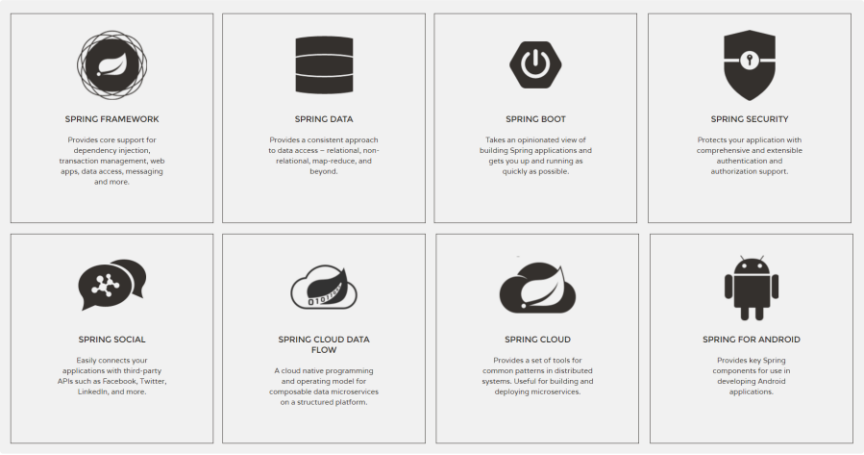
Spring

- Spring Framework es un marco de desarrollo de aplicaciones Java que facilita la creación de aplicaciones empresariales robustas y modulares al proporcionar un conjunto integral de funcionalidades, como inversión de control, contenedor de beans o gestión de transacciones. Utilizar Spring y Spring Boot proporciona ventajas como la flexibilidad, la modularidad, la escalabilidad y una amplia gama de características integradas que aceleran el desarrollo de aplicaciones empresariales.
- Inicialmente era un ejemplo hecho para el libro "J2EE design and development" de Rod Johnson en 2003, que defendía alternativas a la "visión oficial" de aplicación JavaEE basada en EJBs. Actualmente es un framework open source que facilita el desarrollo de aplicaciones java JEE & JSE (no esta limitado a aplicaciones Web ni a java).
- Provee de un contenedor encargado de manejar el ciclo de vida de los objetos (beans) para que los desarrolladores se enfoquen a la lógica de negocio. Permite integración con diferentes frameworks. Surge como una alternativa a EJB’s
- Actualmente es un framework completo compuesto por múltiples módulos/proyectos que cubre todas las capas de la aplicación, con decenas de desarrolladores y miles de descargas al día.

Características

- Ligero
  - No se refiere a la cantidad de clases sino al mínimo impacto que se tiene al integrar Spring.
- No intrusivo
  - Generalmente los objetos que se programan no tienen dependencias de clases específicas de Spring
- Flexible
  - Aunque Spring provee funcionalidad para manejar las diferentes capas de la aplicación (vista, lógica de negocio, acceso a datos) no es necesario usarlo para todo. Brinda la posibilidad de utilizarlo en la capa o capas que queramos.
- Multiplataforma
  - Escrito en Java, corre sobre JVM

Proyectos



Módulos necesarios

- Spring Framework
  - Spring Core
    - Contenedor IoC (inversión de control) - inyector de dependencia
  - Spring MVC
    - Framework basado en MVC para aplicaciones web y servicios REST
- Spring Data
  - Simplifica el acceso a los datos: JPA, bases de datos relacionales / NoSQL, nube
- Spring Boot
  - Simplifica el desarrollo de Spring: inicio rápido con menos codificación

Spring Boot

- Spring tiene una gran cantidad de módulos que implican multitud de configuraciones. Estas configuraciones pueden requerir mucho tiempo, pueden ser desconocidas para principiantes y suelen ser repetitivas.
- La solución de Spring es Spring Boot, que aplica el concepto de Convention over Configuration (CoC).
- CoC es un paradigma de programación que minimiza las decisiones que tienen que tomar los desarrolladores, simplificando tareas.
- No obstante, la flexibilidad no se pierde, ya que a pesar de otorgar valores por defecto, siempre se puede configurar de forma extendida.
- De esta forma se evita la repetición de tareas básicas a la hora de construir un proyecto.
- Spring Boot es una herramienta que nace con la finalidad de simplificar aun más el desarrollo de aplicaciones basadas en el framework Spring Core: que el desarrollador solo se centre en el desarrollo de la solución, olvidándose por completo de la compleja configuración que actualmente tiene Spring Core para poder funcionar.

Spring Boot

- Resolución de dependencias:
  - Con Spring Boot solo hay que determinar que tipo de proyecto estaremos utilizando y el se encarga de resolver todas las librerías/dependencias para que la aplicación funcione.
- Configuración:
  - Spring Boot cuenta con un complejo módulo que autoconfigura todos los aspectos de nuestra aplicación para poder simplemente ejecutar la aplicación, sin tener que definir absolutamente nada.
- Despliegue:
  - Spring Boot se puede ejecutar como una aplicación Stand-alone, pero también es posible ejecutar aplicaciones web, ya que es posible desplegar las aplicaciones mediante un servidor web integrado, como es el caso de Tomcat, Jetty o Undertow.

Spring Boot

- Métricas:
  - Por defecto, Spring Boot cuenta con servicios que permite consultar el estado de salud de la aplicación, permitiendo saber si la aplicación está encendida o apagada, memoria utilizada y disponible, número y detalle de los Bean's creado por la aplicación, controles para el prendido y apagado, etc.
- Extensible:
  - Spring Boot permite la creación de complementos, los cuales ayudan a que la comunidad de Software Libre cree nuevos módulos que faciliten aún más el desarrollo.
- Productividad:
  - Herramientas de productividad para desarrolladores como Spring Initializr, Lombok, LiveReload y Auto Restart, funcionan en su IDE favorito: Spring Tool Suite, IntelliJ IDEA y NetBeans.

Dependencias: starters

- Los starters son un conjunto de descriptores de dependencias convenientes (versiones compatibles, ya probadas) que se pueden incluir en la aplicación.
- Se obtiene una ventanilla única para el módulo de Spring y la tecnología relacionada que se necesita, sin tener que buscar a través de códigos de ejemplo y copiar/pegar cargas de descriptores de dependencia.
- Por ejemplo, si desea comenzar a utilizar Spring con JPA para un acceso CRUD a base de datos, basta con incluir la dependencia spring-boot-starter-data-jpa en el proyecto.

devtools

- Al realizar nuevos cambios en nuestra aplicación, podemos hacer que el arranque se reinicie automáticamente. Para eso es necesario incluir una dependencia Maven extra: spring-boot-devtools.
- Durante el tiempo de ejecución, Spring Boot supervisa la carpeta que se encuentra en classpath (en maven, las carpetas que están en la carpeta "target"). Solo necesitamos activar la compilación de las fuentes en los cambios que causarán la actualización de la carpeta 'destino' y Spring Boot reiniciará automáticamente la aplicación. Si estamos utilizando Eclipse IDE, la acción de guardar puede desencadenar la compilación.
- El módulo spring-boot-devtools incluye un servidor LiveReload incorporado que activa una actualización del navegador cuando se cambia un recurso. Las extensiones del navegador LiveReload están disponibles gratuitamente para Chrome, Firefox y Safari desde:
  http://livereload.com/extensions/
- Para configurar LiveReload:

```
spring.devtools.restart.additional-paths=META-INF/resources/**
spring.devtools.livereload.enabled=true # por defecto
```

Spring Tools

https://spring.io/tools

- Spring Tool Suite (STS) es un IDE basado en la versión Java EE de Eclipse, pero altamente personalizable para trabajar con Spring Framework.
  - IDE gratuito, personalización del Eclipse

- Spring Tools 4 es la próxima generación de herramientas Spring para los entorno de codificación favoritos. Proporciona soporte de primera clase para el desarrollo de aplicaciones empresariales basadas en Spring, ya sea que se prefiera Eclipse, Visual Studio Code o Theia IDE.
  - Help → Eclipse Marketplace …
    - Spring Tools 4 for Spring Boot

Spring Initializr

- Spring Initializr proporciona una API extensible para generar proyectos basados en JVM con implementaciones para varios conceptos comunes:
  - Generación de lenguaje básico para Java, Kotlin y Groovy.
  - Crea una abstracción del sistema con implementaciones para Apache Maven y Gradle.
  - Define el .gitignore de apoyo.
  - Define varios puntos de enganche para la generación de recursos personalizados.
- Las diversas opciones para los proyectos se expresan en un modelo de metadatos que permite configurar la lista de dependencias (starters), versiones de JVM y plataformas compatibles, etc.
- Spring Initializr también expone una aplicación web para generar un proyecto real mediante un interfaz de usuario y genera un fichero comprimido con el proyecto completamente configurado y personalizado.
  - https://start.spring.io/

Crear proyecto

- Desde web:
  - https://start.spring.io/
  - Descomprimir en el workspace
  - Import → Maven → Existing Maven Project
- Desde Eclipse:
  - New Project → Sprint Boot → Spring Started Project
- Dependencias
  - Web
  - JPA
  - JDBC (o proyecto personalizado)

pom.xml

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.0</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

- Con las etiquetas <parent></parent> se indica que nuestro POM hereda del de Spring Boot. La dependencia spring-boot-starter-web es la necesaria para poder empezar con un proyecto de tipo MVC, pero a medida que crece la aplicación se iran añadiendo más dependencias.
- Para alterar la configuración dada por defecto se ofrecen campos que se añadirán y asignaran en el archivo application.properties de la carpeta src/main/resources del proyecto.

Application

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

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

	@Override
	public void run(String... args) throws Exception {
		// Opcional: Procesar los args una vez arrancado SprintBoot

	}

}
```


Configuración

- @Configuration: Indica que esta es una clase usada para configurar el contenedor Spring.
- @ComponentScan: Escanea los paquetes de nuestro proyecto en busca de los componentes que hayamos creado, ellos son, las clases que utilizan las siguientes anotaciones: @Component, @Service, @Controller, @Repository.
- @EnableAutoConfiguration: Habilita la configuración automática, esta herramienta analiza el classpath y el archivo application.properties para configurar nuestra aplicación en base a las librerías y valores de configuración encontrados, por ejemplo: al encontrar el motor de bases de datos H2 la aplicación se configura para utilizar este motor de datos, al encontrar Thymeleaf se crearan los beans necesarios para utilizar este motor de plantillas para generar las vistas de nuestra aplicación web.
- @SpringBootApplication: Es el equivalente a utilizar las anotaciones: @Configuration, @EnableAutoConfiguration y @ComponentScan

Configuración

- Editar src/main/resources/application.properties:

```
server.port=${PORT:8080}
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=hr
spring.datasource.password=hr
spring.datasource.driver-class=oracle.jdbc.driver.OracleDriver

# MySQL settigs
spring.datasource.url=jdbc:mysql://localhost:3306/sakila
spring.datasource.username=root
spring.datasource.password=root
#spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n
logging.level.org.hibernate.SQL=debug
```

- Eclipse: Run Configurations → Arguments → VM Arguments: -DPORT=8888

Configuración del proxy: Maven

- Crear fichero setting.xml o editar %MAVEN_ROOT%/conf/setting.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
    <localRepository>C:\directorio\local\.m2\repository</localRepository>
    <proxies>
        <proxy>
        <id>optional</id>
        <active>true</active>
        <protocol>http</protocol>
        <username>usuario</username>
        <password>contraseña</password>
        <host>proxy.dominion.com</host>
        <port>8080</port>
        <nonProxyHosts>local.net|some.host.com</nonProxyHosts>
      </proxy>
    </proxies>
</settings>
```

- Referenciarlo en Window → Preferences → Maven → User setting → User setting, browse…, seleccionar fichero recién creado, aceptar, update setting, aplicar y cerrar.

Instalación con Docker

- Docker Toolbox
  - Windows 10 Pro ++: https://docs.docker.com/docker-for-windows/install/
  - Otras: https://github.com/docker/toolbox/releases
- Ejecutar Docker QuickStart
- Para crear el contenedor de MySQL con la base de datos Sakila:
  - docker run -d --name mysql-sakila -e MYSQL_ROOT_PASSWORD=root -p 3306:3306 jamarton/mysql-sakila
- Para crear el contenedor de MongoDB:
  - docker run -d --name mongodb -p 27017:27017 mongo
- Para crear el contenedor de Redis:
  - docker run --name redis -p 6379:6379 -d redis
  - docker run --name redis-insight -d -v redisinsight:/db -p 6380:8001 redislabs/redisinsight:latest
  - docker run -d --name redis-commander -p 8081:8081 rediscommander/redis-commander:latest
- Para crear el contenedor de RabbitMQ:
  - docker run -d --hostname rabbitmq --name rabbitmq -p 4369:4369 -p 5671:5671 -p 5672:5672 -p 15671:15671 -p 15672:15672 -p 25672:25672 -e RABBITMQ_DEFAULT_USER=admin -e RABBITMQ_DEFAULT_PASS=curso rabbitmq:management-alpine

https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.htm

IOC CON SPRING CORE

Construcción vs Uso

- Los sistemas de software deben separar el proceso de inicio, en el que se instancian los objetos de la aplicación y se conectan las dependencias, de la lógica de ejecución que utilizan las instancias. La separación de conceptos es una de las técnicas de diseño más antiguas e importantes.
- El mecanismo mas potente es la Inyección de Dependencias, la aplicación de la Inversión de Control a la gestión de dependencias. Delega la instanciación en un mecanismo alternativo, que permite la personalización, responsable de devolver instancias plenamente formadas con todas las dependencias establecidas. Permite la creación de instancias bajo demanda de forma transparente al consumidor.
- En lugar de crear una dependencia en sí misma, una parte de la aplicación simplemente declara la dependencia. La tediosa tarea de crear y proporcionar la dependencia se delega a un inyector que se encuentra en la parte superior.
- Esta división del trabajo desacopla una parte de la aplicación de sus dependencias: una parte que no necesita saber cómo configurar una dependencia, y mucho menos las dependencias de la dependencia, etc.

Inversión de Control

- Inversión de control (Inversion of Control en inglés, IoC) es tanto un concepto como unas técnicas de programación:
  - en las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales,
  - en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones.
- Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones.
- En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

Inyección de Dependencias

- Inyección de Dependencias (en inglés Dependency Injection, DI) es un patrón de arquitectura orientado a objetos, en el que se inyectan objetos a una clase en lugar de ser la propia clase quien cree el objeto, básicamente recomienda que las dependencias de una clase no sean creadas desde el propio objeto, sino que sean configuradas desde fuera de la clase.
- La inyección de dependencias (DI) procede del patrón de diseño más general que es la Inversión de Control (IoC).
- Al aplicar este patrón se consigue que las clases sean independientes unas de otras e incrementando la reutilización y la extensibilidad de la aplicación, además de facilitar las pruebas unitarias de las mismas.
- Desde el punto de vista de Java, un diseño basado en DI puede implementarse mediante el lenguaje estándar, dado que una clase puede leer las dependencias de otra clase por medio del API Reflection de Java y crear una instancia de dicha clase inyectándole sus dependencias.

Inyección

- La Inyección de Dependencias proporciona:
  - Código es más limpio
  - Quita la responsabilidad de buscar o crear objetos dependientes, dejando estos configurables. De esta forma, las búsquedas de componentes complejas, como es el uso de JNDI, pueden ser delegadas al contenedor.
  - Desacoplamiento es más eficaz, pues los objetos no deben de conocer donde están sus dependencias ni cuales son.
  - Reduce a cero las dependencias entre implementaciones, favoreciendo el uso de diseño de modelos de objetos basados en interfaces.
  - Facilidad en las pruebas unitaria e integración

JavaBean

- Los JavaBeans son un convenio para crear componentes de software reutilizables que se puedan manipular visualmente en una herramienta de construcción (IDE).
- Se usan para encapsular varios objetos en un único objeto (una vaina o Bean en inglés), para hacer uso de un solo objeto en lugar de varios más simples.
- Los componentes cuentan con propiedades (personalizan el aspecto y el comportamiento), eventos (notifica lo que está ocurriendo) y métodos.
- Un JavaBean:
  - Debe tener un constructor sin argumentos.
  - Sus atributos deben ser privados.
  - Sus propiedades tienen nombre y deben ser accesibles mediante métodos prefijados por get (is en los booleanos), función sin parámetros que devuelve su valor, y set, procedimiento con un parámetro que establece su valor.
  - Sus eventos tienen nombre con el sufijo Listener y deben tener los métodos prefijados por add y remove para establecer o retirar los delegados que actúan como controladores de eventos.
  - Debe ser serializables (implementar java.io.Serializable).

Spring Core IoC

- Spring proporciona un contenedor encargado de la inyección de dependencias (Spring Core Container).
- Este contenedor nos posibilita inyectar unos objetos sobre otros. Para ello, los objetos deberán ser simplemente JavaBeans.
- La inyección de dependencia (DI) es un proceso mediante el cual los objetos definen sus dependencias (es decir, los otros objetos con los que trabajan) sólo a través de argumentos del constructor, argumentos de un método de fábrica, atributos o propiedades que se establecen en la instancia del objeto después de que se construye o devuelto de un método de fábrica.
- La configuración podrá realizarse bien por anotaciones Java o mediante un fichero XML (XMLBeanFactory).
- Para la gestión de los objetos tendrá la clase (BeanFactory).
- Todos los objetos serán creados como singletons sino se especifica lo contrario.

Modulo de dependencias

- Se crea el fichero de configuración applicationContext.xml y se guarda en el directorio src/META-INF.

```
<beans xmlns:aop="http://www.springframework.org/schema/aop"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://www.springframework.org/schema/beans"
      xsi:schemalocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-2.5.xsd">
     <context:component-scan base-package="es.miEspacio.ioc.services">
     </context:component-scan>
</beans>
```

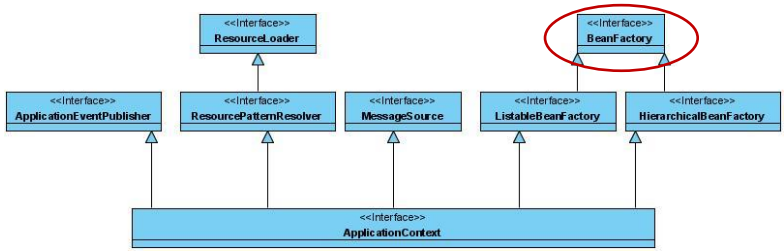

Beans

- Los beans se corresponden a los objetos reales que conforman la aplicación y que requieren ser inyectables: los objetos de la capa de servicio, los objetos de acceso a datos (DAO), los objetos de presentación (como las instancias Action de Struts), los objetos de infraestructura (como Hibernate SessionFactories, JMS Queues), etc.

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
   <!-- services -->
   <bean id="petStore" class="com.samples.PetStoreServiceImpl">
      <property name="accountDao" ref="accountDao"/>
      <property name="itemDao" ref="itemDao"/>
      <!-- additional collaborators and configuration for this bean go here -->
   </bean>
   <!-- more bean definitions for services go here -->
</beans>
```

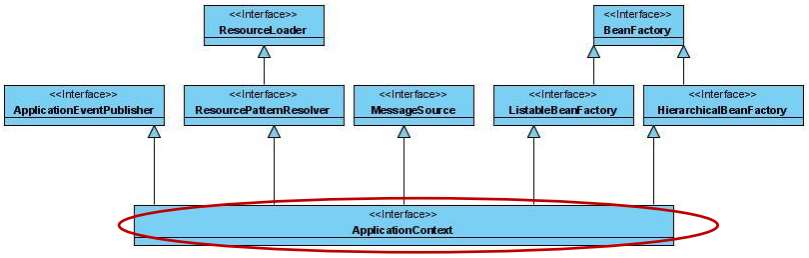
Bean factory

- Denominamos Bean Factory al contenedor Spring.
- Cualquier Bean Factory permite la configuración y la unión de objetos mediante la inyección de dependencia.
- Este Bean Factory también permite una gestión del ciclo de vida de los beans instanciados en él.
- Todos los contenedores Spring (Bean Factory) implementan el interface BeanFactory y algunos sub-interfaces para ampliar funcionalidades



Application Context

- Spring también soporta una "fábrica de beans" algo más avanzado, llamado contexto de aplicación.
- Application Context, es una especificación de Bean Factory que implementa la interface ApplicationContext.
- En general, cualquier cosa que un Bean Factory puede hacer, un contexto de aplicación también lo puede hacer.



Uso de la inyección de dependencias

- Se crea un inyector partiendo de un módulo de dependencias.
- Se solicita al inyector las instancias para que resuelva las dependencias.

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("META-
    INF/applicationContext.xml");
    // AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    BeanFactory factory = context;
    Client client = (Client )factory.getBean("ID_Cliente");
    client.go();
}
```

- Muestra:
  - Este es un servicio...

Anotaciones IoC

- Autodescubrimiento
  - @Component
  - @Repository
  - @Service
  - @Controller
  - @Scope
- Personalización
  - @Configuration
  - @Bean
- Inyección
  - @Autowired (@Inject)
  - @Qualifier (@Named)
  - @Value
  - @PropertySource
  - @~~Required~~
  - @Resource
- Otras
  - @PostConstruct
  - @PreDestroy

Estereotipos

- Spring define un conjunto de anotaciones core que categorizan cada uno de los componentes asociándoles una responsabilidad concreta.
  - @Component: Es el estereotipo general y permite anotar un bean para que Spring lo considere uno de sus objetos.
  - @Repository: Es el estereotipo que se encarga de dar de alta un bean para que implemente el patrón repositorio que es el encargado de almacenar datos en una base de datos o repositorio de información que se necesite. Al marcar el bean con esta anotación Spring aporta servicios transversales como conversión de tipos de excepciones.
  - @Service : Este estereotipo se encarga de gestionar las operaciones de negocio más importantes a nivel de la aplicación y aglutina llamadas a varios repositorios de forma simultánea. Su tarea fundamental es la de agregador.
  - @Controller : El último de los estereotipos que es el que realiza las tareas de controlador y gestión de la comunicación entre el usuario y el aplicativo. Para ello se apoya habitualmente en algún motor de plantillas o librería de etiquetas que facilitan la creación de páginas.
  - @RestController que es una especialización de controller que contiene las anotaciones @Controller y @ResponseBody (escribe directamente en el cuerpo de la respuesta en lugar de la vista).

Configuración por código

- Hay que crear una (o varias) clase anotada con @Configuration que contendrá un método que devuelva cada clase/interfaz (sin estereotipo) que se quiera tratar como un Bean inyectable.
- @Bean indica que un método produce un bean para ser administrado por el contenedor Spring, el método se debería llamar como la clase en notación Camel y devolver la instancia ya creada del tipo de la clase a inyectar. Adicionalmente se puede anotar con @Scope, @Lazy, @DependsOn, @Primary @Qualifier y con @Profile.

```
public class MyBean { … }

@Configuration
public class MyConfig {
    @Bean
    @Scope("prototype")
    public MyBean myBean() { … ]
```

Alcance

- Un aspecto importante del ciclo de vida de los Beans es si el contenedor creara una única instancia o tantas como ámbitos sean necesarios.
  - prototype: No reutiliza instancias, genera siempre una nueva instancia. @Scope("prototype")
  - singleton: (Predeterminado) Instancia única para todo el contenedor Spring IoC. @Scope("singleton") @Singleton
  - Adicionalmente, en el contexto de un Spring Web ApplicationContext: @RequestScope @SessionScope @ApplicationScope
    - request: Instancia única para el ciclo de vida de una sola solicitud HTTP. Cada solicitud HTTP tiene su propia instancia única.
    - session: Instancia única para el ciclo de vida de cada HTTP Session.
    - application: Instancia única para el ciclo de vida de un ServletContext.
    - websocket: Instancia única para el ciclo de vida de un WebSocket.

Inyección basada en constructores

- La inyección basada en constructores se logra cuando el contenedor invoca un constructor con varios argumentos, cada uno de los cuales representa una dependencia. La coincidencia de resolución de argumentos del constructor se produce utilizando el tipo de argumento.

```
@Service
public class ServiceImpl implements Service {
    private Repository dao;
    public ServiceImpl(Repository dao) {
```

- Con la anotación @ConstructorProperties se pueden asociar los nombres de los beans a inyectar:

```
@Configuration
public class MyConfig {
    @Bean int version() { return 2; }
    @Bean() String autor() { return "Yo mismo"; }
:
@Component
public class EjemplosIoC {
    @ConstructorProperties({"version", "autor"})
    public EjemplosIoC(int version, String autor) {
```

Inyección automática

- El contenedor Spring puede conectar automáticamente (Autowired) las relaciones entre beans colaboradores que tiene las siguientes ventajas:
  - permitir la inyección de atributos, setter y métodos
  - reducir significativamente la necesidad de especificar argumentos del constructor
  - actualizar la configuración a medida que los objetos evolucionan sin introducir cambios rupturista
- Esto permite que atributos y propiedades sean inyectados de forma automática después de la inyección del constructores.
- Los valores que puede tomar son los siguientes:
  - byName: Identifica a los beans a través de su propiedad "name".
  - byType: Identifica a los beans a través del tipo.
  - constructor: Encaja con los tipos de argumentos del constructor.
  - autodetect: Primero lo intenta por constructor y luego por tipo.

Inyección automática

- La inyección se realiza con la anotación @Autowired:
  - En atributos:

    ```
    @Autowired
    private MyBeans myBeans;
    ```
  - En propiedades (setter):

    ```
    @Autowired
    public void setMyBeans(MyBeans value) { … }
    ```
  - En constructores
- Por defecto la inyección es obligatoria, se puede marcar como opcional en cuyo caso si no encuentra el Bean inyectará un null.

  ```
  @Autowired(required=false)
  private MyBeans myBeans;
  ```
- Se puede completar @Autowired con la anotación @Lazy para inyectar un proxy de resolución lenta.

Resolución de dependencia

- Si en el contenedor no hay al menos un bean que resuelva la dependencia se genera un error fatal si no se ha marcado la dependencia como opcional.
- Si existe más de uno, también se lanza una excepción fatal, salvo que se indique como elegir el candidato apropiado.
- @Primary indica que se debe dar preferencia a un bean cuando varios candidatos están calificados para conectar automáticamente una dependencia de un solo valor. Si existe exactamente un bean "primario" entre los candidatos, será el valor cableado automáticamente.

```
@Service
@Primary
public class ServiceImpl implements Service {
```

- Cuando se necesite más control sobre el proceso de selección, están disponibles @Qualifier y @Profile.

Cualificación

- Con @Qualifier (@Named) se pueden agrupar o cualificar los beans asociándoles un nombre:

```
public interface MyInterface { ... }

@Component
@Qualifier("old")
public class MyInterfaceImpl implements MyInterface { ... }

@Qualifier("new")
@Component
public class MyNewInterfaceImpl implements MyInterface { ... }

@Autowired(required=false)
@Qualifier("new")
private MyInterface srv;
```

Perfiles

- Spring Profiles proporciona una forma de segregar partes de la configuración de la aplicación y hacer que esté disponible solo en ciertos entornos. Los perfiles se pueden establecer con anotación @Profile y con ficheros de propiedades.
- Los nombres de los perfiles son de libre elección, por ejemplo: dev, test, staging, production o prod. Si no hay ningún perfil activo, se habilita un perfil predeterminado que por defecto se llama default, pero se puede cambiar en application.properties con la propiedad:

```
spring.profiles.default=none
```

- Un perfil es un grupo lógico con nombre de definiciones de beans que se registrarán en el contenedor solo si el perfil dado está activo, proporcionan un mecanismo en el contenedor central que permite el registro de diferentes beans en diferentes entornos.

```
@Service
@Profile("default")
public class ServicioImpl { … }

@Service
@Profile("test")
public class ServicioMockImpl { … }

@Bean@Profile("test")
Servicio getServicio() { … }
```


@Profile

- Cualquier @Component, @Configuration o @ConfigurationProperties se puede marcar con @Profile para limitar cuando se carga.

```
public interface MiPerfil { … }

@Component
@Profile("default")
public class MiPerfilImpl implements MiPerfil { … }

@Component
@Profile("dev")
public class MiPerfilDev implements MiPerfil { … }

@Component
@Profile("production")
public class MiPerfilProd implements MiPerfil { … }

@Autowired
private MiPerfil perfil;
```

- En los nombres de perfil también se puede anteponer con un operador NOT (por ejemplo, !dev) para excluirlos de un perfil.

Activar perfiles

- Hay indicarle a Spring qué perfil está activo, o por defecto utilizara el perfil "default".
- El perfil activo se puede establecer en application.properties, se pueden pasar a través de un parámetro del sistema JVM (Eclipse: Run Configurations → Arguments → Program Arguments) o a través de la variable de entorno:

```
spring.profiles.active=production
-Dspring.profiles.active=dev
export spring_profiles_active=staging
```

- Los perfiles se pueden activar por código:

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("test");
ctx.refresh();
```

- También podemos establecer perfiles directamente en el entorno:

```
@Autowired
private ConfigurableEnvironment env;
...
env.setActiveProfiles("dev");
```

- Las pruebas hacen que sea muy fácil especificar qué perfiles están activos usando la anotación @ActiveProfile para habilitar perfiles específicos:

```
@ActiveProfiles("test")
```


Archivos específicos del perfil

- Además de los archivos de propiedades application.properties o application.yml, Spring Boot también intentará cargar archivos específicos del perfil utilizando la convención de nomenclatura application-{profile}:
  - application-dev.properties
  - application-production.yml
- Las propiedades específicas del perfil se cargan desde las mismas ubicaciones que las estándar application.properties, y los archivos específicos del perfil siempre reemplazan a los no específicos. Si se especifican varios perfiles, se aplica una estrategia de el último gana.

Inyección de valores

- Para recuperar un valor de la configuración:

```
@Value("${mi.valor}")
String miValor;
```

- Para recuperar y crear un componente:

```
// En el fichero .properties
// rango.min=1
// rango.max=10
@Data
@Component
@ConfigurationProperties("rango")
public class Rango {
   private int min;
   private int max;
}
@Autowired
private Rango rango;
```


Acceso a ficheros de propiedades

- Localización (fichero .properties, .yml, .xml):
  - Por defecto: src/main/resources/application.properties
  - En la carpeta de recursos src/main/resources:
    @PropertySource("classpath:my.properties")
  - En un fichero local:
    @PropertySource("file://c:/cng/my.properties")
  - En una URL:
    @PropertySource("http://myserver/application.properties")
- Acceso a través del entorno:

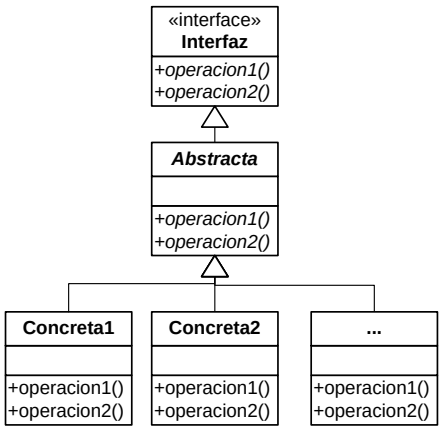
```
@Autowired private Environment env;
env.getProperty("spring.datasource.username")
```

Ciclo de Vida

- Con la inyección el proceso de creación y destrucción de las instancias de los beans es administrada por el contendor.
- Para poder intervenir en el ciclo para controlar la creación y destrucción de las instancias se puede:
  - Implementar las interfaces InitializingBean y DisposableBean de devoluciones de llamada
  - Sobrescribir los métodos init() y destroy()
  - Anotar los métodos con @PostConstruct y @PreDestroy.
- Se pueden combinar estos mecanismos para controlar un bean dado.

Patrón: Doble herencia

- Problema:
  - Mantener las clases que implementan como internas del proyecto (internal o de paquete), pero la interfaz pública.
  - Organizar clases que tienen un comportamiento parecido para que sea consistente.
- Solución:
  - Clase base es abstracta.
  - La clase base puede heredar de mas de una interfaz.
  - Una vez que están escritos los métodos, verifico si hay duplicación en las clases hijas.



Patrón: Doble herencia

- Se crea el interfaz con la funcionalidad deseada:

```
public interface Service {
   public void go();
}
```

- Se implementa la interfaz en una clase (por convenio se usa el sufijo Impl):

```
import org.springframework.stereotype.Service;
@Service
@Singleton
public class ServiceImpl implements Service {
   public void go() {
      System.out.println("Este es un servicio...");
   }
}
```


Cliente

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("ID_Cliente")
public class Client {
  private final Service service;

  @Autowired
  public  void setService(Service service){
    this.service = service;
  }

  public void go(){
    service.go();
  }
}
```

@Autowired establece que deben resolverse los parámetros mediante DI.

Anotaciones estándar JSR-330

- A partir de Spring 3.0, Spring ofrece soporte para las anotaciones estándar JSR-330 (inyección de dependencia). Esas anotaciones se escanean de la misma manera que las anotaciones de Spring.
- Cuando trabaje con anotaciones estándar, hay que tener en cuenta que algunas características importantes no están disponibles.

Anotaciones Spring	Anotaciones Estándar (javax.inject.*) JSR-330
@Autowired	@Inject
@Component	@Named / @ManagedBean
@Scope("singleton")	@Singleton
@Qualifier	@Qualifier / @Named
@Value	-
@Required	-
@Lazy	-

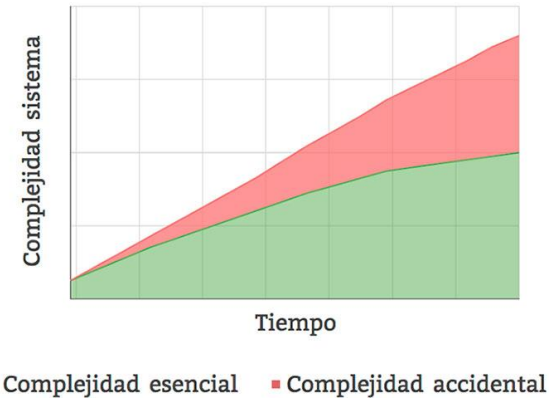
GESTIÓN DE DATOS

Arquitectura ágil

- Es económicamente factible realizar cambios radicales si la estructura del software separa sus aspectos de forma eficaz y dispone de las pruebas unitarias adecuadas de la arquitectura del sistema.
- Es posible iniciar un proyecto de software con una arquitectura simple pero bien desconectada, y ofrecer historias funcionales de forma rápida, para después aumentar la infraestructura.
- Los estándares facilitan la reutilización de ideas y componentes, reclutar personas con experiencia relevante, encapsulan buenas ideas y conectan componentes. Sin embargo, el proceso de creación de estándares a veces puede llevar demasiado tiempo para que la industria espere, y algunos estándares pierden contacto con las necesidades reales de los adoptantes a los que están destinados a servir. Hay que usar estándares cuando añadan un valor demostrable.

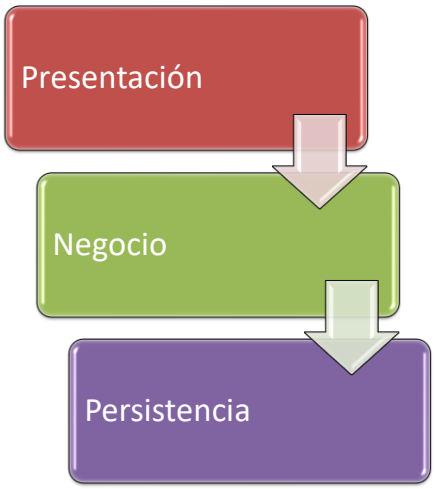
Complejidad

- La complejidad esencial está causada por el problema a resolver y nada puede eliminarla; si los usuarios quieren que un programa haga 30 cosas diferentes, entonces esas 30 cosas son esenciales y el programa debe hacer esas 30 cosas diferentes.
- La complejidad accidental se relaciona con problemas que los desarrolladores crean y pueden solucionar; por ejemplo, los detalles de escribir y optimizar el código o las demoras causadas por el procesamiento por lotes.
- Con el tiempo, la complejidad accidental ha disminuido sustancialmente, los programadores de hoy dedican la mayor parte de su tiempo a abordar la complejidad esencial.

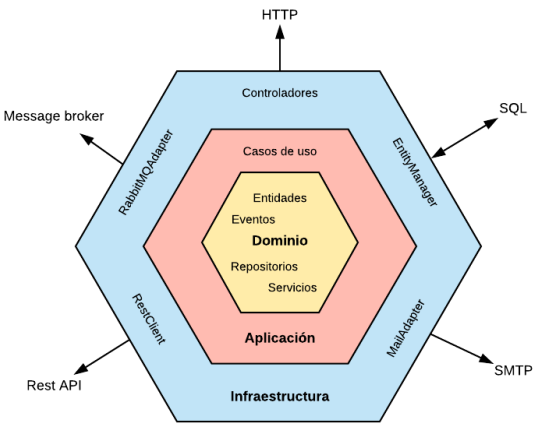


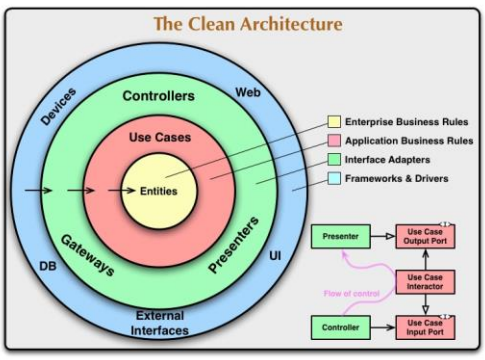
Arquitecturas multicapas

- Cada capa se apoya en la capa subsiguiente (capas horizontales) que depende de las capas debajo de ella, que a su vez dependerá de alguna infraestructura común y servicios públicos.
- El gran inconveniente de esta arquitectura en capas de arriba hacia abajo es el acoplamiento que crea.
- Hay sistemas estrictos (strict layered systems) y relajados (relaxed layered systems) que determinaran el nivel de acoplamiento de las dependencias entre capas.
- Las preocupaciones transversales provocan la aparición de capas verticales.
- Todo esto crea una complejidad accidental innecesaria.



