Índice

1. Anotaciones y frameworks.

1.1. Frameworks.

De una forma básica podemos definir un framework (o marco de trabajo) como un componente de Java que es capaz de gestionar a otros componentes, siendo capaza de: crear instancias de objetos automáticamente, analizar anotaciones, gestionar recursos, e inyectar automáticamente datos en variables.
En Java, ejemplos de frameworks son JavaFx (para gestionar aplicaciones de ventanas gráficas), JavaEE y Spring MVC (para gestionar aplicaciones web), Spring Boot, y otros.
Todos estos framewoks hacen un uso intensivo de anotaciones para aplicar inyección de código utilizando patrones de diseño de inversión de control. La inversión de control utiliza técnicas de inyección de dependencias y reflexión para crear instancias de las clases que necesitemos según el momento y necesidad.

1.1.1. Ejemplo de un framework sencillo.

Dadas sus características, habitualmente los frameworks permiten crear determinados tipos de aplicaciones siguiendo un patrón de diseño.
Veremos un ejemplo sencillo de un framework para diseñar aplicaciones de consola que muestren un menú de opciones. El usuario deberá añadir al proyecto una clase que proporcione métodos para procesar cada opción, de forma que esta clase deberá seguir un determinado patrón de diseño aplicando anotaciones.
Primero crearemos una anotación que permita establecer el texto de las opciones del menú:

```
@Target({ ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface MenuOptions {
  String[] value() default { "Salir" };
}
```

Esta anotación será aplicada sobre una clase que proporcione el proceso del menú. A la anotación se le pasará como argumento un array de opciones menos la última, que está reservada para finalizar el menú. La clase de proceso del menú deberá proporcionar métodos sobre los que se aplicarán la siguiente anotación:

```
@Target({ ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface MenuAction {
  int value();
}
```

Como argumento de esta anotación se asignará el número de opción, contando desde 1.
Ahora la clase para procesar el menú será la siguiente:

```
@MenuOptions({ "Opcion 1" })
public class MyMenu {
  @MenuAction(1)
  public void opcion1() {
    System.out.println("Se eligió la opcion 1.");
  }
}
```

Para simplificar se ha añadido una única opción. El framework añadirá una última opción para finalizar el menú.
Por último, nuestro framework constará de una única clase:

```
public class FrameworkMenu {
  private static Object instanceMenuClass;  // una instancia de la clase que procesa el menú
  private static int numOptions;    // número de opciones totales del menú
  private static Map<Integer, Method> methods;   // Un mapa que asocia {opcion, método}
  private static List<String> options;   // Lista con el texto de las opciones de menú

  // Método principal.
  // Instancia un objeto de la clase que procesa el menú y ejecuta sus métodos.
  public static void run(Class menuClass) throws Exception {
    options = tryGetOptions(menuClass);
    instanceMenuClass = menuClass.getDeclaredConstructor().newInstance();
    numOptions = options.size();
    methods = getMethods(menuClass);
```

```
        boolean end;
        do {
          printMenu();
          int option = inputOption();
          end = processOption(option);
        } while (!end);
      }

      // Recupera el texto de las opciones del menú, o lanza una excepción.
      private static List<String> tryGetOptions(Class menuClass) {
        var menu = (MenuOptions) menuClass.getAnnotation(MenuOptions.class);
        if (menu == null) {
          throw new RuntimeException("Clase no soportada.");
        }
        return Stream.concat(Stream.of(menu.value()), Stream.of("Salir")).toList();
      }

      // Obtiene un mapa que asocia el índice de la opción del menú con un método de la clase dada
      private static Map<Integer, Method> getMethods(Class<?> menuClass) {
        return Stream.of(menuClass.getDeclaredMethods())
          .filter(method -> method.isAnnotationPresent(MenuAction.class))
          .collect(Collectors.groupingBy(
              m -> m.getDeclaredAnnotation(MenuAction.class).value(),
              Collectors.reducing(null, (a, b) -> b))
            );
      }

      // Imprime las opciones del menú
      private static void printMenu() {
        IntStream.range(0, numOptions)
          .forEach(index -> System.out.printf("%d. %s\n", index + 1, options.get(index)));
      }

      // Solita una opción de menú por teclado
      private static int inputOption() {
        return Integer.parseInt(System.console().readLine("Escriba la opción entre 1 y %d: ", numOptions));
      }

      // Ejecuta el método asociado con la opción dada, o no hace nada si no existe
      private static boolean processOption(int option) throws Exception {
        if (option == numOptions) {
          return true; // finalizar
        }
        Method method = methods.get(option);
        if (method == null)
          return false;  // si no existe el método continúa el menú
        method.invoke(instanceMenuClass);   // Ejecuta el método
        return false;
      }
    }
```

En la clase principal del proyecto podemos ejecutar el framework:

```
    public static void main(String[] args) throws Exception {
        FrameworkMenu.run(MyMenu.class);
    }
```

En la pantalla obtendremos el siguiente resultado:

```
    1. Opcion 1
    2. Salir
```

```
Escriba la opción entre 1 y 2: 1
Se elegió la opcion 1.
1. Opcion 1
2. Salir
Escriba la opción entre 1 y 2: 2
```

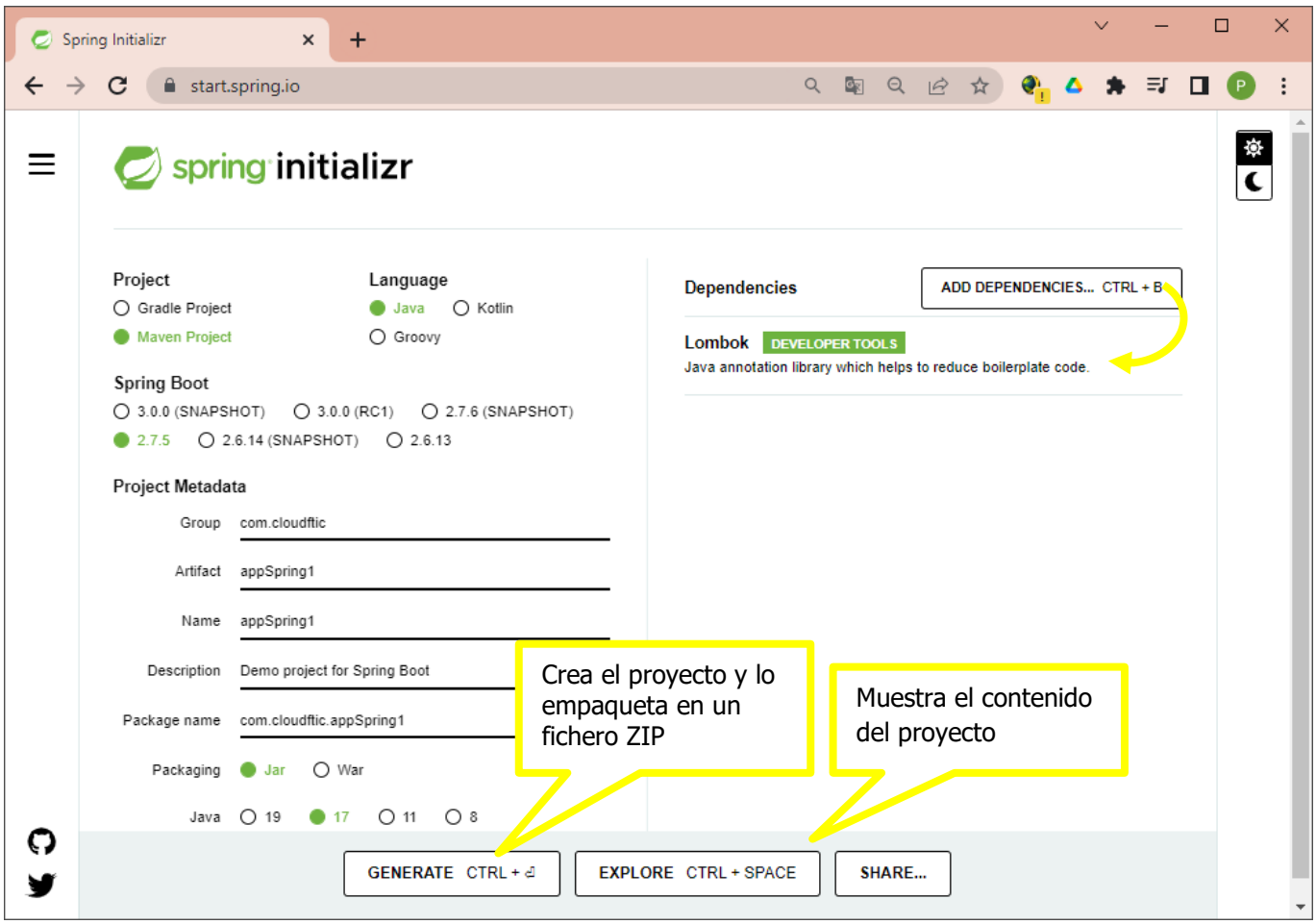
1.2. Fundamentos del framework Spring.

Spring es un Framework Java diseñado para agilizar el desarrollo de aplicaciones empresariales. Implementa un contenedor de inversión de control (IoC), que permite inyectar datos en variables. Puede ser usado para crear aplicaciones web o de escritorio estándar, y cuenta con una gran variedad de módulos que nos facilitan el trabajo.

1.2.1. Creación de aplicaciones Spring Boot.

Spring Boot es un sub-proyecto de Spring que simplifica y agiliza el proceso de creación y desarrollo de aplicaciones web o de escritorio que utilicen el Framework Spring. La configuración requerida para iniciar una aplicación, de cualquier tipo, es mínima y automática, puesto que Spring Boot se auto-configura analizando el *classpath*.

Podemos acceder a la página «https://start.spring.io/» para configurar un proyecto y descargarlo:



Tras seleccionar las opciones adecuadas hay que pulsar el botón [Generate] y se descargará un fichero ZIP que contiene el proyecto.

En VS Code podemos instalar la extensión «Spring Initializr Java Support» para hacer lo mismo. En la paleta de comandos debemos seleccionar «Spring Initializr: Create Maven Project» y seguir los pasos de configuración hasta crear el nuevo proyecto. Si creamos un proyecto con la misma configuración que la mostrada en la imagen previa, incluiría un fichero pom.xml con el siguiente contenido:

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

```
    <version>2.7.5</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  ....
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>
  </dependencies>
  ....
```

Los proyectos de Spring Boot son módulos hijos de un proyecto padre «spring-boot-starter-parent» que define muchas dependencias habituales. En nuestro proyecto simplemente especificaremos las que deseamos sin tener que indicar la versión. La dependencia «spring-boot-starter» da soporte al framework sólo para aplicaciones de escritorio. La dependencia «spring-boot-starter-test» da soporte para pruebas unitarias a través del framework.
La clase principal lanza el framework:

```
package com.cloudftic.appspring1;

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

@SpringBootApplication
public class Appspring1Application {
  public static void main(String[] args) {
    SpringApplication.run(Appspring1Application.class, args);
  }
}
```

La clase principal debe estar anotada con @SpringBootApplication para habilitar todas las características del framework. Esta anotación es equivalente a utilizar las siguientes:

```
@Configuration              // Habilita la inyección de datos mediante Beans
@EnableAutoConfiguration    // Habilita la inyección de Beans predefinidos
@ComponentScan              // Habilita la búsqueda de componentes dentro del paquete principal
```

El método SpringApplication.run() lanza el framework de Spring. Este método retorna un objeto que usaremos posteriormente para acceder a ciertas características:

```
ConfigurableApplicationContext context = SpringApplication.run(Appspring1Application.class, args);
```

1.2.2. Beans e inversión de control.

Una de las características principales del framework Spring es el contenedor de inversión de control (IoC). Este contenedor es el encargado de administrar datos para su inyección en variables. Spring utiliza el concepto de Bean como aquel objeto de Java que es capaz de administrar. Administrar significa que Spring es capaz de instanciar, inicializar y destruir objetos, así como asignarlos automáticamente a variables, campos o parámetros.

Creación de un Bean.

Vemos como crear un Bean y cómo funciona el mecanismo de inversión de control. Empezaremos añadiendo al proyecto una clase para inyección, basada en una interfaz:

```
public interface HelloBean {
  void saludar();
}
```

```
public class HelloBeanIpml implements HelloBean {
  @Override
  public void saludar() {
    System.out.println("Esto funciona.");
  }
}
```

Y ahora crearemos un Bean que para que Spring administre un objeto de tipo HelloBean. Los Beans deben ser creados en clases con la anotación @Configuration o @SpringBootApplication. Básicamente, los Beans se crean anotando un método con @Bean:

```
package com.cloudftic.appspring1;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfiguration {
  @Bean(name="hello")
  public HelloBean hello() {
    return new HelloBeanIpml();
  }
}
```

Nota: los métodos anotados con @Bean no pueden llevar los modificadores private ni final. Se permite que sean static.

Como se puede ver, el método hello() simplemente se anota con @Bean y retorna un objeto de tipo HelloBean. En la anotación se indica el nombre asociado al Bean, pero si no se indica, por defecto tendrá el mismo nombre que el método.

Nota: Para facilitar el descubrimiento de clases de configuración es conveniente ubicar las clase anotadas con @Configuration en el mismo paquete (o en un subpaquete) que la clase principal anotada con @SpringBootApplication. En caso contrario debemos indicar rutas de paquetes en la anotación @SpringBootApplication, como por ejemplo: @SpringBootApplication(scanBasePackages = {"paquetes"})

Durante la ejecución del programa Spring administrará el objeto retornado por este método. Por defecto, Spring siempre servirá el mismo objeto cuando sea solicitado (el denominado comportamiento singletón).

Acceso a un Bean.

Para obtener el bean primero usaremos el método ConfigurableApplicationContext.getBean() en la clase principal.

```
public static void main(String[] args) {
  ConfigurableApplicationContext context = SpringApplication.run(Appspring1Application.class, args);
  HelloBean hello1 = context.getBean(HelloBean.class);
  HelloBean hello2 = context.getBean("hello", HelloBean.class);
  System.out.println("Son la misma instancia: " + (hello1==hello2));
  hello1.saludar();
  context.close();
}
```

El método getBean() permite recuperar un bean por su tipo, o por su nombre. En este ejemplo obtendremos el siguiente resultado:

```
Son la misma instancia: true
Esto funciona.
```

Podemos comprobar que sucesivas recuperaciones del Bean retornan la misma instancia, aunque este comportamiento podremos modificarlo.

Importante: Cuando se recupera un bean por su tipo, sólo debe existir un único bean del tipo (o subtipo) especificado.

Como iremos viendo, Spring crea varios beans predefinidos. Uno de ellos es el que retorna la instancia de la clase principal, la cual podemos obtener de la siguiente manera:

```
var appInstance = context.getBean(Appspring1Application.class);
```

Otras anotaciones para administrar beans.

La anotación @Bean sólo se aplica sobre métodos. Pero disponemos de las anotaciones @Component, @Controller, @Service, @Repository, que se aplican directamente sobre la clase del tipo del Bean. El uso adecuado de estas anotaciones depende del uso que le demos a la clase:

@Controller, anota clases controladoras en aplicaciones web.
@Repository, anota clases que gestionan bases de datos.
@Service, anota clases de servicio.
@Component, anota cualquier otro tipo de clase para inyección.

Usaremos la anotación @Service especificando como argumento el nombre de Bean:

```
public interface HelloBean {
  void saludar();
}

@Service("hello")
public class HelloBeanIpml implements HelloBean {
  @Override
  public void saludar() {
    System.out.println("Esto funciona.");
  }
}
```

En este caso prescindiremos de la clase de configuración y del método anotado con @Bean.

el contenedor Spring analizará el proyecto en busca de clases que contengan estas anotaciones, de encontrarla procederá a crear los beans correspondientes al tipo de anotación, más adelante veremos en detalles cada una de estas anotaciones.

```
public static void main(String[] args) {
  ConfigurableApplicationContext context = SpringApplication.run(Appspring1Application.class, args);
  HelloBean hello1 = context.getBean("hello", HelloBean.class);
  HelloBean hello2 = context.getBean(HelloBean.class);
  System.out.println("Son la misma instancia: " + (hello1==hello2));
  hello1.saludar();
  context.close();
}
```

Primero recuperamos el bean por su nombre "hola", y después por su tipo, y podemos comprobar que obtenemos los mismos resultados que el caso previo.

Importante: Spring es capaz de buscar un Bean a partir de una interfaz que sea implementada por la clase anotada. Pero esto sólo ocurre si no hay ninguna otra clase que implemente dicha interfaz, pues en ese caso se producirá un error.