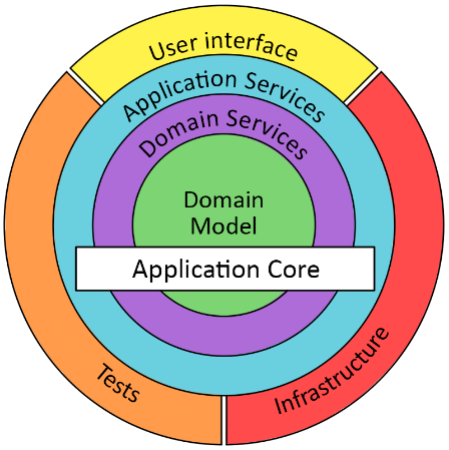
Arquitecturas concéntricas



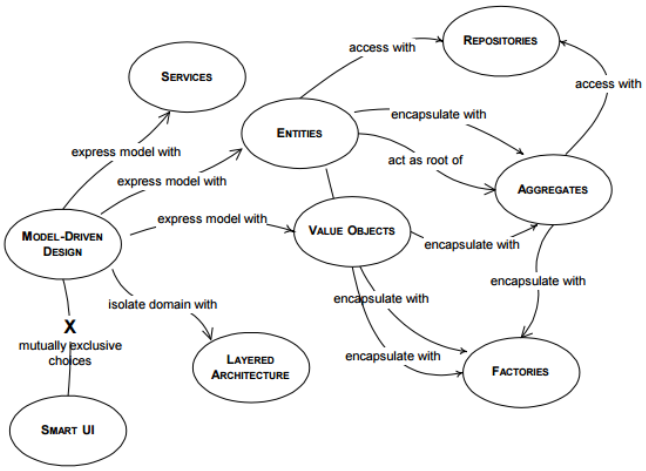


Arquitectura Limpia

- Propone que el dominio sea el núcleo de las capas y que este no se acople a nada externo. Las dependencias solo pueden apuntar hacia adentro, las capas interiores no conocen a las exteriores. Mediante el principio de inversión de dependencias, en vez de hacer uso explícito, nos acoplamos a contratos (interfaces) y no a implementaciones concretas.
  - Independiente de los marcos: La arquitectura no depende de la existencia de alguna biblioteca de software cargado de funciones. Esto permite utilizar dichos marcos como herramientas, en lugar de tener que ajustar el sistema a las limitaciones impuestas.
  - Comprobable: Las reglas de negocio se pueden probar sin la interfaz de usuario, la base de datos, el servidor web o cualquier otro elemento externo.
  - Independiente de la IU: La interfaz de usuario puede cambiar fácilmente, sin cambiar el resto del sistema. Una interfaz de usuario web podría reemplazarse por una interfaz de escritorio, por ejemplo, sin cambiar las reglas de negocio.
  - Independiente de la base de datos: Puede cambiar Oracle o SQL Server por Mongo, BigTable, CouchDB u otra cosa. Las reglas de negocio no están vinculadas a la base de datos.
  - Independiente de cualquier dependencia externa: De hecho, las reglas de negocio simplemente no saben nada sobre el mundo exterior.



Domain Driven Design



Capas de DDD

- Dominio (Domain)
  - Contiene la información sobre el Dominio. Es el núcleo de la parte de la aplicación que contiene las reglas de negocio. Es responsable de mantener el estado de los objetos de negocio. La persistencia de estos objetos se delega en la capa de infraestructura.
- Aplicación (Application)
  - Responsable de coordinar todos los elementos de la aplicación. No contiene lógica de negocio ni mantiene el estado de los objetos de negocio. Es responsable de mantener el estado de la aplicación y del flujo de esta.
- Interface de usuario (User Interface)
  - Responsable de presentar la información al usuario, interpretar sus acciones y enviarlas a la aplicación.
- Infraestructura (Infraestructure)
  - Esta capa es la capa de soporte para el resto de capas. Provee la comunicación entre las otras capas, implementa la persistencia de los objetos de negocio y las librerías de soporte para las otras capas (Interface, Comunicación, Almacenamiento, etc..)

- Dado que son capas conceptuales, su implementación puede ser muy variada y en una misma aplicación, tendremos partes o componentes que formen parte de cada una de estas capas.

Entidades

- Las entidades encapsulan los conceptos del negocio. Una entidad puede ser un objeto con métodos o puede ser un conjunto de funciones y estructuras de datos. No importa siempre que las entidades puedan ser utilizadas por muchas aplicaciones diferentes en la empresa.
- Estas entidades son los objetos de dominio de la aplicación. Encapsulan las reglas más generales y de alto nivel.
- Son los menos propensos a cambiar cuando algo externo cambia. Por ejemplo, no esperaría que estos objetos se vieran afectados por un cambio en la navegación de la página o la seguridad.
- Ningún cambio operativo en una aplicación en particular debería afectar la capa de la entidad.

Patrón Agregado (Aggregate)

- Una Agregación es un grupo de entidades asociadas que deben tratarse como una unidad a la hora de manipular sus datos.
- El patrón Agregado es ampliamente utilizado en los modelos de datos basados en Diseños Orientados al Dominio (DDD).
- Proporciona un forma de encapsular nuestras entidades y los accesos y relaciones que se establecen entre las mismas de manera que se simplifique la complejidad del sistema en la medida de lo posible.
- Cada Agregación cuenta con una Entidad Raíz (root) y una Frontera (boundary):
  - La Entidad Raíz es una Entidad contenida en la Agregación de la que colgarán el resto de entidades del agregado y será el único punto de entrada a la Agregación.
  - La Frontera define qué está dentro de la Agregación y qué no.
- La Agregación es la unidad de persistencia, se recupera toda y se almacena toda.

Repositorio

- Un repositorio separa la lógica empresarial de las interacciones con la base de datos subyacente y centra el acceso a datos en un área, lo que facilita su creación y mantenimiento.
- El repositorio pertenecen a la capa de infraestructura y devuelve los objetos del modelo de dominio.
- Deberían implementar el patrón de doble herencia.
- Forma parte de los Domain Driven Design patterns: Domain Entity, Value-Object, Aggregates, Repository, Unit of Work, Specification, Dependency Injection, Inversion of Control (IoC).

Ventajas del Repositorio

- Proporciona un punto de sustitución para las pruebas unitarias. Es fácil probar la lógica empresarial sin una base de datos y otras dependencias externas.
- Las consultas y los modelos de acceso a datos duplicados se pueden quitar y refactorizar en el repositorio.
- Los métodos del controlador pueden usar parámetros fuertemente tipados, lo que significa que el compilador encontrará errores de tipado de datos en cada compilación en lugar de realizar la búsqueda de errores de tipado de datos en tiempo de ejecución durante las pruebas.
- El acceso a datos está centralizado, lo que brinda las siguientes ventajas:
  - Mayor separación de intereses (SoC), uno de los principios de MVC, lo que facilita más el mantenimiento y la legibilidad.
  - Implementación simplificada de un almacenamiento en caché de datos centralizado.
  - Arquitectura más flexible y con menor acoplamiento, que se puede adaptar a medida que el diseño global de la aplicación evoluciona.
- El comportamiento se puede asociar a los datos relacionados (calcular campos, aplicar relaciones, reglas de negocios complejas entre los elementos de datos de una entidad, …).
- Un modelo de dominio se puede aplicar para simplificar una lógica empresarial compleja.

DTO

- Un objeto de transferencia de datos (DTO) es un objeto que define cómo se enviarán los datos a través de la red.
- Su finalidad es:
  - Desacoplar del nivel de servicio de la capa de base de datos.
  - Quitar las referencias circulares.
  - Ocultar determinadas propiedades que los clientes no deberían ver.
  - Omitir algunas de las propiedades con el fin de reducir el tamaño de la carga.
  - Eliminar el formato de grafos de objetos que contienen objetos anidados, para que sean más conveniente para los clientes.
  - Evitar el "exceso" y las vulnerabilidades por publicación.

Servicio

- Los servicios representan operaciones, acciones o actividades que no pertenecen conceptualmente a ningún objeto de dominio concreto. Los servicios no tienen ni estado propio ni un significado más allá que la acción que los definen. Se anotan con @Service.
- Podemos dividir los servicios en tres tipos diferentes:
  - Domain services
    - Son responsables del comportamiento más específico del dominio, es decir, realizan acciones que no dependen de la aplicación concreta que estemos desarrollando, sino que pertenecen a la parte más interna del dominio y que podrían tener sentido en otras aplicaciones pertenecientes al mismo dominio.
  - Application services
    - Son responsables del flujo principal de la aplicación, es decir, son los casos de uso de nuestra aplicación. Son la parte visible al exterior del dominio de nuestro sistema, por lo que son el punto de entrada-salida para interactuar con la funcionalidad interna del dominio. Su función es coordinar entidades, value objects, domain services e infrastructure services para llevar a cabo una acción.
  - Infrastructure services
    - Declaran comportamiento que no pertenece realmente al dominio de la aplicación pero que debemos ser capaces de realizar como parte de este.

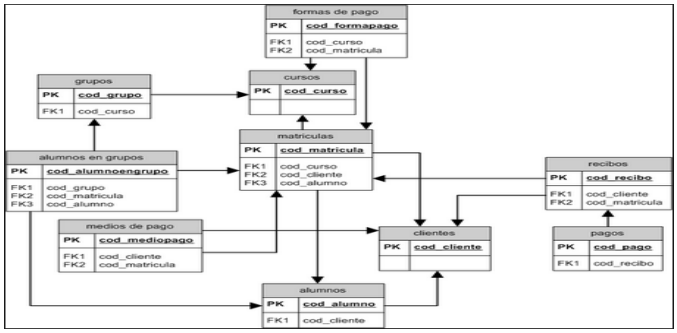
Bases de datos relacionales

- Las bases de datos relacionales son indiscutiblemente el centro de la empresa moderna.
- Los datos de la empresa se basan en entidades que están almacenadas en ubicaciones de naturaleza relacional. (Base de Datos)
- Los actuales lenguajes de programación, como Java, ofrecen una visión intuitiva, orientada a objetos de las entidades de negocios a nivel de aplicación.
- Se han realizado mucho intentos para poder combinar ambas tecnologías (relacionales y orientados a objetos), o para reemplazar uno con el otro, pero la diferencia entre ambos es muy grande.

Discrepancia del Paradigma

- Cuando trabajamos con diferentes paradigmas (Objetos vs Relacional) la discrepancia a la hora de unirlos surge a medida que complicamos el diseño.
- Cuando disponemos de clases sencillas y relaciones sencillas, no hay problemas.

- Esto no es lo habitual. Pero esto SI:



Discrepancia del Paradigma

- Bauer & King (Bauer, C. & King, G. 2007) presentan una lista de los problemas de discrepancia en los paradigmas objeto/relacional
  - Problemas de granularidad
    - Java pueden definir clases con diferentes niveles de granularidad.
      - Cuanto más fino las clases de grano (Direcciones), éstas pueden ser embebida en las clases de grano grueso (Usuario)
    - En cambio, el sistema de tipo de la base de datos SQL son limitados y la granularidad se puede aplicar sólo en dos niveles
      - en la tabla (tabla de usuario) y el nivel de la columna (columna de dirección)
  - Problemas de subtipos
    - La herencia y el polimorfismo son las características básicas y principales del lenguaje de programación Java.
    - Los motores de base de datos SQL en general, no son compatibles con subtipos y herencia de tablas.

Discrepancia del Paradigma

- Problemas de identidad
  - Objetos Java definen dos nociones diferentes de identidad:
    - Identidad de objeto o referencia (equivalente a la posición de memoria, comprobar con un ==).
    - La igualdad como determinado por la aplicación de los métodos equals ().
  - La identidad de una fila de base de datos se expresa como la clave primaria.
  - Ni equals() ni == es equivalente a la clave principal.

- Problemas de Asociaciones
  - El lenguaje Java representa a las asociaciones mediante utilizan referencias a objetos
    - Las asociaciones entre objetos son punteros unidireccionales.
    - Relación de pertenencia → Elementos contenidos
    - Modelo de composición (colecciones) → Modelo jerárquico
  - Las asociaciones en BBDD están representados mediante la migración de claves.
    - Todas las asociaciones en una base de datos relacional son bidireccional

Alternativas

- JDBC (Java Database Connectivity): estándar de conectividad de bases de datos de Java y proporciona un mecanismo para que los programas Java se conecten a las bases de datos.
- Spring JdbcTemplate: realiza las tareas básicas del flujo de trabajo principal de JDBC (como la creación y ejecución de declaraciones), dejando que el código de la aplicación proporcione SQL y extraiga los resultados.
- MyBatis: es un framework de persistencia que soporta SQL, procedimientos almacenados y mapeos avanzados. Elimina casi todo el código JDBC, el establecimiento manual de los parámetros y la obtención de resultados. Puede configurarse con XML o anotaciones y permite mapear diccionarios (mapas) y POJOs (Plain Old Java Objects) con registros de base de datos.
- ORM: técnica de mapear datos desde una representación de modelo de objeto a una representación de modelo de datos relacional (y viceversa).
- Spring Data: unifica y facilita el acceso a distintos tipos de tecnologías de persistencia, tanto a bases de datos relacionales como NoSQL.

ORM

- Las siglas ORM significan "Object-Relational mapping" (Mapeo Objeto-Relacional): técnica de mapear datos desde una representación de modelo de objeto a una representación de modelo de datos relacional (y viceversa).
- Un ORM es un framework de persistencia de nuestros datos (objetos) a una base de datos relacional, es decir, código que escribimos para guardar y recuperar el valor de nuestras clases en una base de datos relacional.
- ORM es el nombre dado a las soluciones automatizadas para solucionar el problema de falta de coincidencia (Objetos-Relacional).
- Un buen número de sistemas de mapeo objeto-relacional se han desarrollado a lo largo de los años, pero su efectividad en el mercado ha sido diversa.

Hibernate	OpenJPA	TopLink	Kodo
CocoBase	DataNucleus	EclipseLink	Amber

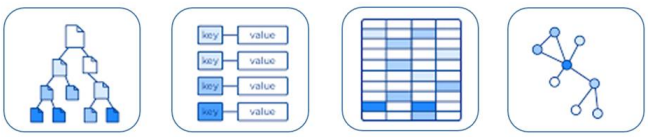
ORM

- Todo ORM deben de cumplir 4 especificaciones:
  - Un API para realizar operaciones básicas CRUD sobre los objetos de las clases persistentes (JDBC).
  - Un lenguaje o API para la especificación de las consultas que hacen referencia a las clases y propiedades de clases (SQL).
  - Un elemento de ORM (SessionFactory) para interactuar con objetos transaccionales y realizar operaciones diversas (conexión, validación, optimización, etc)
  - Un mecanismo para especificar los metadatos de mapeo (fichero XML, anotaciones)

Java Persistence API

- Java Persistence API, más conocida por sus siglas JPA, es la API de persistencia desarrollada para la plataforma Java EE.
- Es la propuesta estándar que ofrece Java para implementar un framework del lenguaje de programación Java que maneja datos relacionales en aplicaciones usando la Plataforma Java en sus ediciones Standard (Java SE) y Enterprise (Java EE).
- La JPA se origina a partir del trabajo del JSR 220 Expert Group el cual correspondía a EJB3. JPA 2.0 sería el trabajo del JSR 317 y posteriormente JPA 2.1 en el JSR 338.
- La persistencia en este contexto cubre tres áreas:
  - La API en sí misma, definida en el paquete javax.persistence
  - El lenguaje de consulta Java Persistence Query Language (JPQL).
  - Metadatos objeto/relacional.
- El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto-relacional), como sí pasaba con EJB2, y permitir usar objetos regulares (conocidos como POJO).

NoSQL



Documento (MongoDB, Azure Cosmos DB, Amazon DynamoDB, Elasticsearch)

- Los datos y los metadatos se almacenan jerárquicamente en documentos, datos semiestructurados, en grupos denominados colecciones.

Par clave-valor (Redis, Amazon ElastiCache)

- Los pares clave-valor se almacenan mediante una tabla hash. Los tipos de pares clave-valor funcionan mejor cuando la clave es conocida y el valor asociado con la clave es desconocido.

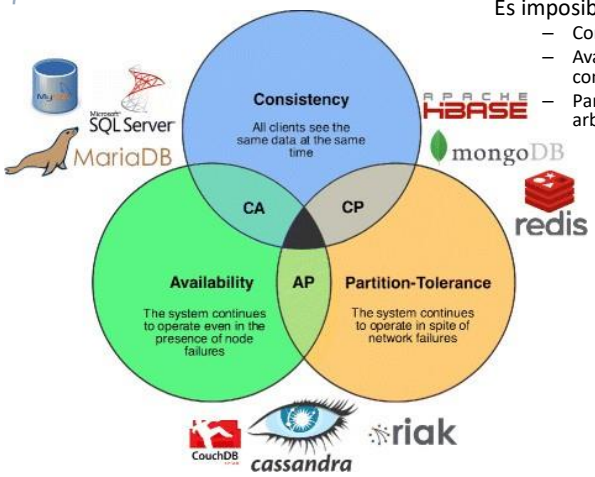
Almacén de columna ancha, Familia de columnas o basadas en columnas (Apache Cassandra)

- Las bases de datos de familia de columnas, basadas en columnas o en forma de columnas almacenan los datos con eficacia y consultan las filas de datos dispersos y resultan útiles para consultar columnas específicas de la base de datos.

Grafo (Neo4J)

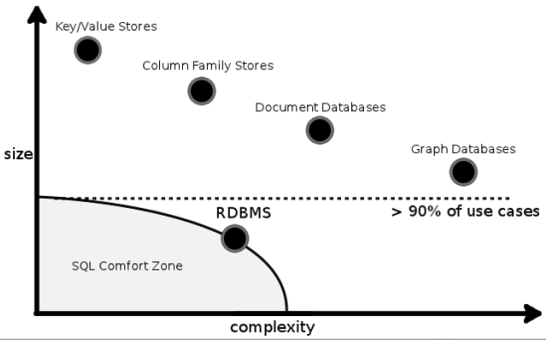
- Las bases de datos de grafos usan un modelo basado en nodos y bordes para representar datos interconectados, como las relaciones entre las personas en una red social, y ofrecen una navegación y un almacenamiento simplificados por las relaciones complejas.

El teorema CAP



Es imposible en un sistema distribuido garantizar simultáneamente:

- Consistency: Todos los nodos ven los mismos datos al mismo tiempo
- Availability: Toda petición obtiene una respuesta tanto en caso de éxito como fallo
- Partition Tolerance: El sistema seguirá funcionando ante pérdidas arbitrarias de información o fallos parciales



SPRING DATA

Spring Data

- Spring Framework ya proporcionaba soporte para JDBC, Hibernate, JPA o JDO, simplificando la implementación de la capa de acceso a datos, unificando la configuración y creando una jerarquía de excepciones común para todas ellas.
- Spring Data es un proyecto (subproyectos) de SpringSource cuyo propósito es unificar y facilitar el acceso a distintos tipos de tecnologías de persistencia, tanto a bases de datos relacionales como a las del tipo NoSQL.
- Spring Data viene a cubrir el soporte necesario para distintas tecnologías de bases de datos NoSQL integrándolas con las tecnologías de acceso a datos tradicionales, simplificando el trabajo a la hora de crear las implementaciones concretas.
- Con cada tipo de tecnología de persistencia, los DAOs (Data Access Objects) ofrecen las funcionalidades típicas de CRUD para objetos de dominio propios, métodos de búsqueda, ordenación y paginación.
- Spring Data proporciona interfaces genéricas para estos aspectos (CrudRepository, PagingAndSortingRepository) e implementaciones específicas para cada tipo de tecnología de persistencia.

Entidades de Dominio

- Una entidad es cualquier objeto del dominio que mantiene un estado y comportamiento más allá de la ejecución de la aplicación y que necesita ser distinguido de otro que tenga las mismas propiedades y comportamientos.
- Es un tipo de clase dedicada a representar un modelo de dominio persistente que:
  - Debe ser publica (no puede ser estar anidada ni final o tener miembros finales)
  - Deben tener un constructor público sin ningún tipo de argumentos.
  - Para cada propiedad que queramos persistir debe haber un método get/set asociado.
  - Debe tener una clave primaria
  - Debería sobrescribir los métodos equals y hashCode
  - Debería implementar el interfaz Serializable para utilizar de forma remota

Anotaciones JPA

Anotación	Descripción
@Entity	- Se aplica a la clase. - Indica que esta clase Java es una entidad a persistir.
@Table(name="Tabla")	- Se aplica a la clase e indica el nombre de la tabla de la base de datos donde se persistirá la clase. - Es opcional si el nombre de la clase coincide con el de la tabla.

@Column(name="Id")	- Se aplica a una propiedad Java para indicar el nombre de la columna de la base de datos en la que se persistirá la propiedad. Se pueden dar detalles de la creación de la columna:
	  - name: nombre
	  - length: longitud
	  - precision: número total de dígitos
	  - scale: número de dígitos decimales
	  - unique: restricción valor único
	  - nullable: restricción valor obligatorio
	  - insertable: es insertable
	  - updatable: es modificable
	- Es opcional si el nombre de la propiedad Java coincide con el de la columna de la base de datos.

@Transient	- Se aplica a una propiedad Java e indica que este atributo no es persistente
@Id	- Se aplica a una propiedad Java e indica que este atributo es la clave primaria.

Anotaciones JPA

Anotación	Descripción

@GeneratedValue	- Proporciona la especificación de estrategias de generación para los valores de claves primarias.
	  - AUTO: Indica que el proveedor de persistencia debe elegir una estrategia adecuada para la base de datos en particular.
	  - IDENTITY: Indica que el proveedor de persistencia debe asignar claves primarias mediante una columna de identidad de base de datos.
	  - SEQUENCE: Indica que el proveedor de persistencia debe asignar claves primarias usando una secuencia de base de datos.
	  - TABLE: Indica que el proveedor de persistencia debe asignar claves primarias mediante una tabla de base de datos subyacente para garantizar la exclusividad.

@Temporal	- Especifica el componente temporal a persistir en los tipos Date y Calendar. (TemporalType: DATE, TIME, TIMESTAMP)
@Enumerated	- Indica que los valores de la propiedad van a estar dentro del rango de un objeto enumerador.
@Lob	- Aplicado sobre una propiedad, indica que es un objeto grande (Large OBject)..
@Embeddable	- Define una clase cuyas instancias se almacenan como parte intrínseca de una entidad propietaria (columnas) y comparten la identidad de la entidad.
@Embedded	- Indica que la propiedad es de tipo Embeddable
@EmbeddedId	- Mapeada una clave primaria compuesta que es una clase incrustable.

Asociaciones

- Uno a uno (Unidireccional)
  - En la entidad fuerte se anota la propiedad con la referencia de la entidad.
  - @OneToOne(cascade=CascadeType.ALL):
    - Esta anotación indica la relación uno a uno de las 2 tablas.
  - @PrimaryKeyJoinColumn:
    - Indicamos que la relación entre las dos tablas se realiza mediante la clave primaria.
- Uno a uno (Bidireccional)
  - Las dos entidades cuentan con una propiedad con la referencia a la otra entidad.

Asociaciones

- Uno a Muchos
  - En Uno
    - Dispone de una propiedad de tipo colección que contiene las referencias de las entidades muchos:
      - List: Ordenada con repetidos
      - Set: Desordenada sin repetidos
    - @OneToMany(mappedBy="propEnMuchos",cascade= CascadeType.ALL)
      - mappedBy: contendrá el nombre de la propiedad en la entidad muchos con la referencia a la entidad uno.
    - @IndexColumn (name="idx")
      - Opcional. Nombre de la columna que en la tabla muchos para el orden dentro de la Lista.
  - En Muchos
    - Dispone de una propiedad con la referencia de la entidad uno.
    - @ManyToOne
      - Esta anotación indica la relación de Muchos a uno
    - @JoinColumn (name="idFK")
      - Indicaremos el nombre de la columna que en la tabla muchos contiene la clave ajena a la tabla uno.

Asociaciones

- Muchos a muchos (Unidireccional)
  - Dispone de una propiedad de tipo colección que contiene las referencias de las entidades muchos.
  - @ManyToMany(cascade=CascadeType.ALL):
    - Esta anotación indica la relación muchos a muchos de las 2 tablas.
- Muchos a muchos(Bidireccional)
  - La segunda entidad también dispone de una propiedad de tipo colección que contiene las referencias de las entidades muchos.
  - @ManyToMany(mappedBy="propEnOtroMuchos"):
    - mappedBy: Propiedad con la colección en la otra entidad para preservar la sincronicidad entre ambos lados

Limites de las agregaciones

- El atributo cascade se utiliza en los mapeos de las asociaciones para indicar cuando se debe propagar la acción en una instancia hacia la instancias relacionadas mediante la asociación.
- Enumeración de tipo CascadeType:
  - ALL = {PERSIST, MERGE, REMOVE, REFRESH, DETACH}
  - DETACH (Separar), MERGE (Modificar), PERSIST (Crear)
  - REFRESH (Releer), REMOVE (Borrar), NONE
- Acepta múltiples valores:
  - @OneToMany(mappedBy="profesor", cascade={CascadeType.PERSIST, CascadeType.MERGE})
- Al propagar en las inserciones de claves primarias auto calculadas como claves ajenas es necesario rectificarlas en las renacidas una vez generadas:

```
@PrePersist
private void prePersiste() {
    if (id == null) {
        setId(new ComposePK(user.getId(), role.getId()));
    }
}
```

Mapeo de Herencia

- Tabla por jerarquía de clases
  - Padre:
    - @Table("Account")
    - @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
    - @DiscriminatorColumn(name="PAYMENT_TYPE")
  - Hija:
    - @DiscriminatorValue(value = "Debit")
- Tabla por subclases
  - Padre:
    - @Table("Account")
    - @Inheritance(strategy = InheritanceType.JOINED)
  - Hija:
    - @Table("DebitAccount")
    - @PrimaryKeyJoinColumn(name = "account_id")
- Tabla por clase concreta
  - Padre:
    - @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
  - Hija:
    - @Table("DebitAccount")

Repositorio

- Un repositorio es una clase que actúa de mediador entre el dominio de la aplicación y los datos que le dan persistencia.
- Su objetivo es abstraer y encapsular todos los accesos a la fuente de datos.
- Oculta completamente los detalles de implementación de la fuente de datos a sus clientes.
- El interfaz expuesto por el repositorio no cambia aunque cambie la implementación de la fuente de datos subyacente (diferentes esquemas de almacenamiento).
- Se crea un repositorio por cada entidad de dominio que ofrece los métodos CRUD (Create-Read-Update-Delete), de búsqueda, ordenación y paginación.

Repositorio

- Con el soporte de Spring Data, la tarea repetitiva de crear las implementaciones concretas de DAO para las entidades se simplifica porque solo vamos se necesita un interfaz que extienda uno de los siguientes interfaces:
  - CrudRepository<T,ID>, ListCrudRepository<T, ID>
    - count, delete, deleteAll, deleteById, existsById, findAll, findAllById, findById, save, saveAll
  - PagingAndSortingRepository<T,ID>, ListPagingAndSortingRepository<T, ID>
    - findAll(Pageable pageable), findAll(Sort sort)
  - JpaRepository<T,ID>
    - deleteAllInBatch, deleteInBatch, flush, getOne, saveAll, saveAndFlush
  - Implementaciones para otros modelos: MongoRepository<T,ID>, …
- En el proceso de inyección Spring implementa la interfaz antes de inyectarla:

```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {}
@Autowired
private ProfesorRepository repository;
```


Repositorio

- El interfaz puede ser ampliado con nuevos métodos que serán implementados por Spring:
  - Derivando la consulta del nombre del método directamente.
  - Mediante el uso de una consulta definida manualmente.
  - Usando especificaciones (JPA Criteria API) para crear consultas mediante programación.
- La implementación se realizará mediante la decodificación del nombre del método, dispone de una sintaxis especifica para crear dichos nombre:

```
List<Profesor> findByNombreStartingWiths(String nombre);
List<Profesor> findByApellido1AndApellido2OrderByEdadDesc(String apellido1, String
      apellido2);

List<Profesor> findByTipoIn(Collection<Integer> tipos);
int deleteByEdadGreaterThan(int valor);
```

Repositorio

- Prefijo consulta derivada:
  - find (read, query, get, search, stream), exists, count, delete (remove)
- Opcionalmente, limitar los resultados de la consulta:
  - Distinct, TopNumFilas y FirstNumFilas
- Expresión de propiedad: ByPropiedad
  - Operador (Between, LessThan, GreaterThan, Like, …) por defecto equal.
  - Se pueden concatenar varias con And y Or
  - Opcionalmente admite el indicador IgnoreCase y AllIgnoreCase.
- Opcionalmente, OrderByPropiedadAsc para ordenar,
  - se puede sustituir Asc por Desc, admite varias expresiones de ordenación.
- Parámetros:
  - un parámetro por cada operador que requiera valor y debe ser del tipo apropiado
- Parámetros opcionales:
  - Sort → Sort.by("nombre", "apellidos").descending(), Sort.unsorted()
  - Pageable → PageRequest.of(0, 10, Sort.by("nombre")), Pageable.unpaged().

Repositorio

Palabra clave	Muestra	Fragmento de JPQL
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
(nada) ,Is,Equals	findByFirstname, findByFirstnameIs, findByFirstnameEquals	… where x.firstname = ?1
Not, IsNot	findByLastnameNot	… where x.lastname <> ?1
Between, IsBetween	findByStartDateBetween	… where x.startDate between ?1 and ?2
LessThan, IsLessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual, IsLessThanEqual	findByAgeLessThanEqual	… where x.age <= ?1
GreaterThan, IsGreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual, IsGreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1

Repositorio

Palabra clave	Muestra	Fragmento de JPQL
Before, IsBefore	findByStartDateBefore	… where x.startDate < ?1
After, IsAfter	findByStartDateAfter	… where x.startDate > ?1
Null, IsNull	findByAgeIsNull	… where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	… where x.age not null
Like, IsLike	findByFirstnameLike	… where x.firstname like ?1
NotLike, IsNotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith, IsStartingWith, StartsWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parámetro enlazado con % anexado)
EndingWith, IsEndingWith, EndsWith	findByFirstnameEndingWith	… where x.firstname like ?1(parámetro enlazado con % antepuesto)

Repositorio

P. Clave	Muestra	Fragmento de JPQL
Containing, IsContaining, Contains	findByFirstnameContaining	… where x.firstname like ?1(parámetro enlazado entre %)
In, IsIn	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn, IsNotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1
True, IsTrue	findByActiveTrue()	… where x.active = true
False, IsFalse	findByActiveFalse()	… where x.active = false
IgnoreCase, IgnoringCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?1)
AllIgnoreCase, AllIgnoringCase	findBy…AllIgnoreCase	
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc

Repositorio

- Valor de retorno de consultas síncronas:
  - find, read, query, get:
    - List<Entidad>, Iterator<T>, Collection<T>
    - Stream<Entidad>
    - Optional<T>
  - exists:
    - boolean
  - count:
    - long
  - delete, remove:
    - void
- Valor de retorno de consultas asíncronas (deben ir anotadas con @Async):
  - Future<Entidad>
  - CompletableFuture<Entidad>
  - ListenableFuture<Entidad>

Repositorio

- Mediante consultas JPQL:

```
@Query("from Profesor p where p.edad > 67")
List<Profesor> findJubilados();
```

- Mediante consultas SQL nativas (nativeQuery=true):

```
@Query(value="select * from Profesores p where p.edad between ?1 and ?2", nativeQuery=true)
List<Profesor> findActivos(int inicial, int final);
```

- Se puede usar el enlace de parámetros basado en la posición (?1, ?2, …) o la anotación @Param para dar un nombre concreto a un parámetro del método y vincular el nombre en la consulta:

```
@Query("from Profesor p where p.edad > :edad")
List<Profesor> findJubilados(@Param("edad") int edad);
```

- Las consultas que modifican las base de datos deben ir anotadas con @Modifying:

```
@Modifying
@Query("update Profesor u set u.salario = ?1 * u.salario")
int revisaSalario(double delta);
@Modifying @Query("delete from Profesor p where p.edad > ?1")
List<Profesor> deleteJubilados(int edad);
```

Spring Data & Spring Security

- Spring Security proporciona la integración de Spring Data que permite referirse al usuario actual dentro de sus consultas.
- Hay que agregar la dependencia org.springframework.security:spring-security-data y proporcionar un bean de tipo SecurityEvaluationContextExtension.

```
@Bean
public SecurityEvaluationContextExtension securityEvaluationContextExtension() {
    return new SecurityEvaluationContextExtension();
}
```

- Ahora se puede utilizar Spring Security dentro de las consultas:

```
@Repository
public interface MessageRepository extends PagingAndSortingRepository<Message,Long> {
    @Query("select m from Message m where m.to.id = ?#{ principal?.id }")
    Page<Message> findInbox(Pageable pageable);
}
```


Especificaciones

- Las especificaciones se pueden usar fácilmente para crear un conjunto extensible de predicados sobre una entidad que luego se puede combinar y usar en un JpaRepository sin necesidad de declarar una consulta (método) para cada combinación necesaria. Al escribir un criteria, se define la cláusula where (predicado) de una consulta para una clase de dominio.
- Para admitir especificaciones, se extiende el interfaz del repositorio con el interfaz JpaSpecificationExecutor:

```
public interface ProfesorRepository extends JpaRepository<Profesor, Long>,
        JpaSpecificationExecutor<Profesor> {}
```

- La interfaz adicional tiene métodos que le permiten ejecutar especificaciones:

```
List<T> findAll(Specification<T> spec);
```

- La interfaz Specification se define de la siguiente manera:

```
public interface Specification<T> {
  Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder builder);
}
```

- Para crear las consultas mediante programación

```
repository.findAll((root, query, builder) -> builder.greaterThan(root.get("edad"), 67))
```

Extender repositorios

- Si muchos repositorios en la aplicación deben tener el mismo conjunto de métodos, se puede definir interfaz base una propia para extender. Dicha interfaz debe anotarse con @NoRepositoryBean. Esto evita que Spring Data intente crear una instancia directamente y falle porque no puede determinar la entidad para ese repositorio, ya que todavía contiene una variable de tipo genérico.

```
@NoRepositoryBean
public interface ProjectionsAndSpecificationJpaRepository<E, ID>
      extends JpaRepository<E, ID>, JpaSpecificationExecutor<E> {
      <T> List<T> findAllBy(Class<T> tipo);
      <T> List<T> findAllBy(Sort orden, Class<T> tipo);
      <T> Page<T> findAllBy(Pageable page, Class<T> tipo);
}
```


Transacciones

- Por defecto, los métodos CRUD en las instancias del repositorio son transaccionales. Para las operaciones de lectura, el indicador readOnly de configuración de transacción se establece en true para optimizar el proceso. Todos los demás se configuran con un plano @Transactional para que se aplique la configuración de transacción predeterminada.
- Cuando se van a realizar varias llamadas al repositorio o a varios repositorios se puede anotar con @Transactional el método para que todas las operaciones se encuentren dentro de la misma transacción.

```
@Transactional
public void create(Pago pago) { … }
```

- Para que los métodos de consulta sean transaccionales:

```
@Override
@Transactional(readOnly = false)
public List<User> findAll();
```

Validaciones

- Desde la versión 3, Spring ha simplificado y potenciado en gran medida la validación de datos, gracias a la adopción de la especificación JSR 303. Este API permite validar los datos de manera declarativa, con el uso de anotaciones. Esto nos facilita la validación de los datos enviados antes de llegar al controlador REST. (**Dependencia: Starter I/O > Validation**)
- Las anotaciones se pueden establecer a nivel de clase, atributo y parámetro de método.
- Para realizar la validación manualmente:

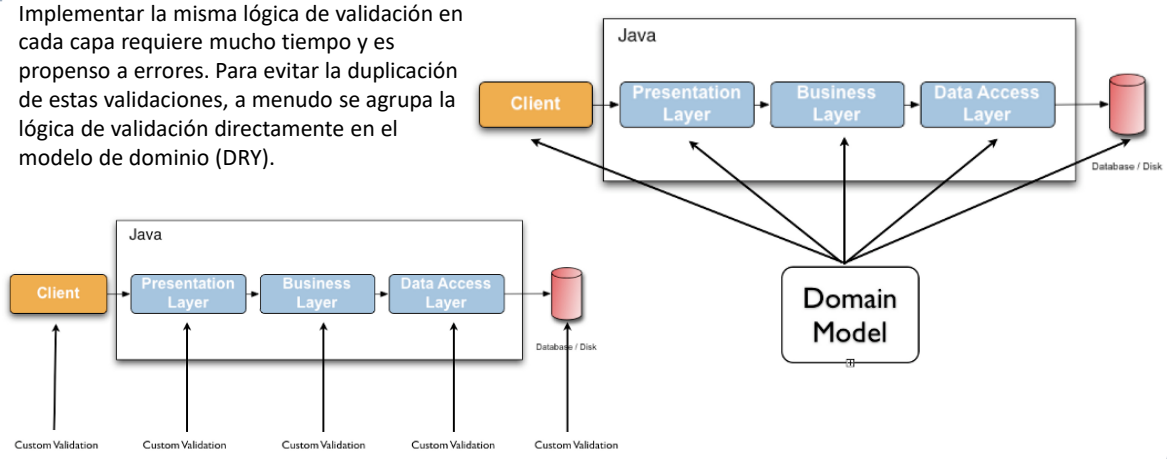
```
@Autowired    // Validation.buildDefaultValidatorFactory().getValidator()
private Validator validator;
Set<ConstraintViolation<Persona>> constraintViolations =  validator.validate( persona );
Set<ConstraintViolation<Persona>> constraintViolations = validator.validateProperty( persona,
      "nombre" );
```

- La anotación @Valid se puede usar para marcar la validación en cascada de las propiedades y elementos de las colecciones.
- Se puede indicar mediante @Valid que el elemento debe validarse antes de ser inyectado.

```
public ResponseEntity<Object> create(@Valid @RequestBody Persona item)
```


Validaciones

Implementar la misma lógica de validación en cada capa requiere mucho tiempo y es propenso a errores. Para evitar la duplicación de estas validaciones, a menudo se agrupa la lógica de validación directamente en el modelo de dominio (DRY).



Validaciones (JSR 303)

@Null : Comprueba que el valor anotado es null

@NotNull : Comprueba que el valor anotado no sea null

@NotEmpty : Comprueba si el elemento anotado no es nulo ni está vacío

@NotBlank : Comprueba que la secuencia de caracteres anotados no sea nula y que la longitud recortada sea mayor que 0. La diferencia @NotEmpty es que esta restricción solo se puede aplicar en secuencias de caracteres y que los espacios en blanco finales se ignoran.

@AssertFalse : Comprueba que el elemento anotado es falso.

@AssertTrue : Comprueba que el elemento anotado es verdadero

Validaciones (JSR 303)

@Max(value=) : Comprueba si el valor anotado es menor o igual que el máximo especificado

@Min(value=) : Comprueba si el valor anotado es mayor o igual que el mínimo especificado

@Negative : Comprueba si el elemento es estrictamente negativo. Los valores cero se consideran inválidos.

@NegativeOrZero : Comprueba si el elemento es negativo o cero.

@Positive : Comprueba si el elemento es estrictamente positivo. Los valores cero se consideran inválidos.

@PositiveOrZero : Comprueba si el elemento es positivo o cero.

@DecimalMax(value=, inclusive=) : Comprueba si el valor númerico anotado es menor que el máximo especificado, cuando inclusive= falso. De lo contrario, si el valor es menor o igual al máximo especificado.

@DecimalMin(value=, inclusive=) : Comprueba si el valor anotado es mayor que el mínimo especificado, cuando inclusive= falso. De lo contrario, si el valor es mayor o igual al mínimo especificado.

Validaciones (JSR 303)

@Digits: El elemento anotado, numérico o secuencia de caracteres , debe ser un número cuyo valor tenga el número de dígitos especificado.

@Email : Comprueba si la secuencia de caracteres especificada es una dirección de correo electrónico válida.

@Pattern(regex=, flags=) : Comprueba si la cadena anotada coincide con la expresión regular regex considerando la bandera dadamatch.

@Size(min=, max=) : Comprueba si el tamaño del elemento anotado está entre min y max (inclusive)

@Past : Comprueba si la fecha anotada está en el pasado

@PastOrPresent : Comprueba si la fecha anotada está en el pasado o en el presente

@Future : Comprueba si la fecha anotada está en el futuro.

@FutureOrPresent : Comprueba si la fecha anotada está en el presente o en el futuro

Validaciones (Hibernate)

@CreditCardNumber(ignoreNonDigitCharacters=): Comprueba que la secuencia de caracteres pasa la prueba de suma de comprobación de Luhn.

@Currency(value=): Comprueba que la unidad monetaria de un javax.money.MonetaryAmount es una de las unidades monetarias especificadas.

@DurationMax(days=, hours=, minutes=, seconds=, millis=, nanos=, inclusive=): Comprueba que el elemento java.time.Duration no sea mayor que el construido a partir de los parámetros de la anotación.

@DurationMin(days=, hours=, minutes=, seconds=, millis=, nanos=, inclusive=): Comprueba que el elemento java.time.Duration no sea menor que el construido a partir de los parámetros de anotación.

@URL(protocol=, host=, port=, regexp=, flags=): Comprueba si la secuencia de caracteres anotada es una URL válida según RFC2396.

@ISBN: Comprueba que la secuencia de caracteres sea un ISBN válido.

Validaciones (Hibernate)

@EAN: Comprueba que la cadena sea un código de barras EAN válido

@CodePointLength(min=, max=, normalizationStrategy=): Valida que la longitud del punto de código de la cadena esté entre min e max incluidos.

@LuhnCheck(startIndex= , endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=): Comprueba que los dígitos de la secuencia de caracteres pasan el algoritmo de suma de comprobación de Luhn.

@Mod10Check(multiplier=, weight=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=): Comprueba que los dígitos dentro de la secuencia de caracteres pasan el algoritmo genérico de suma de comprobación mod 10.

@Mod11Check(threshold=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=, treatCheck10As=, treatCheck11As=): Comprueba que los dígitos dentro de la secuencia de caracteres pasan el algoritmo de suma de comprobación mod 11.

Validaciones (Hibernate)

@UniqueElements: Comprueba que la colección solo contiene elementos únicos.

@Length(min=, max=): Valida que la secuencia de caracteres esté entre min e max incluidos

@Range(min=, max=): Comprueba si el valor anotado se encuentra entre (inclusive) el mínimo y el máximo especificados

@Normalized(form=): Valida que la secuencia de caracteres anotados se normalice de acuerdo con lo dado form.

@SafeHtml(additionalTags=, additionalTagsWithAttributes=, baseURI=, whitelistType=): Comprueba si el valor anotado contiene fragmentos potencialmente maliciosos como <script/>.

@ScriptAssert(lang=, script=, alias=, reportOn=): Comprueba si la secuencia de comandos proporcionada se puede evaluar correctamente con el elemento anotado.

@UUID(allowEmpty=, allowNil=, version=, variant=, letterCase=): Comprueba que la secuencia de caracteres anotada es un identificador único universal válido según RFC 4122.

Mensajes de error

- Todas las anotaciones cuentan con el parámetro message que permite la personalización del mensaje de error, muy recomendable para aportar semántica a determinadas validaciones como el Pattern:

```
@Pattern(
    regexp = "^(?=.*\\d)(?=.*[a-z])(?=.*[A-Z])(?=.*\\W).{8,}$",
    message="debe contener al menos 8 caracteres con letras mayúsculas, minúsculas, números y
        símbolos"
)
private String password;
```

- Es posible utilizar la interpolación con el lenguaje de expresión de Jakarta en los mensajes: ${validatedValue} representa el valor invalido y pueden participar el resto de parámetros de la anotación.

```
@Size(min = 2, max = 50,
    message = "'${validatedValue}' debe tener entre {min} y {max} letras" )
private byte nombre;
```


Validaciones personalizadas

- Anotación personalizada para la validación:

```
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = NifValidator.class)
@Documented
public @interface NIF {
    String message() default "{validation.NIF.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Validaciones personalizadas

- Clase del validador

```
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
public class NifValidator implements ConstraintValidator<NIF, String> {
    @Override
    public void initialize(NIF constraintAnnotation) { // default
        ConstraintValidator.super.initialize(constraintAnnotation);
    }
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        if(value == null) return true;
        value = value.toUpperCase();
        if(!value.matches("^\\d{1,8}[A-Z]$")) return false;
        return "TRWAGMYFPDXBNJZSQVHLCKE".charAt(Integer.parseInt( value.substring(0, value.length() - 1)) % 23)
                == value.charAt(value.length() - 1);
    }
}
```

- En el fichero ValidationMessages.properties

```
validation.NIF.message=${validatedValue} no es un NIF válido.
```


DTO

- Un objeto de transferencia de datos (DTO) es un objeto que define cómo se enviarán los datos a través de la red.
- Su finalidad es:
  - Desacoplar del nivel de servicio de la capa de base de datos.
  - Quitar las referencias circulares.
  - Ocultar determinadas propiedades que los clientes no deberían ver.
  - Omitir algunas de las propiedades con el fin de reducir el tamaño de la carga.
  - Eliminar el formato de grafos de objetos que contienen objetos anidados, para que sean más conveniente para los clientes.
  - Evitar el "exceso" y las vulnerabilidades por publicación.

Lombok

https://projectlombok.org/

- En las clases Java hay mucho código que se repite una y otra vez: constructores, equals, getters y setters. Métodos que quedan definidos una vez que dicha clase ha concretado sus propiedades, y que salvo ajustes menores, serán siempre sota, caballo y rey.
- Project Lombok es una biblioteca de java que se conecta automáticamente al editor y crea herramientas que automatizan la escritura de java.
- Mediante simples anotaciones ya nunca mas vuelves a escribir otro método get o equals.

```
@Data @AllArgsConstructor @NoArgsConstructor public class MyDTO {
    private long id;
    private String name;
}
```

- La anotación @Value (no confundir con la de Spring) crea la versión de solo lectura.
- Es necesario agregar las bibliotecas al proyecto y configurar el entorno.
- Anotar un parámetro con @NonNull genera una declaración de verificación no nula.

Clases registro (v16)

- Las clases de registro, son un tipo especial de clase, que ayudan a modelar agregados de datos simples e inmutables con una sintaxis más concisa que las clases normales. Una declaración de registro especifica en un encabezado una descripción de su contenido. Los campos de un registro son inmutables porque la clase está destinada a servir como un simple "soporte de datos".

```
record MyRecordDTO(long id, @JsonProperty("nombre") String name) { }
```

- Una clase de registro declara automáticamente los siguientes miembros :
  - Para cada componente del encabezado, los siguientes dos miembros:
    - Un campo/atributo private final con el mismo nombre y tipo declarado que el componente de registro, a veces denominado campo de componente.
    - Un método de acceso public con el mismo nombre y tipo del campo componente (getter).
  - Un constructor canónico cuya firma es la misma que la del encabezado. Este constructor asigna cada argumento de la expresión new, que instancia la clase de registro, al campo de componente correspondiente.
  - Implementaciones de los métodos equals y hashCode, que especifican que dos clases de registros son iguales si son del mismo tipo y sus valores de campos de componentes son iguales.
  - Una implementación del método toString que incluye la representación en cadena de todos los componentes de la clase de registro, con sus nombres.
- Pueden implementar sus propios métodos.

ModelMapper

http://modelmapper.org/

- Las aplicaciones a menudo contienen modelos de objetos similares pero diferentes, donde los datos en dos modelos pueden ser similares pero la estructura y las responsabilidades de los modelos son diferentes. El mapeo de objetos facilita la conversión de un modelo a otro, permitiendo que los modelos separados permanezcan segregados.
- ModelMapper facilita el mapeo de objetos, al determinar automáticamente cómo se mapea un modelo de objeto a otro, de acuerdo con las convenciones, de la misma forma que lo haría un ser humano, al tiempo que proporciona una API simple y segura de refactorización para manejar casos de uso específicos.

```
ModelMapper modelMapper = new ModelMapper();
OrderDTO orderDTO = modelMapper.map(order, OrderDTO.class);
```


Proyecciones

- Los métodos de consulta de Spring Data generalmente devuelven una o varias instancias de la raíz agregada administrada por el repositorio. Sin embargo, a veces puede ser conveniente crear proyecciones basadas en ciertos atributos de esos tipos. Spring Data permite modelar tipos de retorno dedicados, para recuperar de forma más selectiva vistas parciales de los agregados administrados.
- La forma más sencilla de limitar el resultado de las consultas solo a los atributos deseados es declarar una interfaz que exponga los métodos de acceso para las propiedades a leer o un DTO *con un único constructor*, que deben coincidir exactamente con las propiedades de la entidad:

```
public interface NamesOnly {
        String getNombre();
        String getApellidos();
}
```

- El motor de ejecución de consultas crea instancias de proxy de esa interfaz en tiempo de ejecución para cada elemento devuelto y reenvía las llamadas a los métodos expuestos al objeto de destino.

```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {
   List<NamesOnly> findByNombreStartingWith(String nombre);
}
```

Proyecciones

- Las proyecciones se pueden usar recursivamente.

```
interface PersonSummary {
  String getNombre();
  String getApellidos();
  DireccionSummary getDireccion();
  interface DireccionSummary {
    String getCiudad();
  }
}
```

- En las proyecciones abiertas, los métodos de acceso en las interfaces de proyección también se pueden usar para calcular nuevos valores:

```
public interface NamesOnly {
    @Value("#{args[0] + ' ' + target.nombre + ' ' + target.apellidos}")
    String getNombreCompleto(String tratamiento);
    default String getFullName() {
          return getNombre.concat(" ").concat(getApellidos());
    }
}
```


Proyecciones

- Se puede implementar una lógica personalizada mas compleja en un bean de Spring y luego invocarla desde la expresión SpEL:

```
@Component
class MyBean {
  String getFullName(Person person) { … }
}
interface NamesOnly {
  @Value("#{@myBean.getFullName(target)}")
  String getFullName();
  …
}
```

- Las proyecciones dinámicas permiten utilizar genéricos en la definición del repositorio para resolver el tipo de devuelto en el momento de la invocación:

```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {
        <T> List<T> findByNombreStartingWith(String prefijo, Class<T> type);
        <T> List<T> findBy(Class<T> type);
        <T> List<T> findAllBy(Class<T> type);
}
dao.findByNombreStartingWith("J", ProfesorShortDTO.class)
```

Serialización Jackson

- Jackson es una librería de utilidad de Java que nos simplifica el trabajo de serializar (convertir un objeto Java en una cadena de texto con su representación JSON), y des serializar (convertir una cadena de texto con una representación de JSON de un objeto en un objeto real de Java) objetos-JSON.
- Jackson es bastante "inteligente" y sin decirle nada es capaz de serializar y des serializar bastante bien los objetos. Para ello usa básicamente la reflexión de manera que si en el objeto JSON tenemos un atributo "name", para la serialización buscará un método "getName()" y para la des serialización buscará un método "setName(String s)".

```
ObjectMapper objectMapper = new ObjectMapper();
String jsonText  = objectMapper.writeValueAsString(person);
Person person = new ObjectMapper().readValue(jsonText, Person.class);
```

- El proceso de serialización y des serialización se puede controlar declarativamente mediante anotaciones:
  https://github.com/FasterXML/jackson-annotations

Serialización Jackson

- **@JsonProperty**: indica el nombre alternativo de la propiedad en JSON.

```
@JsonProperty("name") public String getTheName() { ... }
@JsonProperty("name") public void setTheName(String name) { ... }
```

- **@JsonFormat**: especifica un formato para serializar los valores de fecha/hora.

```
@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy hh:mm:ss")
public Date eventDate;
```

- **@JsonIgnore**: marca que se ignore una propiedad (nivel miembro).

```
@JsonIgnore public int id;
```

Serialización Jackson

- **@JsonIgnoreProperties**: marca que se ignore una o varias propiedades (nivel clase).

```
@JsonIgnoreProperties({ "id", "ownerName" })
@JsonIgnoreProperties(ignoreUnknown=true)
public class Item {
```

- **@JsonInclude**: se usa para incluir propiedades con valores vacíos/nulos/ predeterminados.

```
@JsonInclude(Include.NON_NULL)
public class Item {
```

- **@JsonAutoDetect**: se usa para anular la semántica predeterminada qué propiedades son visibles y cuáles no.

```
@JsonAutoDetect(fieldVisibility = Visibility.ANY)
public class Item {
```


Serialización Jackson

- **@JsonView**: permite indicar la Vista en la que se incluirá la propiedad para la serialización / deserialización.

```
public class Views {
   public static class Partial {}
   public static class Complete extends Partial {}
}
public class Item {
   @JsonView(Views.Partial.class)
   public int id;
   @JsonView(Views.Partial.class)
   public String itemName;
   @JsonView(Views.Complete.class)
   public String ownerName;
}

String result = new ObjectMapper().writerWithView(Views.Partial.class)
      .writeValueAsString(item);
```

Serialización Jackson

- **@JsonFilter**: indica el filtro que se utilizará durante la serialización (es obligatorio suministrarlo).

```
@JsonFilter("ItemFilter")
public class Item {
   public int id;
   public String itemName;
   public String ownerName;
}

FilterProvider filters = new SimpleFilterProvider().addFilter("ItemFilter",
SimpleBeanPropertyFilter.filterOutAllExcept("id", "itemName"));
MappingJacksonValue mapping = new MappingJacksonValue(dao.findAll());
mapping.setFilters(filters);
return mapping;
```


Serialización Jackson

- **@JsonManagedReference** y **@JsonBackReference**: se utilizan para manejar las relaciones maestro/detalle marcando la colección en el maestro y la propiedad inversa en el detalle (multiples relaciones requieren asignar nombres unicos).

```
@JsonManagedReference
public User owner;
@JsonBackReference
public List<Item> userItems;
```

- **@JsonIdentityInfo**: indica la identidad del objeto para evitar problemas de recursión infinita.

```
@JsonIdentityInfo(
  generator = ObjectIdGenerators.PropertyGenerator.class,
  property = "id")
public class Item {
   public int id;
```

Serialización XML (JAXB)

- JAXB (Java XML API Binding) proporciona a una manera rápida, conveniente de crear enlaces bidireccionales entre los documentos XML y los objetos Java. Dado un esquema, que especifica la estructura de los datos XML, el compilador JAXB genera un conjunto de clases de Java que contienen todo el código para analizar los documentos XML basados en el esquema. Una aplicación que utilice las clases generadas puede construir un árbol de objetos Java que representa un documento XML, manipular el contenido del árbol, y regenerar los documentos del árbol, todo ello en XML sin requerir que el desarrollador escriba código de análisis y de proceso complejo.
- Los principales beneficios de usar JAXB son:
  - Usa tecnología Java y XML
  - Garantiza datos válidos
  - Es rápida y fácil de usar
  - Puede restringir datos
  - Es personalizable
  - Es extensible

Anotaciones principales (JAXB)

- Para indicar a los formateadores JAXB como transformar un objeto Java a XML y viceversa se puede anotar (javax.xml.bind.annotation) la clases JavaBean para que JAXP infiera el esquema de unión.
- Las principales anotaciones son:
  - @XmlRootElement(namespace = "namespace"): Define la raíz del XML.
  - @XmlElement(name = "newName"): Define el elemento de XML que se va usar.
  - @XmlAttribute(required=true): Serializa la propiedad como un atributo del elemento.
  - @XmlID: Mapea un propiedad JavaBean como un XML ID.
  - @XmlType(propOrder = { "field2", "field1",.. }): Permite definir en que orden se van escribir los elementos dentro del XML.
  - @XmlElementWrapper: Envuelve en un elemento los elementos de una colección.
  - @XmlTransient: La propiedad no se serializa.

Jackson DataFormat XML

- Extensión de formato de datos para Jackson (http://jackson.codehaus.org) para ofrecer soporte alternativo para serializar POJOs como XML y deserializar XML como pojos. Soporte implementado sobre la API de Stax (javax.xml.stream), mediante la implementación de tipos de API de Streaming de Jackson como JsonGenerator, JsonParser y JsonFactory. Algunos tipos de enlace de datos también se anularon (ObjectMapper se clasificó como XmlMapper).
- Dependencia:

```
<dependency>
        <groupId>com.fasterxml.jackson.dataformat</groupId>
        <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```


MongoDB

- MongoDB (https://www.mongodb.com/) es una base de datos orientada a documentos, open source, que toma su nombre del inglés hu***mongo***us, "enorme“ y se enmarca en la familia de bases de datos NoSQL.
- En lugar de guardar los datos en tablas/filas/columnas, guarda los datos en documentos (estructura jerarquica).
- Los documentos se expresan en JSON y son almacenados en formato BSON (representación binaria de JSON).
- Los documentos se almacenan en colecciones, concepto similar a una tabla de una base de datos relacional.
- Una de las diferencias más importantes con respecto a las bases de datos relacionales, es que no es necesario seguir un esquema.
- Los documentos de una misma colección pueden tener esquemas diferentes, es decir, en una colección se pueden almacenar diferentes tipos de documentos.
- Todos los documentos tienen un identificador único denominado _id, se autogenera en caso de ser necesario

MongoDB

Cuando usar MongoDB

- Prototipos y aplicaciones simples
- Hacer la transición de front a back
- Aplicaciones con mucha carga de escritura
- Agregado de datos a un nivel medio/alto
- Aplicaciones con datos muy heterogéneos
- Enormes colecciones de datos (sharding)
- Almacenar ficheros (sharding)

Cuando no usar MongoDB

- Mongo no puede hacer JOINs
- El lenguaje de consulta menos potente que SQL
- No tiene transacciones
- La velocidad baja al subir la seguridad (escritura)
- Aunque es muy fácil empezar con MongoDB, si la aplicación crece mucho, los modelos complejos van a requerir JOINs

MongoDB: Instalación

- Se puede utilizar un instalador o instalarla manualmente.
- Descargar la última versión estable del Communitu Server desde:
  - https://www.mongodb.com/download-center?jmp=nav#community
- Instalación manual:
  - Crear una carpeta donde realizar la instalación
  - Descomprimir el fichero en el directorio
  - Crear una carpeta \data que contendrá los ficheros de datos
  - En una consola de comandos, en el directorio \bin, hay que levantar el servidor:
    - mongod --dbpath <path\data > --nojournal
  - Es conveniente agregar al PATH la carpeta \bin
- Desde otra consola, sobre el directorio \bin, se puede acceder al interfaz de comandos
  - mongo host:puerto/database -u usuario -p password

MongoDB: Comandos

- **show dbs**: muestra los nombres de las bases de datos
- **show collections**: muestra las colecciones en la base de datos actual
- **show users**: muestra los usuarios en la base de datos actual
- **show profile**: muestra las entradas más recientes de system.profile con el tiempo> = 1ms
- **show logs**: muestra los nombres de los logs accesibles
- **show log [name]**: imprime el último segmento de registro en la memoria, 'global' es el predeterminado
- **use <db_name>**: establece la base de datos actual, creándola si es necesario
- **db**: muestra el nombre de la base de datos actual
- **db.dropDatabase()**: se usa para eliminar una base de datos existente.
- **exit**: sale de la shell de mongo

MongoDB: Comandos

- ***coleccion*.count**: número de documentos
- ***coleccion*.findOne**: recuperar un documento
- ***coleccion*.find**: recuperar varios documentos
- ***coleccion*.insert**: inserta un documento, crea la colección si es necesario
- ***coleccion*.update**: modifica parcialmente documentos
- ***coleccion*.save**: guardar/actualizar un documento
- ***coleccion*.remove**: borrar uno o varios documentos
- ***coleccion*.rename**: cambia de nombre la colección
- ***coleccion*.drop**: elimina la colección

MongoDB: Spring Boot

- Añadir al proyecto:
  - NoSQL > Spring Data MongoDB
- Configurar:

```
#spring.data.mongodb.uri=mongodb://localhost:27017/db
# Por defecto usa la base de datos "test"
spring.data.mongodb.database=db
#spring.data.mongodb.host=localhost
#spring.data.mongodb.port=27017
#spring.data.mongodb.username=
#spring.data.mongodb.password=
#spring.data.mongodb.repositories.enabled=true
```

- Repositorio:
  - MongoRepository<T,ID> extends PagingAndSortingRepository<T,ID>
    - findAll, insert, saveAll

Anotaciones NoSQL

Anotación	Descripción
@Document	de clase, para indicar que esta clase es un documento para la asignación a la base de datos (puede especificarse el nombre de la colección donde se almacenará la base de datos).
@CompoundIndex	de clase, para declarar índices compuestos.
@Id	de campo, para marcar el campo utilizado para la identidad.
@Field	de campo, para indicar el nombre en la base de datos
@DBRef	de campo, para indicar que se debe almacenar utilizando un com.mongodb.DBRef.
@Indexed	de campo, para describir cómo se indexa.
@GeoSpatialIndexed	de campo, para describir cómo se geoindexa.
@Transient	excluye el campo del almacenamiento en la base de datos
@PersistenceConstructor	marca el constructor (incluso en un paquete protegido), para usar cuando se crea una instancia del objeto desde la base de datos. Los argumentos del constructor se asignan por nombre a los valores clave en el DBObject recuperado.
@Value	se puede aplicar a los argumentos del constructor para usar una declaración de Spring Expression Language en la transformación del valor de una clave recuperada en la base de datos antes de que se use para construir un objeto de dominio.

Repositorio (MongoDB)

P. Clave	Muestra	Resultado lógico
After	findByBirthdateAfter(Date date)	{"birthdate" : {"$gt" : date}}
GreaterThan	findByAgeGreaterThan(int age)	{"age" : {"$gt" : age}}
GreaterThanEqual	findByAgeGreaterThanEqual(int age)	{"age" : {"$gte" : age}}
Before	findByBirthdateBefore(Date date)	{"birthdate" : {"$lt" : date}}
LessThan	findByAgeLessThan(int age)	{"age" : {"$lt" : age}}
LessThanEqual	findByAgeLessThanEqual(int age)	{"age" : {"$lte" : age}}
Between	findByAgeBetween(int from, int to)	{"age" : {"$gt" : from, "$lt" : to}}
In	findByAgeIn(Collection ages)	{"age" : {"$in" : [ages…]}}
NotIn	findByAgeNotIn(Collection ages)	{"age" : {"$nin" : [ages…]}}
IsNotNull, NotNull	findByFirstnameNotNull()	{"firstname" : {"$ne" : null}}
IsNull, Null	findByFirstnameNull()	{"firstname" : null}

Repositorio (MongoDB)

P. Clave	Muestra	Resultado lógico
Like, StartingWith, EndingWith	findByFirstnameLike(String name)	{"firstname" : name} (name as regex)
NotLike, IsNotLike	findByFirstnameNotLike(String name)	{"firstname" : { "$not" : name }} (name as regex)
Containing on String	findByFirstnameContaining(String name)	{"firstname" : name} (name as regex)
NotContaining on String	findByFirstnameNotContaining(String name)	{"firstname" : { "$not" : name}} (name as regex)
Containing on Collection	findByAddressesContaining(Address address)	{"addresses" : { "$in" : address}}
NotContaining on Collection	findByAddressesNotContaining(Address address)	{"addresses" : { "$not" : { "$in" : address}}}
Regex	findByFirstnameRegex(String firstname)	{"firstname" : {"$regex" : firstname }}
(No keyword)	findByFirstname(String name)	{"firstname" : name}
Not	findByFirstnameNot(String name)	{"firstname" : {"$ne" : name}}

Repositorio (MongoDB)

P. Clave	Muestra	Resultado lógico
Near	findByLocationNear(Point point)	{"location" : {"$near" : [x,y]}}
Near	findByLocationNear(Point point, Distance max)	{"location" : {"$near" : [x,y], "$maxDistance" : max}}
Near	findByLocationNear(Point point, Distance min, Distance max)	{"location" : {"$near" : [x,y], "$minDistance" : min, "$maxDistance" : max}}
Within	findByLocationWithin(Circle circle)	{"location" : {"$geoWithin" : {"$center" : [[x, y], distance]}}}
Within	findByLocationWithin(Box box)	{"location" : {"$geoWithin" : {"$box" : [[x1, y1], x2, y2]}}}
IsTrue, True	findByActiveIsTrue()	{"active" : true}
IsFalse, False	findByActiveIsFalse()	{"active" : false}
Exists	findByLocationExists(boolean exists)	{"location" : {"$exists" : exists }}

Repositorio

- Consultas basadas en JSON:

```
@Query("{'age' : {'$gt' : 67}}")
List<Profesor> findJubilados();

@Query(value="{ 'nombre' : ?0 }", fields="{ 'nombre' : 1, 'apellidos' : 1}")
List<ProfesorShort> findByNombre(String nombre);
```

- Consultas basadas en JSON con expresiones SpEL

```
@Query("{'lastname': ?#{[0]} }")
List<Person> findByQueryWithExpression(String param0);

@Query("{'id': ?#{ [0] ? {$exists :true} : [1] }}")
List<Person> findByQueryWithExpressionAndNestedObject(boolean param0, String param1);
```

Redis

- Redis (https://redis.io) es un almacén estructurado de datos en memoria de código abierto, que se utiliza como base de datos, caché y agente de mensajes.
- Basado en el almacenamiento en tablas de hashes (clave/valor) es compatible con estructuras de datos como cadenas, hashes, listas, conjuntos, conjuntos ordenados con consultas de rango, mapas de bits, hiperloglogs, índices geoespaciales con consultas de radio y flujos.
- Redis tiene replicación incorporada, scripts Lua, desalojo de LRU, transacciones y diferentes niveles de persistencia en disco, y proporciona alta disponibilidad a través de Redis Sentinel y partición automática con Redis Cluster.
- La popularidad de Redis se debe en gran medida a su espectacular velocidad, ya que mantiene la información en memoria; pero también a su sencillez de uso, productividad y flexibilidad.
- Sin embargo lo que hace que Redis sea tan popular, es que además de permitir asociar valores de tipo string a una clave, permite utilizar tipos de datos avanzados, que junto a las operaciones asociadas a estos tipos de datos, logra resolver muchos casos de uso de negocio, que a priori no pensaríamos que fuéramos capaces con clave-valor. No en vano, Redis se define usualmente como un servidor de estructuras de datos, aunque en sus inicios, haciendo honor a su nombre, fuese un simple diccionario remoto (REmote DIctionary Server).

Redis: Casos de uso

- Almacenamiento de sesiones de usuario
  - Podemos usar Redis como un almacén de sesiones de muy rápido acceso, en el que mantengamos el identificador de sesión junto con toda la información asociada a la misma. Además de reducir los tiempos de latencia de nuestra solución, igual que en el caso anterior evitaremos una vez más accesos a otras bases de datos. Además Redis permite asociar un tiempo de expiración a cada clave, con lo que las sesiones finalizarán automáticamente sin tener que gestionarlo en el código de la aplicación.
- Contadores y estadísticas
  - Para muchos casos de uso es necesario manejar contadores y estadísticas en tiempo real, y Redis tiene soporte para la gestión concurrente y atómica de los mismos. Algunos ejemplos posibles serían el contador de visualización de un producto, votos de usuarios, o contadores de acceso a un recurso para poder limitar su uso.
- Listas de elementos recientes
  - Es muy habitual mostrar listas en las que aparecen las últimas actualizaciones de algún elemento hechas por los usuarios. Por ejemplo, los últimos comentarios sobre un post, las últimas imágenes subidas, los artículos de un catálogo vistos recientemente, etc. Este tipo de operaciones suele ser muy costoso para las bases de datos relacionales, sobre todo cuando el volumen de información se va haciendo mayor, pero Redis es capaz de resolver esta operación con independencia del volumen.

Redis: Casos de uso

- Almacenamiento de carritos de la compra
  - De forma muy similar al almacén de sesiones de usuario, podemos almacenar en Redis los artículos contenidos en la cesta de la compra de un usuario, y la información asociada a los mismos, permitiéndonos acceder en cualquier momento a ellos con una latencia mínima.
- Caché de páginas web
  - Podemos usar Redis como caché de páginas HTML, o fragmentos de estas, lo que acelerará el acceso a las mismas, a la vez que evitamos llegar a los servidores web o de aplicaciones, reduciendo la carga en estos y en los sistemas de bases de datos a los que los mismos accedan. Además de un incremento en la velocidad, esto puede suponer un importante ahorro económico en términos de hardware y licencias de software.
- Caché de base de datos
  - Otra forma de descargar a las bases de datos operacionales es almacenar en Redis el resultado de determinadas consultas que se ejecuten con mucha frecuencia, y cuya información no cambia a menudo, o no es crítico mantener actualizada al instante.
- Base de datos principal
  - Para determinados casos, Redis se puede usar como almacenamiento principal gracias a la potencia de modelado que permiten sus avanzados tipos de datos. Destaca su uso en casos como los microservicios, en los que podemos aprovechar la velocidad de Redis para construir soluciones especializadas, simples de implementar y mantener, que a la vez ofrecen un alto

Redis: Comandos

- Valores:
  - APPEND
  - GET
  - GETBIT
  - GETRANGE
  - GETSET
  - BITCOUNT
  - BITFIELD
  - BITOP
  - BITPOS
  - MGET
  - MSET
  - MSETNX
  - PSETEX
  - SET
  - SETBIT
  - SETEX
  - SETNX
  - SETRANGE
  - STRLEN
- Contadores:
  - INCR
  - INCRBY
  - INCRBYFLOAT
  - DECR
  - DECRBY
- Listas
  - BLPOP
  - BRPOP
  - BRPOPLPUSH
  - LINDEX
  - LINSERT
  - LLEN
  - LPOP
  - LPUSH
  - LPUSHX
  - LRANGE
  - LREM
  - LSET
  - LTRIM
  - RPOP
  - RPOPLPUSH
  - RPUSH
  - RPUSHX
- Conjuntos:
  - SADD
  - SCARD
  - SDIFF
  - SDIFFSTORE
  - SINTER
  - SINTERSTORE
  - SISMEMBER
  - SMEMBERS
  - SMOVE
  - SPOP
  - SRANDMEMBER
  - SREM
  - SSCAN
  - SUNION
  - SUNIONSTORE
- SET Ordenados
- Hash
  - HDEL
  - HEXISTS
  - HGET
  - HGETALL
  - HINCRBY
  - HINCRBYFLOAT
  - HKEYS
  - HLEN
  - HMGET
  - HMSET
  - HSCAN
  - HSET
  - HSETNX
  - HSTRLEN
  - HVALS
- Mensajería:
  - PSUBSCRIBE
  - PUBLISH
  - PUBSUB
  - PUNSUBSCRIBE
  - SUBSCRIBE
  - UNSUBSCRIBE
- Administración
- Y muchos mas

Redis: Instalación

- Se puede utilizar un instalador o instalarla manualmente.
- Descargar la última versión estable para Windows desde https://github.com/MicrosoftArchive/redis/releases
- Puerto: 6379
- Instalación manual:
  - Crear una carpeta donde realizar la instalación
  - Descomprimir el fichero en el directorio
  - Instalar el servicio:
    - redis-server --service-install redis.windows-service.conf --loglevel verbose
- Para arrancar el servicio:
  - redis-server --service-start
- Para parar el servicio:
  - redis-server --service-stop
- Desde otra consola se puede acceder al interfaz de comandos con:
  - redis-cli

Redis: Spring Boot

- Añadir al proyecto:
  - NoSQL > Spring Data Redis (Access+Driver)
- Configurar:

  spring.data.redis.port=6379
  spring.data.redis.host=localhost
  #spring.data.redis.password=
- Uso de repositorios:
  - Anotaciones de entidad:
    - @RedisHash: de clase, para indicar que esta clase es parte de un conjunto para la asignación a la base de datos (debe especificarse el nombre del conjunto donde se almacenará en la base de datos).
    - @Id: de campo, para marcar el campo utilizado para la identidad.
  - Creación del repositorio

    public interface PersonasRepository extends PagingAndSortingRepository<Persona, String> { }

Redis: Repositorios

P.Clave	Muestra	Fragmento Redis
And	findByLastnameAndFirstname	SINTER …:firstname:rand …:lastname:al'thor
Or	findByLastnameOrFirstname	SUNION …:firstname:rand …:lastname:al'thor
Is, Equals	findByFirstname, findByFirstnameIs, findByFirstnameEquals	SINTER …:firstname:rand
IsTrue	FindByAliveIsTrue	SINTER …:alive:1
IsFalse	findByAliveIsFalse	SINTER …:alive:0
Top,First	findFirst10ByFirstname, findTop5ByFirstname	

Redis: RedisTemplate

```
@RestController
@RequestMapping(path = "/me-gusta")
public class ContadorResource {
	public final String ME_GUSTA_CONT="megusta";
	@Autowired
	private StringRedisTemplate template;
	private ValueOperations<String, String> redisValue;
	@PostConstruct
	private void inicializa() {
		redisValue = template.opsForValue();
		if(redisValue.get(ME_GUSTA_CONT) == null)
		    redisValue.set(ME_GUSTA_CONT, "0");
	}
	@GetMapping
	private String get() {
		return "Llevas " + redisValue.get(ME_GUSTA_CONT);
	}
	@PostMapping
	private String add() {
		return "Llevas " + redisValue.increment(ME_GUSTA_CONT);
	}
	@PutMapping("/{id}")
	private String add(@PathVariable int id) {
		long r = 0;
		Date ini = new Date();
		for(int i= 0; i++ < id*1000; r = redisValue.increment(ME_GUSTA_CONT));
		return "Llevas " + r + ". Tardo " + ((new Date()).getTime() - ini.getTime()) + " ms.";
	}
}
```

EVOLUCIÓN DEL DESARROLLO WEB

1990

Cliente

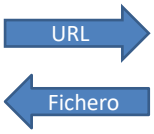
Tim Berners-Lee (CERN)

- ENQUIRE
  - Hipertexto
  - HTTP
  - HTML

Servidor

Robert Cailliau

- CERN httpd
  - Servidor de ficheros
  - HTTP – URL
  - Ficheros estáticos de texto



Origen

- La Web no fue concebida para el desarrollo de aplicaciones. El problema que se pretendía resolver su inventor, Tim Berners-Lee, era el cómo organizar información a través de enlaces.
- De hecho la Web nació en el laboratorio de partículas CERN básicamente para agrupar un conjunto muy grande de información y datos del acelerador de partículas que se encontraba muy dispersa y aislada.
- Mediante un protocolo muy simple (HTTP), un sistema de localización de recursos (URL) y un lenguaje de marcas (HTML) se podía poner a disposición de todo científico en el mundo la información existente en el CERN de tal forma que mediante enlaces se pudiese acceder a información relacionada con la consultada.