1.2.3. Inyección automática con autowiring.

Spring posee un mecanismo de inyección automática llamado *autowiring*, este mecanismo inspecciona el contexto de la aplicación en busca de variables (campos y parámetros) anotadas con @Autowired, y les inyecta automáticamente un bean.

Pero debemos tener en cuenta que la anotación @Autowired solo tendrá sentido dentro de un componente que sea administrado (es decir, que sea un bean).

Podemos aplicar @Autowired sobre un campo de un bean, sobre un método constructor para inyectar un parámetro, o sobre un método setter. Vamos a crear una clase de servicio que haga uso del bean hello de estas tres formas:

```
@Service("saludo")
class GestionSaludo {
  @Autowired
  private @Getter HelloBean hello1;

  private @Getter HelloBean hello2;
  private @Getter HelloBean hello3;
```

```
    @Autowired
    public GestionSaludo(HelloBean hello) {
      this.hello2 = hello;
    }

    @Autowired
    public void setHello(HelloBean hello) {
      this.hello3 = hello;
    }
  }
```

Spring se encargará de inicializar las variables hello1, hello2, y hello3 de manera automática.

```
  public static void main(String[] args) {
    ConfigurableApplicationContext context = SpringApplication.run(Appspring1Application.class, args);
    GestionSaludo gsaludo = context.getBean(GestionSaludo.class);
    gsaludo.getHello1().saludar();
    gsaludo.getHello2().saludar();
    gsaludo.getHello3().saludar();
    context.close();
  }
```

Y obtendremos el resultado:

```
Esto funciona.
Esto funciona.
Esto funciona.
```

Nota: Aplicar a anotación @Autowired sobre un campo estático no tendrá ningún efecto.

Si alguna de las dependencias con @Autowired no se puede resolver (porque no existe un bean del tipo solicitado o hay duplicidad de beans) se producirá una excepción. Podemos usar @Autowired(required=false) para indicar que la dependencia es opcional, si no hay un bean disponible que inyectar no se producirá la excepción.

@Autowired se basa en el tipo de la variable y el nombre de la variable para buscar el bean adecuado. En el ejemplo:

```
    @Autowired
    private @Getter HelloBean hello1;
```

Spring busca un único bean del tipo HelloBean, y si encuentra varios busca el que coincida con el nombre de la variable. También podemos forzar la inyección por nombre del bean con la anotación @Qualifier:

```
  @Autowired
  @Qualifier("hello")
  private @Getter HelloBean hello1;
```

Ciclo de vida de un bean.

Por defecto, Spring mantiene una única instancia de un bean y se encarga de destruirlo. Podemos cambiar este comportamiento con la anotación @Scope:

@Scope("singleton"), el contenedor crea una sola instancia del bean; todas las solicitudes devolverán el mismo objeto, que se almacena en caché. Cualquier modificación al objeto se reflejará en todas las referencias al bean. Este ámbito es el valor predeterminado si no se especifica ningún otro ámbito.

@Scope("prototype"), el contenedor devolverá una instancia diferente cada vez que se solicite el bean.

Los siguientes alcances sólo son aplicables en aplicaciones web:

@Scope("request"), el contenedor crea una instancia diferente en cada solicitud web.

@Scope("session"), el contenedor crea una instancia diferente en cada sesión web.

@Scope("application"), el contenedor crea una única instancia para cada aplicación web.

@Scope("websocket"), el contenedor crea una instancia diferente para cada comunicación con websocket.

El contenedor de Spring se encarga de administrar el ciclo de vida de cada Bean, pero habrá ocasiones donde necesitemos realizar alguna inicialización o finalización sobre un bean. Podemos usar las anotaciones @PostConstruct y @PreDestroy para decorar métodos que actúen de inicializadores y finalizadores:

```
  @Service
  public class MyService {
    public MyService(){
      System.out.println("Se ha instanciado el bean MyService");
    }
```

```
  @PostConstruct
  public void init(){
    System.out.println("El bean MyService ha sido inicializado");
  }
  @PreDestroy
  public void destroy(){
    System.out.println("El bean MyService va a ser destruido.");
  }
}
```

El método init() será invocado justo después de que Spring instancie un bean de esta clase. Y el método destroy() será invocado justo antes de que Spring destruya un bean de esta clase.

1.2.4. Inicializaciones usando beans.

Una de las formas de realizar inicializaciones en la aplicación, es creando un bean que retornen un objeto de tipo CommandLineRunner. Se trata de una interfaz funcional con la siguiente definición:

```
public interface CommandLineRunner extends Runner {
  void run(String... args) throws Exception;
}
```

Podemos utilizar un bean de este tipo para realizar inicializaciones de una base de datos, lectura de configuraciones, etc.

Un ejemplo de uso puede ser el siguiente:

```
@Configuration
public class AppSetup {
  @Bean
  public CommandLineRunner setup() {
    return (args) ->  System.out.println("Esto se ejecuta automáticamente tras el arranque de la aplicación.");
  }
}
```

Se pueden definir varios beans CommandLineRunner dentro del mismo contexto de aplicación y se pueden ordenar mediante la anotación @Order().

1.2.5. Trabajando con archivos de recursos.

Cuando se crea un proyecto de Spring Boot se añade un fichero de propiedades «application.properties» en la carpeta «resources». Éste es el fichero de configuración principal de la aplicación y en él podremos indicar muchos datos de configuración, como conexiones a bases de datos, o información personalizada.

Archivos de propiedades.

Supongamos que el fichero «application.properties» de nuestra aplicación tiene el siguiente contenido:

```
mi_nombre = Pedro
```

Este fichero puede ser accedido de forma general usando la clase Properties:

```
package com.cloudftic.appspring1;
....
@SpringBootApplication(scanBasePackages = {"com.cloudftic"} )
public class Appspring1Application {
  public static void main(String[] args) throws IOException {
    var context = SpringApplication.run(Appspring1Application.class, args);
    var properties = new Properties();
    properties.load(Appspring1Application.class.getResourceAsStream("../../../application.properties"));
    System.out.println(properties);
    context.close();
  }
}
```

Pero Spring ofrece otras forma más amigables de acceder a estas propiedades. La primera forma es utilizando el bean de tipo org.springframework.core.env.Environment, que podemos obtener con getEnvironment():

```
public static void main(String[] args) throws IOException {
  var context = SpringApplication.run(Appspring1Application.class, args);
  String recurso = context.getEnvironment().getProperty("mi_nombre", "un valor por defecto");
  System.out.println(recurso);
  context.close();
```

```
}
```

La clase Environment posee métodos getProperty() que permite especificar un clave y un valor por defecto si no encuentra la clave.

La segunda forma es utilizando las anotaciones siguientes:

```
import org.springframework.context.annotation.PropertySource;
import org.springframework.beans.factory.annotation.Value;
```

La anotación @PropertySource se utiliza para indicar de qué fichero de propiedades queremos leer propiedades. Y la anotación @Value se encarga de inyectar una propiedad en una variable. Veamos un ejemplo:

```
@SpringBootApplication
@PropertySource("classpath:application.properties")
public class Appspring1Application {
  @Value("${mi_nombre}")
  prinvate String miNombre;

  public static void main(String[] args) throws IOException {
    var context =SpringApplication.run(Appspring1Application.class, args);
    System.out.println(context.getBean(Appspring1Application.class).niNombre);
    context.close();
  }
}
```

Primero se aplica la anotación @PropertySource sobre una clase administrada. Con la sintaxis "classpath: recurso" se indica la ubicación del fichero de recursos relativa a la carpeta raíz de los paquetes de nuestra aplicación. Tengamos en cuenta que en un proyecto de Maven, los ficheros que ubiquemos en la subcarpeta «resources» serán copiados en la carpeta raíz de las clases compiladas. Si queremos cargar un fichero ubicado en una ruta absoluta podemos usar la siguiente sintaxis:

```
@PropertySource("file:c:/temp/datos.properties")
```

También se puede indicar más de un fichero de recursos:

```
@PropertySource("classpath:foo.properties")
@PropertySource("classpath:bar.properties")
public class Appspring1Application { ...
```

A continuación, con la anotación @Value indicamos la clave de la propiedad (se utiliza la sintaxis ${clave} para referenciar la clave). Si no ha utilizado la anotación @PropertySource la clave será buscada en el fichero «application.properties» de la aplicación.

Nota: aplicar la anotación @Value sobre campos estáticos no producirá ningún efecto.

El valor asociado a la clave será inyectado en la variable miNombre. En aquellos casos donde la clase puede existir o no, podemos especificar un texto por defecto:

```
@Value("${mi_nombre : no existe}")
```

En el método main() recuperamos el Bean y comprobamos que es el valor asignado en el fichero de propiedades.

Propiedades del sistema y de línea de comandos.

Spring también trata las propiedades del sistema y las propiedades de la línea de comandos como propiedades de aplicación.

Una propiedad de línea de comandos se define en la línea de comando del intérprete con dos guiones seguidos del nombre y el valor. Por ejemplo, se define la propiedad "home" con el valor "Mi casa":

```
(carpeta de trabajo)> java -jar app.jar --home="Mi casa"
```

Existen varias propiedades del sistema predefinidas, y también se pueden añadir en la línea de comando del intérprete con la opción -D:

```
(carpeta de trabajo)> java -Dhome="Mi casa" -jar app.jar
```

Las propiedades del sistema también se pueden recuperar con System.getProperty().

Propiedades con valores aleatorios.

Si en las propiedades queremos asignar valores aleatorios podemos usar la propiedad predefinida "random" de la siguiente manera:

```
aleatorio.int = ${random.int}
aleatorio.long=${random.long}
aleatorio.uuid=${random.uuid}
```

Y recuperar la propiedad de la forma habitual:

```
@Value("${aleatorio.int}")
private int n;
```

O bien inyectar directamente un valor:

```
@Value("${random.int}")
private int n;
```

Propiedades anidadas.

Spring permite que las propiedades de los ficheros de recursos se puedan agrupar. Para agrupar propiedades deben comenzar por las mismas palabras separadas con un punto. Por ejemplo, añadiremos al fichero «application.properties» información sobre un cliente:

```
client.name=Juan Miguel
client.phone=666666666
client.password=
```

Y podemos crear una clase para contener la información de un cliente:

```
@Configuration
@ConfigurationProperties(prefix = "client")
@Data
public class ClientProperties {
  private String name;
  private String phone;
  private String password;
}
```

La anotación @Configuration provocará la creación de un bean. La anotación @ConfigurationProperties buscará en el archivo de propiedades de la aplicación aquellas que comiencen por el prefijo dado, e inyectará los valores en los campos correspondientes.

```
public static void main(String[] args) throws IOException {
  var context = (AnnotationConfigApplicationContext) SpringApplication.run(Appspring1Application.class, args);
  var cliente = context.gettBean(ClientProperties.class)
  ....
```

Otros tipos de recursos.

Con la anotación @Value también es posible cargar archivos de todo tipo: archivos de texto, imágenes, audio, etc. En este caso el tipo de datos debe ser org.springframework.core.io.Resource, el cual nos permitirá obtener la ruta del fichero o un InputStream que podemos usar para leer el archivo.

Supongamos que en la carpeta «resources» del proyecto existe el fichero «txt/informes.txt», y queremos leer su contenido:

```
@Value("${classpath:txt/informes.txt}")
@Bean
public String informes(Resource recurso) {
  try {
    return Files.readString(recurso.getFile().toPath());
  } catch (IOException e) {
    return null;
  }
}

public static void main(String[] args) throws IOException {
  var context = SpringApplication.run(Appspring1Application.class, args);
  System.out.println(context.getBean("informes",String.class));
  context.close();
}
```

Gestión de propiedades personalizada.

Si queremos implementar nuestros propio manejador de propiedades debemos crear un bean que retorne un PropertySourcesPlaceholderConfigurer.

A continuación, se muestra un ejemplo:

```
@Bean
public static PropertySourcesPlaceholderConfigurer properties(){
```

```
    PropertySourcesPlaceholderConfigurer pspc = new PropertySourcesPlaceholderConfigurer();
    // Definimos recursos
    Resource[] resources = new ClassPathResource[ ] { new ClassPathResource( "foo.properties" ) };
    // Los añadimos al proveedor
    pspc.setLocations( resources );
    // Ignora propiedades no encontradas
    pspc.setIgnoreUnresolvablePlaceholders( true );
    return pspc;
  }
```

La clase PropertySourcesPlaceholderConfigurer proporciona métodos para proporcionar varios orígenes de propiedades.

1.2.6. Soporte para mensajes multilenguaje.

Spring soporta el estándar I18N de internalización de mensajes ubicados en ficheros de recursos multilenguaje. Los mensajes serán propiedades de ficheros ".properties" siguiendo la convención de nombrado para países e idiomas. Por defecto, Spring busca ficheros de idioma con el nombre base "messages.properties", y se crean ficheros para idiomas como:

"messages_es.properties" para el español neutro

"messages_en.properties" para el inglés neutro

Y se crean ficheros para idiomas de un país concreto como:

"messages_es_ES.properties" para el español de España

"messages_es_MX.properties" para el español de México

Todos los ficheros de recursos deben colocarse dentro de la carpeta «src/resources» de la aplicación. Supongamos que hemos añadido el siguiente mensaje:

```
saludo=Hola, {0}
```

Con la sintaxis {índice} se indica la posición donde concatenar argumentos para el mensaje.

Ahora podemos recuperar mensajes de varias formas:

1) Usando el objeto ConfigurableApplicationContext:

```
public static void main(String[] args) {
  var context =  SpringApplication.run(Appspring1Application.class, args);
  String mensaje = context.getMessage("saludo", new Object[] {"Juan"},Locale.getDefault());
  System.out.println(mensaje);
  context.close();
}
```

2) Inyectando un MessageSource con @Autowired:

```
@Autowired
private MessageSource messageSource;

public static void main(String[] args)  {
  var context =  SpringApplication.run(Appspring1Application.class, args);
  String mensaje = context.getBean(Appspring1Application.class)
        .messageSource.getMessage("saludo", new Object[] {"Juan"},Locale.getDefault());
  System.out.println(mensaje);
  context.close();
}
```

Si queremos personalizar el nombre del fichero de recursos de idioma podemos proporcionarlo en el fichero «application.properties». El siguiente ejemplo supone que por lo menos existe el fichero «src/resources/i18n/mensajes.properties».

```
spring.messages.basename=i18n.mensajes
```

También podemos crear beans personalizados de tipo MessageSource:

```
@Bean("messagesLabel")
public ResourceBundleMessageSource messageSource() {
  var source = new ResourceBundleMessageSource();
  source.setBasenames("messages/label");
  source.setUseCodeAsDefaultMessage(true);
  return source;
```

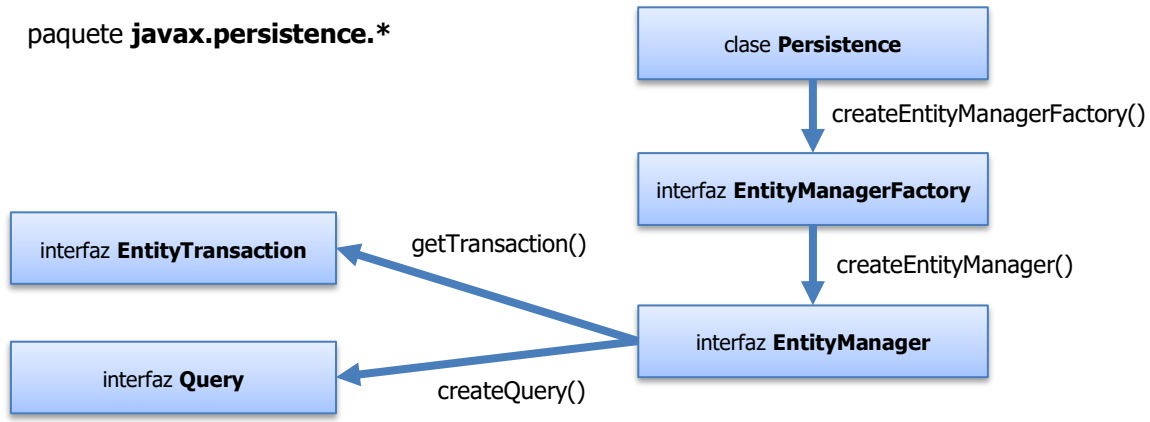
```
}
```

2. El API Java Persistence

El API Java Persistence (JPA) es una tecnología disponible desde Java 5, que permite el salto desde JDBC a objetos del modelo de Java. JDBC no proporciona por sí mismo un modelo objeto-relacional completo que permita trabajar con registros de base de datos a través de objetos personalizados; el API Java Persistence proporciona este modelo.