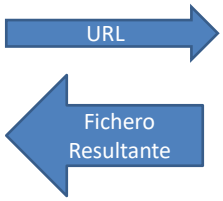
HTML

Cliente

- Navegadores
  - HTML

Servidor

- Servidor Web
  - Servidor de ficheros
  - Ficheros estáticos
    - Texto
    - HTML
  - Ficheros dinámicos
    - CGI (C, Perl)
    - …
    - Servlet



CGI

- Por la necesidad que el servidor Web pudiese devolver páginas Web dinámicas y no únicamente contenido estático residente en ficheros HTML se desarrolló la tecnología CGI (Common Gateway Interface) donde el servidor Web invocaba un programa el cual se ejecutaba, devolvía la página Web y el servidor Web remitía este flujo de datos al navegador.
- Un programa CGI podía ser cualquier programa que la máquina pudiese ejecutar: un programa en C, o en Visual Basic o en Perl. Normalmente se elegía este último por ser un lenguaje de script el cual podía ser traslado con facilidad de una arquitectura a otra. CGI era únicamente una pasarela que comunicaba el servidor Web con el ejecutable que devolvía la página Web.

Formularios

Cliente

- Navegadores
  - HTML 3.2
  - Formularios
  - Plug-in
    - Applet
    - ActiveX
  - Scripting de cliente
    - JavaScript

Servidor

- Servidor Web
  - Servidor de ficheros
  - Ficheros estáticos
    - Texto, HTML, Imágenes, …
  - Ficheros dinámicos
  - Sricpting de servidor
    - ASP
    - PHP
    - …
    - JSP

Scripting

- CGI era una solución cómoda de realizar páginas Web dinámicas pero tenía un grave problema de rendimiento que lo hizo insostenible en cuanto la demanda de la Web comenzó a disparar las peticiones de los servidores Web.
- Para agilizar esto, los principales servidores Web del momento (Netscape e IIS) desarrollaron un sistema para la ejecución dinámica de aplicaciones usando el propio contexto del servidor Web. En el caso de Netscape se le denominó NSAPI (Netscape Server Application Program Interface) y en el caso de IIS se le llamó ISAPI.

Web 2.0

Cliente

- Navegadores
  - HTML 4
  - CSS
  - DOM
  - AJAX
  - RIA
    - Shockwave Flash
    - Silverlight
  - JS Framework

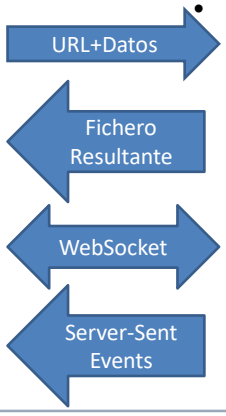
Servidor

- Servidor Web
  - Servidor de ficheros
  - Ficheros estáticos
  - Ficheros dinámicos
  - Sricpting de servidor
  - Servidor de aplicaciones
    - ASP.NET
    - J2EE
  - Web Services
    - WS XML
    - RestFul

Actualidad

Cliente

- Navegadores
  - HTML 5
  - CSS 3
  - ~~RIA~~
  - Móviles
  - JS RIA Framework
  - EcmaScript 6



Servidor

- Servidor Web
  - Servidor de ficheros
  - Ficheros estáticos
  - Ficheros dinámicos
  - Sricpting de servidor
  - Servidor de aplicaciones
    - NodeJS
  - Web Services
  - Server-Sent Events
    - Notificaciones PUSH

Aplicaciones web vs Sitios web

- Aplicaciones web
  - El objetivo principal de una aplicación es que el usuario realice una tarea. También pueden entenderse como un programa que se utiliza desde el navegador. Para crearlos, se usan los lenguajes CSS, HTML, JavaScript y se puede utilizar software gratuito de fuente abierta, como Drupal, Symfony o Meteor.
- Sitios web
  - El objetivo principal de un sitio web es entregar información. Por lo tanto, consumir contenidos es la tarea más importante hacen los usuarios en este tipo de plataformas.
  - Esta idea puede sonar confusa, ya que todos los sitios incluyen algún llamado a la acción adicional, como por ejemplo realizar un contacto o suscribirse a un newsletter. La diferencia está en que estas interacciones representan una parte pequeña y usualmente se pueden lograr solo después de guiar al usuario a través del contenido.

Aplicación Web

- Conjunto de páginas que residen en un directorio web y sus subdirectorios.
- Cualquier página puede ser el punto de entrada de la aplicación, no se debe confundir con el concepto de página principal con el de página por defecto.
- Externamente, no existe el concepto de aplicación como entidad única.
- Internamente, dependiendo de la tecnología utilizada una aplicación puede ser una entidad única o una colección de objetos independientes.

Ventajas e inconvenientes

Ventajas

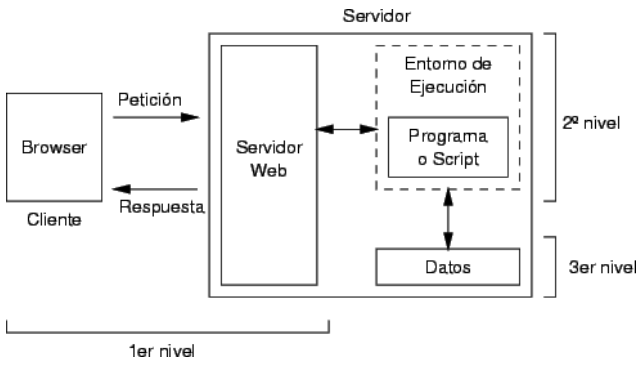
- Inmediatez y accesibilidad
- No ocupan espacio local
- Actualizaciones inmediatas
- No hay problemas de compatibilidad
- Multiplataforma
- Consumo de recursos bajo
- Portables
- Alta escalabilidad y disponibilidad

Inconvenientes

- Interfaz de interacción con el usuario muy limitada
- Base tecnológica inadecuada
- Incompatibilidades entre navegadores
- Bajo rendimiento al ser interpretado
- Cesión tecnológica
- Seguridad menos robusta

Arquitectura Web

Una aplicación Web típica recogerá datos del usuario (primer nivel), los enviará al servidor, que ejecutará un programa (segundo y tercer nivel) y cuyo resultado será formateado y presentado al usuario en el navegador (primer nivel otra vez).



HTTP

Petición

- Método HTTP
- URL (ruta)
  - Segmentos
  - Parámetros (query string)
  - Fragmento
- Encabezados
  - Cookies
- Cuerpo (body)
  - POST FORM

Respuesta

- Código de estado
- Encabezados
  - Cookies
- Cuerpo (body)

Enfoques

- Hay múltiples enfoques generales para crear una interfaz de usuario web moderna:
  - Sitios web:
    - Static Side Generation (SSG): Generación estática del sitio (compilados)
    - Incremental Static Regeneration (ISR): Regeneración estática incremental basada en tiempo o en caches.
  - Aplicaciones que representan la interfaz de usuario desde el servidor.
    - Server-Side Rendering (SSR)
    - Multi Page Application (MPA)
  - Aplicaciones que representan la interfaz de usuario en el cliente.
    - Client-Side Rendering (CSR)
    - Single Page Applications (SPA)
  - Aplicaciones híbridas que aprovechan los enfoques de representación de la interfaz de usuario en servidor y en cliente.

Interfaz de usuario representada en el servidor

- Una aplicación de interfaz de usuario web que se representa en el servidor genera dinámicamente el lenguaje HTML y CSS de la página en el servidor en respuesta a una solicitud del explorador. La página llega al cliente lista para mostrarse.
- Ventajas:
  - Los requisitos de cliente son mínimos porque el servidor realiza el trabajo de la lógica y la generación de páginas: Excelente para dispositivos de gama baja y conexiones de ancho de banda bajo, Permite una amplia gama de versiones de explorador en el cliente, Tiempos de carga de página iniciales rápidos, JavaScript mínimo o inexistente para ejecutar en el cliente.
  - Flexibilidad de acceso a recursos de servidor protegidos: Acceso de bases de datos, Acceso a secretos, como valores para las llamadas API a Azure Storage.
  - Ventajas del análisis de sitios estáticos, como la optimización del motor de búsqueda.
- Inconvenientes:
  - El costo del uso de proceso y memoria se centra en el servidor, en lugar de en cada cliente.
  - Lentitud, las interacciones del usuario requieren un recorrido de ida y vuelta al servidor para generar actualizaciones de la interfaz de usuario.

Interfaz de usuario representada por el cliente

- Una aplicación representada en el cliente representa dinámicamente la interfaz de usuario web en el cliente, actualizando directamente el DOM del explorador según sea necesario.
- Ventajas:
  - Permite una interactividad enriquecida que es casi instantánea, sin necesidad de un recorrido de ida y vuelta al servidor. El control de eventos de la interfaz de usuario y la lógica se ejecutan localmente en el dispositivo del usuario con una latencia mínima. Aprovecha las funcionalidades del dispositivo del usuario.
  - Admite actualizaciones incrementales, guardando formularios o documentos completados parcialmente sin que el usuario tenga que seleccionar un botón para enviar un formulario.
  - Se puede diseñar para ejecutarse en modo desconectado. Las actualizaciones del modelo del lado del cliente se sincronizan finalmente con el servidor una vez que se restablece la conexión.
  - Carga y costo del servidor reducidos, el trabajo se descarga en el cliente. Muchas aplicaciones representadas por el cliente también se pueden hospedar como sitios web estáticos.
- Inconvenientes:
  - El código de la lógica debe descargarse y ejecutarse en el cliente, lo que se agrega al tiempo de carga inicial.
  - Los requisitos de cliente pueden excluir a los usuarios que tienen dispositivos de gama baja, versiones anteriores del explorador o conexiones de ancho de banda bajo.

Single-page application (SPA)

- Un single-page application (SPA), o aplicación de página única es una aplicación web o es un sitio web que utiliza una sola página con el propósito de dar una experiencia más fluida a los usuarios como una aplicación de escritorio.
- En un SPA todo el códigos de HTML, JavaScript y CSS se carga de una sola vez o los recursos necesarios se cargan dinámicamente cuando lo requiera la página y se van agregando, normalmente como respuesta de los acciones del usuario.
- La página no se tiene que cargar otra vez en ningún punto del proceso, tampoco se transfiere a otra página, aunque las tecnologías modernas (como el pushState() API del HTML5) pueden permitir la navegabilidad en páginas lógicas dentro de la aplicación.
- La interacción con las aplicaciones de página única pueden involucrar comunicaciones dinámicas con el servidor web que está por detrás, habitualmente utilizando AJAX o WebSocket (HTML5).

Aplicaciones web progresivas

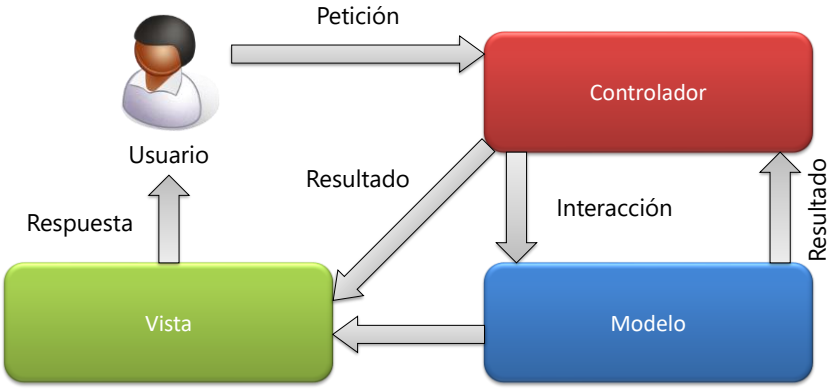
- Las aplicaciones web progresivas (mejor conocidas como PWAs por «Progressive Web Apps») no es un nombre oficial ni formal, es solo una abreviatura utilizada inicialmente por Google para el concepto de crear una aplicación flexible y adaptable utilizando solo tecnologías web.
- Es mucho más fácil y rápido visitar un sitio web que instalar una aplicación, y se pueden compartir enviando un enlace. Por otro lado, las aplicaciones nativas están mejor integradas con el sistema operativo y, por lo tanto, ofrecen una experiencia más fluida para los usuarios, se puede instalar una aplicación nativa para que funcione sin conexión, y los usuarios pueden tocar sus íconos para acceder fácilmente a sus aplicaciones favoritas sin tener que navegar. Las PWA nos brindan la capacidad de crear aplicaciones web que se comportan como nativas.
- Son aplicaciones web que utilizan APIs y funciones emergentes del navegador web junto a una estrategia tradicional de mejora progresiva para ofrecer una aplicación nativa, con la experiencia de usuario de aplicaciones web multiplataforma. Se puede pensar que PWA es similar a AJAX u otros patrones similares, son aplicaciones web desarrolladas con una serie de tecnologías específicas y patrones estándar que les permiten aprovechar las funciones de las aplicaciones nativas y web, aunque no son un estándar formalizado.
- Las PWA deben ser detectables, instalables, enlazables, independientes de la red, progresivas, reconectables, responsivas y seguras.

INTRODUCCIÓN A SPRING MVC

Spring Web MVC

- Spring Web MVC es el marco web original creado para dar soporte a las aplicaciones web de Servlet-Stack basadas en la API de Servlet y desplegadas en los contenedores de Servlet. Está incluido Spring Framework desde el principio.
- El Modelo Vista Controlador (MVC) es un patrón de arquitectura de software (presentación) que separa los datos y la lógica de negocio de una aplicación del interfaz de usuario y del módulo encargado de gestionar los eventos y las comunicaciones.
- Este patrón de diseño se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones, prueba y su posterior mantenimiento.
- Para todo tipo de sistemas (Escritorio, Web, Movil, …) y de tecnologías (Java, Ruby, Python, Perl, Flex, SmallTalk, .Net …)

El patrón MVC



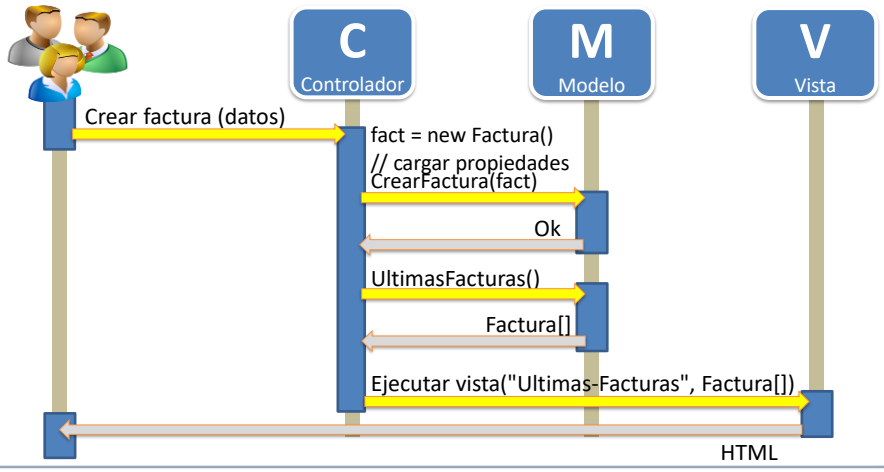
El patrón MVC

M Modelo
- Representación de los **datos del dominio**
- Lógica de **negocio**
- Mecanismos de **persistencia**

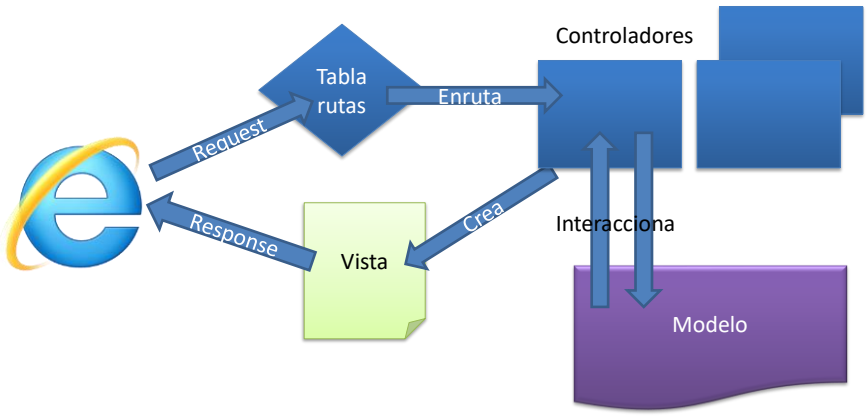
V Vista
- **Interfaz** de usuario
- Incluye elementos de **interacción**

C Controlador
- **Intermediario** entre Modelo y Vista
- **Mapea acciones** de usuario → acciones del Modelo
- **Selecciona** las vistas y les **suministra** información

El patrón MVC



Adaptación del MVC a aplicaciones Web



Controladores

- Los controladores son los componentes que controlan la interacción del usuario, trabajan con el modelo y por último seleccionan una vista para representar la interfaz de usuario.
- Exponen acciones que se encargan de procesar las peticiones
- Cada acción debe devolver un resultado, que es algo que el framework debe hacer (mandar una vista, un fichero binario, un 404, …)
- Hablan con el modelo pero son «tontos»

Modelo

- Los objetos de modelo son las partes de la aplicación que implementan la lógica del dominio de datos de la aplicación.
- A menudo, los objetos de modelo recuperan y almacenan el estado del modelo en una base de datos.
- Encapsula toda la lógica de nuestra aplicación.
- Responde a peticiones de los controladores.
- No interactúan con los usuarios
- Presentan la información en "crudo", no aplican formatos.

Vistas

- Las vistas son los componentes que muestra la interfaz de usuario de la aplicación.
- Normalmente, esta interfaz de usuario se crea a partir de los datos de modelo.
- Se encarga únicamente de temas de presentación.
- Es «básicamente» código HTML (con un poco de server-side)
- NO acceden a BBDD, NO toman decisiones, NO hacen nada de nada salvo…
- … mostrar información

Características

- El modelo MVC ayuda a crear aplicaciones que separan los diferentes aspectos de la aplicación (lógica de entrada, lógica de negocios y lógica de la interfaz de usuario), a la vez que proporciona un vago acoplamiento entre estos elementos.
- El modelo especifica dónde se debería encontrar cada tipo de lógica en la aplicación.
  - La lógica de la interfaz de usuario pertenece a la vista.
  - La lógica de entrada pertenece al controlador.
  - La lógica de negocios pertenece al modelo.
- Esta separación ayuda a administrar la complejidad al compilar una aplicación, ya que le permite centrarse en cada momento en un único aspecto de la implementación.
- El acoplamiento vago entre los tres componentes principales de una aplicación MVC también favorece el desarrollo paralelo.

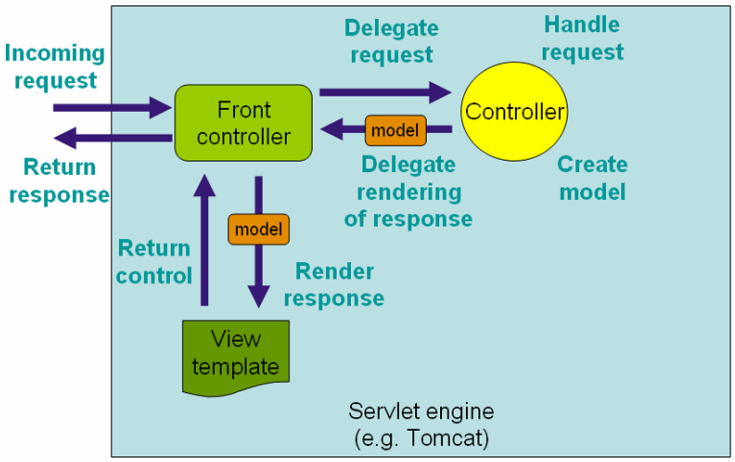
Características de Spring MVC

- Facilidad de mantenimiento.
- Separación limpia entre Controladores, Modelos y Vistas. Lo cual, a su vez, facilita el desarrollo en equipo disciplinado.
- Independencia en la Vista de la aplicación.
- Múltiples tecnologías a la hora de desarrollar las vistas.
- Facilidad a la hora de testear la aplicación.
- Binding y Validaciones.
- Facilidad de trabajar con los componentes gracias a la implementación del patrón de diseño DI "Dependency Injection"

Arquitectura

- El framework Spring MVC está diseñado en base a un DispatcherServlet, que se encarga de distribuir (como su nombre indica) las peticiones a distintos controladores.
- Permite configurar el mapeo de los controladores y resolución de vistas, excepciones, localización, zona horaria y temas.
- Los controladores, anotados con @Controller, son los responsables de preparar un modelo y elegir el nombre de la vista, pero pueden responder directamente y completar la petición si se desea.
- El modelo es una interfaz que permite abstraerse del tipo de tecnología que se está utilizando para la vista, transformándose al formato adecuado cuando se solicita.
- Las vistas pueden generarse utilizando tecnologías como JSP, Velocity, Freemaker o directamente a través de estructuras como XML o JSON.

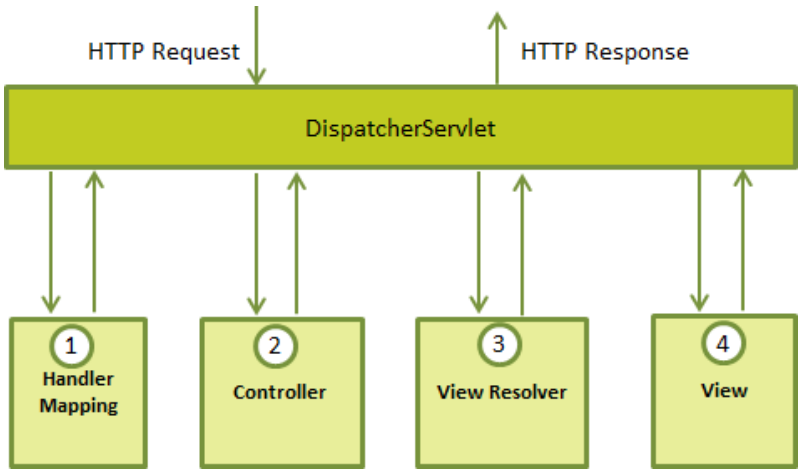
Procesamiento de una solicitud



Procesamiento de una solicitud

- Las peticiones pasan a través de un servlet que actúa de Front Controller (DispatcherServlet).
- El DispatcherServlet consulta a un HandlerMapping para decidir a que controlador le pasa la petición. Usa la URL de la solicitud para decidir.
- El controlador procesa la petición, accede a la lógica de negocio para obtener los resultados y selecciona la vista para presentar el resultado.
  - Para que el controlador no esté acoplado a la vista, se devuelve un identificador lógico (nombre) de vista.
  - Los resultados se encapsulan en un modelo (colección Map)
  - Devuelve al DispatcherServlet un objeto que encapsula ambos: ModelAndView.
- El DispatcherServlet utiliza un ViewResolver para resolver el nombre en una vista concreta
  - Normalmente un JSP, pero se soportan otros Thymeleaf, FreeMarker, XSLT, …
- El DispatcherServlet utiliza la vista para mostrar el modelo al usuario.
  - Puede ser HTML, PDF, XLS, etc.,

Procesamiento de una solicitud



Componentes

- HandlerMapping
  - Asigna una solicitud a un controlador junto con una lista de interceptores para el procesamiento previo y posterior. El mapeo se basa en diferentes criterios, cuyos detalles varían según la implementación HandlerMapping.
  - Las dos implementaciones principales de HandlerMapping son RequestMappingHandlerMapping (basada en las anotaciones @RequestMapping de los métodos) y SimpleUrlHandlerMapping (se define una tabla que asocia URLs a controladores).
- HandlerAdapter
  - Ayuda al DispatcherServlet a invocar al controlador asignado a una solicitud, independientemente de cómo se invoque realmente el controlador. Por ejemplo, invocar un controlador anotado requiere resolver anotaciones.
  - El propósito principal del HandlerAdapter es liberar al DispatcherServlet de tales detalles.

Componentes

- Los Resolver son los encargados de solventar diferentes aspectos a la hora de presentar la respuesta:
  - ViewResolver
    - Resolver la cadena devuelta desde un controlador con el nombre de vista convirtiéndola en una vista real View que procesa la respuesta.
  - HandlerExceptionResolver
    - Estrategia para resolver las excepciones, posiblemente asignándolas a controladores, a vistas de error HTML u otros destinos.
  - LocaleResolver, LocaleContextResolver
    - Resuelve que Locale y, posiblemente, la zona horaria está utilizando un cliente, para poder ofrecer vistas internacionalizadas.
  - ThemeResolver
    - Resuelve los temas que la aplicación web puede usar, por ejemplo, para ofrecer diseños personalizados.
  - MultipartResolver
    - Abstracción para analizar una solicitud de varias partes (por ejemplo, la carga de un archivo desde formulario) con la ayuda de alguna biblioteca.

Configuración

- Con Spring Boot la configuración se puede realizar en application.properties:

  spring.mvc.view.prefix=/WEB-INF/vistas/

  spring.mvc.view.suffix=.jsp
- En un entorno Servlet 3.0+, existe la opción de configurar el contenedor Servlet mediante programación como alternativa o en combinación con un archivo web.xml. El directorio /WEB-INF contiene todos los recursos relacionados con la aplicación web que no se han colocado en el directorio raíz y que no deben servirse al cliente. Esto es importante, ya que este directorio no forma parte del documento público, por lo que ninguno de los ficheros que contenga va a poder ser enviado directamente a través del servidor web.
- La interfaz WebMvcConfigurer define por defecto (Java 8) los métodos que permiten cambiar la configuración MVC de Spring, basta con crear una clase anotada con @Configuration que implemente la interfaz y sobrecargar los métodos que controlen las configuraciones que se desean personalizar.
- El método addViewControllers permite mapear controladores a las URLs dadas para que renderizen la vista dependiendo del nombre dado.

Configuración

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
public class AppMvcConfig implements WebMvcConfigurer {
	@Bean
	public ViewResolver viewResolver() {
		InternalResourceViewResolver resolver = new InternalResourceViewResolver();
		resolver.setPrefix("/WEB-INF/vistas/");
		resolver.setSuffix(".jsp");
		return resolver;
	}
	@Override
	public void addViewControllers(ViewControllerRegistry registry) {
		registry.addViewController("/home").setViewName("home");
	}

}
```


Estilos de comunicación

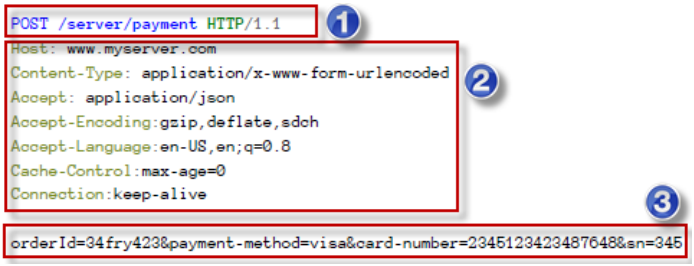
SERVICIOS REST

REST (REpresentational State Transfer)

- En 2000, Roy Fielding propuso la transferencia de estado representacional (REST) como enfoque de arquitectura para el diseño de servicios web. REST **es un estilo de arquitectura** para la creación de sistemas distribuidos basados en hipermedia. REST es independiente de cualquier protocolo subyacente y no está necesariamente unido a HTTP. Sin embargo, en las implementaciones más comunes de REST se usa HTTP como protocolo de aplicación, y esta guía se centra en el diseño de API de REST para HTTP.
- Originalmente se basaba en lo que ya estaba disponible en HTTP:
  - URL como identificadores de recursos
  - HTTP ya define 8 métodos (algunas veces referidos como "verbos") que indica la acción que desea que se efectúe sobre el recurso identificado: HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT + PATCH (HTTP1.1)
  - HTTP permite transmitir en el encabezado la información de comportamiento: Content-type, Accept, Authorization, Cache-control, …
  - HTTP utiliza códigos de estado en la respuesta para indicar como se ha completado una solicitud HTTP específica: respuestas informativas (1xx), respuestas satisfactorias (2xx), redirecciones (3xx), Errores en la petición (4xx) y errores de los servidores (5xx).

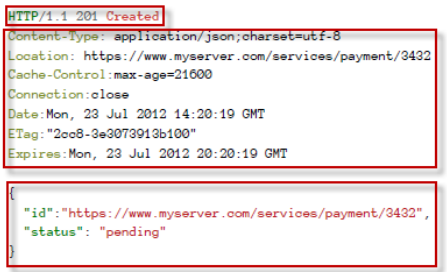
Petición HTTP

- Cuando realizamos una petición HTTP, el mensaje consta de:
  - Primera línea de texto indicando la versión del protocolo utilizado, el verbo y el URI
    - El verbo indica la acción a realizar sobre el recurso web localizado en la URI
  - Posteriormente vendrían las cabeceras (opcionales)
  - Después el cuerpo del mensaje, que contiene un documento, que puede estar en cualquier formato (XML, HTML, JSON → Content-type)



Respuesta HTTP

- Los mensajes HTTP de respuesta siguen el mismo formato que los de envío.
- Sólo difieren en la primera línea
  - Donde se indica un código de respuesta junto a una explicación textual de dicha respuesta.
  - El código de respuesta indica si la petición tuvo éxito o no.



Recursos

- Un recurso es cualquier elemento que dispone de un URI correcto y único, cualquier tipo de objeto, dato o servicio que sea direccionable a través de internet.
- Un recurso es un objeto que es lo suficientemente importante como para ser referenciado por sí mismo. Un recurso tiene datos, relaciones con otros recursos y métodos que operan contra él para permitir el acceso y la manipulación de la información asociada. Un grupo de recursos se llama colección. El contenido de las colecciones y los recursos depende de los requisitos de la organización y de los consumidores.
- En REST todos los recursos comparten una interfaz única y constante, la URI. (https://...)
- Todos los recursos tienen las mismas operaciones (CRUD)
  - CREATE, READ, UPDATE, DELETE
- Normalmente estos recursos son accesibles en una red o sistema.
- Las URI son el único medio por el que los clientes y servidores pueden realizar el intercambio de representaciones.
- Para que un URI sea correcto, debe cumplir los requisitos de formato, REST no indica de forma específica un formato obligatorio.
  - <esquema>://<host>:puerto/<ruta><querystring><fragmento>
- Los URI asociados a los recursos pueden cambiar si modificamos el recurso (nombre, ubicación, características, etc)

Tipos MIME

- Otro aspecto muy importante es la posibilidad de negociar distintos formatos (representaciones) a usar en la transferencia del estado entre servidor y cliente (y viceversa). La representación de los recursos es el formato de lo que se envía un lado a otro entre clientes y servidores. Como REST utiliza HTTP, podemos transferir múltiples tipos de información.
- Los datos se transmiten a través de TCP/IP, el navegador sabe cómo interpretar las secuencias binarias (CONTENT-TYPE) por el protocolo HTTP.
- La representación de un recurso depende del tipo de llamada que se ha generado (Texto, HTML, PDF, etc).
- En HTTP cada uno de estos formatos dispone de su propio tipos MIME, en el formato <tipo>/<subtipo>.
  - application/json application/xml text/html text/plain image/jpeg
- Para negociar el formato:
  - El cliente, en la cabecera ACCEPT, envía una lista priorizada de tipos MIME que entiende.
  - Tanto cliente como servidor indican en la cabecera CONTENT-TYPE el formato MIME en que está codificado el body.
- Si el servidor no entiende ninguno de los tipos MIME propuestos (ACCEPT) devuelve un mensaje con código 406 (incapaz de aceptar petición).

Métodos HTTP

HTTP	REST	Descripción
GET	RETRIEVE	Sin identificador: Recuperar el estado completo de un recurso (HEAD + BODY) Con identificador: Recuperar el estado individual de un recurso (HEAD + BODY)
HEAD		Recuperar la cabecera del estado de un recurso (HEAD)
POST	CREATE or REPLACE	Crea o modifica un recurso (sin identificador)
PUT	CREATE or REPLACE	Crea o modifica un recurso (con identificador)
DELETE	DELETE	Sin identificador: Elimina todo el recurso Con identificador: Elimina un elemento concreto del recurso
CONNECT		Comprueba el acceso al host
TRACE		Solicita al servidor que introduzca en la respuesta todos los datos que reciba en el mensaje de petición
OPTIONS		Devuelve los métodos HTTP que el servidor soporta para un URL específico
PATCH	REPLACE	HTTP 1.1 Reemplaza parcialmente un elemento del recurso

Códigos HTTP (status)

status	statusText	Descripción
100	Continue	Una parte de la petición (normalmente la primera) se ha recibido sin problemas y se puede enviar el resto de la petición
101	Switching protocols	El servidor va a cambiar el protocolo con el que se envía la información de la respuesta. En la cabecera Upgrade indica el nuevo protocolo
200	OK	La petición se ha recibido correctamente y se está enviando la respuesta. Este código es con mucha diferencia el que mas devuelven los servidores
201	Created	Se ha creado un nuevo recurso (por ejemplo una página web o un archivo) como parte de la respuesta
202	Accepted	La petición se ha recibido correctamente y se va a responder, pero no de forma inmediata
203	Non-Authoritative Information	La respuesta que se envía la ha generado un servidor externo. A efectos prácticos, es muy parecido al código 200
204	No Content	La petición se ha recibido de forma correcta pero no es necesaria una respuesta
205	Reset Content	El servidor solicita al navegador que inicialice el documento desde el que se realizó la petición, como por ejemplo un formulario
206	Partial Content	La respuesta contiene sólo la parte concreta del documento que se ha solicitado en la petición

Códigos de redirección

status	statusText	Descripción
300	Multiple Choices	El contenido original ha cambiado de sitio y se devuelve una lista con varias direcciones alternativas en las que se puede encontrar el contenido
301	Moved Permanently	El contenido original ha cambiado de sitio y el servidor devuelve la nueva URL del contenido. La próxima vez que solicite el contenido, el navegador utiliza la nueva URL
302	Found	El contenido original ha cambiado de sitio de forma temporal. El servidor devuelve la nueva URL, pero el navegador debe seguir utilizando la URL original en las próximas peticiones
303	See Other	El contenido solicitado se puede obtener en la URL alternativa devuelta por el servidor. Este código no implica que el contenido original ha cambiado de sitio
304	Not Modified	Normalmente, el navegador guarda en su caché los contenidos accedidos frecuentemente. Cuando el navegador solicita esos contenidos, incluye la condición de que no hayan cambiado desde la última vez que los recibió. Si el contenido no ha cambiado, el servidor devuelve este código para indicar que la respuesta sería la misma que la última vez
305	Use Proxy	El recurso solicitado sólo se puede obtener a través de un proxy, cuyos datos se incluyen en la respuesta
307	Temporary Redirect	Se trata de un código muy similar al 302, ya que indica que el recurso solicitado se encuentra de forma temporal en otra URL

Códigos de error en la petición

status	statusText	Descripción
400	Bad Request	El servidor no entiende la petición porque no ha sido creada de forma correcta
401	Unauthorized	El recurso solicitado requiere autorización previa
402	Payment Required	Código reservado para su uso futuro
403	Forbidden	No se puede acceder al recurso solicitado por falta de permisos o porque el usuario y contraseña indicados no son correctos
404	Not Found	El recurso solicitado no se encuentra en la URL indicada. Se trata de uno de los códigos más utilizados y responsable de los típicos errores de *Página no encontrada*
405	Method Not Allowed	El servidor no permite el uso del método utilizado por la petición, por ejemplo por utilizar el método GET cuando el servidor sólo permite el método POST
406	Not Acceptable	El tipo de contenido solicitado por el navegador no se encuentra entre la lista de tipos de contenidos que admite, por lo que no se envía en la respuesta
407	Proxy Authentication Required	Similar al código 401, indica que el navegador debe obtener autorización del proxy antes de que se le pueda enviar el contenido solicitado
408	Request Timeout	El navegador ha tardado demasiado tiempo en realizar la petición, por lo que el servidor la descarta

Códigos de error en la petición

status	statusText	Descripción
409	Conflict	El navegador no puede procesar la petición, ya que implica realizar una operación no permitida (como por ejemplo crear, modificar o borrar un archivo)
410	Gone	Similar al código 404. Indica que el recurso solicitado ha cambiado para siempre su localización, pero no se proporciona su nueva URL
411	Length Required	El servidor no procesa la petición porque no se ha indicado de forma explícita el tamaño del contenido de la petición
412	Precondition Failed	No se cumple una de las condiciones bajo las que se realizó la petición
413	Request Entity Too Large	La petición incluye más datos de los que el servidor es capaz de procesar. Normalmente este error se produce cuando se adjunta en la petición un archivo con un tamaño demasiado grande
414	Request-URI Too Long	La URL de la petición es demasiado grande, como cuando se incluyen más de 512 bytes en una petición realizada con el método GET
415	Unsupported Media Type	Al menos una parte de la petición incluye un formato que el servidor no es capaz procesar
416	Requested Range Not Suitable	El trozo de documento solicitado no está disponible, como por ejemplo cuando se solicitan bytes que están por encima del tamaño total del contenido
417	Expectation Failed	El servidor no puede procesar la petición porque al menos uno de los valores incluidos en la cabecera Expect no se pueden cumplir

Códigos de error del servidor

status	statusText	Descripción
500	Internal Server Error	Se ha producido algún error en el servidor que impide procesar la petición
501	Not Implemented	Procesar la respuesta requiere ciertas características no soportadas por el servidor
502	Bad Gateway	El servidor está actuando de proxy entre el navegador y un servidor externo del que ha obtenido una respuesta no válida
503	Service Unavailable	El servidor está sobrecargado de peticiones y no puede procesar la petición realizada
504	Gateway Timeout	El servidor está actuando de proxy entre el navegador y un servidor externo que ha tardado demasiado tiempo en responder
505	HTTP Version Not Supported	El servidor no es capaz de procesar la versión HTTP utilizada en la petición. La respuesta indica las versiones de HTTP que soporta el servidor

Estilo de arquitectura

- Las APIs de REST se diseñan en torno a recursos, que son cualquier tipo de objeto, dato o servicio al que puede acceder el cliente.
- Un recurso tiene un identificador, que es un URI que identifica de forma única ese recurso.
- Los clientes interactúan con un servicio mediante el intercambio de representaciones de recursos.
- Las APIs de REST usan una interfaz uniforme, que ayuda a desacoplar las implementaciones de clientes y servicios. En las APIs REST basadas en HTTP, la interfaz uniforme incluye el uso de verbos HTTP estándar para realizar operaciones en los recursos. Las operaciones más comunes son GET, POST, PUT, PATCH y DELETE. El código de estado de la respuesta indica el éxito o error de la petición.
- Las APIs de REST usan un modelo de solicitud sin estado. Las solicitudes HTTP deben ser independientes y pueden producirse en cualquier orden, por lo que no es factible conservar la información de estado transitoria entre solicitudes. El único lugar donde se almacena la información es en los propios recursos y cada solicitud debe ser una operación atómica.
- Las APIs de REST se controlan mediante vínculos de hipermedia.

Estilo de arquitectura

- Métodos GET
  - Petición de consulta a la URL sin identificador (todas las instancias del recurso) o con identificador (una instancia) y la cabecera accept para la respuesta.
  - Normalmente, un método GET correcto devuelve el código de estado HTTP 200 (Correcto), el cuerpo de respuesta contiene una representación del recurso y la cabecera content-type. Si no se encuentra el recurso, el método debe devolver HTTP 404 (No encontrado).
- Métodos POST
  - Petición de crear o reemplazar a la URL sin identificador, la instancia en el cuerpo, la cabecera content-type y, si espera respuesta, la cabecera accept.
  - Si un método POST crea un nuevo recurso, devuelve el código de estado HTTP 201 (Creado), el URI del nuevo recurso se incluye en el encabezado Location de la respuesta. Con el código de estado HTTP 200, el cuerpo de respuesta contiene una representación del recurso y la cabecera content-type.
  - Si no crea un nuevo recurso, puede devolver el código de estado HTTP 200 e incluir el resultado de la operación en el cuerpo de respuesta y la cabecera content-type. O bien, si no hay ningún resultado para devolver, devolverá el código de estado HTTP 204 (Sin contenido) y sin cuerpo de respuesta.
  - Si el cliente coloca datos no válidos en la solicitud, el servidor debe devolver el código de estado HTTP 400 (Solicitud incorrecta), un 409 (Conflicto) o un 412 (fallo de concurrencia). El cuerpo de respuesta puede contener información adicional sobre el error o un vínculo a un URI que proporciona más detalles.

Estilo de arquitectura

- Métodos PUT
  - Petición de crear o reemplazar (idempotente) a la URL con identificador, la instancia en el cuerpo, la cabecera content-type y, si espera respuesta, la cabecera accept.
  - Al igual que con un método POST, si un método PUT crea un nuevo recurso, devuelve el código de estado HTTP 201 (Creado), y si actualiza un recurso existente, devuelve un 200 (Correcto) o un 204 (Sin contenido). Si el cliente coloca datos no válidos en la solicitud, el servidor debe devolver un 400 (Solicitud incorrecta), si no es posible actualizar el recurso existente devolverá un 409 (Conflicto), un 412 (fallo de concurrencia) o el recurso no existe, puede devolver un 404 (No encontrado).
  - Hay que considerar la posibilidad de implementar operaciones HTTP PUT masivas que pueden procesar por lotes las actualizaciones de varios recursos de una colección. La solicitud PUT debe especificar el URI de la colección y el cuerpo de solicitud debe especificar los detalles de los recursos que se van a modificar. Este enfoque puede ayudar a reducir el intercambio de mensajes y mejorar el rendimiento.
- Métodos DELETE
  - Petición de borrado a la URL sin identificador (todas las instancias del recurso) o con identificador (una instancia)
  - El método debe responder con un 204 (Sin contenido), que indica que la operación de eliminación es correcta, el cuerpo de respuesta no contiene información adicional. Si el recurso no existe, puede devolver un 404 (No encontrado), un 409 (Conflicto) o un 412 (fallo de concurrencia).

Estilo de arquitectura

- Métodos PATCH
  - Petición de reemplazo parcial a la URL con identificador, en el cuerpo el conjunto de cambios a aplicar, la cabecera content-type y, si espera respuesta, la cabecera accept.
  - Con una solicitud PATCH, el cliente envía un conjunto de actualizaciones a un recurso existente, en forma de un documento de revisión. El servidor procesa el documento de revisión para realizar la actualización que, por definición, no es idempotente..
  - La estructura del documento de revisión debería de seguir una sintaxis estándar como JSON Patch (RFC6902) o JSON Merge (RFC7386).
  - Los formatos MIME del documento de revisión aceptados por el servidor deberían aparecer en la cabecera Accept-Patch en respuesta a una petición OPTIONS.
  - Si el método actualiza un recurso existente, devuelve un 200 (Correcto) o un 204 (Sin contenido). Si el cliente coloca datos no válidos en la solicitud o el documento de revisión tiene un formato incorrecto, el servidor debe devolver un 400 (Solicitud incorrecta), si no es posible actualizar el recurso existente devolverá un 409 (Conflicto) o un 412 (fallo de concurrencia), si no se admite el formato de documento de revisión devolverá un 415 (Tipo de medio no compatible) o el recurso no existe, puede devolver un 404 (No encontrado).

Encabezado HTTP Cache-Control

- El encabezado HTTP Cache-Control especifica directivas (instrucciones) para almacenar temporalmente (caching) tanto en peticiones como en respuestas. Una directiva dada en una petición no significa que la misma directiva estar en la respuesta.
- Los valores estándar que pueden ser usados por el servidor en una respuesta HTTP son:
  - public: La respuesta puede estar almacenada en cualquier memoria cache.
  - private: La respuesta puede estar almacenada sólo por el cache de un navegador.
  - no-cache: La respuesta puede estar almacenada en cualquier memoria cache pero DEBE pasar siempre por una validación con el servidor de origen antes de utilizarse.
  - no-store: La respuesta puede no ser almacenada en cualquier cache.
  - max-age=<seconds>: La cantidad máxima de tiempo un recurso es considerado reciente.
  - s-maxage=<seconds>: Anula el encabezado max-age o el Expires, pero solo para caches compartidos (e.g., proxies).
  - must-revalidate: Indica que una vez un recurso se vuelve obsoleto, el cache no debe usar su copia obsoleta sin validar correctamente en el servidor de origen.
  - proxy-revalidate: Similar a must-revalidate, pero solo para caches compartidos (es decir, proxies). Ignorado por caches privados.
  - no-transform: No deberían hacerse transformaciones o conversiones al recurso.

Encabezados HTTP ETag, If-Match y If-None-Match

- El encabezado de respuesta de HTTP ETag es un identificador (resumen hash) para una versión específica de un recurso y los encabezados If-Match e If-None-Match de la solicitud HTTP hace que la solicitud sea condicional.
- Para los métodos GET y HEAD con If-None-Match: si el ETag no coincide con los dados, el servidor devolverá el recurso solicitado con un estado 200, si coincide el servidor debe devolver el código de estado HTTP 304 (No modificado) y DEBE generar cualquiera de los siguientes campos de encabezado que se habrían enviado en una respuesta 200 (OK) a la misma solicitud: Cache-Control, Content-Location, Date, ETag, Expires y Vary.
- Para los métodos PUT y DELETE con If-Match: si el ETag coincide con los dados, se realiza la actualización o borrado y se devuelve un estado HTTP 204 (sin contenido) incluyendo el Cache-Control y el ETag de la versión actualizada del recurso en el PUT. Si no coinciden, se ha producido un error de concurrencia, la versión del servidor ha sido modificada desde que la recibió el cliente, debe devolver una respuesta HTTP con un cuerpo de mensaje vacío y un código de estado 412 (Precondición fallida).
- Si los datos solicitados ya no existen, el servido debe devolver una respuesta HTTP con el código de estado 404 (no encontrado).

Estilo de arquitectura

- Request: Método /uri?parámetros
  - GET: Recupera el recurso (200)
    - Todos: Sin identificador
    - Uno: Con identificador
  - POST: Crea o reemplaza un nuevo recurso (201)
  - PUT: Crea o reemplaza el recurso identificado (200, 204)
  - DELETE: Elimina el recurso (204)
    - Todos: Sin identificador
    - Uno: Con identificador
- Cabeceras:
  - Accept: Indica al servidor el formato o posibles formatos esperados, utilizando MIME.
  - Content-type: Indica en que formato está codificado el cuerpo, utilizando MIME
- HTTP Status Code: Código de estado con el que el servidor informa del resultado de la petición.

Diseño de un Servicio Web REST

- Para el desarrollo de los Servicios Web's REST es necesario definir una serie de cosas:
  - Analizar el/los recurso/s a implementar
  - Diseñar la REPRESENTACION del recurso.
    - Deberemos definir el formato de trabajo del recurso: XML, JSON, HTML, imagen, RSS, etc
  - Definir el URI de acceso.
    - Deberemos indicar el/los URI de acceso para el recurso
  - Establecer los métodos soportados por el servicio
    - GET, POST, PUT, DELETE
  - Fijar qué códigos de estado pueden ser devueltos
    - Los códigos de estado HTTP típicos que podrían ser devueltos
- Todo lo anterior dependerá del servicio a implementar.

Definir operaciones

- Sumario y descripción de la operación.
- Dirección: URL
  - Sin identificador
  - Con identificador
  - Con parámetros de consulta
- Método: GET | POST | PUT | DELETE | PATCH
- Solicitud:
  - Cabeceras:
    - ACCEPT: formatos aceptables si espera recibir datos
    - CONTENT-TYPE: formato de envío de los datos en la solicitud
    - Otras cabeceras: Authorization, Cache-control, X-XSRF-TOKEN, …
  - Cuerpo: en caso de envío, estructura de datos formateados según el CONTENT-TYPE.
- Respuesta:
  - Cabeceras:
    - Códigos de estado HTTP: posibles y sus causas.
    - CONTENT-TYPE: formato de envío de los datos en la respuesta
    - Otras cabeceras
  - Cuerpo: en caso de respuesta, estructura de datos según código de estado y formateados según el CONTENT-TYPE.

Richardson Maturity Model

http://www.crummy.com/writing/speaking/2008-QCon/act3.html

- Nivel 0: Definir un URI y todas las operaciones son solicitudes POST a este URI.
- Nivel 1 (Pobre): Crear distintos URI para recursos individuales pero utilizan solo un método.
  - Se debe identificar un recurso
    /entities/?invoices =2 → entities/invoices/2
  - Se construyen con nombres nunca con verbos
    /getUser/{id} → /users/{id}/
    /users/{id}/edit/login → users/{id}/access-token
  - Deberían tener una estructura jerárquica
    /invoices/user/{id} → /user/{id}/invoices
- Nivel 2 (Medio): Usar métodos HTTP para definir operaciones en los recursos.
- Nivel 3 (Óptimo): Usar hipermedia (HATEOAS, se describe a continuación).

Hypermedia

- Uno de los principales propósitos que se esconden detrás de REST es que debe ser posible navegar por todo el conjunto de recursos sin necesidad de conocer el esquema de URI. Cada solicitud HTTP GET debe devolver la información necesaria para encontrar los recursos relacionados directamente con el objeto solicitado mediante los hipervínculos que se incluyen en la respuesta, y también se le debe proporcionar información que describa las operaciones disponibles en cada uno de estos recursos.
- Este principio se conoce como HATEOAS, del inglés Hypertext as the Engine of Application State (Hipertexto como motor del estado de la aplicación). El sistema es realmente una máquina de estado finito, y la respuesta a cada solicitud contiene la información necesaria para pasar de un estado a otro; ninguna otra información debería ser necesaria.
- Se basa en la idea de enlazar recursos: propiedades que son enlaces a otros recursos.
- Para que sea útil, el cliente debe saber que en la respuesta hay contenido hypermedia.
- En content-type es clave para esto
  - Un tipo genérico no aporta nada:
    Content-Type: text/xml
  - Se pueden crear tipos propios
    Content-Type:application/servicio+xml

JSON Hypertext Application Language

- RFC4627 http://tools.ietf.org/html/draft-kelly-json-hal-00
- HATEOAS: Content-Type: application/hal+json

```
{
    "_links": {
      "self": {"href": "/orders/523" },
      "warehouse": {"href": "/warehouse/56" },
      "invoice": {"href": "/invoices/873"}
    },
    , "currency": "USD"
    , "status": "shipped"
    , "total": 10.20
}
```


Características de una API bien diseñada

- **Fácil de leer y trabajar**: con una API bien diseñada será fácil trabajar, y sus recursos y operaciones asociadas pueden ser memorizados rápidamente por los desarrolladores que trabajan con ella constantemente.
- **Difícil de usar mal**: la implementación e integración con una API con un buen diseño será un proceso sencillo, y escribir código incorrecto será un resultado menos probable porque tiene comentarios informativos y no aplica pautas estrictas al consumidor final de la API.
- **Completa y concisa**: Finalmente, una API completa hará posible que los desarrolladores creen aplicaciones completas con los datos que expone. Por lo general, la completitud ocurre con el tiempo, y la mayoría de los diseñadores y desarrolladores de API construyen gradualmente sobre las APIs existentes. Es un ideal por el que todo ingeniero o empresa con una API debe esforzarse.

Guía de diseño

- Organización de la API en torno a los recursos
- Definición de operaciones en términos de métodos HTTP
- Conformidad con la semántica HTTP
- Filtrado y paginación de los datos
- Compatibilidad con respuestas parciales en recursos binarios de gran tamaño
- Uso de HATEOAS para permitir la navegación a los recursos relacionados
- Control de versiones en la API RESTful
- Documentación Open API

Definición de recursos

- La organización de la API en torno a los recursos se centran en las entidades de dominio que debe exponer la API. Por ejemplo, en un sistema de comercio electrónico, las entidades principales podrían ser clientes y pedidos. La creación de un pedido se puede lograr mediante el envío de una solicitud HTTP POST que contiene la información del pedido. La respuesta HTTP indica si el pedido se realizó correctamente o no.
- Un recurso no tiene que estar basado en un solo elemento de datos físico o tablas de una base de datos relacional. La finalidad de REST es modelar entidades y las operaciones que un consumidor externo puede realizar sobre esas entidades, no debe exponerse a la implementación interna.
- Es necesario adoptar una convención de nomenclatura coherente para los URI. Los URI de recursos deben basarse en nombres (de recurso), nunca en verbos (las operaciones en el recurso) y, en general, resulta útil usar nombres plurales que hagan referencia a colecciones. Debe seguir una estructura jerárquica que refleje las relaciones entre los diferentes tipos de recursos.
- Hay que considerar el uso del enfoque HATEOAS para permitir el descubrimiento y la navegación a los recursos relacionados o el enfoque del patrón agregado.

Definición de recursos

- Exponer una colección de recursos con un único URI puede dar lugar a que las aplicaciones capturen grandes cantidades de datos cuando solo se requiere un subconjunto de la información. A través de la definición de parámetros de cadena de consulta se pueden realizar particiones horizontales con filtrado, ordenación y paginación, o particiones verticales con la proyección de las propiedades a recuperar:
  - https://host/users?page=1&rows=20&projection=userId,name,lastAccess
- La definición de operaciones con el recurso se realiza en términos de métodos HTTP, estableciendo cuales serán soportadas. Las operaciones no soportadas por métodos HTTP deben sustantivarse al crearles una URI especifica y utilizar el método HTTP semánticamente mas próximo.
  - DELETE https://host/users/bloqueo (desbloquear)
  - POST https://host/pedido/171/factura (facturar)
- Hay que establecer el tipo o tipos de formatos mas adecuados para las representaciones de recursos. Los formatos se especifican mediante el uso de tipos de medios, también denominados tipos MIME. En el caso de datos no binarios, la mayoría de las APIs web admiten JSON (application/json) y, posiblemente, XML (application/xml).

Políticas de versionado

- Es muy poco probable que una API permanezca estática. Conforme los requisitos empresariales cambian, se pueden agregar nuevas colecciones de recursos, las relaciones entre los recursos pueden cambiar y la estructura de los datos de los recursos debe adecuarse.
- Los cambios rupturistas no son compatibles con la versión anterior, el consumidor tendrá que adaptar su código para pasar su aplicación existente a la nueva versión y evitar que se rompa.
- Hay dos razones principales por las que las APIs HTTP se comportan de manera diferente al resto de las APIs:
  - El código del cliente dicta lo que lo romperá: Un proveedor de API no tiene control sobre las herramientas que un consumidor puede usar para interpretar una respuesta de la API y la tolerancia al cambio que tienen esas herramientas varían ampliamente, si es rupturista o no.
  - El proveedor de API elige si los cambios son opcionales o transparentes: Los proveedores de API pueden actualizar su API y los cambios en las respuestas afectarán inmediatamente a los clientes. Los clientes no pueden decidir si adoptar o no la nueva versión, lo que puede generar fallos en cascada en los cambios rupturistas.

Políticas de versionado

- El control de versiones permite que una API indique la versión expuesta y que una aplicación cliente pueda enviar solicitudes que se dirijan a una versión específica con una característica o un recurso.
  - Sin control de versiones: Este es el enfoque más sencillo y puede ser aceptable para algunas APIs internas. Los grandes cambios podrían representarse como nuevos recursos o nuevos vínculos.
  - Control de versiones en URI: Cada vez que modifica la API web o cambia el esquema de recursos, agrega un número de versión al URI para cada recurso. Los URI ya existentes deben seguir funcionando como antes y devolver los recursos conforme a su esquema original.

    http://host/v2/users
  - Control de versiones en cadena de consulta: En lugar de proporcionar varios URI, se puede especificar la versión del recurso mediante un parámetro dentro de la cadena de consulta anexada a la solicitud HTTP:

    http://host/users?versión=2.0

Políticas de versionado

  - Control de versiones en encabezado: En lugar de anexar el número de versión como un parámetro de cadena de consulta, se podría implementar un encabezado personalizado que indica la versión del recurso. Este enfoque requiere que la aplicación cliente agregue el encabezado adecuado a las solicitudes, aunque el código que controla la solicitud de cliente puede usar un valor predeterminado (versión actual) si se omite el encabezado de versión.

    GET https://host/users HTTP/1.1

    Custom-Header: api-version=1
  - Control de versiones por MIME (tipo de medio): Cuando una aplicación cliente envía una solicitud HTTP GET a un servidor web, debe prever el formato del contenido que puede controlar mediante el uso de un encabezado Accept.

    GET https:// host/users/3 HTTP/1.1

    Accept: application/vnd.mi-api.v1+json
- Si la versión no está soportada, el servicio podría generar un mensaje de respuesta HTTP 406 (no aceptable) o devolver un mensaje con un tipo de medio predeterminado.
- Los esquemas de control de versiones de URI y de cadena de consulta son compatibles con la caché HTTP puesto que la misma combinación de URI y cadena de consulta hace referencia siempre a los mismos datos.

Políticas de versionado

- Dentro de la política de versionado es conveniente planificar la obsolescencia y la política de desaprobación.
- La obsolescencia programada establece el periodo máximo, como una franja temporal o un número de versiones, en que se va a dar soporte a cada versión, evitando los sobrecostes derivados de mantener versiones obsoletas indefinidamente.
- Dentro de la política de desaprobación, para ayudar a garantizar que los consumidores tengan tiempo suficiente y una ruta clara de actualización, se debe establecer el número de versiones en que se mantendrá una característica marcada como obsoleta antes de su desaparición definitiva.
- La obsolescencia programada y la política de desaprobación beneficia a los consumidores de la API porque proporcionan estabilidad y sabrán qué esperar a medida que las APIs evolucionen.
- Para mejorar la calidad y avanzar las novedades, se podrán realizar lanzamientos de versiones Beta y Release Candidatos (RC) o revisiones para cada versión mayor y menor. Estas versiones provisionales desaparecerán con el lanzamiento de la versión definitiva.

Guía de implementación

- Procesamiento de solicitudes
  - Las acciones GET, PUT, DELETE, HEAD y PATCH deben ser idempotentes.
  - Las acciones POST que crean nuevos recursos no deben tener efectos secundarios no relacionados.
  - Evitar implementar operaciones POST, PUT y DELETE que generen mucha conversación.
  - Seguir la especificación HTTP al enviar una respuesta.
  - Admitir la negociación de contenido.
  - Proporcionar vínculos que permitan la navegación y la detección de recursos de estilo HATEOAS.

Guía de implementación

- Administración de respuestas y solicitudes de gran tamaño
  - Optimizar las solicitudes y respuestas que impliquen objetos grandes.
  - Admitir la paginación de las solicitudes que pueden devolver grandes cantidades de objetos.
  - Implementar respuestas parciales para los clientes que no admitan operaciones asincrónicas.
  - Evitar enviar mensajes de estado 100-Continuar innecesarios en las aplicaciones cliente.
- Mantenimiento de la capacidad de respuesta, la escalabilidad y la disponibilidad
  - Ofrecer compatibilidad asincrónica para las solicitudes de ejecución prolongada.
  - Comprobar que ninguna de las solicitudes tenga estado.
  - Realizar un seguimiento de los clientes e implementar limitaciones para reducir las posibilidades de ataques de denegación de servicio.
  - Administrar con cuidado las conexiones HTTP persistentes.

Guía de implementación

- Control de excepciones
  - Capturar todas las excepciones y devolver una respuesta significativa a los clientes.
  - Distinguir entre los errores del lado cliente y del lado servidor.
  - Evitar las vulnerabilidades por exceso de información.
  - Controlar las excepciones de una forma coherente y registrar la información sobre los errores.
- Optimización del acceso a los datos en el lado cliente
  - Admitir el almacenamiento en caché del lado cliente.
  - Proporcionar ETags para optimizar el procesamiento de las consultas.
  - Usar ETags para admitir la simultaneidad optimista.

Guía de implementación

- Publicación y administración de una API web
  - Todas las solicitudes deben autenticarse y autorizarse, y debe aplicarse el nivel de control de acceso adecuado.
  - Una API web comercial puede estar sujeta a diversas garantías de calidad relativas a los tiempos de respuesta. Es importante asegurarse de que ese entorno de host es escalable si la carga puede variar considerablemente con el tiempo.
  - Puede ser necesario realizar mediciones de las solicitudes para fines de monetización.
  - Es posible que sea necesario regular el flujo de tráfico a la API web e implementar la limitación para clientes concretos que hayan agotado sus cuotas.
  - Los requisitos normativos podrían requerir un registro y una auditoría de todas las solicitudes y respuestas.
  - Para garantizar la disponibilidad, puede ser necesario supervisar el estado del servidor que hospeda la API web y reiniciarlo si hiciera falta.

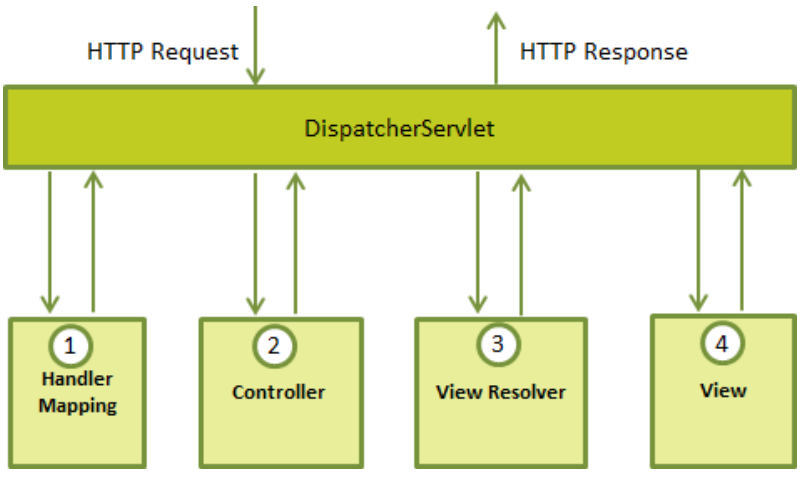
Guía de implementación

- Pruebas de la API
  - Ejercitar todas las rutas y parámetros para comprobar que invocan las operaciones correctas.
  - Verificar que cada operación devuelve los códigos de estado HTTP correctos para diferentes combinaciones de entradas.
  - Comprobar que todas las rutas estén protegidas correctamente y que estén sujetas a las comprobaciones de autenticación y autorización apropiadas.
  - Verificar el control de excepciones que realiza cada operación y que se devuelve una respuesta HTTP adecuada y significativa de vuelta a la aplicación cliente.
  - Comprobar que los mensajes de solicitud y respuesta están formados correctamente.
  - Comprobar que todos los vínculos dentro de los mensajes de respuesta no están rotos.

Spring Web MVC

- Spring Web MVC es el marco web original creado para dar soporte a las aplicaciones web de Servlet-stack basadas en la API de Servlet y desplegadas en los contenedores de Servlet. Está incluido Spring Framework desde el principio.
- El Modelo Vista Controlador (MVC) es un patrón de arquitectura de software (presentación) que separa los datos y la lógica de negocio de una aplicación del interfaz de usuario y del módulo encargado de gestionar los eventos y las comunicaciones.
- Este patrón de diseño se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones, prueba y su posterior mantenimiento.
- Para todo tipo de sistemas (Escritorio, Web, Movil, …) y de tecnologías (Java, Ruby, Python, Perl, Flex, SmallTalk, .Net …)

Procesamiento de una solicitud



Recursos

- Son clases Java con la anotación @RestController (@Controller + @ResponseBody) y representan a los servicios REST, son controller que reciben y responden a las peticiones de los clientes.

```
@RestController
public class PaisController {
        @Autowired
        private PaisRepository paisRepository;
```

- Los métodos de la clase (métodos de acción u operaciones) que interactúan con el cliente deben llevar la anotación @RequestMapping, con la subruta y el RequestMethod.

```
@RequestMapping(value = "/paises/{id}", method = RequestMethod.GET)
public ResponseEntity<Pais> getToDoById(@PathVariable("id") String id) {
      return new ResponseEntity<Pais>(paisRepository.findById(id).get(), HttpStatus.OK);
}
```


RequestMapping

- La anotación @RequestMapping permite asignar solicitudes a los métodos de acción de los controladores.
- Tiene varios atributos para definir URL, método HTTP, parámetros de solicitud, encabezados y tipos de medios.
- Se puede usar a el nivel de clase para expresar asignaciones compartidas o a el nivel de método para limitar a una asignación de endpoint específica.
- También hay atajos con el método HTTP predefinido:
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping
  - @PatchMapping

Patrones de URI

- Establece que URLs serán derivadas al controlador.
- Puede asignar solicitudes utilizando los siguientes patrones globales y comodines:

?	Coincide con un carácter
*	Coincide con cero o más caracteres dentro de un segmento de ruta
**	Coincide con cero o más segmentos de ruta hasta el final de la ruta. Solo puede ir al final del patrón.
{param}	Coincide con un segmento de ruta (*) y lo captura como un parámetro
	Con {param:\d+} debe coincidir con la expresión regular
	Con {*param} captura hasta el final de la ruta. Solo puede ir al final del patrón.

- También puede declarar variables en la URI y acceder a sus valores con anotando con @PathVariable los parámetros, debe respetarse la correspondencia de nombres:

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
  // ...
}
```


Restricciones

- cosumes: formatos MIME permitidos del encabezado Content-type

```
@PostMapping(path = "/pets", consumes = "application/json")
public void addPet(@RequestBody Pet pet) {
```

- produces : formatos MIME permitidos del encabezado Accept

```
@GetMapping(path = "/pets/{petId}", produces = "application/json;charset=UTF-8")
@ResponseBody
public Pet getPet(@PathVariable String petId) {
```

- params: valores permitidos de los QueryParams
- headers: valores permitidos de los encabezados

```
@GetMapping(path = "/pets/{petId}", params = "myParam=myValue", headers =
    "myHeader=myValue")
public void findPet(@PathVariable String petId) {
```

Inyección de Parámetros

- El API decodifica la petición e inyecta los datos como parámetros en el método.
- Es necesario anotar los parámetros para indicar la fuente del dato a inyectar.
- En las anotaciones será necesario indicar el nombre del origen en caso de no existir correspondencia de nombres con el de los parámetros.
- El tipo de origen, en la mayoría de los casos, es String que puede discrepar con los tipos de los parámetros, en tales casos, la conversión de tipo se aplica automáticamente en función de los convertidores configurados.
- Por defecto los parámetros son obligatorios, se puede indicar que sean opcionales, se inicializaran a null si no reciben en la petición salvo que se indique el valor por defecto:

```
@RequestParam(required=false, defaultValue="1")
```


Inyección de Parámetros

Anotación	Descripcción
@PathVariable	Para acceder a las variables de la plantilla URI.
@MatrixVariable	Para acceder a pares nombre-valor en segmentos de ruta URI.
@RequestParam	Para acceder a los parámetros de solicitud del Servlet (QueryString o Form), incluidos los archivos de varias partes. Los valores de los parámetros se convierten al tipo de argumento del método declarado.
@RequestHeader	Para acceder a las cabeceras de solicitud. Los valores de encabezado se convierten al tipo de argumento del método declarado.
@CookieValue	Para el acceso a las cookies. Los valores de las cookies se convierten al tipo de argumento del método declarado.

Inyección de Parámetros

Anotación	Descripcción
@RequestBody	Para acceder al cuerpo de la solicitud HTTP. El contenido del cuerpo se convierte al tipo de argumento del método declarado utilizando implementaciones HttpMessageConverter.
@RequestPart	Para acceder a una parte en una solicitud multipart/form-data, convertir el cuerpo de la parte con un HttpMessageConverter.
~~@ModelAttribute~~	~~Para acceder a un atributo existente en el modelo (instanciado si no está presente) con enlace de datos y validación aplicada.~~
@SessionAttribute	Para acceder a cualquier atributo de sesión, a diferencia de los atributos de modelo almacenados en la sesión como resultado de una declaración @SessionAttributes de nivel de clase .
@RequestAttribute	Para acceder a los atributos de solicitud.

Inyecciones adicionales

- Se definen los parámetros de los tipos adecuados Spring inyectara los objetos indicados.
- Para acceder a los valores originales de los parámetros anotados:
  - WebRequest, NativeWebRequest, MultipartRequest, ServletRequest, ServletResponse, MultipartHttpServletRequest, HttpSession, SessionStatus, HttpMethod,
- Para acceder a la configuración regional de la solicitud actual y la zona horaria asociada
  - java.util.Locale, java.util.TimeZone
- Para acceder al usuario autenticado actual
  - java.security.Principal
- Para acceder al cuerpo de la solicitud o de la respuesta sin procesar
  - java.io.InputStream, java.io.Reader, java.io.OutputStream, java.io.Writer
- Para acceder al modelo
  - java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap
- Para acceder a los errores de validación y enlace de datos
  - Errors, BindingResult
- Para preparar una URL relacionada con la solicitud actual
  - UriComponentsBuilder

Paginación y Ordenación

QueryString	Descripción
page	Número de página en base 0. Por defecto: página 0.
size	Tamaño de página. Por defecto: 20.
sort	Propiedades de ordenación en el formato property,property(,ASC|DESC). Por defecto: ascendente. Hay que utilizar varios sort para diferente direcciones (?sort=firstname&sort=lastname,asc)

```
@GetMapping
public List<Employee> getAll(@PageableDefault(size = 10) Pageable pageable) {
    if(pageable.isPaged()) {
        return dao.findAll(pageable).getContent();
    } else
        return dao.findAll();
}
```

Los nombres de los parámetros se pueden configurar:

```
spring.data.web.pageable.page-parameter=_page
spring.data.web.pageable.size-parameter=_rows
spring.data.rest.sort-param-name=_sort
```


@RequestBody

- Se puede utilizar la anotación @RequestBody para que el cuerpo de la solicitud se lea y se deserialice a parámetro a través de HttpMessageConverter. Se pueden usar convertidores de mensajes para configurar o personalizar la conversión de mensajes.

```
@PostMapping("/accounts")
public void handle(@RequestBody Account account) {
```

- En combinación con la anotación javax.validation.Valid o la de Spring @Validated, se aplica la validación estándar de Bean. De forma predeterminada, los errores de validación causan una MethodArgumentNotValidException, que se convierte en una respuesta 400 (BAD_REQUEST). Alternativamente, se pueden manejar los errores de validación localmente dentro del controlador a través de un argumento Errors o BindingResult:

```
@PostMapping("/region")
public void handle(@Valid @RequestBody Region item, BindingResult result) {
```

Inyección de Parámetros

```
// http://localhost:8080/params/1?nom=kk

@GetMapping("/params/{id}")
public String cotilla(
      @PathVariable String id,
      @RequestParam String nom,
      @RequestHeader("Accept-Language") String language,
      @CookieValue("JSESSIONID") String cookie) {
   StringBuilder sb = new StringBuilder();
   sb.append("id: " + id + "\n");
   sb.append("nom: " + nom + "\n");
   sb.append("language: " + language + "\n");
   sb.append("cookie: " + cookie + "\n");
   return sb.toString();
}
```


Respuesta

- La anotación @ResponseBody (incluida en el @RestController) en un método indica que el retorno será serializado en el cuerpo de la respuesta a través de un HttpMessageConverter.

```
@PostMapping("/invierte")
@ResponseBody
public Punto body(@RequestBody Punto p) {
      int x = p.getX();
      p.setX(p.getY());
      p.setY(x);
      return p;
}
```

- El código de estado de la respuesta se puede establecer con la anotación @ResponseStatus:

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public void add(@RequestBody Punto p) { … }
```

Respuesta personalizada

- La clase ResponseEntity permite agregar estado y encabezados a la respuesta (no requiere la anotación @ResponseBody).

```
@GetMapping(value="/pais")
public ResponseEntity<List<Pais>> getAll(){
    return new ResponseEntity<List<Pais>>(
            paisRepository.findAll(),
            HttpStatus.OK);
}
```

- La clase ResponseEntity dispone de builder para generar la respuesta:

```
return ResponseEntity.ok().eTag(etag).build(body);
```


Mapeo de respuestas genéricas a excepciones.

- Spring Framework no incluye automáticamente los detalles de error en el cuerpo de la respuesta porque la representación es específica de la aplicación.
- Los métodos de acción de los controladores pueden capturar las excepciones (try catch) y devuelven un ResponseEntity que permite establecer el estado y el cuerpo de la respuesta, tanto de las correctas como las erróneas.
- Una clase @RestController puede contar con métodos anotados con @ExceptionHandler que intercepten determinadas excepciones producidas en el resto de los métodos de la clase y pueden devolver un ResponseEntity, un ProblemDetail o estar anotadas con un @ResponseStatus y generar el cuerpo con los detalles de error.
- Esto mismo se puede hacer globalmente en clases anotadas con @ControllerAdvice que solo tienen los correspondientes métodos @ExceptionHandler.
- @RestControllerAdvice es una anotación compuesta que se anota con @ControllerAdvice y @ResponseBody, lo que esencialmente significa que los métodos @ExceptionHandler se representan en el cuerpo de la respuesta a través de la conversión del mensaje (en comparación con la resolución de la vista o la representación de la plantilla).

Excepciones personalizadas

```
public class NotFoundException extends Exception {
    private static final long serialVersionUID = 1L;
    public NotFoundException() {
        super("NOT FOUND");
    }
    public NotFoundException(String message) {
        super(message);
    }
    public NotFoundException(Throwable cause) {
        super("NOT FOUND", cause);
    }
    public NotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
    public NotFoundException(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}
```

Nota: *Pueden extender a org.springframework.web.ErrorResponseException si pertenecen a las capas de presentación.*

Error Personalizado

```
public class ErrorMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    private String error, message;
    public ErrorMessage(String error, String message) {
        this.error = error;
        this.message = message;
    }
    public String getError() { return error; }
    public void setError(String error) { this.error = error; }
    public String getMessage() { return message; }
    public void setMessage(String message) { this.message = message; }
}
```

Respuestas de error

- Un requisito común para los servicios REST es incluir detalles en el cuerpo de las respuestas de error. Spring Framework admite la especificación RFC 7807 para "Detalles del problema para las API HTTP". Las principales abstracciones son:
  - ProblemDetail: representación RFC 7807 del detalle de un problema; un contenedor simple para los campos estándar y no estándar definidos en la especificación.
  - ErrorResponse: interface para exponer los detalles de la respuesta de error de HTTP, incluido el estado de HTTP, los encabezados de respuesta y un cuerpo en el formato de RFC 7807; esto permite que las excepciones encapsulen y expongan los detalles de cómo se asignan a una respuesta HTTP. Todas las excepciones de Spring MVC lo implementan.
  - ErrorResponseException: implementación base de ErrorResponse que otras clases de excepción pueden usar como clase base.
  - ResponseStatusException: subclase de ErrorResponseException con el estado y una razón que, de forma predeterminada, asigna al status y detail de un ProblemDetail.
  - ResponseEntityExceptionHandler: clase base conveniente para un @ControllerAdvice que maneja todas las excepciones de Spring MVC, y cualquiera ErrorResponseException, y genera una respuesta de error con un cuerpo.

Problem Details (RFC 7807)

- El objeto de detalles del problema puede tener los siguientes miembros:
  - "type" (cadena): URI que identifica el tipo de problema y proporciona documentación legible por humanos para el tipo de problema. El valor "about:blank" (predeterminado) indica que el problema no tiene semántica adicional a la del código de estado HTTP.
  - "title" (cadena): Breve resumen legible por humanos del problema escribe. NO DEBE cambiar de una ocurrencia a otra del mismo problema, excepto para fines de localización. Con "type": "about:blank", DEBE coincidir con la versión textual del status.
  - "status" (número): Código de estado HTTP (por conveniencia, opcional, debe coincidir).
  - "detail" (cadena): Explicación legible por humanos específica de la ocurrencia concreta del problema.
  - "instance" (cadena): URI de referencia que identifica el origen de la ocurrencia del problema.
- Las definiciones de tipo de problema PUEDEN extender el objeto con miembros adicionales.

@RestControllerAdvice

```
@RestControllerAdvice
public class ApiExceptionHandler {
    @ExceptionHandler({ NotFoundException.class })
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ErrorMessage notFoundRequest(Exception exception) {
        return new ErrorMessage(exception.getMessage(), ServletUriComponentsBuilder.fromCurrentRequest().build().toUriString());
    }
    @ExceptionHandler({ NotFoundException.class })
    public ProblemDetail notFoundRequest(Exception exception) {
        return ProblemDetail.forStatus(HttpStatus.NOT_FOUND);
    }
    @ExceptionHandler({ ErrorResponseException.class })
    public ProblemDetail defaultResponse(ErrorResponse  exception) {
        return exception.getBody();
    }
    @ExceptionHandler({ BadRequestException.class, DuplicateKeyException.class })
    public ProblemDetail badRequest(Exception exception) {
        return ProblemDetail.forStatusAndDetail(HttpStatus.BAD_REQUEST, exception.getMessage());
    }
}
```


Controlador de error global

- La aplicación Spring Boot tiene una configuración predeterminada para el manejo de errores. Se puede configurar con:
  server.error.include-stacktrace=never
  server.error.include-binding-errors=always
- Si la aplicación tiene un controlador que lo implementa ErrorController, reemplaza a BasicErrorController.