2.1. Arquitectura de Java Persistence.

El siguiente diagrama muestra la relación entre los componentes principales de la arquitectura de JPA:



La interfaz javax.persistence.EntityManager es el núcleo de esta tecnología. Mediante un EntityManager se crean consultas y se recuperan datos desde el origen de datos como objetos Java Bean (estos objetos se denominan entidades de persistencia).

Pero varias de las interfaces de este diagrama son solo necesarias para su utilización fuera de un servidor de aplicaciones que soporte JPA. Por ejemplo, los servidores de aplicaciones que soportan EJBs son capaces de instanciar automáticamente un EntityManager, con lo cual no es necesario usar Persistence ni EntityManagerFactory. Por otra parte, las transacciones dentro de un servidor de aplicaciones se controlan mediante un mecanismo estándar de controles, y por lo tanto la interfaz EntityTransaction tampoco es utilizada en ese entorno.

A continuación, se detalla el uso de estos elementos:

- ***Clases de entidad Java***. Son clases Java Bean con propiedades y anotaciones para mapear (o asociar) las propiedades (o campos) con las columnas de una consulta.
- ***Archivo «persistence.xml»***. Es un archivo de configuración con información sobre la cadena de conexión con la base de datos y el mapeado entre columnas y clases de entidad.
- ***Persistence***. La clase javax.persistence.Persistence contiene métodos estáticos de ayuda para obtener una instancia de EntityManagerFactory de una forma independiente al proveedor de la implementación de JPA.
- ***EntityManagerFactory***. La clase javax.persistence.EntityManagerFactory nos ayuda a crear objetos de tipo EntityManager utilizando el patrón de diseño Factory.
- ***EntityManager***. La clase javax.persistence.EntityManager es la interfaz principal de JPA utilizada para la persistencia de las aplicaciones. Cada EntityManager puede realizar operaciones de creación, lectura, actualización y borrado sobre un conjunto de objetos persistentes.
- ***EntityTransaction***. Cada instancia de EntityManager tiene una relación de uno a uno con una instancia de javax.persistence.EntityTransaction, la cual permite operaciones sobre los datos persistentes. Mediante EntityTransaction, varias operaciones agrupadas forman una unidad de trabajo transaccional, de forma que todo el grupo sincroniza su estado de persistencia en la base de datos o todas las operaciones fallan en el intento.
- ***Query***. La interfaz javax.persistence.Query está implementada por cada proveedor de JPA para encontrar objetos persistentes manejando cierto criterio de búsqueda. JPA estandariza el soporte para consultas utilizando Java Persistence Query Language (JPQL) y Structured Query Language (SQL). Podemos obtener una instancia de Query desde una instancia de un EntityManager.

• ***Entity***. La clase javax.persistence.Entity es una anotación Java que decora las clases de entidad. De esta forma, cada objeto puede almacenar los datos de un registro.

2.1.1. Unidades de persistencia.

La tecnología de persistencia se basa en el uso de un fichero «persistence.xml» donde se declaran unidades de persistencia. Este fichero se ubica habitualmente en la carpeta META-INF de nuestro proyecto.

Un contenido posible de este fichero puede ser el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="AppTestPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>entidades.Empresa</class>
    <class>entidades.Empleado</class>
    <properties>
      <property name="javax.persistence.jdbc.url"
          value="jdbc:derby://localhost:1527/Negocio"/>
      <property name="javax.persistence.jdbc.user" value="APP"/>
      <property name="javax.persistence.jdbc.driver"
          value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.password" value="app"/>
    </properties>
  </persistence-unit>
</persistence>
```

En este archivo se crean unidades de persistencia mediante el elemento <persistence-unit />. Cada unidad de persistencia configura las propiedades de conexión con la base de datos y las clases de entidad. En este ejemplo, la entidad entidades.Empesa va a mapear los registros de una tabla EMPRESA, y la entidad entidades.Empleado va a mapear los registros de una tabla EMPLEADO.

Cada unidad de persistencia se identifica por un nombre; en este ejemplo: AppTestPU. Este nombre será utilizado para crear un EntityManager.

Proveedores de persistencia.

Existen varias librerías que implementan JPA, siendo los proveedores más utilizados EclipseLink e Hibernate. Para utilizar el proveedor EclipseLink debemos incluir las siguientes dependencias:

```
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>eclipselink</artifactId>
  <version>4.0.0</version>
</dependency>
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>org.eclipse.persistence.json</artifactId>
  <version>4.0.0</version>
</dependency>
```

Con lo cual tendremos que usar los tipos del paquete «jakarta.persistence.*» en vez de «javax.persistence.*». Y el fichero de persistencia puede ser como sigue:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
      http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">
  <persistence-unit name="negocioPU">
    <!-- clases de entidad -->
    <class>entidades.Empesa</class>
    <class>entidades.Empleado</class>
```

```
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
      <!-- Propiedades de acceso JDBC -->
      <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver" />
      <property name="jakarta.persistence.jdbc.url" value="jdbc:h2:mem:negociodb" />
      <property name="jakarta.persistence.jdbc.user" value="sa" />
      <property name="jakarta.persistence.jdbc.password" value="" />
      <!-- Propiedades para generación de tablas a partir de clases de entidad -->
      <property name="eclipselink.ddl-generation" value="drop-and-create-tables" />
      <property name="eclipselink.ddl-generation.output-mode" value="database" />
    </properties>
  </persistence-unit>
</persistence>
```

Para utilizar el proveedor Hibernate debemos incluir las siguientes dependencias:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>5.6.14.Final</version>
</dependency>
```

Y el fichero de persistencia puede ser como sigue:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">
  <persistence-unit name="negocioPU">
    <!-- clases de entidad -->
    <class>entidades.Empesa</class>
    <class>entidades.Empleado</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
      <!-- Propiedades de acceso JDBC -->
      <property name="javax.persistence.jdbc.driver" value="org.h2.Driver" />
      <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:negociodb" />
      <property name="javax.persistence.jdbc.user" value="sa" />
      <property name="javax.persistence.jdbc.password" value="" />
      <!-- La clase de Hibernate para el dialecto de la base de datos H2-->
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />
      <!-- Propiedad para generación de tablas a partir de clases de entidad -->
      <property name="hibernate.hbm2ddl.auto" value="update" />
    </properties>
  </persistence-unit>
</persistence>
```

El proveedor Hibernate proporciona clases de tipo Dialect para saber convertir su lenguaje de consulta propio a consultas SQL. Esta clase debe ser especificada como una de las propiedades.

Gestión de transacciones.

Cuando se utiliza JPA en un servidor de aplicaciones, éste puede gestionar automáticamente transacciones. De hecho, JPA soporta dos tipos de gestión de transacciones:

1) RESOURCE_LOCAL: cuando las transacciones deben gestionarse explícitamente usando el EntityManager.
2) JTA: cuando las transacciones pueden ser gestionadas por la propia aplicación.

Aunque cada tipo de aplicación aplica el tipo de gestión adecuado, podemos explicitarlo en el nodo <persistence-unit>. Por ejemplo:

```
<persistence-unit name="negocioPU" transaction-type="RESOURCE_LOCAL">
```

2.1.2. Entidades de persistencia.

Una entidad de persistencia es un objeto Java Bean personalizado que permite acceder a los datos contenidos en los registros del origen de datos.

Supongamos que en la base de datos Negocio se han creado las siguientes tablas:

```
CREATE TABLE APP.EMPRESA (
  ID INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  NOMBRE VARCHAR(50) NOT NULL,
  FECHACREACION DATE NOT NULL
);
CREATE TABLE APP.EMPLEADO (
  ID INTEGER PRIMARY KEY,
  NOMBRE VARCHAR(50) NOT NULL,
  REFERENCIA CLOB,
  IDEMPRESA INTEGER REFERENCES EMPRESA(ID) NOT NULL
)
```

Cada empresa consta de uno o varios empleados y cada empleado pertenece a una única empresa. La clave primaria de Empresa es autogenerada, pero la de Empleado no.

En el caso de entidades relacionadas, las relaciones se reflejarán mediante colecciones. En este caso una empresa constará de una colección de empleados. En la forma más simple, las clases quedarían de la siguiente manera:

```
package entidades;
...
@Data
@NoArgsConstructor
@EqualsAndHashCode(of="id")
@Entity
public class Empresa implements Serializable {
  private static final long serialVersionUID=1L;
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Integer id;
  @Basic(optional=false)
  private String nombre;
  @Basic(optional=false)
  @Temporal(TemporalType.DATE)
  private Date fechacreacion;
  @OneToMany(cascade=CascadeType.ALL, mappedBy="empresa")
  private List<Empleado> empleadoList;

  public Empresa(Integer id, String nombre, Date fechacreacion) {
    this.id = id;
    this.nombre = nombre;
    this.fechacreacion = fechacreacion;
  }
}
```

Una clase anotada con @Entity se convierte en una entidad de persistencia. El mapeado de los campos de los registros y las propiedades de la entidad se realiza en tiempo de ejecución mediante las anotaciones @Table y @Column. La propiedad correspondiente a la clave primaria se marca con la anotación @Id.

En este caso, para reflejar la relación entre empresa y empleados se añade la propiedad empleadoList con la anotación @OneToMany.

Por su parte, la clase entidades.Empleado es así:

```
package entidades;
...
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(of = "id")
```

```
@Entity
public class Empleado implements Serializable {
  private static final long serialVersionUID=1L;
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Integer id;
  @Basic(optional=false)
  private String nombre;
  @Lob
  private String referencia;
  @JoinColumn(name="IDEMPRESA", referencedColumnName="ID", nullable=false)
  @ManyToOne(optional=false)
  private Empresa empresa;
}
```

Para reflejar la relación de un empleado con una empresa se añade la propiedad empresa anotada con @ManyToOne.

Una clase de entidad debe cumplir con los siguientes requerimientos:

- La clase debe estar anotada con la anotación javax.persistence.Entity.
- La clase debe tener un constructor sin argumentos público o protegido. La clase puede tener otros constructores.
- La clase no puede ser declarada final. Tampoco métodos o campos persistentes deben ser declarados final.
- Si las instancias de la clase son pasadas a través a métodos de beans de sesión remotos, la clase debe implementar la interfaz Serializable.
- Una clase de entidad puede extender a otra clase de entidad o de no entidad.

2.1.3. El administrador de entidades de persistencia.

Las entidades de persistencia son gestionadas por un EntityManager. Cada instancia de EntityManager se asocia con un contexto de persistencia, en el cual se crean, eliminan y modifican las instancias de las entidades.

Para nuestra aplicación de ejemplo, podemos crear un EntityManager de la siguiente manera:

```
EntityManager em = Persistence.createEntityManagerFactory("NegocioPU").createEntityManager();
// ... operaciones con el gestor de entidades ...
em.close();
```

Para obtener una instancia de EntityManager, primero se debe obtener una instancia de EntityManagerFactory. (En servidores de aplicaciones como JEE y Spring Boot se podrán crear objetos EntityManager usando anotaciones de inyección.)

Configuración de propiedades mediante un mapa.

El método createEntityManagerFactory() también permite especificar un mapa con las propiedades de persistencia. Este mapa complementará las propiedades del fichero «persistence.xml» o las sustituirán. Por ejemplo:

```
var props = Map.of(
    PersistenceUnitProperties.JDBC_DRIVER, "org.h2.Driver",
    PersistenceUnitProperties.JDBC_URL, "jdbc:h2:mem:negociodb",
    PersistenceUnitProperties.JDBC_USER, "sa",
    PersistenceUnitProperties.JDBC_PASSWORD, "");
var em = Persistence.createEntityManagerFactory("negocioPU", props).createEntityManager();
```

Uso de un «DataSource».

Podemos configurar la unidad de persistencia para que haga uso de un pool de conexiones.

Con EclipseLink podemos configurar el pool mediante propiedades. Primero debemos forzar el modo RESOURCE_LOCAL:

```
var props = Map.of(
  PersistenceUnitProperties.TRANSACTION_TYPE, PersistenceUnitTransactionType.RESOURCE_LOCAL.name(),
  PersistenceUnitProperties.JDBC_DRIVER, "org.h2.Driver",
  PersistenceUnitProperties.JDBC_URL,"jdbc:h2:mem:negociodb",
  PersistenceUnitProperties.JDBC_USER, "sa",
  PersistenceUnitProperties.JDBC_PASSWORD, "",
  PersistenceUnitProperties.CONNECTION_POOL_MIN, "1",
  PersistenceUnitProperties.DDL_GENERATION, "drop-and-create-tables",
```

```
        PersistenceUnitProperties.DEFAULT_DDL_GENERATION_MODE, "database");
    var em = Persistence.createEntityManagerFactory("negocioPU", props).createEntityManager();
```

Con Hibernate debemos realizar una configuración similar:

```
    var props = Map.of(
        "transaction-type", PersistenceUnitTransactionType.RESOURCE_LOCAL.name(),
        "javax.persistence.jdbc.driver", "org.h2.Driver",
        "javax.persistence.jdbc.url", "jdbc:h2:mem:negociodb",
        "javax.persistence.jdbc.user", "sa",
        "javax.persistence.jdbc.password", "",
        "hibernate.dialect", "org.hibernate.dialect.H2Dialect",
        "hibernate.hbm2ddl.auto", "update",
        "hibernate.c3p0.min_size", 1);
```

2.2. Trabajando con entidades de persistencia.

Se gestionan las entidades de persistencia invocando operaciones mediante un EntityManager. Cada EntityManager crea en memoria un contexto de persistencia que le permite realizar un seguimiento sobre lo que ocurre con cada instancia de entidad que gestiona.

De esta manera, cada instancia de una clase de entidad puede pasar por cuatro estados:

- ***Es nueva***. Las instancias nuevas no tienen identidad de persistencia y no están asociadas con ningún contexto de persistencia. Simplemente han sido instanciadas desde código asignándoles unos datos.
- ***Está asociada***. Las instancias asociadas están registradas en un contexto de persistencia y o bien están marcada como nuevas o bien se corresponden con registros de la base de datos. El EntityManager realizará un seguimiento de cambios sobre la instancia.
- ***Está desasociada***. Las instancias desasociadas se corresponden con registros de la base de datos, pero actualmente no están asociadas a un contexto de persistencia.
- ***Está eliminada***. Las instancias eliminadas tienen una identidad de persistencia (se corresponden con registros de la base de datos), están asociadas a un contexto de persistencia, y están marcadas para ser removidas de la base de datos.

2.2.1. Buscar entidades por clave.

El método EntityManager.find() permite recuperar una entidad almacenada a partir de su clave primaria. Por ejemplo, el siguiente método permite recuperar un objeto Empresa a partir de su clave:

```
    public Empresa buscarEmpresa(EntityManager em, int id) {
      return em.find(Empesa.class, id);
    }
```

Todas las instancias recuperadas mediante el método find() quedan asociadas al contexto de persistencia. De hecho, si la instancia buscada ya existe en el contexto de persistencia se recupera inmediatamente si realizar una consulta a la base de datos. Si la instancia no existe el método retorna null.

2.2.2. Agregar nuevas instancias de entidad.

Usaremos un EntityManager para agregar nuevos registros de EMPRESA usando instancias de la clase entidades.Empresa.

Se pueden crear nuevas entidades usando el método persist(). Pero la operación de persistencia debe realizarse en el contexto de una transacción si queremos trasladarla a la base de datos.

El siguiente método permite añadir una nueva empresa a la base de datos:

```
    public void addEmpresa(EntityManager em, Empresa empresa) {
      em.getTransaction().begin(); // Se inicia una transacción.
      em.persist(empresa); // Se asocia la empresa al contexto de persistencia como nueva
      em.getTransaction().commit(); // Se inserta el registro en la base de datos.
    }
```

El parámetro empresa se corresponde con una instancia nueva que todavía no está asociada al contexto de persistencia. Se inicia una transacción usando el método begin(). Esto hace que el contexto de persistencia haga un seguimiento sobre cualquier cambio que se realice sobre las entidades asociadas o las nuevas que se asocien. Después de asociar la entidad usando el método persist(), el método commit() detecta que en el contexto de persistencia existe una entidad nueva y la persiste creando el registro correspondiente en la base de datos. Si ya existe en la base de datos un registro con la misma clave el método commit() lanza una excepción.