```
@RestController
public class CustomErrorController implements ErrorController {
   @RequestMapping(path = "/error")
   public Map<String, Object> handle(HttpServletRequest request) {
      Map<String, Object> map = new HashMap<String, Object>();
      map.put("status", request.getAttribute("javax.servlet.error.status_code"));
      map.put("error", request.getAttribute("javax.servlet.error.message"));
      return map;
   }
```

Servicio Web RESTful

```
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Valid;
import jakarta.validation.Validator;
import jakarta.websocket.server.PathParam;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;
import org.springframework.http.HttpStatus;

import curso.api.exceptions.BadRequestException;
import curso.api.exceptions.NotFoundException;
import curso.model.Actor;
import curso.repositories.ActorRepository;
```


Servicio Web RESTful

```
@RestController
@RequestMapping("/api/actores")
public class ActorResource {
	@Autowired
	private ActorRepository dao;

	@Autowired
	private Validator validator;

	@GetMapping
	public List<Actor> getAll() {
		// …
	}

	@GetMapping(path = "/{id}")
	public Actor getOne(@PathVariable int id) throws NotFoundException {
		// …
	}
```

Servicio Web RESTful

```
    @PostMapping
    public ResponseEntity<Object> create(@Valid @RequestBody Actor item) throws BadRequestException {
        // ...
        URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
            .buildAndExpand(newItem.getActorId()).toUri();
        return ResponseEntity.created(location).build();

    }

    @PutMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void update(@PathVariable int id, @Valid @RequestBody Actor item) throws BadRequestException, NotFoundException {
        // ...
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void delete(@PathVariable int id) {
        // ..
    }
}
```


HATEOAS

- HATEOAS es la abreviación de Hypermedia as the Engine of Application State (hipermedia como motor del estado de la aplicación).
- Es una restricción de la arquitectura de la aplicación REST que lo distingue de la mayoría de otras arquitecturas.
- El principio es que un cliente interactúa con una aplicación de red completamente a través de hipermedia proporcionadas dinámicamente por los servidores de aplicaciones.
- El cliente REST debe ir navegando por la información y no necesita ningún conocimiento previo acerca de la forma de interactuar con cualquier aplicación o servidor más allá de una comprensión genérica de hipermedia.
- En otras palabras cuando el servidor nos devuelva la representación de un recurso parte de la información devuelta serán identificadores únicos en forma de hipervínculos a otros recursos asociados.
- Spring HATEOAS proporciona algunas API para facilitar la creación de representaciones REST que siguen el principio de HATEOAS cuando se trabaja con Spring y especialmente con Spring MVC. El problema central que trata de resolver es la creación de enlaces y el ensamblaje de representación

Spring HATEOAS

- La clase base ResourceSupport con soporte para la colección _links.

```
class PersonaDTO extends ResourceSupport {
```

- El objeto de valor Link sigue la definición de enlace Atom que consta de los atributos rel y href. Contiene algunas constantes para relaciones conocidas como self, next, etc.
- Spring Hateoas ahora proporciona una ControllerLinkBuilder que permite crear enlaces apuntando a clases de controladores:

```
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.*;
```

- Para añadir una referencia a si mismo:

```
personaDTO.add(linkTo(PersonaResource.class).withSelfRel());
personaDTO.add(linkTo(PersonaResource.class).slash(personaDTO.getId()).withSelfRel());
```

- Para añadir una referencia a si mismo como método:

```
personaDTO.add(linkTo(PersonaResource.class.getMethod("get", Long.class),
      personaDTO.getId()).withSelfRel());
```

- Para crear una referencia a un elemento interno:

```
personaDTO.add(linkTo(PersonaResource.class).
      slash(personaDTO.getId()).slash("direciones").withRel("direciones"));
```


Spring HATEOAS

- La interfaz EntityLinks permite generar la referencia a partir de la entidad del modelo.
- Para configurarlo:

```
@Configuration
@EnableEntityLinks
public class MyConfig {
```

- Hay que asociar las entidades a los RestController:

```
@RestController
@RequestMapping(value = "/api/personas")
@ExposesResourceFor(Persona.class)
public class PersonaResource {
```

- Se inyecta:

```
@Autowired EntityLinks entityLinks;
```

- Para añadir una referencia:

```
personaDTO.add(entityLinks.linkToSingleResource(PersonaResource.class, personaDTO.getId()).withSelfRel());
```

Spring Data Rest

- Spring Data REST se basa en los repositorios de Spring Data y los exporta automáticamente como recursos REST. Aprovecha la hipermedia para que los clientes encuentren automáticamente la funcionalidad expuesta por los repositorios e integren estos recursos en la funcionalidad relacionada basada en hipermedia. Spring Data REST es en sí misma una aplicación Spring MVC y está diseñada de tal manera que puede integrarse con las aplicaciones Spring MVC existentes con un mínimo esfuerzo.
- De forma predeterminada, Spring Data REST ofrece los recursos REST en la URI raíz, '/', se puede cambiar la URI base configurando en el fichero application.properties:
  - spring.data.rest.basePath=/api
- Dado que la funcionalidad principal de Spring Data REST es exportar como recursos los repositorios de Spring Data, el artefacto principal será la interfaz del repositorio.

Spring Data Rest

- Spring Data REST es expone los metodos del repositorio como métodos REST:
  - GET: findAll(), findAll(Pageable), findAll(Sort)
    - Si el repositorio tiene capacidades de paginación, el recurso toma los siguientes parámetros:
      - page: El número de página a acceder (base 0, por defecto a 0).
      - size: El tamaño de página solicitado (por defecto a 20).
      - sort: Una colección de directivas de género en el formato ($propertyname,)+[asc|desc]?.
  - POST, PUT, PATCH: save(item)
  - DELETE: deleteById(id)
- Devuelve el conjunto de códigos de estado predeterminados:
  - 200 OK: Para peticiones GET .
  - 201 Created: Para solicitudes POST y PUT que crean nuevos recursos.
  - 204 No Content: Para solicitudes PUT, PATCH y DELETE cuando está configurada para no devolver cuerpos de respuesta para actualizaciones de recursos (RepositoryRestConfiguration.returnBodyOnUpdate). Si se configura incluir respuestas para PUT, se devuelve 200 OK para las actualizaciones y 201 Created si crea nuevos recursos.

Spring Data Rest

- Para cambiar la configuración predeterminada del REST:

```
@RepositoryRestResource(path="personas", itemResourceRel="persona", collectionResourceRel="personas")
public interface PersonaRepository extends JpaRepository<Persona, Integer> {
    @RestResource(path = "por-nombre")
    List<Person> findByNombre(String nombre);
    // http://localhost:8080/personas/search/nombre?nombre=terry
```

- Para ocultar ciertos repositorios, métodos de consulta o campos

```
@RepositoryRestResource(exported = false)
interface PersonaRepository extends JpaRepository<Persona, Integer> {
    @RestResource(exported = false)
    List<Person> findByName(String name);
    @Override
    @RestResource(exported = false)
    void deleteById(Long id);
```


Spring Data Rest

- Spring Data REST presenta una vista predeterminada del modelo de dominio que exporta. Sin embargo, a veces, es posible que deba modificar la vista de ese modelo por varias razones.

  Mediante un interfaz **en el paquete de las entidades o en uno de subpaquetes** se crea un proyección con nombre:

```
@Projection(name = "personasAcortado", types = { Personas.class })
public interface PersonaProjection {
    public int getPersonaId();
    public String getNombre();
    public String getApellidos();
}
```

- Para acceder a la proyección:
  - http://localhost:8080/personas?projection=personasAcortado
- Para fijar la proyección por defecto:

```
@RepositoryRestResource(excerptProjection = PersonaProjection.class)
public interface PersonaRepository extends JpaRepository<Persona, Integer> {
```

Spring Data Rest

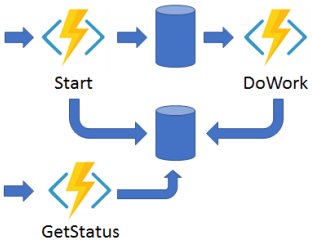
- Spring Data REST usa HAL para representar las respuestas, que define los enlaces que se incluirán en cada propiedad del documento devuelto.
- Spring Data REST proporciona un documento ALPS (Semántica de perfil de nivel de aplicación) para cada repositorio exportado que se puede usar como un perfil para explicar la semántica de la aplicación en un documento con un tipo de medio agnóstico de la aplicación (como HTML, HAL, Collection + JSON, Siren, etc.).
  - http://localhost:8080/profile
  - http://localhost:8080/profile/personas

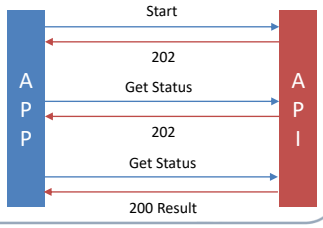
WebHooks

- Los webhooks son eventos HTTP que desencadenan acciones. Su nombre se debe a que funcionan como «enganches» de los programas en Internet y casi siempre se utilizan para la comunicación entre sistemas. Son la manera más sencilla de obtener un aviso cuando algo ocurre en otro sistema y para el intercambio de datos entre aplicaciones web, permitiendo las llamada asíncronas en HTTP.
- Un webhook es una retro llamada HTTP, una solicitud HTTP GET/POST insertada en una página web, que interviene cuando ocurre algo (una notificación de evento a través de HTTP GET/POST).
- Los webhooks se utilizan para las notificaciones en tiempo real (con los datos del evento como parámetros o en cuerpo en JSON o XML) a una determinada dirección http:// o https://, que puede:
  - almacenar los datos del evento en JSON o XML
  - generar una respuesta que permita actualizarse al sistema donde se produce el evento
  - ejecutar un proceso en el sistema receptor del evento (Ej: enviar un correo electrónico)
- Los webhook están pensados para su utilización desde páginas web y sus diferentes consumidores: navegadores, correo electrónico, webapps, …
- Un ejemplo típico es su utilización en correos electrónico de marketing para notificar al servidor que debe enviar un nuevo correo electrónico porque el usuario a abierto el mensaje.
- Pueden considerarse una versión especializada y simplificada de los servicios REST (solo GET/POST).

WebHooks

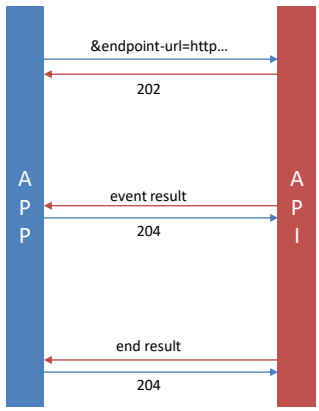
- El patrón Async HTTP APIs soluciona el problema de coordinar el estado de las operaciones de larga duración con los clientes externos. Una forma habitual de implementar este patrón es que un punto de conexión HTTP desencadene la acción de larga duración. A continuación, el cliente se redirige a un punto de conexión de estado al que sondea para saber el estado actual del proceso y cuando finaliza la operación.
- Un endpoint HTTP desencadena el proceso con la acción de larga duración y devuelve inmediatamente un 202 si la petición es correcta y, opcionalmente, como carga útil las URLs de los endpoints de estado y control. Las peticiones incorrectas recibirán el 4XX apropiado.
- El cliente sondea periódicamente el endpoint de estado y obtiene:
  - 202 con el estado actual del proceso como carga útil mientras el proceso este en marcha.
  - 200 con el estado final (correcto o fallido) del proceso como carga útil cuando haya finalizado la operación.
- El endpoint HTTP desencadenador puede suministrar endpoints adicionales para pausar, reanudar, cancelar/terminar, reiniciar o enviar eventos al proceso en marcha.





WebHooks

- El patrón Reverse API soluciona el problema del sondeo periódico, las API inversas invierten esta situación, para que sea la API invocada la que notifique automáticamente cuando se ha producido un determinado evento. El cliente debe crear su propia API para recibir las notificaciones.
- Al realizar la petición al endpoint HTTP desencadenador, se suministra un endpoint propio para que el proceso desencadenado pueda notificar cuándo ocurre algo de interés. Le da la vuelta a la comunicación, el cliente pasa a ser servidor y el servidor a cliente. El desencadenador devuelve inmediatamente un 202 si la petición es correcta o el 4XX apropiado se es incorrecta.
- Este esquema de funcionamiento tiene muchas ventajas en ambos extremos:
  - **Ahorro de recursos y tiempo**: con el sondeo se harán muchas llamadas "para nada", que no devolverán información relevante. Los dos extremos gastan recursos para hacer y responder a muchas llamadas que no tienen utilidad alguna (no nos llame, ya les llamaremos).
  - **Eliminación de los retrasos**: la aplicación usuaria recibirá una llamada en el momento exacto en el que se produce y no tendrá retrasos al próximo sondeo.
  - **Velocidad de las llamadas**: generalmente la llamada que se hace a un webhook es muy rápida porque solo se envía una pequeña información sobre el evento y se suele procesar asíncronamente. Muchas veces ni siquiera se espera por el resultado: se hace una llamada del tipo "fire and forget" (o sea, dispara y olvídate), pues se trata de notificar el evento y listo.



DOCUMENTACIÓN

Enfoque API First

- El enfoque basado en API First significa que, para cualquier proyecto de desarrollo dado, las APIs se tratan como "ciudadanos de primera clase": que todo sobre un proyecto gira en torno a la idea de que el producto final es un conjunto de APIs consumido por las aplicaciones del cliente.
- El enfoque de API First implica que los desarrollos de APIs sean consistentes y reutilizables, lo que se puede lograr mediante el uso de un lenguaje formal de descripción de APIs para establecer un contrato sobre cómo se supone que se comportará la API. Establecer un contrato implica pasar más tiempo pensando en el diseño de una API.
- A menudo también implica una planificación y colaboración adicionales con las partes interesadas, proporcionando retroalimentación de los consumidores sobre el diseño de una API antes de escribir cualquier código evitando costosos errores.

Beneficios de API First

- Los equipos de desarrollo pueden trabajar en paralelo.
  - Los equipos pueden simular APIs y probar sus dependencias en función de la definición de la API establecida.
- Reduce el coste de desarrollar aplicaciones
  - Las APIs y el código se pueden reutilizar en muchos proyectos diferentes.
- Aumenta la velocidad de desarrollo.
  - Gran parte del proceso de creación de API se puede automatizar mediante herramientas que permiten importar archivos de definición de API y generar el esqueleto del backend y el cliente frontend, así como un mocking server para las pruebas.

Beneficios de API First

- Asegura buenas experiencias de desarrollador
  - Las APIs bien diseñadas, bien documentadas y consistentes brindan experiencias positivas para los desarrolladores porque es más fácil reutilizar el código y los desarrollos integrados, reduciendo la curva de aprendizaje.
- Reduce el riesgo de fallos
  - Reduce el riesgo de fallos al facilitar las pruebas para garantizar que las APIs sean confiables, consistentes y fáciles de usar para los desarrolladores.

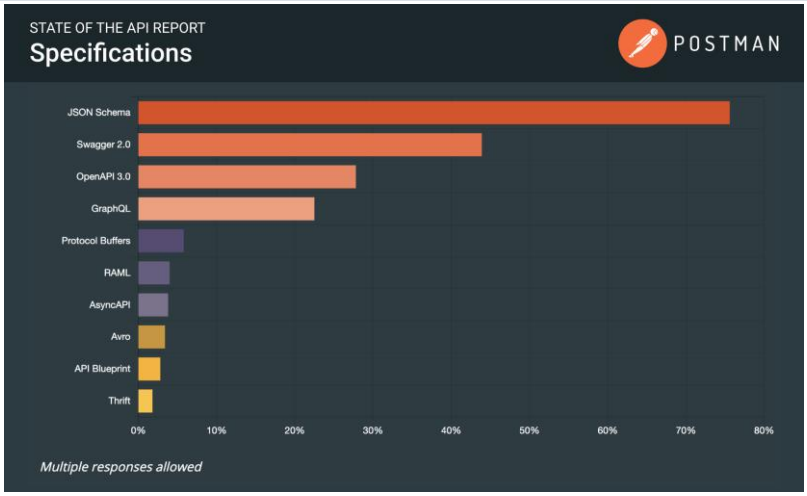
Documentar servicios Rest

- Dado que las API están diseñadas para ser consumidas, es importante asegurarse de que el cliente o consumidor pueda implementar rápidamente una API y comprender qué está sucediendo con ella. Desafortunadamente, muchas API hacen que la implementación sea extremadamente difícil, frustrando su propósito.
- La documentación es uno de los factores más importantes para determinar el éxito de una API, ya que la documentación sólida y fácil de entender hace que la implementación de la API sea muy sencilla, mientras que la documentación confusa, desincronizada, incompleta o intrincada hace que sea una aventura desagradable, una que generalmente conduce a desarrolladores frustrados a utilizar las soluciones de la competencia.
- Una buena documentación debe actuar como referencia y como formación, permitiendo a los desarrolladores obtener rápidamente la información que buscan de un vistazo, mientras también leen la documentación para obtener una comprensión de cómo integrar el recurso / método que están viendo.
- Con la expansión de especificaciones abiertas como OpenApi, RAML, … y las comunidades que las rodean, la documentación se ha vuelto mucho más fácil, aun así requiere invertir tiempo y recursos, todo ello con una cuidadosa planificación.

Documentar servicios Rest

- Web Application Description Language (WADL) (https://www.w3.org/Submission/wadl/)
  - Especificación de W3C, que la descripción XML legible por máquina de aplicaciones web basadas en HTTP (normalmente servicios web REST). Modela los recursos proporcionados por un servicio y las relaciones entre ellos. Está diseñado para simplificar la reutilización de servicios web basados en la arquitectura HTTP existente de la web. Es independiente de la plataforma y del lenguaje, tiene como objetivo promover la reutilización de aplicaciones más allá del uso básico en un navegador web.
- Spring REST Docs (https://spring.io/projects/spring-restdocs)
  - Documentación a través de los test (casos de uso), evita enterrar el código entre anotaciones.
- RAML (https://raml.org/)
  - RESTful API Modeling Language es una forma práctica de describir un API RESTful de una manera que sea muy legible tanto para humanos como para maquinas.
- Open API (anteriormente Swagger)
  - Especificación para describir, producir, consumir y visualizar servicios web RESTful. Es el más ampliamente difundido y cuenta con un ecosistema propio.
- JSON Schema (https://json-schema.org/)
  - JSON Schema es una especificación para definir, anotar y validar las estructuras de datos JSON.

Especificaciones mas utilizadas



Swagger

https://swagger.io/

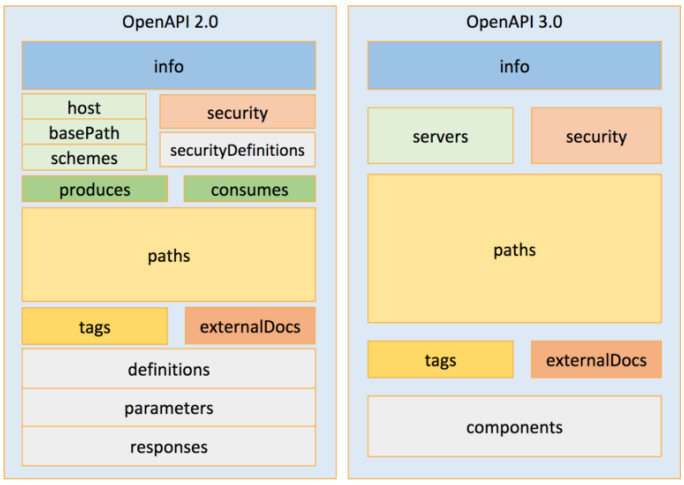
- Swagger (OpenAPI Specification) es una especificación abierta y su correspondiente implementación para probar y documentar servicios REST. Uno de los objetivos de Swagger es que podamos actualizar la documentación en el mismo instante en que realizamos los cambios en el servidor.
- Un documento Swagger es el equivalente de API REST de un documento WSDL para un servicio web basado en SOAP.
- El documento Swagger especifica la lista de recursos disponibles en la API REST y las operaciones a las que se puede llamar en estos recursos.
- El documento Swagger especifica también la lista de parámetros de una operación, que incluye el nombre y tipo de los parámetros, si los parámetros son necesarios u opcionales, e información sobre los valores aceptables para estos parámetros.

OpenAPI

https://www.openapis.org/

- OpenAPI es un estándar para definir contratos de API. Los cuales describen la interfaz de una serie de servicios que vamos a poder consumir por medio de una signatura. Conocido previamente como Swagger, ha sido adoptado por la Linux Foundation y obtuvo el apoyo de compañías como Google, Microsoft, IBM, Paypal, etc. para convertirse en un estándar para las APIs REST.
- Las definiciones de OpenAPI se pueden escribir en JSON o YAML. La versión actual de la especificación es la 3.0.3 y orientada a YAML y la versión previa la 2.0, que es idéntica a la especificación 2.0 de Swagger antes de ser renombrada a "Open API Specification".
- Actualmente nos encontramos en periodo de transición de la versión 2 a la 3, sin soporte en muchas herramientas.

Cambio de versión



Sintaxis

- Un documento de OpenAPI que se ajusta a la especificación de OpenAPI es en sí mismo un objeto JSON con propiedades, que puede representarse en formato JSON o YAML.
- YAML es un lenguaje de serialización de datos similar a XML pero que utiliza el sangrado para indicar el anidamiento, estableciendo la estructura jerárquica, y evitar la necesidad de tener que cerra los elementos.
- Para preservar la capacidad de ida y vuelta entre los formatos YAML y JSON, se RECOMIENDA la versión 1.2 de YAML junto con algunas restricciones adicionales:
  - Las etiquetas DEBEN limitarse a las permitidas por el conjunto de reglas del esquema JSON .
  - Las claves utilizadas en los mapas YAML DEBEN estar limitadas a una cadena escalar, según lo definido por el conjunto de reglas del esquema YAML Failsafe.
- Todos los nombres de propiedades o campos de la especificación distinguen entre mayúsculas y minúsculas. Esto incluye todas los propiedades que se utilizan como claves asociativas, excepto donde se indique explícitamente que las claves no distinguen entre mayúsculas y minúsculas .
- El esquema expone dos tipos de propiedades:
  - propiedades fijas: tienen el nombre establecido en el estándar
  - propiedades con patrón: sus nombres son de creación libre pero deben cumplir una expresión regular (patrón) definida en el estándar y deben ser únicos dentro del objeto contenedor.

Sintaxis

- El sangrado utiliza espacios en blanco, no se permite el uso de caracteres de tabulación.
- Los miembros de las listas van entre corchetes ([ ]) y separados por coma espacio (,), o uno por línea con un guion (-) inicial.
- Los vectores asociativos se representan usando los dos puntos seguidos por un espacio, "clave: valor", bien uno por línea o entre llaves ({ }) y separados por coma seguida de espacio (,).
- Un valor de un vector asociativo viene precedido por un signo de interrogación (?), lo que permite que se construyan claves complejas sin ambigüedad.
- Los valores sencillos (o escalares) por lo general aparecen sin entrecomillar, pero pueden incluirse entre comillas dobles ("), o apostrofes (').

Sintaxis

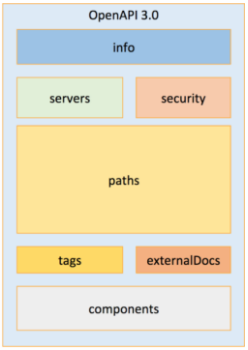
- Los comentarios vienen encabezados por la almohadilla (#) y continúan hasta el final de la línea.
- Es sensible a mayúsculas y minúsculas, todas las propiedades (palabras reservadas) de la especificación deben ir en minúsculas y terminar en dos puntos (:).
- Las propiedades requieren líneas independiente, su valor puede ir a continuación en la misma línea (precedido por un espacio) o en múltiples líneas (con sangrado)
- Las descripciones textuales pueden ser multilínea y admiten el dialecto CommonMark de Markdown para una representación de texto enriquecido. El HTML es compatible en la medida en que lo proporciona CommonMark (Bloques HTML en la Especificación 0.27 de CommonMark).
- $ref permite sustituir, reutilizar y enlazar una definición local con una externa.

CommonMark de Markdown

- Requiere doble y triple salto de línea para saltos de párrafos y cierre de bloques. Dos espacios al final de la línea lo convierte en salto de línea.
- Regla horizontal (separador): ---
- Énfasis: *cursiva* **negrita** ***cursiva y negrita***
- Enlaces: <http://www.example.com> [texto](http://www. example.com)
- Imágenes: ![Image](http://www.example.com/logo.png "icon")
- Citas: > Texto de la cita con sangría
- Bloques de códigos: `Encerrados entre tildes graves`
- Listas: Dos espacios en blanco por nivel de sangrado.
  + Listas desordenadas 1. Listas ordenadas
- Encabezado: dos líneas debajo del texto, añadir cualquier número de caracteres = para el nivel de título 1, <h1> … <h6> (el # es interpretado como comentario).

Estructura básica

- Un documento de OpenAPI puede estar compuesto por un solo documento o dividirse en múltiples partes conectadas a discreción del usuario. En el último caso, los campos $ref deben utilizarse en la especificación para hacer referencia a esas partes.
- Se recomienda que el documento raíz de OpenAPI se llame: openapi.json u openapi.yaml.
- La especificación de la API se puede dividir en 3 secciones principales:
  - Meta información
  - Elementos de ruta (puntos finales):
    - Parámetros de las solicitud
    - Cuerpo de las solicitud
    - Respuestas
  - Componentes reutilizables:
    - Esquemas (modelos de datos)
    - Parámetros
    - Respuestas
    - Otros componentes



Estructura básica

```yaml
openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description in ...
  version: 0.1.9
servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        '200':    # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

Estructura básica (cont)

```
components:
  schemas:
    User:
      properties:
        id:
          type: integer
        name:
          type: string
      # Both properties are required
      required:
        - id
        - name
  securitySchemes:
      BasicAuth:
        type: http
        scheme: basic
security:
  - BasicAuth: []
```


Prologo

- Cada definición de API debe incluir la versión de la especificación OpenAPI en la que se basa el documento en la propiedad openapi.
- La propiedad info contiene información de la API:
  - title es el nombre de API.
  - description es información extendida sobre la API.
  - version es una cadena arbitraria que especifica la versión de la API (no confundir con la revisión del archivo o la versión del openapi).
  - también admite otras palabras clave para información de contacto (nombre, url, email), licencia (nombre, url), términos de servicio (url) y otros detalles.
- La propiedad servers especifica el servidor API y la URL base. Se pueden definir uno o varios servidores (elementos precedidos por -).
- Con la propiedad externalDocs se puede referenciar la documentación externa adicional.

Rutas

- La sección paths define los puntos finales individuales (rutas) en la API y los métodos (operaciones) HTTP admitidos por estos puntos finales.
- Las ruta es relativa a la ruta del objeto Server.
- Los parámetros de la ruta se pueden usar para aislar un componente específico de los datos con los que el cliente está trabajando. Los parámetros de ruta son parte de la ruta y se expresan entre llaves (/users/{userId}), participan en la jerarquía de la URL y, por lo tanto, se apilan secuencialmente. Los parámetros de ruta deben describirse obligatoriamente en parameters (común para todas las operaciones) o a nivel de operación individual.
- No puede haber dos rutas iguales o ambiguas, que solo se diferencian por el parámetro de ruta.
- La definición de la ruta pude tener con un resumen (summary) y una descripción (description).
- Una ruta debe contar con un conjunto de operaciones, al menos una.
- Opcionalmente, servers permite dar una matriz alternativa de server que den servicio a todas las operaciones en esta ruta.

Rutas

```
'/users/{id}/roles':
  get:
    summary: Returns a list of users's roles.
    operationId: getDirecciones
    parameters:
      - in: path
        name: id
        description: User ID
        required: true
        schema:
          type: number
      - in: query
        name: size
        schema:
          type: string
          enum: [long, medium, short]
        required: true
      - in: query
        name: page
        schema:
          type: integer
          minimum: 0
          default: 0
```

```
responses:
  '200':
    description: List of roles
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Roles'
  '400':
    description: Bad request. User ID must be an
      integer and larger than 0.
  '401':
    description: Authorization information is missing
      or invalid.
  '404':
    description: A user with the specified ID was not
      found.
  '5XX':
    description: Unexpected error.
  default:
    description: Default error sample response
```

Operaciones

- Describe una única operación de API en una ruta y se identifica con el nombre del método HTTP: get, put, post, delete, options, head, patch, trace.
- Una definición de operación puede incluir un breve resumen de lo que hace (summary), una explicación detallada del comportamiento (description), una referencia a documentación externa adicional (externalDocs), un identificador único para su uso en herramientas y bibliotecas (operationId) y si está obsoleta y debería dejar de usarse (deprecated).
- Las operaciones pueden tener parámetros pasados a través de la ruta URL (/users/{userId}), cadena de consulta (/users?role=admin), encabezados (X-CustomHeader: Value) o cookies (Cookie: debug=0).
- Si la petición (POST, PUT, PATCH) envía un cuerpo en la solicitud (body), la propiedad requestBody permite describir el contenido del cuerpo y el tipo de medio.
- Para cada las respuestas de la operación, se pueden definir los posibles códigos de estado y el schema del cuerpo de respuesta. Los esquemas pueden definirse en línea o referenciarse mediante $ref. También se pueden proporcionar ejemplos para los diferentes tipos de respuestas.

Parámetros

- Un parámetro único se define mediante una combinación de nombre (name) y ubicación (in: "query", "header", "path" o "cookie") en la propiedad parameters.
- Opcionalmente puede ir acompañado por una breve descripción del parámetro (description), si es obligatorio (required), si permite valores vacíos (allowemptyvalue) y si está obsoleto y debería dejar de usarse (deprecated).
- Las reglas para la serialización del parámetro se especifican dos formas:
  - Para los escenarios más simples, con schema y style se puede describir la estructura y la sintaxis del parámetro.
  - Para escenarios más complejos, la propiedad content puede definir el tipo de medio y el esquema del parámetro.
- Un parámetro debe contener la propiedad schema o content, pero no ambas.
- Se puede proporcionar un example o examples pero debe seguir la estrategia de serialización prescrita para el parámetro.

Parámetros

```
paths:
  /users:
    get:
      description: Returns a list of users
      parameters:
        - name: rows
          in: query
          description: Limits the number of items on a page
          schema:
            type: integer
        - name: page
          in: query
          description: Specifies the page number of the users to be displayed
          schema:
            type: integer
```


Cuerpo de la solicitud

- En versiones anteriores, el cuerpo de la solicitud era un parámetro mas in: body.
- Actualmente se utiliza la propiedad requestBody con una breve descripción (description) y si es obligatorio para la solicitud (required), ambas opcionales.
- La descripción del contenido (content) es obligatoria y se estructura según los tipos de medios que actúan como identificadores. Para las solicitudes que coinciden con varias claves, solo se aplica la clave más específica (text/plain → text/* → */*).
- Por cada tipo de medio se puede definir el esquema del contenido de la solicitud (schema), uno (example) o varios (examples) ejemplos y la codificación (encoding).
- El requestBody sólo se admite en métodos HTTP donde la especificación HTTP 1.1 RFC7231 haya definido explícitamente semántica para cuerpos de solicitud.

Cuerpo de la solicitud

```
paths:
  /users:
    post:
      description: Lets a client post a new user
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required:
                - username
              properties:
                username:
                  type: string
                password:
                  type: string
                  format: password
                name:
                  type: string
```


Respuestas

- Es obligaría la propiedad responses con la lista de posibles respuestas que se devuelven al ejecutar esta operación.
- No se espera necesariamente que la documentación cubra todos los códigos de respuesta HTTP posibles porque es posible que ni se conozcan de antemano. Sin embargo, se espera que cubra la respuesta de la operación cuando tiene éxito y cualquier error previsto.
- La posibles respuestas se identifican con el código de respuesta HTTP. Con default se puede definir las respuesta por defecto para todos los códigos HTTP que no están cubiertos por la especificación individual.
- La respuesta cuenta con una breve descripción de la respuesta (description) y, opcionalmente, el contenido estructurado según los tipos de medios (content), los encabezados (headers) y los enlaces de operaciones que se pueden seguir desde la respuesta (links).

Respuestas

```
paths:
  /users:
    post:
      description: Lets a client post a new user
      requestBody: # …
      responses:
        '201':
          description: Successfully created a new user
        '400':
          description: Invalid request
          content:
            application/json:
              schema:
                type: object
                properties:
                  code:
                    type: integer
                  message:
                    type: string
```


Etiquetas

- Las etiquetas son metadatos adicionales que permiten organizar la documentación de la especificación de la API y controlar su presentación. Las etiquetas se pueden utilizar para la agrupación lógica de operaciones por recursos o cualquier otro calificador. El orden de las etiquetas se puede utilizar para reflejar un orden en las herramientas de análisis.
- Cada nombre de etiqueta en la lista debe ser único (name) y puede ir acompañado por una explicación detallada (description) y una referencia a documentación externa adicional (externalDocs).
- Las etiquetas se pueden declarar en la propiedad tags del documento:

```
tags:
  - name: security-resource
    description: Gestión de la seguridad
```

Etiquetas

- Las etiquetas se aplican en la propiedad tags de las operaciones:

```
paths:
  /users:
    get:
      tags:
        - security-resource
  /roles:
    get:
      tags:
        - security-resource
        - read-only-resource
```

- No es necesario declarar todas las etiquetas, se pueden usar directamente pero no se podrá dar información adicional y se mostraran ordenadas al azar o según la lógica de las herramientas.

Componentes

- La propiedad global components permite definir las estructuras de datos comunes utilizadas en la especificación de la API: Contiene un conjunto de objetos reutilizables para diferentes aspectos de la especificación.
- Todos los objetos definidos dentro del objeto de componentes no tendrán ningún efecto en la API a menos que se haga referencia explícitamente a ellos desde propiedades fuera del objeto de componentes.
- La sección components dispones de propiedades para schemas, responses, parameters, examples, requestBodies, headers, securitySchemes, links y callbacks.
- Se puede hacer referencia a ellos con $ref cuando sea necesario. $ref acepta referencias internas con # o externas con el nombre de un fichero. La referencia debe incluir la trayectoria para encontrar el elemento referenciado:

```
$ref: '#/components/schemas/Rol'
$ref: responses.yaml#/404Error
```

- El uso de referencias permite la reutilización de elementos ya definidos, facilitando la mantenibilidad y disminuyendo sensiblemente la longitud de la especificación, por lo que se deben utilizar extensivamente. Las referencias no interfieren con la presentación en el UI.

Esquemas de datos

- Los schemas definen los modelos de datos consumidos y devueltos por la API.
- Los tipos de datos OpenAPI se basan en un subconjunto extendido del JSON Schema Specification Wright Draft 00 (también conocido como Draft 5).
- Los tipos base son string, number, integer, boolean, array y object.
- Con la propiedad format se pueden especificar otros tipos especiales partiendo de los tipos base: long, float, double, byte, binary, date, dateTime, password.
- Los tipos array se definen como una colección de ítems y en dicha propiedad se define el tipo y la estructura de los elementos que lo componen. Los objetos son un conjunto de propiedades, cada una definida dentro de properties.
- Cada tipo y propiedad se identifica por un nombre que no debe estar repetido en su ámbito.
- Cada propiedad puede definir description, default, minimum, maximum, maxLength, minLength, pattern, required, readOnly, …
- Para una propiedad se pueden definir varios tipos (tipos mixtos o unión).
- Los tipos pueden hacer referencia a otros tipos.