Para hacer un seguimiento de cualquier otra operación debemos iniciar una nueva transacción.
Si en un caso dado, en vez de confirmar las operaciones registradas en el contexto de persistencia, queremos anularlas, podemos usar el método rollback().

```
try {
  em.getTransaction().begin();
  Empresa empresa = em.find(Empresa.class, 1);
  empresa.setFechacreacion(new Date());
  em.persist(empresa);
  em.getTransaction().commit();
} catch (Exception e) {
  em.getTransaction().rollback();
}
```

2.2.3. Asociar y desasociar instancias al contexto de persistencia.

Como se ha visto, si instanciamos un objeto Empresa, no quedará asociado al contexto de persistencia hasta que lo agreguemos con el método persist().
Si queremos asociar una nueva instancia de Empresa con un registro existente del almacén de datos debemos utilizar el método merge(). En el siguiente código se instancia una empresa con una clave existente y un nuevo nombre, y se asocia con la empresa existente en el contexto de persistencia.

```
...
Empresa e1 = new Empresa(1, "Nuevo nombre");
em.getTransaction().begin();
if ( ! em.contains(e1) ) {
  e1 = em.merge(e1);
}
em.getTransaction.commit(); // se persiste el nuevo nombre
```

Aunque no es necesario, antes de asociar la instancia de empresa se comprueba que no existe ya en el contexto con el método contains(). El método merge() lanzará una IllegalArgumentException si la instancia no es una entidad de persistencia o fue eliminada del contexto.
Se puede desasociar una instancia del contexto con el método detach(). Por ejemplo, el siguiente método desasocia una instancia de empresa si está asociada:

```
private void desasociarEntidad(EntityManager em, Empresa empresa) {
  try {
    if (em.contains(p))
      em.detach(p);
  } catch (Exception e) {
  }
}
```

2.2.4. Eliminar instancias de entidad.

Las instancias de entidad se eliminan directamente invocando al método remove(). El registro correspondiente a la entidad se eliminará de la base de datos cuando se complete la transacción actual.
El siguiente método elimina un Empleado:

```
private void eliminarEmpleado(EntityManager em, Empleado empleado) {
  if (em.contains(empleado)) {
    em.getTransaction().commit();
    em.remove(empleado);
    em.getTransaction().begin();
  }
}
```

Este método primero comprueba que la entidad empleado está asociada al contexto. Si no fuese así, el método remove() daría un error.

2.3. Consultas.

Además del método find() para buscar una entidad a través de su clave, JPA permite realizar consultas al estilo SQL, pero utilizando un lenguaje propio llamado JPQL. Antes de meternos a fondo con este lenguaje veremos las diversas formas de ejecutar consultas con JPA.

2.3.1. Ejecutando consultas con nombre.

Cuando se utiliza el asistente de NetBeans para crear clases de entidad agrega unas consultas a cada clase:

```
@Entity
@Table(name = "EMPLEADO")
@NamedQueries({
  @NamedQuery(name = "Empleado.findAll",
      query = "SELECT e FROM Empleado e")
  , @NamedQuery(name = "Empleado.findById",
      query = "SELECT e FROM Empleado e WHERE e.id = :id")
  , @NamedQuery(name = "Empleado.findByNombre",
      query = "SELECT e FROM Empleado e WHERE e.nombre = :nombre")})
public class Empleado implements Serializable {
  ...
}
```

La anotación @NamedQuery declara una consulta JPQL con nombre, que puede ser ejecutada usando el método EntityManager.createNamedQuery().

En cada anotación @NamedQuery se especifica un nombre de consulta (name) y la sentencia asociada (query). Cada nombre de consulta debe ser único para todas las clases de entidad.

La consulta "SELECT e FROM Empleado e" permite recuperar todas las entidades de empleado. Podemos ejecutarla de la siguiente manera:

```
EntityManager em = obtenerElEntityManager();
TypedQuery<Empleado> consulta = em
      .createNamedQuery("Empleado.findAll", Empleado.class);
List<Empleado> empleados = consulta.getResultList();
```

En este caso se utiliza el método getResultList() para retornar una lista con todos los empleados.

Si ejecutamos una consulta que retorne un único empleado, como "Empleado.findById", las instrucciones serían las siguientes:

```
TypedQuery<Empleado> consulta = em
  .createNamedQuery("Empleado.findById", Empleado.class);
Empleado empleado1 = consulta
  .setParameter("id", 1)
  .getSingleResult();
```

La consulta con nombre "Empleado.findById" define un parámetro :id, al cual debemos dar valor antes de resolver la consulta. En este caso se utiliza getSingleResult() para obtener un único resultado.

2.3.2. Ejecutando consultas sin nombre.

Se utiliza el método EntityManager.createQuery() para crear consultas sin nombre usando la sintaxis del lenguaje JPQL.

El siguiente ejemplo crea una consulta simple para obtener las empresas creadas a partir del año 2014:

```
EntityManager em = obtenerElEntityManager();
TypedQuery<Empresa> consulta = em
  .createQuery("select e from Empresa e where e.fechacreacion>='1/1/2014'", Empresa.class);
List<Empresa> empresas = consulta.getResultList();
```

Habrá ocasiones donde simplemente necesitemos recuperar datos concretos. Por ejemplo, para recuperar la fecha de creación de la empresa de id 3:

```
EntityManager em = obtenerElEntityManager();
TypedQuery<Date> consulta = em
  .createQuery("select e.fechacreacion from Empresa e where e.id=3", Date.class);
Date fecha = consulta.getSingleResult();
```

También podemos recuperar más de un dato por entidad. En este caso se retornará un array de objetos (cada celda contendrá uno de los datos solicitados) por cada entidad. Por ejemplo, podemos recuperar el nombre y el año de creación de cada empresa:

```
EntityManager em = obtenerElEntityManager();
Query consulta = em
  .createQuery("select e.nombre, FUNC('YEAR', e.fechacreacion) from Empresa e");
List<Object[]> resultados  = consulta.getResultList();
```

```
for (Object[] registro : resultados) {
  System.out.printf("Empresa: %s, creada en el año %d\n", registro[0], registro[1]);
}
```

2.3.3. Ejecutando consultas dinámicas.

El método createQuery() también permite ejecutar consultas sin usar JPQL creando un objeto CriteriaQuery. Los objetos CriteriaQuery son genéricos y permiten ir creando una consulta encadenando métodos. Dada su complejidad veremos sólo un ejemplo sencillo para recuperar los empleados ordenados por su nombre:

```
EntityManager em = obtenerElEntityManager();
// Primero obtenemos un fabricador de consultas
CriteriaBuilder builder = em.getCriteriaBuilder();
// Creamos la consulta para retornar objetos Empleado
CriteriaQuery<Empleado> criteria = builder.createQuery(Empleado.class);
// Obtiene la raíz de la consulta para entidades Empleado
Root<Empleado> root = criteria.from(Empleado.class);
// Filtramos los empleados con nombres nulos.
criteria.where(builder.isNotNull(root.get("nombre")));
// Aplica ordenación ascendente por nombre
criteria.orderBy(builder.asc(root.get("nombre")));
// Crea una consulta tipada a partir del criterio dinámico
TypedQuery<Empleado> consulta = em.createQuery(criteria);
// Obtiene el resultado de la consulta
List<Empleado> empleadosOrdeandos = consulta.getResultList();
```

Este tipo de consultas se construyen combinando objetos CriteriaBuilder, CriteryQuery y Root.

2.3.4. Métodos para filtrar la consulta.

Además de las condiciones que establezcamos en la consulta, los objetos Query proporcionan métodos para filtrar los resultados de esta. Estos métodos retornan la propia consulta y pueden encadenarse entre ellos.

- setFirstResult(int posicionInicial), determina la posición del primer resultado que se recuperará. Las posiciones se numeran desde cero.
- setMaxResults(int maxResultados), determina el máximo número de resultados que se recuperarán.

Por ejemplo, la siguiente consulta retorna sólo cinco empresas cuya fecha sea previa a la actual:

```
Query consulta = em
  .createQuery("SELECT e FROM Empresa e WHERE e.fechacreacion < :fecha")
  .setParameter("fecha", new java.util.Date())
  .setMaxResults(5);
```

2.3.5. Persistencia de cambios en los datos de las entidades.

Como se ha visto, con las inserciones y borrados, un commit() confirma la operación realizada en el contexto de la transacción actual. Pero esa confirmación afecta también a los cambios realizados en las propiedades de las entidades de persistencia recuperadas mediante una consulta.

Por tanto, cada vez que se invoque el método commit() se trasladará a la base de datos cualquier cambio en las propiedades de las entidades asociadas con el contexto actual.

```
em.getTransaction().commit();
// Se confirman inserciones, borrados y actualizaciones pendientes.
em.getTransaction().begin();
// Se inicia una nueva transacción
```

Pero también podemos persistir cambios usando el método flush(), aunque bajo ciertas condiciones. Por ejemplo:

```
em.setFlushMode(FlushModeType.AUTO); // Valor por defecto
em.getTransaction().begin();
Empleado e1 = em.find(Empleado.class, 1);
e1.setNombre("Nuevo nombre");
em.flush();
```

En este caso, el método flush() forzará la sincronización de las entidades de persistencia con los registros de la base de datos. Pero si tras ejecutar este código consultamos directamente la base de datos veremos que aún no ha cambiado el nombre. Sin embargo, cualquier consulta en código que hagamos sobre la entidad de empleado cambiada sí reflejará el cambio:

```
Object consultaNombre = em
```

```
            .createQuery("select e.nombre from Empleado e where e.id=1")
            .getSingleResult();
    System.out.println(consultaNombre);
    // Se imprime: Nuevo nombre
```

Hasta que no hagamos un commit() no se confirmarán físicamente los cambios en la base de datos.

2.4. Características de las clases de entidad.

En este capítulo desarrollaremos algunas de las características de JPA a la hora de definir campos, generar claves automáticamente, uso de campos LOB, etc.

2.4.1. Campos y propiedades en la clase de entidad.

El estado de un objeto persistente se establece mediante los valores en sus campos o en sus propiedades. Java Persistence admite campos y propiedades de los tipos:

- Tipos primitivos de Java: int, long, double, char, etc.
- java.lang.String
- Tipos serializables.
- Tipos enumerados
- Colecciones de entidad
- Clases embebidas.

Se puede utilizar una combinación de campos y propiedades para persistir el estado de una entidad. Para ello se deben aplicar las anotaciones adecuadas sobre el campo o sobre el método getter de la propiedad.

2.4.2. Campos y propiedades persistentes.

Los campos no estáticos que no estén anotados con javax.persistence.Transient o con transient son considerados persistentes y JPA intentará mapearlos con columnas de la base de datos. Por ejemplo, en la clase Empresa del ejemplo previo:

```
@Entity
@Table(name = "EMPRESA")
public class Empresa implements Serializable {
  @Transient
  private PropertyChangeSupport changeSupport = new PropertyChangeSupport(this);
  private static final long serialVersionUID = 1L;
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Basic(optional = false)
  @Column(name = "ID", nullable = false)
  private Integer id;
  ...
}
```

El campo changeSupport, al estar anotado con @Transient no es considerado persistente. El campo serialVersionUID, al ser estático tampoco es considerado persistente. El campo id será considerado persistente y la anotación @Column(name="ID") lo asocia con la columna ID de la tabla EMPRESA.

En el caso de las propiedades debemos aplicar las anotaciones sobre el método getter (si es una propiedad de tipo boolean su nombre empezará con is, en los demás casos con get).

2.4.3. Campos enumerados.

Java Persistence permite mapear valores de columnas de tipo entero y de tipo texto con tipos enumerados de Java. Por ejemplo, supongamos la siguiente tabla:

```
CREATE TABLE CUENTA (
  ID INTEGER PRIMARY KEY,
  TIPO NUMERIC(2) CHECK (TIPO>=1 AND TIPO<=4),
);
```

La tabla CUENTA define una columna TIPO que sólo puede tomar valores entre 1 y 4. Estos valores van a determinar el tipo de la cuenta. Cuando hagamos el mapeado de la tabla CUENTA con la clase de entidad Cuenta puede interesarnos darles un significado más explícito a estos valores haciéndolos corresponder con una enumeración:

```
package entity;
import java.io.Serializable;
import javax.persistence.*;
```

```
@Entity
@Table(name = "CUENTA")
public class Cuenta implements Serializable {
  public enum TipoCuenta {Ahorro, Personal, Familiar, Empresarial}
  @Id
  @Column(name = "ID")
  private Integer id;
  @Enumerated(EnumType.ORDINAL)
  @Column(name = "TIPO")
  private TipoCuenta tipo;
  ...
}
```

En este caso se ha definido una enumeración TipoCuenta interna y se ha aplicado la anotación @Enumerated sobre el campo tipo. El valor EnumType.ORDINAL le dice a JPA que debe hacer corresponder el valor numérico de la columna TIPO con un valor de la enumeración; siendo el criterio por seguir que la primera constante de la enumeración se corresponde con el valor 1, la siguiente con el valor 2, etc.

También es posible mapear una columna de texto con valores de una enumeración. En este caso se utiliza la anotación:

```
@Enumerated(EnumType.STRING)
```

Y debe ocurrir que el valor en la columna se corresponda exactamente con un nombre de la constante de la enumeración. Si no es así, se lanzará una excepción.

2.4.4. Campos Lob.

Un campo Lob es un objeto grande. Las columnas Lob se usan para almacenar textos muy grandes y ficheros binarios. Hay dos clases de Lob: CLOB y BLOB. El primero es un lob de caracteres y se puede usar para almacenar texto; es una alternativa al tipo VARCHAR, el cual tiene limitaciones de tamaño. El segundo es un lob binario y se puede usar para almacenar ficheros binarios.

Por ejemplo, en la siguiente tabla LIBRO, se utiliza el campo CONTENIDO para almacenar todo el contenido del libro, y el campo PORTADA para almacenar la imagen de portada:

```
CREATE TABLE LIBRO (
  ID INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  TITULO VARCHAR(150),
  CONTENIDO CLOB,
  PORTADA BLOB
);
```

El mapeado de esta tabla se corresponderá con la siguiente clase de entidad:

```
package entity;
import java.io.Serializable;
import javax.persistence.*;
import javax.validation.constraints.Size;
@Entity
@Table(name = "LIBRO")
public class Libro implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Basic(optional = false)
  @Column(name = "ID")
  private Integer id;
  @Size(max = 150)
  @Column(name = "TITULO")
  private String titulo;
  @Lob
  @Column(name = "CONTENIDO")
  private String contenido;
  @Lob
  @Column(name = "PORTADA")
```

```
    private Serializable portada;
    ...
}
```

Ambos campos, contenido y portada quedan anotados con @Lob, pero uno se define de tipo String y otro de tipo Serializable. Los strings de Java admiten contenidos muy grandes y no tienen problemas para mapearse con un CLOB e incluso con un BLOB. Aunque el campo portada sea definido como un Serializable, en realidad no hay problema en mapearlo como un array de bytes, un string o un java.sql.Blob:

```
@Lob
@Column(name = "PORTADA")
private byte[] portada;
```

O bien:

```
@Lob
@Column(name = "PORTADA")
private java.sql.Blob portada;
```

2.4.5. Claves primarias en entidades.

Cada entidad debe tener un único valor identificador. Una entidad personalizada tendrá normalmente un identificador de tipo numérico. Este identificador único o clave primaria permite a las aplicaciones cliente localizar una instancia de entidad concreta. Cada entidad de persistencia debe tener obligatoriamente una clave primaria, que puede ser simple o compuesta.

Las claves primarias simples se anotan con @Id, bien sobre el campo o bien sobre el método getter.

Las claves primarias compuestas incluyen más de un campo. Por eso motivo se deben definir en una clase de clave primaria, y deben ser anotadas con @IdClass.

Supongamos que queremos crear una tabla COMENTARIO que contendrá registros con comentarios sobre un empleado concreto. Cada comentario tendrá un número de orden y un texto de comentario. Como clave principal de la tabla COMENTARIO usaremos el id de empleado y el número de orden. La tabla se creará en la base de datos con el siguiente comando:

```
CREATE TABLE COMENTARIO (
  idEmpresa INTEGER REFERENCES EMPRESA(id),
  orden INTEGER,
  texto VARCHAR(120),
  PRIMARY KEY(idEmpresa, orden)
);
```

Ahora agregaremos una entidad de persistencia para la tabla COMENTARIO en nuestro proyecto.

```
@Data
@Embeddable
public class ComentarioPK implements Serializable {
  @Basic(optional = false)
  @Column(name = "IDEMPRESA", nullable = false)
  private int idempresa;
  @Basic(optional = false)
  @Column(name = "ORDEN", nullable = false)
  private int orden;
}

@Data
@Entity
@Table(name = "COMENTARIO")
public class Comentario implements Serializable {
  private static final long serialVersionUID = 1L;
  @EmbeddedId
  protected ComentarioPK comentarioPK;
  ... ... ...
}
```

La clase ComentarioPK se corresponde con la clave primaria compuesta de la entidad Comentario, y como puede verse define un atributo que referencia el id de empresa y el orden del comentario. Es definida como una clase

embebida con la anotación @Embeddable. Sus campos se mapean con la correspondiente columna de la tabla mediante la anotación @Basic, donde se especifica optional=false para indicar que no admiten valores nulos.
En la clase Comentario, en este caso se indica la clave primaria compuesta con la anotación @EmbeddedId. Pero también podemos mapear los campos de la clave compuesta por separado de la siguiente manera:

```
@Entity
@IdClass(ComentarioPK.class)
public class Comentario implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id
  protected int idempresa;
  @Id
  protected int orden;
  ... ... ...
}
```

En este caso se especifica la clase de clave compuesta con la anotación @IdClass, y se declaran campos idempresa y orden anotados con @Id con sus correspondientes métodos accesores. Es importante que los nombres de estos campos coincidan con los de la clase de la clave compuesta.