Tipos de datos

type	format	Comentarios
boolean		Booleanos: true y false
integer	int32	Enteros con signo de 32 bits
integer	int64	Enteros con signo de 64 bits (también conocidos como largos)
number	float	Reales cortos
number	double	Reales largos
string		Cadenas de caracteres
string	password	Una pista a las IU para ocultar la entrada.
string	date	Según lo definido por full-date RFC3339 (2018-11-13)
string	date-time	Según lo definido por date-time- RFC3339 (2018-11-13T20:20:39+00:00)
string	byte	Binario codificados en base64
string	binary	Binario en cualquier secuencia de octetos
array		Colección de items
object		Colección de properties

Propiedades de los objetos de esquema

- type: integer, number, boolean, string, array, object
- format: long, float, double, byte, binary, date, dateTime, password
- title: Nombre a mostrar en el UI
- description: Descripción de su uso
- maximum: Valor máximo
- exclusiveMaximum: Valor menor que
- minimum: Valor mínimo
- exclusiveMinimum: Valor mayor que
- multipleOf: Valor múltiplo de
- maxLength: Longitud máxima
- minLength: Longitud mínima
- pattern: Expresión regular del patrón
- deprecated: Si está obsoleto y debería dejar de usarse
- nullable: Si acepta nulos
- default: Valor por defecto
- enum: Lista de valores con nombre
- example: Ejemplo de uso
- externalDocs: referencia a documentación externa adicional
- items: Definición de los elementos del array
- maxItems: Número máximo de elementos
- minItems: Número mínimo de elementos
- uniqueItems: Elementos únicos
- properties: Definición de las propiedades del objeto,
- maxProperties: Número máximo de propiedades
- minProperties: Número mínimo de propiedades
- readOnly: propiedad de solo lectura
- writeOnly: propiedad de solo escritura
- additionalProperties: permite referenciar propiedades adicionales
- required: Lista de propiedades obligatorias

Modelos de entrada y salida

```
components:
  schemas:
    Roles:
      type: array
      items:
        $ref: '#/components/schemas/Rol'
    Rol:
      type: object
      description: Roles de usuario
      properties:
        rolId:
          type: integer
          format: int32
          minimum: 0
          maximum: 255
        name:
          type: string
          maxLength: 20
        description:
          type: string
        last_updated:
          type: string
          format: dateTime
          readOnly: true
        level:
            type: string
            description: Nivel de permisos
            enum:
            - high
            - normal
            - low
            default: normal
      required:
      - rolId
      - name
```

Autenticación

- La propiedad securitySchemes de components y la propiedad security del documento se utilizan para describir y establecer los métodos de autenticación utilizados en la API.
- securitySchemes define los esquemas de seguridad que pueden utilizar las operaciones. Los esquemas admitidos son la autenticación HTTP, una clave API (ya sea como encabezado, parámetro de cookie o parámetro de consulta), los flujos comunes de OAuth2 (implícito, contraseña, credenciales de cliente y código de autorización) tal y como se define en RFC6749 y OpenID Connect Discovery. Cada esquema cuenta con un identificador, un tipo (type: "apiKey", "http", "oauth2", "openIdConnect") y opcionalmente puede ir acompañado por una breve descripción (description).
- Según el tipo seleccionado será obligatorio:
  - apiKey: ubicación (in: "query", "header" o "cookie") y su nombre (name) de parámetro, encabezado o cookie.
  - http: esquema de autorización HTTP que se utilizará en el encabezado Authorization (scheme): Basic, Bearer, Digest, OAuth, … y, si es Bearer, prefijo del token de portador (bearerFormat).
  - openIdConnect: URL de OpenId Connect para descubrir los valores de configuración de OAuth2 (openIdConnectUrl).
  - oauth2: objeto que contiene información de configuración para los tipos de flujo admitidos (flows).
- La propiedad security enumera los esquemas de seguridad que se pueden utilizar en la API.

Autenticación

```
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
    JWTAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
    ApiKeyAuth:
      type: apiKey
      name: x-api-key
      in: header
    ApiKeyQuery:
      type: apiKey
      name: api-key
      in: query
security:
  - ApiKeyAuth: []
  - ApiKeyQuery: []
```

Ejemplos

- Los ejemplos son fundamentales para la correcta comprensión de la documentación. La especificación permite proporcionar uno (example) o varios (examples) ejemplos asociados a las estructuras de datos.
- Por cada uno se puede dar un resumen del ejemplo (summary), una descripción larga (description), el juego de valores de las propiedades de la estructura (value) o una URL que apunta al ejemplo literal para ejemplos que no se pueden incluir fácilmente en documentos JSON o YAML (externalValue). value y externalValue son mutuamente excluyentes. Cuando son varios ejemplos deber estar identificados por un nombre único.

```
examples:
  first-page:
    summary: Primera página
    value: 0
  second-page:
    summary: Segunda página
    value: 1
```

- Los ejemplos pueden ser utilizados automáticamente por las herramientas de UI y de generación de pruebas.

Ecosistema Swagger

- Swagger Open Source Tools
  - Swagger Editor: Diseñar APIs en un potente editor de OpenAPI que visualiza la definición y proporciona comentarios de errores en tiempo real.
  - Swagger Codegen: Crear y habilitar el consumo de su API generando la fontanería del servidor y el cliente.
  - Swagger UI: Generar automáticamente la documentación desde la definición de OpenAPI para la interacción visual y un consumo más fácil.
- Swagger Pro Tools
  - SwaggerHub: La plataforma de diseño y documentación para equipos e individuos que trabajan con la especificación OpenAPI.
  - Swagger Inspector: La plataforma de pruebas y generación de documentación de las APIs
- https://openapi.tools/

springdoc-openapi

https://springdoc.org/

- La biblioteca Java springdoc-openapi ayuda a automatizar la generación de documentación de la API utilizando proyectos de spring boot. springdoc-openapi funciona examinando la aplicación en tiempo de ejecución para inferir la semántica de la API en función de las configuraciones de Spring, la estructura de clases y varias anotaciones.
- Genera automáticamente documentación en API de formato JSON/YAML y HTML. Esta documentación se puede completar con comentarios usando anotaciones swagger-api.
  - https://github.com/swagger-api/swagger-core/wiki/Swagger-2.X---Annotations
- Esta biblioteca admite:
  - OpenAPI 3
  - Spring Boot (1 y 2: v1, 3: v2), Actuator, Mvc, WebFlux, Security, Data Rest y Hateoas
  - JSR-303, específicamente para @NotNull, @Min, @Max y @Size.
  - Swagger-ui
  - OAuth 2
  - Imágenes nativas de GraalVM

Instalación (v3.0)

- Se debe añadir la dependencia Maven del starter de springdoc.

```
<properties><org.springdoc.version>2.1.0</org.springdoc.version></properties>
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>${org.springdoc.version}</version>
</dependency>
```

- Se pueden cambiar las configuraciones por defecto en el fichero application.properties

```
springdoc.swagger-ui.use-root-path=true
springdoc.show-actuator=true
springdoc.swagger-ui.path=/open-api
springdoc.packagesToScan=com.example.applications.resources
springdoc.pathsToMatch=/v1/**, /api/**, /auto/**
```

- Para acceder a la documentación:
  - http://localhost:8080/swagger-ui/index.html (versión HTML)
  - http://localhost:8080/v3/api-docs (versión JSON) o http://localhost:8080/v3/api-docs.yaml (versión YAML)

Soporte adicional (v3.0)

- Para habilitar la compatibilidad con javadoc, que mejora el soporte de etiquetas y comentarios javadoc, es necesario incluir:

```
<dependency>
    <groupId>com.github.therapi</groupId>
    <artifactId>therapi-runtime-javadoc</artifactId>
    <version>0.13.0</version>
</dependency>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <annotationProcessorPaths>
          <path>
            <groupId>com.github.therapi</groupId>
            <artifactId>therapi-runtime-javadoc-scribe</artifactId>
            <version>0.15.0</version>
          </path>
        </annotationProcessorPaths>
      </configuration>
    </plugin>
  </plugins>
</build>
```


Anotar el modelo

- **@Schema**:
  - documenta la entidad, con nombre alternativo al de la clase (name) y una descripción más larga (description).

```
@Schema(name = "Entidad Personas", description = "Información completa de la personas")
public class Persona {
```

  - documenta las propiedades, con una descripción (description), ocultarla (hidden) y el resto de las propiedades de los objetos de esquema como format, maximum, exclusiveMaximum, minimum, exclusiveMinimum, multipleOf, maxLength, minLength, pattern, deprecated, nullable, default, required.

```
@ Schema(description = "Identificador de la persona", mínimum=0, required = true)
private Long id;
```

  - Dispone de soporte para las anotaciones de validación de bean JSR-303, específicamente para @NotNull (@NotEmpty, @NotBlank), @Min, @Max, @Size y @Pattern, que documenta automáticamente.

Anotar el modelo

- **@ArraySchema**: se usa para definir un esquema de tipo "array". Las anotaciones ArraySchema y Schema no pueden coexistir. Los parámetros arraySchema permite documentar el array dentro de su esquema y schema documenta los elementos dentro del array. Están disponibles propiedades adicionales como uniqueItems, maxItems o maxItems.

```
@ArraySchema(arraySchema = @Schema(description = "Lista de
        correos electrónicos"), schema = @Schema(format = "email",
        maxLength = 100), maxItems = 3, uniqueItems = true)
private List<String> correos;
```


Anotar el servicio

- **@Tag**: documenta el servicio REST en si. Permite establecer el nombre y la descripción.

```
@RestController
@Tag(name = "Microservice Personas", description = "API que permite el mantenimiento de personas")
public class PersonasResource {
```

- **@Operation**: documenta cada método del servicio con el summary, una breve descripción, y description. También puede definir:
  - etiquetas:@Tag
  - documentos externos: @ExternalDocumentation
  - parámetros: @Parameter
  - cuerpo de solicitud: @RequestBody
  - respuestas: @ApiResponse
  - seguridad: @SecurityRequirement
  - servidores: @Server
  - extensiones: @Extension
  - ocultar: @Hidden

```
@GetMapping(path = "/{id}")
@Operation(summary="Buscar una persona",  description = "Devuelve una persona por su identificador")
public Persona getOne(@Parameter(description = "Identificador de la persona", required = true) @PathVariable int id) {
```

Anotar el servicio

- @**Parameter**: documenta los parámetros de cada método del servicio.

```
public Persona getOne(@Parameter(description = "Identificador de la persona", required = true) @PathVariable int id) {
```

- @**ParameterObject**: puede extraer cada campo del objeto parámetro como un parámetro de solicitud independiente.

```
public Page<Persona> getAll(@ParameterObject Pageable page) {
```

- @**RequestBody**: documenta el cuerpo de solicitud de la operación y permite definir propiedades adicionales a las disponibles en @Parameter

```
@PostMapping
public ResponseEntity<Object> create(@Valid @RequestBody(description = "Datos de la
        persona", required = true, content = @Content(mediaType = "application/json",
        schema = @Schema(implementation = Persona.class))) Persona item) throws
        BadRequestException, DuplicateKeyException, InvalidDataException {
```


Anotar el servicio

- @**ApiResponse**: documenta las posibles respuestas del método, con un mensaje explicativo. El uso de @ResponseStatus en métodos en una clase @RestControllerAdvice generará automáticamente la documentación para los códigos de respuesta. @ApiResponses es un contenedor de respuestas.

```
@ApiResponse(responseCode = "200", description = "Persona encontrada")
@ApiResponse(responseCode = "404", description = "Persona no encontrada")
public Persona getOne(@PathVariable int id) throws NotFoundException {
```

- @**Hidden**: Marca un recurso u operación como oculto en la documentación.

```
@Hidden
@RestController
public class PersonasResource {
@Hidden
public Persona getOne(@PathVariable int id) throws NotFoundException {
```

Anotar el servicio

```
@PutMapping("/{id}")
@ResponseStatus(HttpStatus.ACCEPTED)
@Operation(summary = "Modificar una persona",
    description = "Sustituye una persona con los nuevos datos, los identificadores deben coincidir.",
    tags = {"Microservice Personas", "Modificaciones"},
    parameters = {
        @Parameter(in = ParameterIn.PATH, name = "id", required = true, description = "Identificador de la persona")
    },
    requestBody = @io.swagger.v3.oas.annotations.parameters.RequestBody(description = "Datos de la persona", required = true, content
        = @Content(mediaType = "application/json", schema = @Schema(implementation = Persona.class))),
    responses = {
        @ApiResponse(responseCode = "202", description = "Persona modificada"),
        @ApiResponse(responseCode = "400", description = "Datos invalidos", content = @Content(mediaType = "application/json",
            schema = @Schema(implementation = ErrorMessage.class))),
        @ApiResponse(responseCode = "404", description = "Persona no encontrada", content = @Content(mediaType =
            "application/json", schema = @Schema(implementation = ErrorMessage.class)))
    }
)
public void update(@PathVariable int id, @Valid @RequestBody ActorShort item) throws BadRequestException, NotFoundException,
InvalidDataException {
```


Configuración (OpenApi)

```
@OpenAPIDefinition(
    info = @Info(title = "Microservicio: Demos",  version = "1.0",
        description = "**Demos** de Microservicios.",
        license = @License(name = "Apache 2.0", url = "https://www.apache.org/licenses/LICENSE-2.0.html"),
        contact = @Contact(name = "Javier Martín", url = "https://github.com/jmagit", email = "support@example.com")
    ),
    externalDocs = @ExternalDocumentation(description = "Documentación del proyecto", url = "https://github.com/jmagit/curso")
)
public class PrincipalApplication implements CommandLineRunner {

@Configuration
public class OpenApiConfiguration {
    @Bean
    public OpenAPI springShopOpenAPI() {
        return new OpenAPI()
            .info(new Info().title("Microservicio: Demos")
                .description("**Demos** de Microservicios.").version("1.0")
                .license(new License().name("Apache 2.0").url("https://www.apache.org/licenses/LICENSE-2.0.html"))
                .contact(new Contact().name("Javier Martín").url("https://github.com/jmagit").email("support@example.com"))
            ).externalDocs(new ExternalDocumentation().description("Documentación del proyecto").url("https://github.com/jmagit/curso"));
    }
}
```

Configuración (swagger-ui)

- Para ordenar los servicios y sus operaciones, en el fichero application.properties

```
springdoc.swagger-ui.tagsSorter=alpha
springdoc.swagger-ui.operationsSorter=alpha
springdoc.swagger-ui.docExpansion=none
```

- Para ordenar los modelos del esquema:

```
@Bean
public OpenApiCustomiser sortSchemasAlphabetically() {
   return openApi -> {
      var schemas = openApi.getComponents().getSchemas();
      openApi.getComponents().setSchemas(new TreeMap<>(schemas));
   };
}
```


Archivo de propiedades

- Hay que crear en resources un archivo de propiedades, por ejemplo, openapi.properties
- Insertar los mensajes deseados como pares clave-valor donde la clave se usará como marcador de posición:

```
person.id.value = Identificador único de la persona
```

- En lugar del texto en la anotación, se inserta un marcador de posición:

```
@Parameter(description = "${person.id.value}", required = true)
```

- Hay que registrar el archivo de propiedades de la configuración a nivel de clase:

```
@PropertySource ("classpath: openapi.properties")
```

Seguridad

- @SecurityScheme: Permite definir los esquemas de seguridad, una por definición. La definición es global y complementan a la anotación @OpenAPIDefinition.

```
@SecurityScheme(name = "bearerAuth", type = SecuritySchemeType.HTTP, scheme =
"bearer", bearerFormat = "JWT")

return new OpenAPI()
        .components(new Components()
        .addSecuritySchemes("bearerAuth",
        new SecurityScheme().type(SecurityScheme.Type.HTTP)
            .scheme("bearer").bearerFormat("JWT")));
```

- @SecurityRequirement: A nivel de operación, indica que requiere autorización y el esquema de autenticación:

```
@SecurityRequirement(name = "bearerAuth")
```


Que debe incluir

- Una explicación clara de lo que hace el método / recurso.
- Una lista de los parámetros utilizados en este recurso / método,
- Posibles respuestas, que comparten información importante con los desarrolladores, incluidas advertencias y errores
- Descripción de los tipos, formatos especial, reglas y restricciones.
- Una invocación y una respuesta de ejemplo, incluido los cuerpos con los media-type correspondientes.
- Ejemplos de código para varios lenguajes, incluido todo el código necesario (por ejemplo, Curl con PHP, así como ejemplos para Java, .Net, Ruby, etc.)
- Ejemplos de SDK (si se proporcionan SDK) que muestren cómo acceder al recurso / método utilizando el SDK para los lenguajes en que se suministra.
- Experiencias interactivas para probar las llamadas API.
- Preguntas frecuentes / escenarios con ejemplos de código
- Enlaces a recursos adicionales (otros ejemplos, blogs, etc.)
- Una sección de comentarios donde los usuarios pueden compartir / discutir el código.

CONSULTAS ENTRE SERVICIOS

Eureka

- Eureka permite registrar y localizar microservicios existentes, informar de su localización, su estado y datos relevantes de cada uno de ellos. Además, permite el balanceo de carga y tolerancia a fallos.
- Eureka dispone de un módulo servidor que permite crear un servidor de registro de servicios y un módulo cliente que permite el auto registro y descubrimiento de microservicios.
- Cuando un microservicio arranca, se comunicará con el servidor Eureka para notificarle que está disponible para ser consumido. El servidor Eureka mantendrá la información de todos los microservicios registrados y su estado. Cada microservicio le notificará, cada 30 segundos, su estado mediante heartbeats.
- Si pasados tres periodos heartbeats no recibe ninguna notificación del microservicio, lo eliminará de su registro. Si después de sacarlo del registro recibe tres notificaciones, entenderá que ese microservicio vuelve a estar disponible.
- Cada cliente o microservicio puede recuperar el registro de otros microservicios registrados y quedará cacheado en dicho cliente.
- Para los servicios que no están basados en Java, hay disponibles clientes Eureka para otros lenguaje y el servidor Eureka expone todas sus operaciones a través de un API REST que permiten la creación de clientes personalizados.

Eureka Server

- Añadir al proyecto:
  - Spring Boot + Cloud Discovery: Eureka Server + Core: Cloud Bootstrap
- Anotar aplicación:

```
@EnableEurekaServer
@SpringBootApplication
public class MsEurekaServiceDiscoveryApplication {
```

- Configurar:

```
#Servidor Eureka Discovery Server
eureka.instance.hostname: localhost
eureka.client.registerWithEureka: false
eureka.client.fetchRegistry: false
server.port: ${PORT:8761}
```

- Arrancar servidor
- Acceder al dashboard de Eureka: http://localhost:8761/

Auto registro de servicios

- Añadir al proyecto:
  - Eureka Discovery, Cloud Bootstrap
- Anotar aplicación (ahora es opcional):

```
~~@EnableEurekaClient~~@EnableDiscoveryClient
@SpringBootApplication
public class MsEurekaServiceDiscoveryApplication {
```

- Configurar:

```
# Service registers under this name
spring.application.name=educado-service
# Discovery Server Access
eureka.client.serviceUrl.defaultZone=${DISCOVERY_URL:http://localhost:8761}/eureka/
```

- Arrancar microservicio y refrescar dashboard de Eureka:
  - http://localhost:8761/
- Se puede usar EurekaClient o DiscoveryClient para descubrir las instancias de un servicio:

```
@Autowired
private EurekaClient discoveryClient;

InstanceInfo instance = discoveryClient.getNextServerFromEureka(nombre, false);
return instance.getHomePageUrl();
```

HttpClient nativo

- El HttpClient nativo se introdujo como un módulo incubador en Java 9 y de forma definitiva en Java 11 como parte de JEP 321 .
- HTTPClient reemplaza la clase heredada HttpUrlConnection presente en el JDK desde las primeras versiones de Java.
- Algunas de sus características incluyen:
  - Soporte para HTTP/1.1, HTTP/2 y Web Socket.
  - Soporte para modelos de programación sincrónicos y asincrónicos.
  - Manejo de cuerpos de solicitud y respuesta como flujos reactivos.
  - Soporte para cookies.

HttpClient nativo

```
HttpClient client = HttpClient.newBuilder()
                        .version(Version.HTTP_2)
                        .followRedirects(Redirect.NORMAL)
                        .build();

HttpRequest request = HttpRequest.newBuilder()
                        .uri(new URI("https://picsum.photos/v2/list?limit=10"))
                        .GET()
                        .header("Accept", "application/json")
                        .timeout(Duration.ofSeconds(10))
                        .build();

client.sendAsync(request, BodyHandlers.ofString())
                        .thenApply(HttpResponse::body)
                        .thenAccept(System.out::println)
                        .join();
```

RestTemplate

- La RestTemplate proporciona un API de nivel superior sobre las bibliotecas de cliente HTTP y facilita la invocación de los endspoint REST en una sola línea. Para incorporarlo en Maven:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-web</artifactId>
</dependency>
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
</dependency>
```

- Para poder inyectar la dependencia:

```
@Bean public RestTemplate restTemplate(RestTemplateBuilder builder) {
      return builder.build();
}
@Autowired RestTemplate srvRest;
```


RestTemplate

Grupo de métodos	Descripción
getForObject	Recupera una representación a través de GET.
getForEntity	Recupera un ResponseEntity(es decir, estado, encabezados y cuerpo) utilizando GET.
headForHeaders	Recupera todos los encabezados de un recurso utilizando HEAD.
postForLocation	Crea un nuevo recurso utilizando POST y devuelve el encabezado Location de la respuesta.
postForObject	Crea un nuevo recurso utilizando POST y devuelve la representación del objeto de la respuesta.
postForEntity	Crea un nuevo recurso utilizando POST y devuelve la representación de la respuesta.
put	Crea o actualiza un recurso utilizando PUT.

RestTemplate

Grupo de métodos	Descripción
patchForObject	Actualiza un recurso utilizando PATCH y devuelve la representación de la respuesta.
delete	Elimina los recursos en el URI especificado utilizando DELETE.
optionsForAllow	Recupera los métodos HTTP permitidos para un recurso utilizando ALLOW.
exchange	Versión más generalizada (y menos crítica) de los métodos anteriores que proporciona flexibilidad adicional cuando es necesario. Acepta a RequestEntity (incluido el método HTTP, URL, encabezados y cuerpo como entrada) y devuelve un ResponseEntity.
execute	La forma más generalizada de realizar una solicitud, con control total sobre la preparación de la solicitud y la extracción de respuesta a través de interfaces de devolución de llamada.

RestTemplate

- Para recuperar uno:

```
PersonaDTO rslt = srvRest.getForObject(
      "http://localhost:8080/api/personas/{id}", PersonaDTO.class, 1);
```

- Para recuperar todos (si no se dispone de una implementación de List<PersonaDTO>):

```
ResponseEntity<List<PersonaDTO>> response =
    srvRest.exchange("http://localhost:8080/api/personas",
    HttpMethod.GET,
    HttpEntity.EMPTY, new
    ParameterizedTypeReference<List<PersonaDTO>>() {
    });
List<PersonaDTO> rslt = response.getBody();
```

RestTemplate

- Para crear o modificar un recurso:

```
ResponseEntity<PersonaDTO> httpRslt = srvRest.postForEntity(
    "http://localhost:8080/api/personas", new PersonaDTO("pepito",
    "grillo")), PersonaDTO.class);
```

- Para crear o modificar un recurso con identificador:

```
srvRest.put("http://localhost:8080/api/personas/{id}", new
    PersonaDTO(new Persona("Pepito", "Grillo"))), 111);
```

- Para borrar un recurso con identificador:

```
srvRest.delete("http://localhost:8080/api/personas/{id}", 111);
```


RestTemplate

- De forma predeterminada, RestTemplate lanzará una de estas excepciones en caso de un error de HTTP:
  - HttpClientErrorException: en estados HTTP 4xx
  - HttpServerErrorException: en estados HTTP 5xx
  - UnknownHttpStatusCodeException: en caso de un estado HTTP desconocido.
- Para vigilar las excepciones:

```
} catch (HttpClientErrorException e) {
    switch (e.getStatusCode()) {
    case BAD_REQUEST:
    case NOT_FOUND:
            // …
            break;
```

LinkDiscoverers

- Cuando se trabaja con representaciones habilitadas para hipermedia, una tarea común es encontrar un enlace con un tipo de relación particular en ellas.
- Spring HATEOAS proporciona implementaciones basadas en JSONPath de la interfaz LinkDiscoverer.

```
<dependency>
   <groupId>com.jayway.jsonpath</groupId>
   <artifactId>json-path</artifactId>
</dependency>
```

- Para acceder a un enlace:

```
String resp = srvRest.getForObject("http://localhost:8080/personas/1", String.class);
LinkDiscoverer discoverer = new HalLinkDiscoverer();
Link link = discoverer.findLinkWithRel("direcciones", resp);
if(link != null)
      direccionesURL = link.getHref();
```


Feign

- Feign es un cliente declarativo de servicios web.
- Facilita la escritura de clientes de servicios web (proxies) mediante la creación de una interfaz anotada.
- Tiene soporte de anotación conectable que incluye anotaciones Feign y JAX-RS.
- Feign también soporta codificadores y decodificadores enchufables.
- Spring Cloud agrega soporte para las anotaciones de Spring MVC y para usar el mismo HttpMessageConverters usado de forma predeterminada en Spring Web.
- Spring Cloud integra Ribbon y Eureka para proporcionar un cliente http con equilibrio de carga cuando se usa Feign.
- Dispone de un amplio juego de configuraciones.

Feign

- Dependencia: Spring Cloud Routing > OpenFeign
- Anotar la clase principal con:
  @EnableFeignClients("com.example.proxies")
- Crear un interfaz por servicio:

```
@FeignClient(name = "personas", url = "http://localhost:8002")
// @FeignClient(name = "personas-service") // Eureka
public interface PersonaProxy {
      @GetMapping("/personas")
      List<PersonaDTO> getAll();
      @GetMapping("/personas/{id}")
       PersonaDTO getOne(@PathVariable int id);
       @PutMapping(value = "/personas/{id}", consumes = "application/json")
       PersonaDTO update(@PathVariable("id") id, PersonaDTO persona);
```

- Inyectar la dependencia:

```
@Autowired
PersonaProxy srvRest;
```


@HttpExchange (v.6)

- En Spring, una interfaz de servicio HTTP es una interfaz Java con métodos @HttpExchange. El método anotado se trata como un punto final HTTP, y los detalles se definen estáticamente a través de atributos de anotación, así como a través de los tipos de argumentos del método de entrada.
- @HttpExchange es la anotación genérica para especificar un punto final HTTP. Cuando se utiliza a nivel de interfaz, se aplica a todos los métodos. Está disponible como @GetExchange, @PostExchange, @PutExchange, @PatchExchange, @DeleteExchange.
- @HttpExchange permite definir a nivel de interfaz url (ruta base), method, accept y contentType.
- Los métodos de intercambio admiten los siguientes parámetros en la firma del método:
  - @PathVariable: sustituye un marcador de posición por un valor en la URL de la solicitud.
  - @RequestBody: proporciona el cuerpo de la solicitud.
  - @RequestParam: añade los parámetros de la petición. Cuando content-type está configurado como application/x-www-form-urlencoded, los parámetros de la petición se codifican en el cuerpo de la petición. En caso contrario, se añaden como parámetros de consulta de la URL.
  - @RequestHeader: añade los nombres y valores de las cabeceras de la petición.
  - @RequestPart: se puede utilizar para añadir una parte de la petición (campo de formulario, recurso o HttpEntity).
  - @CookieValue: añade cookies a la petición.

@HttpExchange (v.6)

```
@HttpExchange(url = "/actores/v1", accept = "application/json", contentType = "application/json")
public interface ActoresProxy {
    public record ActorShort(int id, String nombre) {}
    public record ActorEdit(int id, String nombre, String apellidos) {}

    @GetExchange
    List<ActorShort> getAll();
    @GetExchange("/{id}")
    ActorEdit getOne(@PathVariable int id);
    @PostExchange
    ResponseEntity<ActorEdit> add(@RequestBody ActorEdit item);
    @PutExchange("/{id}")
    void change(@PathVariable int id, @RequestBody ActorEdit item);
    @DeleteExchange("/{id}")
    void delete(@PathVariable int id);
}
```


@HttpExchange (v.6)

- Un método de intercambio HTTP puede devolver:
  - Clases (modo bloqueante) o clases reactivas (Mono/Flux).
  - ResponseEntity<T> que contiene el estado, los encabezados y el cuerpo des serializado
  - void si el método se trata como sólo de ejecución
- HttpServiceProxyFactory es una fábrica para crear un proxy de cliente a partir de una interfaz de servicio HTTP.

```
@Bean
WebClient webClient() {
    return WebClient.builder().baseUrl("http://localhost:8010/").build();
}
@Bean
ActoresProxy actoresProxy(WebClient webClient) {
    HttpServiceProxyFactory httpServiceProxyFactory = HttpServiceProxyFactory
                .builder(WebClientAdapter.forClient(webClient)).build();
    return httpServiceProxyFactory.createClient(ActoresProxy.class);
}
```

Spring Cloud LoadBalancer

- Un balanceador o equilibrador de carga fundamentalmente es un dispositivo de hardware o software que se interpone al frente de un conjunto de servidores que atienden una aplicación y, tal como su nombre lo indica, asigna o reparte las solicitudes que llegan de los clientes a los servidores usando algún algoritmo (desde un simple round-robin hasta algoritmos más sofisticados).
- Spring Cloud proporciona su propia abstracción e implementación del equilibrador de carga del lado del cliente. Para el mecanismo de equilibrio de carga, ReactiveLoadBalancer, se ha agregado una interfaz y se le han proporcionado implementaciones basadas en Round-Robin y Random.
- El balanceo de carga se basada en el descubrimiento de servicios que utiliza el cliente de descubrimiento disponible en la ruta de clases, como Spring Cloud Netflix Eureka, Spring Cloud Consul Discovery o Spring Cloud Zookeeper Discovery.
- Spring Cloud LoadBalancer puede integrarse con:
  - Spring RestTemplate como cliente de equilibrador de carga
  - Spring WebClient como cliente de equilibrador de carga
  - Spring OpenFeign como cliente de equilibrador de carga
  - Spring WebFlux WebClient con ReactorLoadBalancerExchangeFilterFunction
- Dependencia: Spring Cloud Routing > Cloud LoadBalancer

Spring Cloud LoadBalancer

- Spring Cloud LoadBalance permite:
  - Cambiar entre los algoritmos de equilibrio de carga
  - Almacenamiento en caché
  - Equilibrio de carga basado en zonas
  - Comprobación del estado de las instancias (HealthCheck)
  - Establecer preferencia de Misma instancia, Sesión fija basada en solicitudes, basado en sugerencias.
  - Transformar la solicitud HTTP en le proceso de equilibrio de carga antes de enviarla.

Clientes Load Balancer

- La anotación @LoadBalanced configura los diferentes clientes para que utilicen el balanceo de carga:

```
@LoadBalanced
@Bean RestTemplate restTemplate(RestTemplateBuilder builder) { return builder.build(); }
@LoadBalanced
@Bean public WebClient.Builder webClientBuilder() { return WebClient.builder(); }
```

- Las peticiones sustituyen en la URL el nombre del dominio por el nombre registrado en el servidor de descubrimiento para que utilicen el balanceo de carga:

```
return restTemplate.getForObject("lb://personas-service/resource", String.class);
return webClientBuilder.build().get().uri("lb://personas-service/resource")
                .retrieve().bodyToMono(String.class);

@FeignClient(name = "personas-service")
public interface PersonaProxy {
```


Spring Cloud Gateway

- Spring Cloud Gateway se puede definir como un proxy inverso o edge service (fachada) que va a permitir tanto enrutar y filtrar las peticiones de manera dinámica, así como monitorizar, balancear y securizar las mismas.
- Este componente actúa como un punto de entrada a los servicios públicos, es decir, se encarga de solicitar una instancia de un microservicio concreto a Eureka y de su enrutamiento hacia el servicio que se desea consumir.
- Las peticiones pasarán de manera individual por cada uno de los filtros que componen la configuración de Spring Cloud Gateway. Estos filtros harán que la petición sea rechazada por determinados motivos de seguridad en función de sus características, sea dirigida a la instancia del servicio apropiada, que sea etiquetada y registrada con la intención de ser monitorizada.

Spring Cloud Gateway

- Añadir proyecto:
  - Spring Cloud Routing Gateway, Eureka Discovery Client
- Anotar la aplicación:

  @EnableDiscoveryClient
  @EnableEurekaClient
- Configurar:

```
eureka:
  client:
    fetchRegistry: true
    registerWithEureka: false
    serviceUrl:
      defaultZone: ${DISCOVERY_URL:http://localhost:8761}/eureka/
  instance:
    appname: apigateway-server
server:
  port: ${PORT:8080}
```


Spring Cloud Gateway

- Ruta: el bloque de construcción básico de la puerta de enlace. Está definido por un ID, un URI de destino, una colección de predicados y una colección de filtros. Una ruta coincide si el predicado agregado es verdadero.
- Predicado: Patrón de coincidencia con las solicitud (Spring FrameworkServerWebExchange), es un predicado de función de Java 8. El tipo de entrada, permite hacer coincidir cualquier cosa desde la solicitud HTTP, como encabezados o parámetros. Spring Cloud Gateway incluye múltiples factorías de predicados de ruta integradas.
- Filtro: Se pueden modificar las solicitudes y respuestas antes o después de enviar la solicitud descendente. Spring Cloud Gateway incluye múltiples factorías de filtros integradas.
- Los clientes realizan solicitudes a Spring Cloud Gateway. Si la asignación del controlador de la puerta de enlace determina que una solicitud coincide con una ruta, se envía al controlador web de la puerta de enlace. Este controlador ejecuta la solicitud a través de una cadena de filtros que es específica de la solicitud. Los filtros pueden ejecutar la lógica antes y después de que se envíe la solicitud del proxy.
- Hay dos formas de configurar predicados y filtros: imperativamente por código o declarativamente en application.properties.

Spring Cloud Gateway

```
spring:
  cloud:
    gateway:
      routes:
      - id: serv-catalogo
          #Se utiliza el esquema lb:// cuando se va a acceder a través de Eureka
        uri: lb://catalogo-service
        predicates:
          - Path=/catalogo/**
        filters:
          - RewritePath=/catalogo/*, /
      - id: serv-clientes
        uri: lb://clientes-service
        predicates:
          - Path=/clientes/**
        filters:
          - RewritePath=/clientes/*, /
      - id: serv-search
        uri: https://www.google.com/
        predicates:
          - Path=/search/**
        filters:
          - RewritePath=/search/*, /
```


SEGURIDAD

Spring Cloud Config

- Para conectarse con recursos protegidos y otros servicios, las aplicaciones típicamente necesitan usar cadenas de conexión, contraseñas u otras credenciales que contengan información confidencial.
- Estas partes de información sensible se llaman secretos.
- Es una buena práctica no incluir secretos en el código fuente y, sobre todo, no almacenar secretos en el sistema de control de versiones.
- En su lugar, debería utilizar el modelo de configuración para leer los secretos desde ubicaciones más seguras.
- Se deben separar los secretos para acceder a los recursos de desarrollo y pre-producción (staging) de los que se usan para acceder a los recursos de producción, porque diferentes individuos necesitarán acceder a esos conjuntos diferentes de secretos. Para almacenar secretos usados durante el desarrollo, los enfoques comunes son almacenar secretos en variables de entorno. Para un almacenamiento más seguro en entornos de producción, los microservicios pueden almacenar secretos en una Key Vault.
- Los servidores de configuración de Spring Cloud soportan los siguientes orígenes (backends): GIT, Vault y JDBC
- Los recursos con los nombres de archivos application*(application.properties, application.yml, application-*.properties, etc.) son compartidos entre todas las aplicaciones cliente.

Spring Cloud Config: Servidor

- Añadir proyecto:
  - Spring Cloud Config > Config Server
- Anotar la aplicación:
  @EnableConfigServer
- Crear repositorio (local):
  - Crear directorio
  - Desde la consola de comandos posicionada en el directorio: git init
  - Crear un fichero que se llame como el spring.application.name del cliente que va a solicitar los datos y extensión **.properties**, con la configuración. Se pueden incluir perfiles añadiéndoselos al nombre: -production.properties
  - Añadir el fichero al repositorio: git add mi-service.properties ó git add .
  - Realizar un commit del fichero: git commit -m “Comentario a la versión”
- Configurar:

```
server.port= ${PORT:8888}
spring.cloud.config.server.git.uri=file://C:/mi/configuration-repository
#spring.cloud.config.server.git.uri=https://github.com/jmagit/mi-config.git
```

Spring Cloud Config: Cliente

- Añadir proyecto:
  - Spring Cloud Config > Config Client
- Para poder refrescar la configuración en caliente, se añadirá el starter Actuator
- Configurar:

```
server.port= ${PORT:8001}
spring.application.name=mi-service
spring.config.import=optional:configserver:${CONFIG_URI:http://localhost:8888}
#spring.profiles.active=production
management.endpoints.web.exposure.include=refresh
```

- Eclipse: Run Configurations → Arguments → Program Arguments: --spring.profiles.active=production
- De forma predeterminada, los valores de configuración solo se leen en el inicio del cliente. Puede forzar a un bean a que actualice su configuración (vuelva a leer), para ello debe anotarse con @RefreshScope (clase o método).
- Para refrescar la configuración en caliente después de realizar un commit al repositorio hay que hacer un POST a:
  - http://localhost:8001/actuator/refresh

Spring Cloud Config: Cliente

- Para recuperar un valor de la configuración:

```
@Value("${mi.valor}")
String miValor;
```

- Para recuperar y crear un componente:

```
// En el fichero .properties
// rango.min=1
// rango.max=10
@Data
@Component
@ConfigurationProperties("rango")
public class Rango {
   private int min;
   private int max;
}
@Autowired
private Rango rango;
```

CORS

- La ejecución de aplicaciones JavaScript puede suponer un riesgo para el usuario que permite su ejecución.
- Por este motivo, los navegadores restringen la ejecución de todo código JavaScript a un entorno de ejecución limitado.
- Las aplicaciones JavaScript no pueden establecer conexiones de red con dominios distintos al dominio en el que se aloja la aplicación JavaScript.
- Los navegadores emplean un método estricto para diferenciar entre dos dominios ya que no permiten ni subdominios ni otros protocolos ni otros puertos.
- Si el código JavaScript se descarga desde la siguiente URL: http://www.ejemplo.com
- Las funciones y métodos incluidos en ese código no pueden acceder a:
  - https://www.ejemplo.com/scripts/codigo2.js
  - http://www.ejemplo.com:8080/scripts/codigo2.js
  - http://scripts.ejemplo.com/codigo2.js
  - http://192.168.0.1/scripts/codigo2.js

CORS

- Un recurso hace una solicitud HTTP de origen cruzado cuando solicita otro recurso de un dominio distinto al que pertenece.
- XMLHttpRequest sigue la política de mismo-origen, por lo que, una aplicación usando XHR solo puede hacer solicitudes HTTP a su propio dominio. Para mejorar las aplicaciones web, los desarrolladores pidieron que se permitieran a XHR realizar solicitudes de dominio cruzado.
- El Grupo de Trabajo de Aplicaciones Web del W3C recomienda el nuevo mecanismo de Intercambio de Recursos de Origen Cruzado (CORS, Cross-origin resource sharing: https://www.w3.org/TR/cors). Los servidores deben indicar al navegador mediante cabeceras si aceptan peticiones cruzadas y con que características:
  - "Access-Control-Allow-Origin", "*"
  - "Access-Control-Allow-Headers", "Origin, Content-Type, Accept, Authorization, X-XSRF-TOKEN"
  - "Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS"
  - "Access-Control-Allow-Credentials", "true"
- Soporte: Chrome 3+ Firefox 3.5+ Opera 12+ Safari 4+ Internet Explorer 8+

CORS

- Para configurar CORS en la interfaz del repositorio

```
@CrossOrigin(origins = "http://myDomain.com", maxAge = 3600, methods={RequestMethod.GET,
RequestMethod.POST })
public interface PersonaRepository extends JpaRepository<Persona, Integer> {
```

- Para configurar CORS globalmente

```
@Configuration @EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
   @Override
   public void addCorsMappings(CorsRegistry registry) {
      registry.addMapping("/api/**")
         .allowedOrigins("*")
         .allowedMethods("GET", "POST", "PUT", "DELETE")
         .allowedHeaders("origin", "content-type", "accept", "authorization")
         .allowCredentials(true).maxAge(3600);
   }
}
```


Autenticación y Autorización

- La autenticación es un proceso en el que un usuario o una aplicación se identifica proporcionando credenciales que después se comparan con las almacenadas en un sistema operativo, base de datos, aplicación o recurso para validar que es realmente quién asegura ser. La autenticación puede crear una o varias identidades para el usuario o aplicación autenticado.
- La autorización se refiere al proceso que determina lo que una identidad puede hacer en función a los permisos otorgados a una identidad concreta sobre un recurso concreto. La autorización es ortogonal e independiente de la autenticación. Sin embargo, la autorización requiere un mecanismo de autenticación. La autorización puede utilizar un modelo basado en roles, sencillo y declarativo, o un modelo avanzado basado en directivas y evidencias.

Spring Security

- Spring Security es un framework de apoyo al marco de trabajo Spring, que dota al mismo de una serie servicios de seguridad aplicables para sistemas basados en la arquitectura JEE, enfocado particularmente sobre proyectos construidos usando Spring Framework. De esta dependencia, se minimiza la curva de aprendizaje si ya se conoce Spring.
- Los procesos de seguridad están destinados principalmente, a comprobar la identidad del usuario mediante la autenticación y los permisos asociados al mismo mediante la autorización. La autorización, basada en roles, es dependiente de la autenticación ya que se produce posteriormente a su proceso.
- Por regla general muchos de estos modelos de autenticación son proporcionados por terceros o son desarrollados por estándares importantes como el IETF. Adicionalmente, Spring Security proporciona su propio conjunto de características de autenticación:
  - In-Memory, JDBC, LDAP, OAuth 2.0, Kerberos, SAML …
- El proceso de autorización se puede establecer a nivel de recurso individual o mediante configuración que cubra múltiples recursos.

Spring Boot

- Si Spring Security está en la ruta de clase, las aplicaciones web están protegidas de forma predeterminada. Spring Boot se basa en la estrategia de negociación de contenido de Spring Security para determinar si se debe usar httpBasic o formLogin.
- Para agregar seguridad a nivel de método a una aplicación web, también puede agregar @EnableMethodSecurity en la configuración que desee.
- El valor predeterminado del UserDetailsService tiene un solo usuario. El nombre del usuario es “user” y la contraseña se genera aleatoriamente al arrancar y se imprime como INFO:
  - Using generated security password: e4918bc4-d8ac-4179-9916-c37825c7eb55
- Puede cambiar el nombre de usuario y la contraseña proporcionando un spring.security.user.name y spring.security.user.password en application.properties.
- Las características básicas predeterminadas en una aplicación web son:
  - Un bean UserDetailsService con almacenamiento en memoria y un solo usuario con una contraseña generada.
  - Inicio de sesión basado en formularios o seguridad básica HTTP (según el tipo de contenido) para toda la aplicación (incluidos los endpoints).
  - Un DefaultAuthenticationEventPublisher para la publicación de eventos de autenticación.

Seguridad MVC

- La configuración de seguridad predeterminada se implementa en SecurityAutoConfiguration y UserDetailsServiceAutoConfiguration. SecurityAutoConfiguration importa SpringBootWebSecurityConfiguration para la seguridad web y UserDetailsServiceAutoConfiguration configura la autenticación, que también es relevante en aplicaciones no web.
- Para desactivar completamente la configuración de seguridad de la aplicación web predeterminada,se puede agregar un bean de tipo WebSecurityConfigurerAdapter (al hacerlo, no se desactiva la configuración UserDetailsService).
- Para cambiar la configuración del UserDetailsService, se puede añadir un bean de tipo UserDetailsService, AuthenticationProvider o AuthenticationManager.
- Las reglas de acceso se pueden anular agregando una personalización de WebSecurityConfigurerAdapter, que proporciona métodos de conveniencia que se pueden usar para anular las reglas de acceso para los puntos finales del actuador y los recursos estáticos.
- EndpointRequest se puede utilizar para crear un RequestMatcher que se basa en la propiedad management.endpoints.web.base-path. PathRequest se puede usar para crear recursos RequestMatcher en ubicaciones de uso común

Elementos principales

- SecurityContextHolder contiene información sobre el contexto de seguridad actual de la aplicación, que contiene información detallada acerca del usuario que está trabajando actualmente con la aplicación. Utiliza el ThreadLocal para almacenar esta información, por lo que el contexto de seguridad siempre está disponible para la ejecución de los métodos en el mismo hilo de ejecución (Thread). Para cambiar eso, se puede utilizar un método estático SecurityContextHolder.setStrategyName (estrategia de cadena).
- SecurityContext contiene un objeto de autenticación, es decir, la información de seguridad asociada con la sesión del usuario.
- Authentication es, desde punto de vista Spring Security, un usuario (Principal)
- GrantedAuthority representa la autorización dada al usuario de la aplicación.
- UserDetails estandariza la información del usuario independientemente del sistema de autenticación.
- UserDetailsService es la interfaz utilizada para crear el objeto UserDetails.

Proceso de Autenticación

- Para poder tomar decisiones sobre el acceso a los recursos, es necesario que el participante se identifique para realizar las comprobaciones necesarias sobre su identidad. Mediante la interfaz Authentication, se pueden acceder a tres objetos bien diferenciados:
  - principal, normalmente hace referencia al nombre del participante
  - credenciales (del usuario) que permiten comprobar su identidad, normalmente su contraseña, aunque también puede ser otro tipo de métodos como certificados, etc...
  - autorizaciones, un lista de los roles asociados al participante.
- Si un usuario inicia un proceso de autenticación, se crea un objeto Authentication, con los elementos Principal y Credenciales. Si realiza la autenticación mediante el empleo de contraseña y nombre usuario, se crea un objeto UsernamePasswordAuthenticationToken. El framework Spring Security aporta un conjunto de clases que permite que esta autenticación se realice mediante nombre de usuario y contraseña. Para ello, utiliza la autenticación que proporciona el contenedor o utiliza un servicio de identificación basado en Single Sign On (sólo se identifica una vez).

Proceso de Autenticación

- Una vez se ha obtenido el objeto Authentication se envía al AuthenticationManager. Una vez aquí, se realiza una comprobación del contenido de los elementos del objeto principal y las credenciales. Se comprueban que concuerdan con las esperadas, añadiéndole al objeto Authentication las autorizaciones asociadas a esa identidad o generandon una excepción de tipo AuthenticationException.
- El propio framework ya tiene implementado un gestor de autenticación que es válido para la mayoría de los casos, el ProviderManager. El bean AuthenticationManager es del tipo ProviderManager, lo que significa que actúa de proxy con el AuthenticationProvider.
- Este es el encargado de realizar la comprobación de la validez del nombre de usuario/contraseña asociada y de devolver las autorizaciones permitidas a dicho participante (roles asociados).
- Esta clase delega la autenticación en una lista que engloba a los proveedores y que, por tanto, es configurable. Cada uno de los proveedores tiene que implementar el interfaz AuthenticationProvider.

Proceso de Autenticación

- Cada aplicación web tendrá una estrategia de autenticación por defecto. Cada sistema de autenticación tendrá su AuthenticationEntryPoint propio, que realiza acciones como enviar avisos para la autenticación.
- Cuando el navegador decide presentar sus credenciales de autenticación (ya sea como formulario HTTP o HTTP header) tiene que existir algo en el servidor que "recoja" estos datos de autenticación. A este proceso se le denomina "mecanismo de autenticación". Una vez que los detalles de autenticación se recogen en el agente de usuario, un objeto "solicitud de autenticación" se construye y se presenta a un AuthenticationProvider.
- El último paso en el proceso de autenticación de seguridad es un AuthenticationProvider. Es el responsable de tomar un objeto de solicitud de autenticación y decidir si es o no válida. El Provider decide si devolver un objeto de autenticación totalmente lleno o una excepción.
- Cuando el mecanismo de autenticación recibe de nuevo el objeto de autenticación, si se considera la petición válida, debe poner la autenticación en el SecurityContextHolder, y hacer que la solicitud original se ejecute. Si, por el contrario, el AuthenticationProvider rechazó la solicitud, el mecanismo de autenticación mostrara un mensaje de error.

Proceso de Autenticación

- El DaoAuthenticationProvider es una implementación de la interfaz de autenticación centrada en el acceso a los datos que se encuentran almacenados dentro de una base de datos. Este proveedor específico requiere una atención especial.
- Esta implementación delega a su vez en un objeto de tipo UserDetailsService, un interfaz que define un objeto de acceso a datos con un único método loadUserByUsername que permite obtener la información de un usuario a partir de su nombre de usuario devolviendo un UserDetails que estandariza la información del usuario independientemente del sistema de autenticación.
- El UserDetails contiene el nombre de usuario, contraseña, los flags isAccountNonExpired, isAccountNonLocked, isCredentialsNonExpired, IsEnabled y los roles del usuario.
- Los roles de usuario son cadenas que por defecto llevan el prefijo de “ROLE_”.

Cifrado de claves

- Nunca se debe almacenar las contraseñas en texto plano, uno de los procesos básicos de seguridad contra robo de identidad es el cifrado de las claves de usuario.
- Spring Security ofrece algoritmos de encriptación que se pueden aplicar de forma rápida al resto de la aplicación.
- Para esto hay que utilizar una clase que implemente la interfaz PasswordEncoder, que se utilizará para cifrar la contraseña introducida a la hora de crear el usuario.
- Además, hay que pasárselo al AuthenticationManagerBuilder cuando se configura para que cifre la contraseña recibida antes de compararla con la almacenada.
- Spring suministra BCryptPasswordEncoder que es una implementación del algoritmo BCrypt, que genera una hash segura como una cadena de 60 caracteres.

```
@Autowired private PasswordEncoder passwordEncoder;

String encodedPass = passwordEncoder.encode(userDTO.getPassword());
```


Configuración de Autenticación

- Para realizar la configuración se crea una clase, anotada con @Configuration y @EnableWebSecurity, que extienda a WebSecurityConfigurerAdapter. La sobreescritua del método configure(AuthenticationManagerBuilder) permite fijar el UserDetailsService y el PasswordEncoder.

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   @Autowired
   UserDetailsService userDetailsService;
   @Bean
   public PasswordEncoder passwordEncoder() {  return new BCryptPasswordEncoder();  }
   @Autowired
   public void configure(AuthenticationManagerBuilder auth) throws Exception {
      auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
   }
}
```

UserDetailsService

```
@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {
	@Autowired
	private PasswordEncoder passwordEncoder;
	@Override
	public UserDetails loadUserByUsername(final String username) throws UsernameNotFoundException {
		switch(username) {
		case "user":		return this.userBuilder(username, passwordEncoder.encode("user"), "USER");
		case "manager":		return this.userBuilder(username, passwordEncoder.encode("manager"), "MANAGER");
		case "admin":		return this.userBuilder(username, passwordEncoder.encode("admin"), "USER", "MANAGER", "ADMIN");
		default: throw new UsernameNotFoundException("Usuario no encontrado");
		}
	}
	private User userBuilder(String username, String password, String... roles) {
		List<GrantedAuthority> authorities = new ArrayList<>();
		for (String role : roles) {
			authorities.add(new SimpleGrantedAuthority("ROLE_" + role));
		}
		return new User(username, password, /* enabled */ true, /* accountNonExpired */ true,
				/* credentialsNonExpired */ true, /* accountNonLocked */ true, authorities);
	}}
```


InMemoryAuthentication

```
@Autowired
  public void configureAuth(AuthenticationManagerBuilder auth) throws Exception {
   auth.inMemoryAuthentication()
       .withUser("user")
           .password("user").roles("USER")
       .and()
       .withUser("manager")
           .password("manager").roles("MANAGER")
       .and()
       .withUser("admin")
           .password("admin").roles("USER", "ADMIN");
  }
```

Autorización

- El AccessDecisionManager es la interfaz que atiende la llamada AbstractSecurityInterceptor producida tras interceptar una petición. Esta interfaz es la responsable final de la toma de decisiones sobre el control de acceso.
- AccessDecisionManager delega la facultad de emitir votos en objetos de tipo AccessDecisionVoter. Se proporcionan dos implementaciones de éste último interfaz:
  - RoleVoter, que comprueba que el usuario presente un determinado rol, comprobando si se encuentra entre sus autorizaciones (authorities).
  - BasicAclEntryVoter, que a su vez delega en una jerarquía de objetos que permite comprobar si el usuario supera las reglas establecidas como listas de control de acceso.
- El acceso por roles se puede fijar para:
  - URLs, permitiendo o denegando completamente
  - Servicios, controladores o métodos individuales

Configuración

- La sobreescritua del método configure(HttpSecurity) permite configurar el http.authorizeRequests():

  .requestMatchers("/static/**").permitAll() acceso a los recursos

  .anyRequest().authenticated() se requiere estar autenticado para todas las peticiones.

  .requestMatchers("/**").permitAll() equivale a anyRequest()

  .requestMatchers("/privado/**", "/config/**").authenticated() equivale a @PreAuthorize("authenticated")

  .requestMatchers("/admin/**").hasRole("ADMIN") equivale a @PreAuthorize("hasRole('ROLE_ADMIN')")
- El método .and() permite concatenar varias definiciones.

Seguridad: Configuración

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   // …
   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http.csrf().disable()
         .authorizeRequests()
            .requestMatchers("/**").permitAll()
            .requestMatchers("/privado/**").authenticated()
            .requestMatchers("/admin/**").hasRole("ADMIN")
         .and().formLogin().loginPage("/login").permitAll()
         .and().logout().permitAll();
   }
}
```


Basada en anotaciones

- Desde la versión 2.0 en adelante, Spring Security ha mejorado sustancialmente el soporte para agregar seguridad a los métodos de capa de servicio proporcionando soporte para la seguridad con anotación JSR-250, así como la anotación original @Secured del marco. A partir de la 3.0 también se puede hacer uso de nuevas anotaciones basadas en expresiones: @PreAuthorize y @PostAuthorize.
- Se puede habilitar la seguridad basada anotaciones en cualquier instancia @Configuration utilizando la anotación @EnableMethodSecurity:
  - prePostEnabled habilita las anotaciones previas y posteriores
  - secureEnabled determina si las anotaciones @Secured deben estar habilitadas
  - jsr250Enabled permite usar las anotaciones @RoleAllowed
- **@Secured**: Anotación para definir una lista de atributos de configuración de seguridad para métodos de un servicio y se puede utilizar como una alternativa a la configuración XML.

```
@Secured({ "ROLE_USER" }) public void create(Contact contact) {
@Secured({ "ROLE_USER", "ROLE_ADMIN" }) public void update(Contact contact) {
@Secured({ "ROLE_ADMIN" }) public void delete(Contact contact){
```

Basada en anotaciones

- **@PreAuthorize**: Anotación para especificar una expresión de control de acceso al método que se evaluará para decidir si se permite o no una invocación del método.

```
@PreAuthorize("isAnonymous()")
@PreAuthorize("isAuthenticated()")
@PreAuthorize("hasAuthority('ROLE_TELLER')")
@PreAuthorize("hasRole('USER') or hasRole('ROLE_EDITOR')")
@PreAuthorize("hasPermission(#contact, 'admin')")
```

- Se puede usar un argumento del método como parte de la expresión:

```
@PreAuthorize("#item.username == authentication.name")
public void putProfile(@RequestBody User item) { … }
```

- **@PostAuthorize**: Anotación para especificar una expresión de control de acceso al método que se evaluará después de que se haya invocado un método.

```
@PostAuthorize("returnObject.username == authentication.principal.nickName")
public CustomUser loadUserDetail(String username) {
   return userRoleRepository.loadUserByUserName(username);
}
```

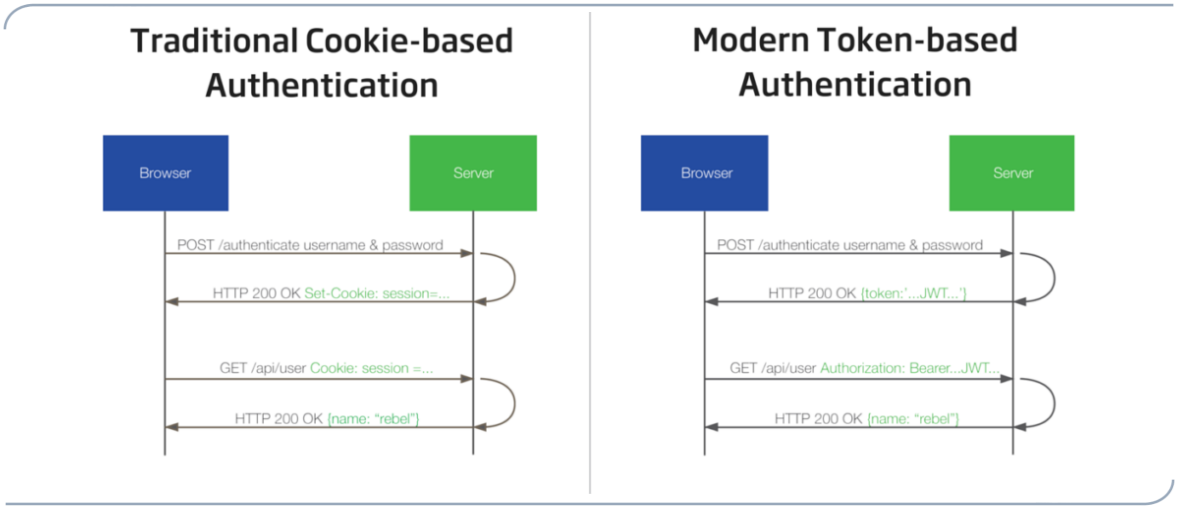

Control de acceso basado en expresiones

Expresión	Descripción
hasRole([role])	Devuelve true si el principal actual tiene el rol especificado. De forma predeterminada, si el rol proporcionado no comienza con 'ROLE_' se agregará. Esto se puede personalizar modificando el defaultRolePrefix en DefaultWebSecurityExpressionHandler.
hasAnyRole([role1,role2])	Se devuelve true si el principal actual tiene alguno de los roles proporcionados (lista de cadenas separadas por comas).
hasAuthority([authority])	Devuelve true si el principal actual tiene la autoridad especificada.
hasAnyAuthority([authority1,authority2])	Se devuelve true si el principal actual tiene alguna de las autorizaciones proporcionadas (se proporciona como una lista de cadenas separadas por comas)
principal	Permite el acceso directo al objeto principal que representa al usuario actual.
authentication	Permite el acceso directo al objeto Authentication actual obtenido del SecurityContext

Control de acceso basado en expresiones

Expresión	Descripción
permitAll	Siempre se evalúa a true
denyAll	Siempre se evalúa a false
isAnonymous()	Devuelve true si el principal actual es un usuario anónimo
isRememberMe()	Devuelve true si el principal actual es un usuario de recordarme
isAuthenticated()	Devuelve true si el usuario no es anónimo
isFullyAuthenticated()	Se devuelve true si el usuario no es un usuario anónimo o recordado
hasPermission(Object target, Object permission)	Devuelve true si el usuario tiene acceso al objetivo proporcionado para el permiso dado. Por ejemplo, hasPermission(domainObject, 'read')
hasPermission(Object targetId, String targetType, Object permission)	Devuelve true si el usuario tiene acceso al objetivo proporcionado para el permiso dado. Por ejemplo, hasPermission(1, 'com.example.domain.Message', 'read')

Autenticación



Autenticación sin estado basada en tokens

- Para solucionar los problemas de sobrecarga y escalabilidad provocados la autenticación basada en sesiones y cookies, surge la autenticación sin estado (stateless). Esto significa que el servidor no va a almacenar ninguna información, ni tampoco la sesión.
- Cuando el usuario se autentica con sus credenciales o cualquier otro método, en la respuesta recibe un token (access token) y, opcionalmente, un refresh token. El token es una cadena encriptada firmada, para evitar alteraciones y ser confiable, con una fecha de expiración corta para evitar vulnerabilidades de seguridad.
- A partir de ese momento, todas las peticiones que se hagan al API llevarán este token en una cabecera HTTP de modo que el servidor pueda identificar qué usuario hace la petición y, una vez verificado el token, confiar en las credenciales suministradas sin necesidad de buscar en base de datos ni en ningún otro sistema de almacenamiento o agente externo.
- Con este enfoque, la aplicación pasa a ser escalable, ya que es el propio cliente el que almacena su información de autenticación, y no el servidor. Así las peticiones pueden llegar a cualquier instancia del servidor y podrá ser atendida sin necesidad de sincronizaciones. Así mismo, diferentes plataformas podrán usar el mismo API. Además se incrementa la seguridad, evitando vulnerabilidades CSRF, al no existir sesiones.
- El refresh token es una identidad verificada y se usa para generar un nuevo access token cuando este expira. Típicamente, si el access token tiene fecha de expiración corta, una vez que caduca, el usuario tendría que autenticarse de nuevo para obtener un nuevo access token. Con el refresh token, que identifica al usuario y tiene una expiración mas generosa, este paso se puede saltar y con una petición al API obtener un nuevo access token que permita al usuario seguir accediendo de forma transparente a los recursos de la aplicación.

Bearer Authentication

- La autenticación de portador (también llamada token de autenticación) es un esquema de autenticación HTTP que involucra tokens de seguridad llamados tokens de portador (Bearer). El nombre "Autenticación de portador" puede entenderse como "dar acceso al portador de este token". El token portador es una cadena encriptada, generalmente generada por el servidor en respuesta a una solicitud de inicio de sesión (Access token). El cliente debe enviar este token en el encabezado Authorization al realizar solicitudes a recursos protegidos:

  Authorization: Bearer <token>
- El esquema de autenticación Bearer se creó originalmente como parte de OAuth 2.0 en RFC 6750, pero a veces también se usa solo. De manera similar a la autenticación básica, la autenticación de portador solo debe usarse a través de HTTPS (SSL).

Single Sign On (SSO)

- El Single SingOn (SSO o inicio de sesión única) es un patrón arquitectónico para permitir a los usuarios el acceso a varias aplicaciones mediante una solo autenticación, utilizando para ello, un proveedor de autenticación en común. SSO permite delegar el proceso de autenticación a una entidad externa, la cual tiene como única responsabilidad autenticar que el usuario es quien dice ser, una vez autenticado recibe un token de identidad.
- Esto quieres decir que un usuario podría entrar a varias aplicaciones sin necesidad de tener que autenticarse en cada una, en su lugar, solo requerirá autenticarse la primera vez en cualquiera de las aplicaciones y posteriormente podrá acceder al resto si pasar por el proceso de autenticación.
- Hay varios tipos principales de SSO, también llamados reduced sign-on systems ("sistemas de autenticación reducida").
  - Enterprise SSO (E-SSO), también llamado legacy sso, funciona para una autenticación primaria, interceptando los requisitos de login presentados por las aplicaciones secundarias para completar los mismos con el usuario y contraseña. Los sistemas E-SSO permiten interactuar con sistemas que pueden deshabilitar la presentación de la pantalla de login.
  - Web SSO (Web-SSO), también llamado gestión de acceso web (web access management, Web-AM o WAM) trabaja solamente con aplicaciones y recursos accedidos vía web.
  - Kerberos es un método popular de externalizar la autenticación de los usuarios. Los usuarios se registran en el servidor Kerberos y reciben un tique, luego las aplicaciones cliente lo presentan para obtener acceso.
  - Identidad federada es una nueva manera de enfrentar el problema de la autenticación, también para aplicaciones Web. Utiliza protocolos basados en estándares para habilitar que las aplicaciones puedan identificar los clientes sin necesidad de autenticación redundante.
  - OpenID es un proceso de SSO distribuido y descentralizado donde la identidad se compila en un Localizador Uniforme de Recursos (URL) que cualquier aplicación o servidor puede verificar.

Ventajas y Desventajas del SSO

- Ventajas
  - Acceso rápido a las aplicaciones: Con el SSO, los usuarios pueden acceder rápidamente a múltiples aplicaciones y servicios con una sola autenticación.
  - Simplificación de la experiencia del usuario.
  - Mayor seguridad: Al utilizar el SSO, se puede implementar una autenticación más robusta y segura.
  - Administración simplificada: El SSO simplifica la administración de usuarios y contraseñas en un entorno empresarial. Los administradores pueden gestionar de manera centralizada las cuentas de usuario y los permisos de acceso, lo que facilita la incorporación y desactivación de usuarios en los
- Desventajas
  - Vulnerabilidad única: Si el SSO se ve comprometido, todas las aplicaciones y servicios vinculados a él también pueden estar en riesgo.
  - Dependencia de la disponibilidad del sistema SSO: Si el sistema SSO experimenta una interrupción o se vuelve inaccesible, los usuarios podrían perder el acceso a todas las aplicaciones y servicios vinculados. Es un potencial cuello de botellas.
  - Complejidad de implementación: La implementación de un sistema SSO puede ser compleja, especialmente en entornos empresariales con múltiples aplicaciones y sistemas.
  - Privacidad y confianza: Al utilizar el SSO, los usuarios deben confiar en que su proveedor de SSO protegerá adecuadamente sus datos personales y de inicio de sesión.

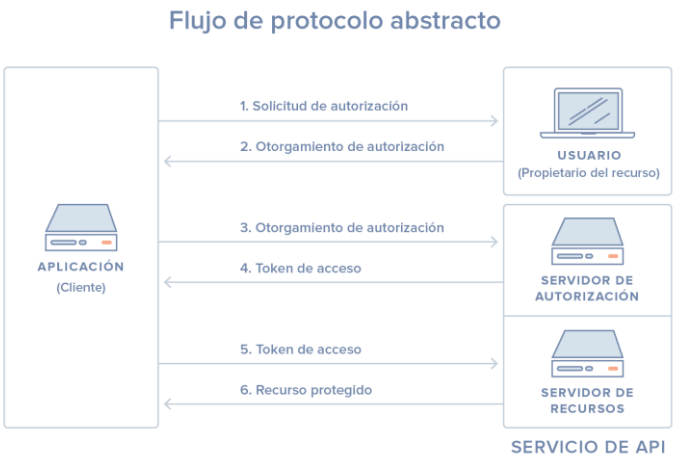
Patrones de diseño

- Patrón: Token de acceso
  - Ha aplicado la arquitectura de microservicios y los patrones de API Gateway . La aplicación consta de numerosos servicios. La puerta de enlace API es el único punto de entrada para las solicitudes de los clientes. Autentica las solicitudes y las reenvía a otros servicios, que a su vez podrían invocar otros servicios. ¿Cómo comunicar la identidad del solicitante a los servicios que tramitan la solicitud? El API Gateway autentica la solicitud y pasa un token de acceso (por ejemplo, JSON Web Token) que identifica de forma segura al solicitante en cada solicitud a los servicios. Un servicio puede incluir el token de acceso en las solicitudes que realiza a otros servicios.
- Patrón: Valet Key
  - Usa un token que proporciona a los clientes acceso directo restringido a un recurso específico, con el fin de descargar la transferencia de datos desde la aplicación. Esto es especialmente útil en aplicaciones que usan sistemas o colas de almacenamiento hospedado en la nube, ya que puede minimizar los costes y maximizar la escalabilidad y el rendimiento.
- Patrón: Federated Identity
  - La autenticación se delega a un proveedor de identidad externo. Esto puede simplificar el desarrollo, minimizar los requisitos de administración de usuarios y mejorar la experiencia del usuario de la aplicación.

OAuth 2

- OAuth 2 es un protocolo de autorización que permite a las aplicaciones obtener acceso limitado a los recursos de usuario en un servicio HTTP, como Facebook, GitHub y Google. Delega la autenticación del usuario al servicio que aloja la cuenta del mismo y autoriza a las aplicaciones de terceros el acceso a dicha cuenta de usuario. OAuth define cuatro roles:
  - Propietario del recurso: Una entidad capaz de otorgar acceso a un recurso protegido. Cuando el propietario del recurso es una persona, se le conoce como usuario final.
  - Servidor de recursos: El servidor que aloja los recursos protegidos, capaz de aceptar y responder a solicitudes de recursos protegidos utilizando tokens de acceso.
  - Cliente: Una aplicación que realiza solicitudes de recursos protegidos en nombre del propietario del recurso y con su autorización. El término "cliente" no implica ninguna característica de implementación particular.
  - Servidor de autorizaciones: El servidor que emite tokens de acceso al cliente después de haber realizado correctamente la autenticación y validar la concesión del propietario del recurso.

OAuth 2



JWT: JSON Web Tokens

- JSON Web Token (JWT) es un estándar abierto (RFC-7519) basado en JSON para crear un token que sirva para enviar datos entre aplicaciones o servicios y garantizar que sean válidos y seguros.
- El caso más común de uso de los JWT es para manejar la autenticación en aplicaciones móviles o web. Para esto cuando el usuario se quiere autenticar manda sus datos de inicio del sesión al servidor, este genera el JWT y se lo manda a la aplicación cliente, posteriormente en cada petición el cliente envía este token que el servidor usa para verificar que el usuario este correctamente autenticado y saber quien es.
- Se puede usar con plataformas IDaaS (Identity-as-a-Service) como Auth0 que eliminan la complejidad de la autenticación y su gestión.
- También es posible usarlo para transferir cualquier dato entre servicios de nuestra aplicación y asegurarnos de que sean siempre válido. Por ejemplo, si tenemos un servicio de envío de email, otro servicio podría enviar una petición con un JWT junto al contenido del mail o cualquier otro dato necesario y que estemos seguros que esos datos no fueron alterados de ninguna forma.

https://jwt.io

Tokens

- Los tokens son una serie de caracteres cifrados y firmados con una clave compartida entre servidor OAuth y el servidor de recurso o para mayor seguridad mediante clave privada en el servidor OAuth y su clave pública asociada en el servidor de recursos, con la firma el servidor de recursos el capaz de comprobar la autenticidad del token sin necesidad de comunicarse con él.
- Se componen de tres partes separadas por un punto: una cabecera con el algoritmo hash utilizado y tipo de token, un documento JSON con datos y una firma de verificación.
- El hecho de que los tokens JWT no sea necesario persistirlos en base de datos elimina la necesidad de tener su infraestructura, como desventaja es que no es tan fácil de revocar el acceso a un token JWT y por ello se les concede un tiempo de expiración corto.

- La infraestructura requiere varios elementos configurables de diferentes formas:
  - El servidor OAuth que realiza la autenticación y proporciona los tokens.
  - El servicio al que se le envía el token, es el que decodifica el token y decide conceder o no acceso al recurso.
  - En el caso de múltiples servicios con múltiples recursos es conveniente un gateway para que sea el punto de entrada de todos los servicios, de esta forma se puede centralizar las autorizaciones liberando a los servicios individuales.

Servidor de Autenticación/Autorizacion

- Dependencias: Spring Web y Spring Security
- En pom.xml

```
<dependency>
   <groupId>com.auth0</groupId>
   <artifactId>java-jwt</artifactId>
   <version>4.4.0</version>
</dependency>
```

- Configurar:

```
server.port=8081
spring.application.name=autentication-service
autenticacion.clave.secreta=Una clave secreta al 99% segura
autenticacion.expiracion.min=10
```

API de autenticación y obtención del token

```
@RestController
public class UserResource {
	@Value("${jwt.secret}")
	private String SECRET;
	@Value("${jwt.expiracion.mim:10}")
	private int EXPIRES_IN_MINUTES = 10;
	@Autowired
	PasswordEncoder passwordEncoder;
	@Autowired
	UsuarioRepositoy dao;

	@PostMapping(path = "/login", consumes = "application/json")
	public AuthToken loginJSON(@Valid @RequestBody BasicCredential credential) {
		var item = dao.findById(credential.getUsername());
		if (item.isEmpty() || !passwordEncoder.matches(credential.getPassword(), item.get().getPassword()))
			return new AuthToken();
		var usr = item.get();
		String token = JWT.create().withIssuer("MicroserviciosJWT").withClaim("usr", usr.getIdUsuario()).withArrayClaim("roles", usr.getRoles().toArray(new
			String[0])).withIssuedAt(new Date(System.currentTimeMillis())).withExpiresAt(new Date(System.currentTimeMillis() + EXPIRES_IN_MINUTES *
			60_000)).sign(Algorithm.HMAC256(SECRET));
		return new AuthToken(true, "Bearer " + token, usr.getNombre());
	}
}
```


Filtro de decodificación del token

```
public class JWTAuthorizationFilter extends OncePerRequestFilter {
	private final String HEADER = "Authorization";
	private final String PREFIX = "Bearer ";
	private String secret;

	public JWTAuthorizationFilter(String secret) {
		super();
		this.secret = secret;
	}

	@Override
	protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws ServletException, IOException {
		try {
			String authenticationHeader = request.getHeader(HEADER);
			if (authenticationHeader != null && authenticationHeader.startsWith(PREFIX)) {
				DecodedJWT token = JWT.require(Algorithm.HMAC256(secret)).withIssuer("MicroserviciosJWT").build().verify(authenticationHeader.substring(PREFIX.length()));
				List<GrantedAuthority> authorities = token.getClaim("roles").asList(String.class).stream().map(role -> new SimpleGrantedAuthority(role)).collect(Collectors.toList());
				UsernamePasswordAuthenticationToken auth = new UsernamePasswordAuthenticationToken(token.getClaim("usr").toString(), null, authorities);
				SecurityContextHolder.getContext().setAuthentication(auth);
			}
		} catch(JWTVerificationException ex) {
			response.sendError(403, ex.getMessage());
		} finally {
			chain.doFilter(request, response);
		}
	}
}
```

Configurar con el filtro

```
@Configuration
@EnableMethodSecurity(prePostEnabled = true, securedEnabled = true, jsr250Enabled = true)
public class WebSecurityConfig {
  @Value("${jwt.secret}")
  private String SECRET;

   @Bean
   SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
      return http
            .cors(Customizer.withDefaults())
            .csrf((csrf) -> csrf.disable())
            .sessionManagement((session) -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .addFilterAfter(new JWTAuthorizationFilter(SECRET), UsernamePasswordAuthenticationFilter.class)
            .authorizeHttpRequests(requests -> requests
               .requestMatchers(HttpMethod.GET, "/publico").permitAll()
               .anyRequest().authenticated()
             )
            .build();
   }
}
```