2.4.6. Generación automática de claves.

Muchas bases de datos soportan la auto-generación de valores para claves primarias de tipo entero. Cada base de datos tiene su manera de generar valores para la clave, bien mediante campos de identidad o mediante generadores de secuencia. Al crear la clase de entidad podemos especificar si la clave primaria (si es entera) puede ser generada desde una secuencia, si se auto-genera o si se debe insertar desde una tabla.
La anotación javax.persistence.GeneratedValue permite especificar una estrategia para generar los valores de la clave. Se pueden dar tres estrategias:

• GenerationType.IDENTITY

Algunas bases de datos como MySQL o Microsoft SQL Server permiten la generación de campos de tipo identidad. Estos campos generan un valor automático para la clave cuando se inserta un nuevo registro. Por ejemplo, supongamos la siguiente tabla creada con MySQL:

```
CREATE TABLE USUARIO (
  USUARIO_ID BIGINT NOT NULL AUTO_INCREMENT,
  NOMBRE VARCHAR(255) NOT NULL,
  PRIMARY KEY(USUARIO_ID)
);
```

La cláusula AUTO_INCREMENT le dice a la base de datos que debe generar un valor para la clave USUARIO_ID. El mapeado con la clase de entidad Usuario debe declarar esta estrategia de la siguiente manera:

```
@Entity
@Table( name = "USUARIO")
public class Usuario implements Serializable {
  @Id
  @GeneratedValue( strategy = GenerationType.IDENTITY )
  @Column(name = "USUARIO_ID")
  private Long usuarioId;
  ... ... ...
}
```

Las bases de datos que soportan esta estrategia son: Java Derby, MySQL, Microsoft SQL Server e IBM DB2 ver 7.1 posterior.

• GenerationType.SEQUENCE

Bases de datos como Oracle DB utilizan generadores de secuencias para asignar valores a claves primarias. Por ejemplo, supongamos la siguiente tabla y secuencia creadas en Oracle DB:

```
CREATE TABLE USUARIO (
  USUARIO_ID NUMBER(7) NOT NULL,
  NOMBRE VARCHAR2(255) NOT NULL,
  PRIMARY KEY(USUARIO_ID)
);
CREATE SEQUENCE USUARIOS_SEQ START WITH 1 INCREMENT BY 1 NOCACHE NOCYCLE;
```

La secuencia USUARIOS_SEQ permitirá obtener números secuenciales desde el valor 1. Ahora podemos especificar en la clase de entidad Usuario que utilice dicha secuencia para asignar automáticamente la clave primaria cada vez que se persistan objetos Usuario:

```
@Entity
@Table( name = "USUARIO")
public class Usuario implements Serializable {
  @Id
  @SequenceGenerator(name="usuarioSeq", sequenceName="USUARIOS_SEQ",
      allocationSize=1, initialValue=1)
  @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="usuarioSeq" )
  @Column( name = "USUARIO_ID" )
  private Long usuarioId;
  ... ... ...
}
```

La anotación @SequenceGenerator define una secuencia de la base de datos y la identifica con un nombre (atributo name). En la anotación @GeneratedValue se hace referencia a este nombre en el atributo generator.

• GenerationType.TABLE

Las dos estrategias anteriores tienen el inconveniente de que son dependientes de una base de datos concreta. En esta estrategia se utiliza una tabla auxiliar que mantendrá el valor del último valor de una clave generada por JPA. Esta tabla auxiliar debe disponer de dos columnas: un nombre de secuencia de tipo string, y el valor actual de la secuencia de tipo entero. La siguiente tabla puede ser un ejemplo:

```
CREATE TABLE ALMACEN_SECUENCIAS (
  NOMBRE_SECUENCIA VARCHAR(255) PRIMARY KEY,
  VALOR_SECUENCIA INTEGER NOT NULL
);
INSERT INTO ALMACEN_SECUENCIAS VALUES ('UsuarioPK', 0);
```

Se ha añadido a la tabla ALMACEN_SECUENCIAS un registro que identifica una secuencia llamada UsuarioPK con el valor inicial cero. Si queremos definir en esta tabla otras secuencias basta con añadir un nuevo registro asegurando que el nombre de secuencia será único. Ahora podemos mapear la entidad Usuario de la siguiente manera:

```
@Entity
@Table( name = "USUARIO")
public class Usuario implements Serializable {
  @Id
  @TableGenerator(name="usuarioStore", table="ALMACEN_SECUENCIAS",
      pkColumnName="NOMBRE_SECUENCIA", pkColumnValue="UsuarioPK",
      valueColumnName="VALOR_SECUENCIA", initialValue=1, allocationSize=1)
  @GeneratedValue(strategy=GenerationType.TABLE, generator="usuarioStore")
  @Column( name = "USUARIO_ID" )
  private Long usuarioId;
  ... ... ...
}
```

Primero se especifica en la anotación @TableGenerator la tabla auxiliar y cómo se llaman sus columnas. En el atributo pkColumnValue se especifica el nombre de la secuencia (y por tanto de qué registro de la tabla auxiliar) se extraerán los valores, y la identifica con un nombre (atributo name). En la anotación @GeneratedValue se hace referencia a este nombre en el atributo generator.

2.4.7. Clases embebidas en entidades.

Se utilizan clases embebidas para representar el estado de una entidad, pero no tienen una identidad de persistencia por sí mismas. Las instancias de una clase embebida comparten la identidad de su entidad propietaria, y por tanto sólo existen como parte del estado de otra entidad.

Las clases embebidas tienen las mismas reglas que una clase de entidad pero están anotadas con javax.persistence.Embeddable en vez de la anotación @Entity.

Por ejemplo, podemos necesitar incluir en la tabla EMPLEADO su dirección, como una localidad y una calle. Para ello añadiremos dos nuevas columnas a esta tabla: LOCALIDAD será el nombre de la ciudad o pueblo donde vive, y DIRECCION su dirección:

```
ALTER TABLE EMPLEADO ADD LOCALIDAD VARCHAR(120);
ALTER TABLE EMPLEADO ADD DIRECCION VARCHAR(250);
```

Si deseamos encapsular los datos de la dirección podemos crear una nueva clase embebida:

```
package entidades;
import javax.persistence.Column;
import javax.persistence.Embeddable;
@Embeddable
public class Direccion implements java.io.Serializable {
  @Column(name="LOCALIDAD")
  private String localidad;
  @Column(name="DIRECCION")
  private String direccion;
  // Getters y setters ...
}
```

Esta clase embebida se puede utilizar en la clase Empleado de la siguiente manera:

```
@Entity
public class Empleado implements java.io.Serializable {
  @Embedded
  private Direccion direccion;
  ... ... ...
}
```

A veces se puede reutilizar una clase embebida en varias clases de entidad correspondientes a diferentes tablas. Si las tablas tienen nombres de columnas diferentes podemos configurar el mapeado de las columnas de la tabla con las propiedades de la clase embebida en la clase de entidad:

```
@Entity
public class Empleado implements java.io.Serializable {
  @Embedded
  @AttributeOverrides({
    @AttributeOverride(name="localidad", column=@Column(name="LOCALIDAD")),
    @AttributeOverride(name="direccion", column=@Column(name="DIRECCION"))
  })
  private Direccion direccion;
  ... ... ...
}
```

2.5. Relaciones entre entidades de persistencia

Cuando una base de datos se compone de varias tablas, lo habitual es que existan relaciones entre las mismas. Realizar consultas para recuperar los registros de una tabla junto con sus datos relacionados implica ejecutar consultas de "join" entre tablas. Este tipo de consultas pueden ser más o menos complejas dependiendo del tipo de relación entre tablas.

2.5.1. Relaciones de multiplicidad entre entidades.

El API Persistence simplifica el acceso a los datos de entidades relacionadas mediante el uso de propiedades de navegación en cada entidad. Habitualmente la entidad principal poseerá una propiedad de tipo colección que contendrá las instancias de la entidad relacionada, y la entidad secundaria poseerá una propiedad que referencia la instancia de la entidad principal.

Si analizamos el proyecto de ejemplo de la base de datos NEGOCIO, podemos comprobar que la entidad Empresa posee una propiedad que referencia sus empleados:

```
@Entity
@Table(name = "EMPRESA")
public class Empresa implements Serializable {
  ... ... ...
  @OneToMany(cascade = CascadeType.ALL, mappedBy = "empresa")
  private List<Empleado> empleadoList;

  public List<Empleado> getEmpleadoList() {
    return empleadoList;
```

```
    }
    public void setEmpleadoList(List<Empleado> empleadoList) {
      this.empleadoList = empleadoList;
    }
    ... ... ...
  }
```

La propiedad empleadoList es una propiedad de navegación que permite acceder a los registros de la tabla Empleado asociados con el registro de Empresa actual.
Por su parte, en la clase de entidad Empleado podemos comprobar que se ha creado una propiedad de navegación empresa, la cual referencia el objeto Empresa asociado al empleado:

```
@Entity
@Table(name = "EMPLEADO")
public class Empleado implements Serializable {
  ... ... ...
  @JoinColumn(name = "IDEMPRESA", referencedColumnName = "ID",
      nullable = false)
  @ManyToOne(optional = false)
  private Empresa empresa;

  public Empresa getEmpresa() {
    return empresa;
  }
  public void setEmpresa(Empresa empresa) {
    this.empresa = empresa;
  }
  ... ... ...
}
```

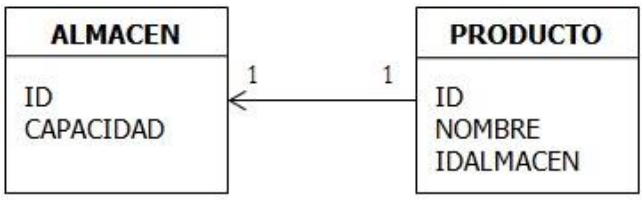
Java Persistence permite los siguientes tipos de relación entre entidades:

- **Uno-a-uno**: Cada instancia de entidad se relaciona sólo con una instancia de la entidad relacionada.
- **Uno-a-varios**. Una instancia de entidad puede relacionarse con varias instancias de otra entidad relacionada.
- **Varios-a-uno**. Varias instancias de entidad se pueden relacionar con una misma instancia de otra entidad relacionada.
- **Varios-a-varios**. Las instancias de una entidad pueden relacionarse con varias instancias de la otra entidad y viceversa.

La dirección entre una relación puede ser unidireccional o bidireccional. Una relación bidireccional implica una propiedad de navegación en cada una de las entidades, mientras que una relación unidireccional implica sólo una propiedad de navegación en una de las entidades.

2.5.2. Relaciones uno-a-uno.

En una relación uno-a-uno, cada instancia de entidad se relaciona sólo con una instancia de la entidad relacionada. Por ejemplo, en una base de datos de gestión de almacenes puede darse el caso de que cada almacén contiene un único producto, y cada producto se puede almacenar en único almacén:



Relación 1 a 1

En este caso la tabla PRODUCTO referencia a su ALMACEN mediante una clave foránea IDALMACEN. Las relaciones uno-a-uno utilizan la anotación javax.persistence.OneToOne sobre la correspondiente propiedad de navegación.

```
@Entity
@Table(name = "ALMACEN")
public class Almacen implements Serializable {
  @Id
  @Column(name = "ID")
  private int id;
  @Column(name = "CAPACIDAD")
  private int capacidad;
  ... ... ...
}
```

```
@Entity
@Table(name = "PRODUCTO")
public class Producto implements Serializable {
  @Id
  @Column(name = "ID")
  private int id;
  @Column(name = "NOMBRE")
  private String nombre;
  @OneToOne(opcional=false)
  private Almacen almacen;
  ... ... ...
}
```

En este ejemplo se establece una relación unidireccional, ya que desde un producto podemos referenciar su almacén, pero desde un almacén no se puede referenciar su producto. La anotación @OneToOne admite los siguientes atributos:

- optional, establece si obligatoriamente un producto debe referenciar una instancia de almacén (valor false), o si permite que la propiedad de navegación pueda tomar el valor nulo (valor true) al ser persistido.
- cascade, indica cómo se reflejan los cambios sobre esta entidad en la entidad relacionada. Posibles valores son:
  - CascadeType.PERSIST. Provoca que si creamos un nuevo producto y lo asociamos con un nuevo almacén, al persistir el producto se persistirá automáticamente el almacén.
  - CascadeType.REMOVE. Provoca que si eliminamos un producto también se eliminará su almacén asociado.
  - CascadeType.MERGE. Provoca que si asociamos un objeto producto con un producto existente en el contexto de persistencia, también se asociarán los objetos almacén correspondientes.
  - CascadeType.DETACH. Provoca que si desasociamos un objeto producto del contexto de persistencia también se desasocie su objeto almacén asociado.
  - CascadeType.REFRESH. Provoca que si actualizamos un objeto producto desde la base de datos también se actualice su objeto almacén asociado.
  - CascadeType.ALL. Aplica todas las opciones.
- mappedBy, se utiliza en relaciones bidireccionales para asociar la propiedad correspondiente en la instancia de la entidad relacionada.
- fetch, establece la estrategia de recuperación de los datos relacionados. Posibles valores son:
  - FetchType.EAGER, es la estrategia inmediata e indica que cuando se recupera un producto se recuperan también inmediatamente los datos de su almacén.
  - FetchType.LAZY, es la estrategia perezosa e indica que si se recupera un producto no se recuperan los datos de su almacén asociado hasta que se intente acceder a dichos datos mediante la propiedad de navegación.
- orphanRemoval, indica si se deben o no eliminar los almacenes que dejen de ser referenciados desde un producto.
- targetEntity, establece el tipo de la clase relacionada.

Junto con la anotación @OneToOne también se puede utilizar la anotación @JoinColumn o @JoinColumns para especificar cuál es el campo de la tabla que actúa de clave foránea o clave foránea compuesta, respectivamente.

```
@OneToOne(opcional=false)
@JoinColumn(name="IDALMACEN", referencedColumnName="ID")
private Almacen almacen;
```

Si deseamos crear una relación bidireccional basta con añadir a la entidad Almacen una propiedad que haga referencia a la entidad Producto y anotarla con el atributo @OneToOne:

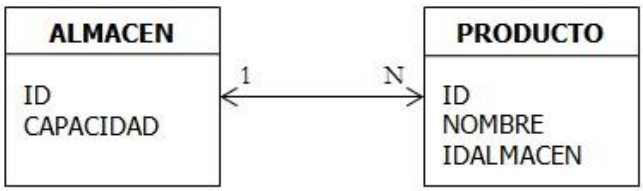
```
@Entity
@Table(name = "ALMACEN")
public class Almacen implements Serializable {
  @Id
  @Column(name = "ID")
  private int id;
  @Column(name = "CAPACIDAD")
  private int capacidad;
  @OneToOne(opcional=false, mappedBy="almacen")
  private Producto producto;
  ... ... ...
}
```

```
@Entity
@Table(name = "PRODUCTO")
public class Producto implements Serializable {
  @Id
  @Column(name = "ID")
  private int id;
  @Column(name = "NOMBRE")
  private String nombre;
  @OneToOne(opcional=false)
  private Almacen almacen;
  ... ... ...
}
```

El lado inverso de la relación (en este caso Almacen) se refiere al lado propietario usando el atributo mappedBy de @OneToOne.

2.5.3. Relaciones uno-a-varios y varios-a-uno.

Trataremos a la vez ambos tipos de relaciones puesto que son la misma dependiendo del punto de vista de la relación. Por ejemplo, puede ocurrir que la base de datos de gestión de almacenes permita que en un mismo almacén se puedan guardar varios productos, pero cada producto sólo puede almacenarse en un único almacén. Desde el punto de vista de Almacen tiene una relación uno-a-varios con Producto. Pero desde el punto de vista de Producto tiene una relación varios-a-uno con Almacen.



Relación 1 a varios

Las relaciones uno-a-varios utilizan la anotación javax.persistence.OneToMany sobre la correspondiente propiedad de navegación, mientras que las relaciones varios-a-uno utilizan la anotación javax.persistence.ManyToOne sobre la correspondiente propiedad de navegación:

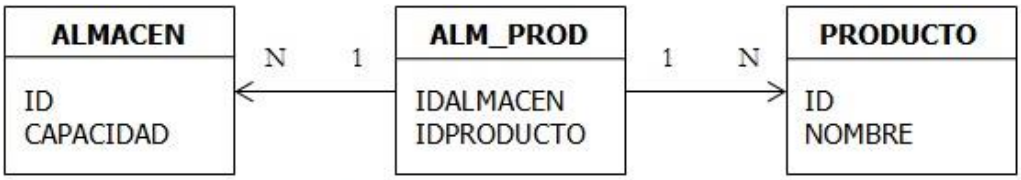
```
@Entity
@Table(name = "ALMACEN")
public class Almacen implements Serializable {
  @Id
  @Column(name = "ID")
  private int id;
  @Column(name = "CAPACIDAD")
  private int capacidad;
  @OneToMany(cascade=CascadeType.ALL,
      mappedBy="almacen")
  private List<Producto> productoList =
      new ArrayList<>(0);
  ... ... ...
}
```

```
@Entity
@Table(name = "PRODUCTO")
public class Producto implements Serializable {
  @Id
  @Column(name = "ID")
  private int id;
  @Column(name = "NOMBRE")
  private String nombre;
  @ManyToOne(optional = false)
  @JoinColumn(name="IDALMACEN",
      referencedColumnName="ID")
  private Almacen almacen;
  ... ... ...
}
```

La anotación @OneToMany tiene los mismos atributos que @OneToOne excepto optional. Por su parte, la anotación @ManyToOne sólo tiene los atributos cascade, fecth, optional y targetEntity.

2.5.4. Relaciones varios-a-varios.

Se dan relaciones varios-a-varios cuando las instancias de dos entidades se pueden relacionar entre ellas sin restricciones. Por ejemplo, puede ocurrir que la base de datos de gestión de almacenes permita que en un mismo almacén se puedan guardar varios productos, y que un producto se pueda almacenar en varios almacenes:



Relación varios a varios

En los modelos relacionales, una relación varios-a-varios implica siempre una tabla de relación entre las dos entidades relacionadas. Esta tabla de relación hereda las claves de las tablas relacionadas. Java Persistence hace que esta tabla de relación sea transparente.

Las relaciones varios-a-varios utilizan la anotación javax.persistence.ManyToMany sobre la correspondiente propiedad de navegación.

```
@Entity
@Table(name = "ALMACEN")
public class Almacen implements Serializable {
  @Id
  @Column(name = "ID")
  private int id;
  @Column(name = "CAPACIDAD")
  private int capacidad;
  @ManyToMany
  @JoinTable( name="ALM_PROD",
    joinColumns={
      @JoinColumn(name="IDALMACEN",
      referencedColumnName="ID")
    }, inverseJoinColumns={
      @joinColumn(name=""IDPRODUCTO,
        referencedColumnName="ID")
    } )
  private List<Producto> productoList =
    new ArrayList<>(0);
  ... ... ...
}
```

```
@Entity
@Table(name = "PRODUCTO")
public class Producto implements Serializable {
  @Id
  @Column(name = "ID")
  private int id;
  @Column(name = "NOMBRE")
  private String nombre;
  @ManyToMany(mappedBy="productoList")
  private List<Almacen> almacenList =
    new ArrayList<>(0);
  ... ... ...
}
```

2.5.5. Estrategias de actualización y recuperación de entidades relacionadas.

Java Persistence permite simplificar las operaciones entre entidades relacionadas de dos maneras: trasladando operaciones en cascada y aplicando dos estrategias de recuperación de datos relacionados.

Operaciones en cascada.

En el caso de tablas relacionadas podemos configurar el atributo cascade en el nodo que define la colección que contiene los objetos relacionados. Este atributo permite determinar cómo afecta una operación de actualización realizada sobre una entidad en sus entidades relacionadas.

Siguiendo con el ejemplo de las clases de entidad Empresa y Empleado:

```
public class Empresa implements Serializable {
  ... ... ...
  @OneToMany(cascade = CascadeType.ALL, mappedBy = "empresa")
  private List<Empleado> empleadoList = new ArrayList<>(0);
  ... ... ...
}
```

La asignación cascade=CascadeType.ALL es equivalente a asignar en este atributo un array con los siguientes valores:

- CascadeType.DETACH, si la entidad empresa se desasocia del contexto también lo harán los empleados contenidos en su colección de navegación.
- CascadeType.MERGE, si la entidad empresa se mezcla con el contexto también lo harán los empleados contenidos en su colección de navegación.
- CascadeType.PERSIST, si una nueva entidad empresa se persiste en el contexto también lo harán los empleados nuevos añadidos a su colección de navegación.

- CascadeType.REFRESH, si una entidad empresa se refresca desde la base de datos también lo harán los empleados contenidos en su colección de navegación.
- CascadeType.REMOVE, si una entidad empresa se elimina del contexto también se eliminarán sus empleados relacionados.

Por tanto, este atributo permite realizar cambios en la colección de las entidades relacionadas a través de la propiedad de navegación de la entidad principal y estos cambios se trasladarán en cadena.
Por ejemplo, supongamos que queremos añadir un nuevo empleado a la empresa de código 1. Primero recuperamos la entidad empresa:

```
EntityManager em = obtenerEntityManager();
Empresa empresa1 = em.find(Empresa.class, 1);
```

Y ahora creamos y persistimos el nuevo empleado:

```
em.getTransaction().begin();
Empleado nuevo1 = new Empleado(12, "Empleado12");
nuevo1.setReferencia("Referencia del empleado12");
nuevo1.setEmpresa(empresa1); // Esta asignación es necesaria
em.persist(nuevo1);
em.getTransaction().commit();
```

Pero gracias a la configuración en cascada también podemos persistir el empleado a través de su empresa:

```
em.getTransaction().begin();
Empresa empresa1 = em.find(Empresa.class, 1);
Empleado nuevo1 = new Empleado(12, "Empleado12");
nuevo1.setReferencia("Referencia del empleado12");
nuevo1.setEmpresa(empresa1); // Esta asignación es necesaria
empresa1.getEmpleadoList().add(nuevo1);
em.getTransaction().commit();
```

Si realizamos una operación a través de la propiedad de navegación, y no la hemos habilitado en el atributo cascade se producirá una excepción al realizar un commit().

Estrategias de carga de datos.

Cuando recuperamos datos de una entidad, mediante el método find() o una consulta, JPA utiliza por defecto una estrategia perezosa (lazy) de recuperación de datos relacionados. En el ejemplo de empresa y empleados esto es equivalente a:

```
@OneToMany(fetch = FetchType.LAZY, mappedBy = "empresa")
private List<Empleado> empleadoList = new ArrayList<>();
```

La estrategia perezosa implica que si recuperamos los datos de una empresa su lista de empleados no se recupera al momento, sino que se recupera bajo demanda, cuando necesitemos acceder a ellos. Por el contrario, con la estrategia inmediata (fetch = FetchType.EAGER) la colección de empleados se recupera al momento.
Podemos ver esto evaluando el tipo de la lista devuelta por la propiedad empleadoList:

```
Empresa empresa1 = em.find(Empresa.class, 1);
System.out.println(empresa1.getEmpleadoList().getClass());
```

Con estrategia perezosa se imprime:

```
class org.eclipse.persistence.internal.indirection.jdk8.IndirectList
```

Una IndirectList es una colección de tipo proxy, que no contiene directamente los objetos Empleado, sino que contiene los algoritmos necesarios para acceder a los datos bajo demanda. Esto significa que con la estrategia perezosa los datos sólo serán consultados, y se cargarán en la memoria, cuando realmente los necesitemos en tiempo de ejecución.
Con esta estrategia perezosa, a todos los efectos, podremos trabajar con la colección empleadoList como si contuviese a todos los empleados de la empresa, pero en realidad no es así, ya que realiza las operaciones de base de datos necesarias sólo cuando hagan falta. Por tanto, esta estrategia es ideal cuando trabajamos solamente con un proceso local y no debemos transmitir entidades entre procesos.
En el caso de la estrategia inmediata se imprime:

```
class java.util.Vector
```

En este caso, un Vector contendrá directamente los objetos Empleado. Esta estrategia será necesaria si necesitamos transmitir un objeto Empresa con sus empleados a otros procesos través de comunicaciones de red.

2.5.6. Herencia en entidades de persistencia.

Las entidades de persistencia soportan herencia de clases, asociaciones polimórficas y consultas polimórficas. Recordemos que el polimorfismo es el mecanismo por el cual podemos gestionar objetos de una clase mediante variables de una superclase. Las clases de entidad pueden extender a clases que no son de entidad, y las clases que no son de entidad pueden extender a clases de entidad. Las clases de entidad pueden ser tanto abstractas como no abstractas.

Entidades abstractas.

Una clase abstracta también se puede declarar como una entidad decorándola con el atributo @Entity. Las clases de entidad abstractas son como las clases de entidad concretas, solo que no se pueden instanciar.

Se utilizan clases de entidad abstractas para realizar consultas sobre subclases suyas y poder realizar un tratamiento común.

Partamos del siguiente ejemplo. Se crea una clase de entidad abstracta Empleado:

```
package entity;
@Entity
public abstract class Empleado {
  @Id
  protected Integer id;
  ... ... ...
}
```

Y ahora se crean dos subclases:

```
package entity;
@Entity
public class EmpleadoFijo extends Empleado {
  protected Integer salario;
  ... ... ...
}

package entity;
@Entity
public class EmpleadoTemporal extends Empleado {
  protected Float pagoHora;
}
```

Mapeado de superclases.

Se pueden mapear las clases del ejemplo anterior sobre tablas de base de datos. Pero podemos seguir varias estrategias a la hora de mapear las entidades con las tablas de la base de datos.

La estrategia es configurada asignando el atributo strategy de la anotación @Inheritance a uno de los valores de la enumeración javax.persistence.InheritanceType. Se pueden dar cuatro escenarios:

1) Superclases de no-entidad.
   Las entidades pueden tener una superclase que no sea de entidad, y esta superclase pude ser abstracta o concreta. El estado de no-entidad de la superclase es no-persistente, y cualquier estado heredado desde la superclase por una clase de entidad será no-persistente. Cualquier mapeado de columnas o relaciones en la superclase no-entidad serán ignoradas.
   En este escenario la superclase puede proporcionar propiedades no persistentes y métodos de proceso comunes. Las subclases se mapean con tablas de la base de datos de manera normal.
2) Mapeado de todas las entidades sobre una sola tabla.
   Puede darse el caso de que la base de datos posea una única tabla EMPLEADO que contenga todas las columnas de datos de empleados, más una columna de discriminación que determina el tipo de empleado.
   En este caso la tabla puede crearse de la siguiente manera:

```
CREATE TABLE EMPLEADO (
  ID INTEGER PRIMARY KEY,
  NOMBRE VARCHAR(150),
  SALARIO NUMERIC(8,2),
  PAGO_HORA NUMERIC(6,2),
  TIPO VARCHAR(8)
);
```

Esta estrategia se corresponde con InheritanceType.SINGLE_TABLE, y todas las clases de la jerarquía son mapeada con esta única tabla. Esta tabla tiene un discriminador, la columna TIPO, que contiene un valor que identifica la subclase cuya instancia se creará.
La columna de discriminación se especifica mediante la anotación javax.persistence.DiscriminatorColumn en la clase raíz de la jerarquía. Esta anotación tiene los siguientes atributos: name (el nombre de la columna discriminadora), discriminatorType (puede ser DiscriminatorType.STRING, DiscriminatorType.CHAR o DiscriminatorType.INTEGER), length (la longitud de la columna discriminadora cuando es un string).
Si no se especifica @DiscriminatorColumn en la superclase se asume que la columna de discriminación se llama DTYPE y es de tipo DiscriminatorType.STRING.
La superclase puede ser abstracta o no, y podrá usarse para consultas. Teniendo esto en cuenta la superclase quedará como sigue:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TIPO", length = 8)
public class Empleado implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id
  @Basic(optional = false)
  @Column(name = "ID")
  protected int id;
  @Column(name = "NOMBRE", length=150)
  protected String nombre;
  @Column(name = "TIPO", length=8)
  protected String tipo;
  ... ... ...
}
```

Ahora se usa la anotación javax.persistence.DiscriminatorValue en cada subclase para especificar su valor de discriminación. Si no se especifica este atributo se considerará el nombre de la subclase como el valor de discriminación.
Teniendo esto en cuenta las subclases quedarán como sigue:

```
@Entity
@DiscriminatorValue("fijo")
public class EmpleadoFijo extends Empleado {
  @Column(name = "SALARIO")
  private double salario;
  ... ... ...
}

@Entity
@DiscriminatorValue("temporal")
public class EmpleadoTemporal extends Empleado {
  @Column(name = "PAGO_HORA")
  private double pagoHora;
  ... ... ...
}
```

3) Mapeado de cada entidad concreta con una tabla.
Otro escenario que puede darse es que existan dos tablas, EMPLEADO_FIJO y EMPLEADO_TEMPORAL, con la circunstancia de que comparten definiciones de columnas. En este caso las tablas pueden crearse de la siguiente manera:

```
CREATE TABLE EMPLEADO_FIJO (
  ID INTEGER PRIMARY KEY,
  NOMBRE VARCHAR(150),
  SALARIO NUMERIC(8,2)
);
CREATE TABLE EMPLEADO_TEMPORAL (
  ID INTEGER PRIMARY KEY,
```

```
  NOMBRE VARCHAR(150),
  PAGO_HORA NUMERIC(6,2)
);
```

Esta estrategia se corresponde con InheritanceType.TABLE_PER_CLASS, y cada clase concreta es mapeada a una de las tablas en la base de datos.

La superclase debe ser abstracta y no podrá usarse para hacer consultas.

```
@MappedSuperclass
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
abstract public class Empleado implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id
  @Basic(optional = false)
  @Column(name = "ID")
  protected int id;
  @Column(name = "NOMBRE", length=150)
  protected String nombre;
  ... ... ...
}
```

En cada subclase se debe especificar la tabla con la que se mapea:

```
@Entity
@Table(name="EMPLEADO_FIJO")
public class EmpleadoFijo extends Empleado {
  @Column(name = "SALARIO")
  private double salario;
  ... ... ...
}

@Entity
@Table(name="EMPLEADO_TEMPORAL")
public class EmpleadoTemporal extends Empleado {
  @Column(name = "PAGO_HORA")
  private double pagoHora;
  ... ... ...
}
```

4) Estrategia de subclases asociadas.

El último caso se da cuando en la base de datos se crea una tabla EMPLEADO con todos los datos comunes a empleados y tablas asociadas EMPLEADO_FIJO y EMPLEADO_TEMPORAL con los datos particulares de estos tipos de empleado. En este caso las tablas pueden crearse de la siguiente manera:

```
CREATE TABLE EMPLEADO (
  ID INTEGER PRIMARY KEY,
  NOMBRE VARCHAR(150)
);
CREATE TABLE EMPLEADO_FIJO (
  ID INTEGER PRIMARY KEY REFERENCES EMPLEADO(ID),
  SALARIO NUMERIC(8,2)
);
CREATE TABLE EMPLEADO_TEMPORAL (
  ID INTEGER PRIMARY KEY REFERENCES EMPLEADO(ID),
  PAGO_HORA NUMERIC(6,2)
);
```

Esta estrategia se corresponde con InheritanceType.JOINED, y cada subclase se corresponde con una tabla independiente que sólo tiene los campos específicos, además de compartir la clave de la superclase.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Empleado implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id
```

```
  @Basic(optional = false)
  @Column(name = "ID")
  protected int id;
  @Column(name = "NOMBRE", length=150)
  protected String nombre;
  ... ... ...
}

@Entity
@Table(name="EMPLEADO_FIJO")
public class EmpleadoFijo extends Empleado {
  @Column(name = "SALARIO")
  private double salario;
  ... ... ...
}

@Entity
@Table(name="EMPLEADO_TEMPORAL")
public class EmpleadoTemporal extends Empleado {
  @Column(name = "PAGO_HORA")
  private double pagoHora;
  ... ... ...
}
```

Esta estrategia proporciona un buen soporte para relaciones con polimorfismo, pero requiere una o más operaciones de join cuando se instancian las subclases de entidad. Esto puede provocar un peor rendimiento en jerarquías grandes.

IMPORTANTE. Algunos proveedores de API Persistence, incluyendo el proveedor de GlassFish Server y Java Derby, requieren una columna discriminadora con esta estrategia.

2.6. El lenguaje de consultas JPQL

El lenguaje de consultas Java Persistence (JPQL) define una sintaxis de consultas para entidades y su estado de persistencia. Este lenguaje también permite escribir consultas portables que trabajan sobre el almacén de datos subyacentes.

Aunque es un lenguaje parecido a SQL, este lenguaje de consultas usa el esquema de entidades persistentes, incluyendo sus relaciones, en vez de las tablas de la base de datos.

2.6.1. Sintaxis básica de las consultas JPQL.

La sintaxis de las consultas JPQL es similar a la sintaxis de las consultas SQL estándar, pero con algunas diferencias. En SQL se consultan tablas, en JPQL se consultan entidades de persistencia.

Los comandos de consulta permiten estas cláusulas: SELECT, DISTINCT, FROM, JOIN, WHERE, GROUP BY, HAVING y ORDER BY. Las cláusulas SELECT y FROM son obligatorias.

Una consulta que recupera entidades tiene la siguiente sintaxis:

```
SELECT alias FROM Entidad alias WHERE condición
```

Por ejemplo, podemos obtener las empresas cuyo año de creación sea mayor que el 2006:

```
SELECT e FROM Empresa e WHERE FUNC('year', e.fechacreacion) > 2006
```

La cláusula GROUP BY permite realizar agrupaciones en los resultados de acuerdo con un conjunto de propiedades cuyos valores se repiten. De cada grupo podemos retornar las propiedades comunes y expresiones de agrupación. Cualquier condición sobre un grupo deberá ser especificada en una cláusula HAVING.

Por ejemplo, podemos obtener cuántas empresas se han creado en cada fecha anterior a la actual:

```
SELECT e.fechacreacion, COUNT(e) FROM Empresa e
  GROUP BY e.fechacreacion HAVING e.fechacreacion < CURRENT_DATE
```

La consulta agrupa por la fecha de creación y de cada grupo retorna la propiedad fechacrecaion y el número de resultados en ese grupo.

Los comandos de actualización tienen la siguiente sintaxis:

```
UPDATE Entidad alias SET alias.propiedad=valor WHERE condición
```

El comando UPDATE es similar al usado en SQL y sigue sus mismas reglas. En el siguiente ejemplo se actualiza el nombre de la empresa de id 1:

```
UPDATE Empresa e SET e.nombre = UPPER(e.nombre) WHERE e.id = 1
```

Los comandos de borrado tienen la siguiente sintaxis:

```
DELETE Entidad alias WHERE condición
```

El comando DELETE es similar al usado en SQL y sigue sus mismas reglas. En el siguiente ejemplo se eliminan las empresas cuyo año de creación sea el 2010:

```
DELETE FROM Emprea e  WHERE FUNC('year', e.fechacreacion) = 2010
```

2.6.2. Consultas que navegan a entidades relacionadas.

Al realizar consultas sobre entidades y no sobre tablas, JPQL permite expresiones que navegan a entidades relacionadas a través de las propiedades de navegación.

Por ejemplo, en una base de datos de EMPRESA y COMENTARIO, podemos realizar un producto cartesiano entre las entidades Empresa y Comentario de la siguiente manera:

```
SELECT DISTINCT e FROM Empresa e, IN(e.comentarioSet) c
```

Donde comentarioSet es la propiedad de navegación de la clase Empresa que retornaba sus comentarios como un conjunto. Con la palabra clave IN se indica que comentarioSet es una colección de entidades relacionadas. Y por tanto se devuelven aquellas empresas que tienen algún comentario.

Pero también se puede usar la cláusula JOIN para escribir la misma consulta:

```
SELECT DISTINCT e FROM Emprea e JOIN e.comentarioSet c
```

O bien se puede evaluar la misma propiedad de navegación:

```
SELECT DISTINCT e FROM Empresa e WHERE e.comentarioSet IS NOT EMPTY
```

2.6.3. Consultas de Join.

Como se ha visto, se puede usar la cláusula JOIN para navegar por entidades relacionadas, o en general para relacionar dos entidades persistentes.

La cláusula JOIN e INNER JOIN son equivalentes. Se utilizan para unir entidades relacionadas. Por ejemplo, la siguiente consulta recupera el id de una empresa junto con el texto de cada uno de sus comentarios.

```
SELECT e.id, c.texto FROM Empresa e JOIN Comentario c where c.empresa=e
```

Pero esta sintaxis no es la más apropiada. Lo mejor es utilizar la propiedad de navegación comentarioSet:

```
SELECT e.id, c.texto FROM Empresa e, IN(e.comentarioSet) c
```

En ambos casos sólo se recuperan aquellas empresas que tienen algún comentario. Si queremos forzar empresas que no tienen comentarios podemos utilizar la cláusula LEFT JOIN.

```
SELECT e.id, c.texto FROM Empresa e LEFT JOIN e.comentarioSet c
```

En este caso se recuperarán id's de empresa que no tienen comentarios, pero la columna de texto se rellenará con el valor NULL.

Si la estrategia de recuperación de datos es perezosa (LAZY), al devolver los resultados de las consultas no se incluirán los datos de las entidades relacionadas de forma inmediata. Si queremos forzar esta recuperación inmediata podemos agregar la cláusula FETCH a los JOIN. Por ejemplo:

```
SELECT e.id, c.texto FROM Empresa e LEFT JOIN FETCH e.comentarioSet c
```

2.6.4. Navegando a campos de entidades relacionadas.

Podemos usar la cláusula JOIN para navegar a las propiedades de una entidad relacionada.

```
SELECT e FROM Empesa e JOIN e.comentarioSet c WHERE c.orden > 3
```

En esta consulta se recuperan las empresas que tienen al menos 3 comentarios.

También podemos recuperar los comentarios de una empresa dada usando un JOIN:

```
SELECT c FROM Empresa e JOIN e.comentarioSet c WHERE e.id = 1
```

Pero también podemos hacerlo accediendo directamente a la propiedad de navegación:

```
SELECT e.comentarioSet FROM Empresa e WHERE e.id = 1
```

2.6.5. Expresiones de condición en las consultas.

Las consultas JPQL permiten el uso de operadores similares a los de SQL. A continuación se describe algunos de ellos:

- El operador LIKE permite casar un texto con un patrón. Por ejemplo, obtener aquellos comentarios que comiencen por la palabra 'Com' y tengan algún carácter más:

```
SELECT c FROM Comentario c WHERE c.texto LIKE 'Com%_'
```

El símbolo % se utiliza para indicar una secuencia de cero o varios caracteres, mientras que el símbolo _ se utiliza para indicar un caracter obligatorio en esa posición.

• El operador IS NULL permite evaluar si una expresión es nula. Por ejemplo, obtener aquellas empresas cuyo nombre no sea nulo:

```
SELECT e FROM Empresa e WHERE NOT e.nombre IS NULL
```

• El operador IS EMPTY permite evaluar si una expresión está vacía. Por ejemplo, obtener aquellas empresas que no tengan comentarios:

```
SELECT e FROM Empresa e WHERE e.comentarioSet IS EMPTY
```

• El operador BETWEEN permite evaluar un rango de valores. Por ejemplo, obtener aquellos comentarios cuyo orden esté entre 1 y 3 inclusivos:

```
SELECT c FROM Comentario c WHERE c.orden BETWEEN 1 and 3
```

• El operador IN permite evaluar la inclusión de un valor dentro de una lista. Por ejemplo, obtener los comentarios cuyo orden es igual a 1, 3 o 5:

```
SELECT c FROM Comentario c WHERE c.orden IN (1, 3, 5)
```

• Los operadores MEMBER OF y NOT MEMBER OF permiten evaluar si un valor está contenido dentro de una colección. Por ejemplo, obtener empresas que contengan el objeto comentario pasado por parámetro:

```
SELECT e FROM Empresa e WHERE :comentario MEMBER OF e.comentarioSet
```

2.6.6. Subconsultas.

El lenguaje JPQL también admite el uso de subconsultas. Se pueden usar para evaluar expresiones en las cláusulas WHERE y HAVING. Las subconsultas siempre deben ir entre paréntesis.

En el siguiente ejemplo se recuperan las empresas que tienen más de 3 comentarios usando una subconsulta:

```
SELECT e FROM Empresa e WHERE (SELECT COUNT(c) FROM e.comentarioSet c) > 3
```

Con el operador EXISTS podemos evaluar si una subconsulta devuelve resultados. Por ejemplo, obtener aquellas empresas que tienen algún comentario:

```
SELECT e FROM Empresa e WHERE EXISTS (SELECT c FROM e.comentarioSet c)
```

Con el operador ANY podemos evaluar si un valor pertenece al resultado de una subconsulta. Por ejemplo, obtener comentarios cuyo orden sea menor que el orden de los comentarios de la empresa 1:

```
SELECTc FROM Comentario c WHERE c.orden
  <ANY (SELECT d.orden FROM Comentario d WHERE d.idempresa=1)
```

Esta consulta es trivial, pero demuestra el uso del operador ANY, el cual siempre debe combinarse con un operador relacional: >, >=, <, <=, =, !=

Por su parte el operador ALL evalúa si un valor cumple una condición con todos los resultados de una subconsulta. Por ejemplo, obtener todos los comentarios cuyo orden sea menor o igual que todos los demás órdenes:

```
SELECT c FROM Comentario c WHERE c.orden <=ALL (SELECT d.orden FROM Comentario d)
```

Con esta consulta siempre retornaríamos el comentario con el menor orden.

2.6.7. Funciones predefinidas.

Al igual que SQL, las consultas JPQL incluyen varias funciones predefinidas para manipular texto y fechas, y funciones aritméticas.

Funciones sobre texto.

A continuación se resume el uso de las funciones que manipulan strings:

CONCAT(String, String)	Devuelve el resultado de concatenar dos strings.
LENGTH(String)	Devuelve la longitud de un string.
LOCATE(String, String [, start])	Devuelve la posición del primer string dentro del segundo string. Se puede indicar una posición de búsqueda inicial.
SUBSTRING(String, start, length)	Devuelve un substring dentro de un string desde una posición inicial y una longitud.
TRIM([[LEADING|TRAILING|BOTH] char FROM] String)	Devuelve un string del cual se han eliminado ciertos caracteres del principio o final o de ambos lados.
LOWER(String)	Devuelve un string en minúsculas.
UPPER(String)	Devuelve un string en mayúsculas.

Por ejemplo, si queremos recuperar el texto de los comentarios eliminando espacios en blanco iniciales y finales usaríamos la función TRIM:

```
SELECT TRIM(c.texto) FROM Comentario c
```

Si el texto puede tener puntos iniciales o finales y queremos eliminarlos podemos usar también:

```
SELECT TRIM(BOTH '.' FROM c.texto) FROM Comentario c
```

Funciones aritméticas.

A continuación se resume el uso de las funciones aritméticas:

ABS(number)	Retorna el valor absoluto de un número.
MOD(int, int)	Retorna el resto de la división entera de dos números.
SQRT(double)	Retorna la raíz cuadrada de un número.
SIZE(Collection)	Retorna el tamaño de una colección.

Funciones para fechas.

A continuación se resume el uso de las funciones para manipular fechas:

CURRENT_DATE	Retorna la fecha actual en formato java.sql.Date.
CURRENT_TIME	Retorna la hora actual en formato java.sql.Time.
CURRENT_TIMESTAMP	Retorna la fecha y hora actuales en formato java.sql.Timestamp.

JPA no incorpora funciones específicas para obtener partes de una fecha, puesto que funciones SQL como Year(), Month(), etc. no son estándar para todas las bases de datos.

Uso de funciones nativas.

JPA incorpora la función FUNC() o FUNCTION() para aplicar funciones nativas de la base de datos específica. Por ejemplo, podemos aplicar la siguiente consulta para obtener los años de creación de las empresas:

```
SELECT FUNCTION('YEAR', e.fechacreacion) FROM Empresa e
```

Siempre y cuando la base de datos soporte la función YEAR(), como es el caso de Java Derby. Pero, por ejemplo, para una base de datos de Oracle debemos usar:

```
SELECT FUNCTION('TO_CHAR', 'YYYY', e.fechacreacion) FROM Empresa e
```

La expresión «CASE».

La expresión case permite seleccionar un valor según el cumplimiento de una condición. En ese sentido es similar a la instrucción swith/case de Java.

Por ejemplo, si queremos obtener de cada empresa su id y fecha de creación y un valor "antigua" o "moderna" podemos evaluar si su año de creación es mayor o menor que 2010 de la siguiente manera:

```
SELECT e.id,
  e.fechacreacion,
  CASE WHEN FUNC('YEAR', e.fechacreacion) < 2010 THEN 'anitgua' ELSE 'moderna' END
FROM Empresa e
```

La expresión CASE admite varias sintaxis. En este caso permite evaluar condiciones con cláusulas WHEN y si todas fallan evaluar la cláusula ELSE. También permite comparar un valor contra varias expresiones. Por ejemplo, en una entidad Empleado podemos evaluar el tipo de empleado y retornar un texto personalizado:

```
SELECT e.nombre,
        CASE e.tipo WHEN 'Directivo' THEN 'A' WHEN 'Subordinado' THEN 'B' ELSE '?' END
  FROM Empleado e
```

Funciones de agregado.

Al igual que en las consultas SQL disponemos también de funciones de agrupación. La siguiente tabla las resume:

AVG	Retorna un Double con la media aritmética de los resultados.
COUNT	Retorna un Long con el número de resultados.
MAX	Retorna el valor máximo de una expresión en todos los resultados.
MIN	Retorna el valor mínimo de una expresión en todos los resultados.
SUM	Retorna la suma de los resultados.

Por ejemplo, podemos calcular el precio medio de todos los pedidos de una entidad Pedido:

```
SELECT AVG(p.precio) FROM Pedido p
```

Las funciones AVG, MAX, MIN y SUM retornan null si no hay resultados. La función COUNT retorna cero si no hay resultados.

2.6.8. Expresiones de constructor.

Las expresiones de constructor permiten retornar instancias de Java para almacenar los resultados de una consulta en vez de que se retorne un array de objetos.

Por ejemplo, la siguiente consulta retorna un array de objetos:

```
SELECT p.id, p.fecha FROM Pedido p WHERE p.id=1
```

Si existe el pedido de id 1, la consulta retorna un Object[] de dos celdas. La primera celda contiene el id, y el segundo la fecha.

Si tenemos una clase personalizada como DetallePedido:

```
package entity;
public class DetallePedido {
  private Integer id;
  private Date fecha;
  public DetallePedido(Integer id, Date fecha) {
    this.id = id;
    this.fecha = fecha;
  }
  // Métodos getter y setter ……
}
```

Entonces podemos modificar la consulta previa de la siguiente manera:

```
SELECT NEW entity.DetallePedido(p.id, p.fecha) FROM Pedido p WHERE p.id=1
```

Ahora la consulta retorna un objeto DetallePedido que encapsulará el id y la fecha. Si la consulta retornase más de un resultado retornaría un List<DetallePedido>.

3. Spring Data JPA

Las aplicaciones de Spring Boot permiten integrar fácilmente JPA, e incluyen tipos y anotaciones especiales para generar fácilmente pools de conexiones y repositorios.

3.1. Spring Hibernate JPA

Aunque Spring permite utilizar diversos proveedores de JPA, está especialmente orientado a utilizar Hibernate con un DataSource por defecto.

Las dependencias requeridas por Spring Framework para Hibernate son las siguientes:

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

3.1.1. Inicialización de la base de datos.

Podemos configurar un pool de conexiones usando el fichero de propiedades de Spring. A continuación se muestra propiedades habituales:

Fichero «src/main/resources/application.properties»

```
# Propiedades JBDC de conexión
spring.datasource.url=jdbc:h2:mem:negociodb
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver.class=org.h2.Driver
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.H2Dialect

# Crear el esquema de la base de datos a partir de las clases de entidad: true o false
spring.jpa.generate-ddl=false
# Modo DDL: create, create-drop, validate, update o none
spring.jpa.hibernate.ddl-auto=create-drop

#Inicialización adicional de la BBDD usando los ficheros schema.sql y data.sql
spring.jpa.defer-datasource-initialization=true

# En BBDD no integradas obligar a usar schema.sql y data.sql
```

```
spring.sql.init.mode=always
```

Como Spring soporta tres motores de bases de datos integradas, (HSQLDB, H2, DERBY) es opcional indicar el driver y el dialecto para estos tipos de bases de datos en el fichero de propiedades.
Cuando se utiliza una base de datos integrada, por defecto se buscan dos ficheros de recursos: «schema.sql» y «data.sql». El primero es usado para crear la estructura de la base de datos, las tablas y columnas, etc., el segundo añade datos a las tablas recién creadas, y ambos archivos deben estar ubicado en la raíz del classpath. Un contenido de ejemplo puede ser el siguiente:

Archivo «schema.sql»	Archivo «data.sql»

CREATE TABLE Cliente (
  ID INTEGER PRIMARY KEY,
  nombre VARCHAR(20),
  ciudad VARCHAR(50)
);
 	
CREATE TABLE Producto(
  ID INTEGER PRIMARY KEY,
  nombre VARCHAR(30),
  precio DECIMAL ,
  clienteId INTEGER REFERENCES Cliente(ID)
);
	INSERT INTO Cliente VALUES(1, 'Laura', 'Madrid');
	INSERT INTO Producto VALUES(1, 'Iron Iron', 54, 1);
	INSERT INTO Cliente VALUES(2, 'Susana', 'Sevilla');
	INSERT INTO Producto VALUES(2, 'Chair Shoe', 248, 2);
	INSERT INTO Cliente VALUES(3, 'Ana', 'Cádiz');
	INSERT INTO Producto VALUES(3, 'Telephone Clock', 248, 3);

Para que el Framework ejecute estos ficheros se debe dar uno de estos casos:

1) Hemos deshabilitado la generación del esquema a partir de las clases de entidad:
   spring.jpa.hibernate.ddl-auto=none
2) La generación automática está habilitada y forzamos la inicialización posterior:
   spring.jpa.hibernate.ddl-auto=create-drop
   spring.jpa.defer-datasource-initialization=true

   En ese segundo caso, se usará «schema.sql» para actualizar el esquema, y «data.sql» para poblar las tablas.
3) Para bases de datos no integradas se fuerza su uso:
   spring.sql.init.mode=always

3.1.2. Trabajando con el «EntityManager».

Con la información de configuración, el Framework es capaz de instanciar automáticamente un EntityManager y dejarlo disponible mediante un Bean. Pero debemos tener en cuenta que Spring Boot crea unidades de persistencia con transacciones de tipo JTA. Esto quiere decir que el Framework se encarga de gestionar transacciones de operaciones realizadas dentro de métodos. Estos métodos (o su clase) tendremos que anotarlos con @Transactional para que sean transaccionales.
Una DAO se corresponde con una clase que sigue el modelo CRUD (Create-Read-Update-Delete/Crear-Leer-Actualizar-Eliminar). Debe incluir métodos para persistir, recuperar, actualizar y eliminar entidades de datos.
A continuación se muestra un ejemplo de una clase DAO:

```
@Transactional
public interface EntityDao<T, K> {
  EntityManager getEntityManager();
  Class<T> getEntityClass();

  default T save(T entity) {
    getEntityManager().persist(entity);
```

```
    return entity;
  }
  default void delete(T entity) {
      getEntityManager().remove(entity);
  }
  default T update(T entity) {
    getEntityManager().merge(entity);
    return entity;
  }
  default T find(K key) {
    return getEntityManager().find(getEntityClass(), key);
  }
  default List<T> findAll() {
    CriteriaQuery<T> criteria = getEntityManager().getCriteriaBuilder().createQuery(getEntityClass());
    criteria.select(criteria.from(getEntityClass()));
    return getEntityManager().createQuery(criteria).getResultList();
  }
}

@Repository
public class ClienteDao implements EntityDao<Cliente, Integer> {
  @PersistenceContext
  private EntityManager em;
  @Override
  public EntityManager getEntityManager() {
    return em;
  }
  @Override
  public Class<Cliente> getEntityClass() {
    return Cliente.class;
  }
}

@SpringBootApplication
public class SpringjdbcApplication {
  @Autowired
  private ClienteDao clienteDao;

  public static void main(String[] args) {
    var context = SpringApplication.run(SpringjdbcApplication.class, args);
    var app = context.getBean(SpringjdbcApplication.class);
    app.clienteDao.save(new Cliente(10, "A", "C"));
    System.out.println(app.clienteDao.findAll());
  }

}
```

Como vemos en este ejemplo, si lo basamos en una interfaz genérica (EntityDao) podemos crear métodos default, de forma que la clase DAO (ClienteDao) simplemente debe proporcionando el EntityManager y su Class.

La anotación @PersistenceContext permite inyectar directamente un EntityManager, o bien podemos recuperarlo como un Bean de tipo EnityManager:

```
EntityManage = context.getBean(EntityManager.class);
```

El patrón de diseño DAO aplicado a una clase está pensado para gestionar un único tipo de entidades, mientras que el patrón Repository está pensado para que una clase gestione varios tipos de entidades.

3.1.3. Control de transacciones.

Básicamente, el Framework de Spring Boot habilita la gestión de transacciones de forma predeterminada. Con la anotación org.springframework.transaction.annotation.Transactional podremos decorar un Bean a nivel de clase o de método para controlar las transacciones.

```
@Repository
@Transactional
public class ClienteRepository {
  //...
}
```

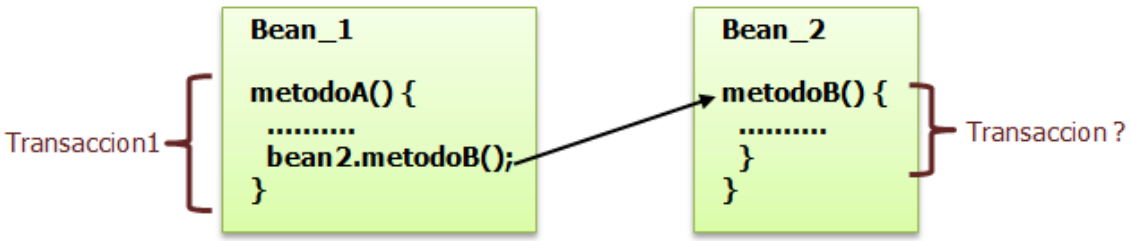
La anotación admite configuraciones adicionales :

- El tipo de propagación de la transacción.
- El nivel de aislamiento de la transacción.
- Un tiempo de espera para la operación envuelta por la transacción.
- Un indicador de solo lectura : una sugerencia para el proveedor de persistencia de que la transacción debe ser de solo lectura.
- Las reglas de reversión para la transacción (rollback).

Hay que tener en cuenta que, de forma predeterminada, la reversión se produce en el tiempo de ejecución, solo cuando se producen excepciones de tipo unchecked. La excepción checked no desencadenan una reversión de la transacción. Aunque podemos configurar este comportamiento con los parámetros rollbackFor y noRollbackFor de la anotación.

Propagación de transacciones.

La propagación de la transacción controla el alcance de esta. Esto es importante si se da un caso como el que se ilustra a continuación:



El método metodoA() es transaccional, y este método invoca a otro método metodoB() también transaccional. Cuando metodoA() ejecuta metodoB(), ¿se crea una nueva transacción? La respuesta depende de la propagación aplicada sobre metodoB().

Los tipos de propagación soportadas son las siguientes:

Propagation.MANDATORY	Si el llamador se ejecuta dentro de una transacción e invoca el método del bean, el método se ejecuta dentro de la transacción del llamador. Si el llamador no está asociado a una transacción el contenedor lanza una excepción.
Propagation.NEVER	Si el llamador se ejecuta dentro de una transacción e invoca el método del bean, el contenedor lanza una RemoteException. Si el llamador no está asociado a ninguna transacción, el contenedor no inicia una nueva transacción antes de ejecutar el método.
Propagation.NOT_SUPPORTED	Si el llamador se ejecuta dentro de una transacción e invoca el método del bean, se suspende la transacción del llamador antes de invocar el método. Después de que el método se completa, se reanuda la transacción del llamador. Si el llamador no está asociado a una transacción no se inicia una nueva transacción antes de ejecutar el método. Es el tipo por defecto, y se usa para métodos que no necesitan transacciones.
Propagation.REQUIRED	Si el llamador se está ejecutando dentro de una transacción e invoca el método del bean, el método se ejecuta dentro de la transacción del llamador. Si el llamador no está asociado con una transacción, se inicia una nueva transacción antes de ejecutar el método.

Propagation.REQUIRES_NEW	Si el llamador se ejecuta dentro de una transacción e invoca el método del bean, se siguen los siguientes pasos: - Se suspende la transacción del llamador. - Se inicia una nueva transacción. - Se delega la llamada al método. - Se reanuda la transacción del llamador después de que el método se completa. Si el llamador no está asociado a una transacción se inicia una nueva transacción antes de ejecutar el método.
Propagation.SUPPORTS	Si el llamador se ejecuta dentro de una transacción e invoca el método del bean, el método se ejecuta dentro de la transacción del llamador. Si el llamador no está asociado a ninguna transacción, no inicia una nueva transacción antes de ejecutar el método.

Según esto, supongamos el siguiente ejemplo:

```
@Transactional(propagation = Propagation.MANDATORY)
public void save(Cliente cliente) {
  getEntityManager().persist(cliente);
}
public void update(Cliente cliente) {
  if (getEntityManager().find(Cliente.class, cliente.getId()) == null) {
    save(cliente);
  } else {
    getEntityManager().merge(entity);
  }
}
```

El método save() es transaccional y el método update() no. La ejecución de update() provocará un error cuando ejecute save() puesto que su tipo de propagación es MANDATORY y requiere que su llamador sea transaccional.

Cambiar el nivel de aislamiento.

Podemos cambiar el nivel de aislamiento de la transacción:

```
@Transactional(isolation = Isolation.SERIALIZABLE)
```

Con SERIALIZABLE se evitan lecturas endebles de los datos, evitando situaciones en la que una transacción lee todas las filas que cumplen una condición WHERE, una segunda transacción inserta una fila que cumple esa condición y la primera transacción vuelve a leer la misma condición, recuperando la fila adicional "fantasma" en la segunda lectura. Con la constante READ_COMMITTED sólo se leerían filas de datos consolidados. Con la constante READ_UNCOMMITTED se leerían también filas de datos no consolidados, lo cual puede dar lugar a lectura de filas fantasmas.

Transacciones de solo lectura.

La opción readOnly impide que se confirmen operaciones de actualización a través de la transacción.

```
@Transactional(readOnly = true)
```

Reversión de transacciones.

Cuando un método transaccional genera una excepción unchecked, por defecto se revierte la transacción (se aplica un rollback). Hay dos formas de controlar el comportamiento de revertir una transacción: declarativa y programática.

En el enfoque declarativo, se anotan los métodos con @Transactional. Esta anotación utiliza los atributos rollbackFor o rollbackForClassName para revertir las transacciones y los atributos noRollbackFor o noRollbackForClassName para evitar la reversión cuando se generen las excepciones enumeradas.

Veamos un ejemplo simple provocando con una excepción que deshaga una inserción:

```
@Transactional
public void save(Cliente cliente) {
  getEntityManager.persist(cliente);
  throw new DataIntegrityViolationException("Throwing exception for demoing Rollback!!!");
}
```

A continuación, revertiremos una transacción cuando se produzcan las excepciones enumeradas, en este caso una SQLException :

```
@Transactional(rollbackFor = { SQLException.class })
public void save(Cliente cliente) throws SQLException {
  getEntityManager.persist(cliente);
  throw new SQLException("Throwing exception for demoing rollback");
}
```

Ahora, un uso simple del atributo noRollbackFor para evitar la reversión de la transacción para la excepción enumerada:

```
@Transactional(noRollbackFor = { SQLException.class })
```

Con un enfoque programático, debemos revertimos las transacciones usando TransactionAspectSupport :

```
public void save(Cliente cliente) {
  try {
    getEntityManager.persist(cliente);
  } catch (Exception e) {
    TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
  }
}
```

3.2. Configuración de un DataSource programáticamente.

Aunque la forma más simple de configuración el DataSource es usando el fichero de propiedades de la aplicación, habrá veces donde necesitemos programar manualmente el DataSource. Para ello debemos crear un Bean que retorne el DataSource y para crearlo Spring Data proporciona la clase DataSourceBuilder:

```
@Configuration
public DataSource getDataSource() {
  return DataSourceBuilder.create()
       .driverClassName("org.h2.Driver")
       .url("jdbc:h2:mem:test")
       .username("SA")
       .password("")
       .build();
  }
}
```

También será posible externalizar parcialmente nuestra configuración del DataSource . Por ejemplo, podríamos definir algunas propiedades básicas en el método fabricador del Bean:

```
@Bean
public DataSource getDataSource() {
  return DataSourceBuilder.create()
       .username("SA")
       .password("")
       .build();
  }
```

Y luego podemos especificar algunas propiedades adicionales en el archivo «application.properties»:

```
spring.datasource.url=jdbc:h2:mem:test
spring.datasource.driver-class-name=org.h2.Driver
```

3.3. Uso de DAO/Repositorio.

Crear clases aplicando el patrón DAO o Repository generalmente es una tarea pesada, pues consta de una gran cantidad de código repetitivo que se puede y debe simplificar. Spring Data permite esta simplificación y la lleva un paso más allá y hace posible eliminar por completo las implementaciones de DAO. La interfaz del DAO será el único artefacto que necesitemos definir explícitamente.

3.3.1. Definición y uso de un Bean DAO.

Las interfaces DAO deben extender una de las interfaces de repositorio proporcionadas por Spring Data: CrudRepository o JpaRepository. Esto permitirá que Spring Data reconozca esta interfaz y cree automáticamente una implementación para ella. Por ejemplo:

```
public interface ClienteRepository extends CrudRepository<Cliente, Integer> {
```

```
}
```

Al ser genérica, la interfaz JpaRepository pide el tipo de la clase de entidad, y el tipo de la clave primaria. Y Sping Data se encarga de todo lo demás, dejando disponible en la interfaz los métodos CRUD más relevantes para el acceso a datos disponibles en un DAO estándar.

```
@SpringBootApplication
public class SpringDataApplication {
  @Autowired
  private ClienteRepository clienteRepository;

  public static void main(String[] args) {
    var context = SpringApplication.run(SpringjdbcApplication.class, args);
    var app = context.getBean(SpringjdbcApplication.class);

    var repo = app.clienteRepository;
    repo.save(new Cliente(10, "A", "C"));          // se añade un cliente
    repo.findAll().forEach(System.out::println);   // se recuperan todos los clientes
  }
}
```

3.3.2. Método de acceso personalizado y consultas.

Como se mencionó, al extender CrudRepository o JpaRepository, el DAO ya tendrá definidos e implementados algunos métodos CRUD básicos (y consultas). Para definir métodos de acceso más específicos, Spring JPA admite varias opciones:

- Simplemente definir un nuevo método en la interfaz con un nombre adecuado.
- Proporcionar la consulta JPQL real utilizando la anotación @Query sobre el método.
- Utilizar la especificación más avanzada y la compatibilidad con Querydsl en Spring Data.
- Definir consultas personalizadas a través de JPA Named Queries.

Consultas personalizadas automáticas.

Cuando Spring Data crea una nueva implementación de repositorio, analiza todos los métodos definidos por las interfaces e intenta generar consultas automáticamente a partir de los nombres de los métodos. Si bien esto tiene algunas limitaciones, es una forma muy poderosa y elegante de definir nuevos métodos de acceso personalizados con muy poco esfuerzo.

Veamos un ejemplo sobre la base de datos de clientes. Si queremos recuperar un cliente por su nombre (propiedad nombre) definiremos un método findByNombre(), y esto generará automáticamente la consulta correcta:

```
public interface ClienteRepository extends CrudRepository<Cliente, Integer> {
  Cliente findByNombre(String nombre);
}
```

En ese caso se supone que la consulta retornará un único resultado, si no es así, el método tendrá que ser declarado para que retorne una lista.

Este es un ejemplo relativamente simple. El mecanismo de creación de consultas admite un conjunto mucho mayor de palabras clave. La siguiente tabla resume las palabras clave soportadas para JPA.

Palabra:	Ejemplos:	Consulta JPQL generada:
Distinct	findDistinctByNombre(nombre)	select distinct … where x.nombre = ?1
And	findByNombreAndCiudad(nombre, ciudad)	… where x.nombre = ?1 and x.ciuad = ?2
Or	findByNombreOrCiudad(nombre, ciudad)	… where x.nombre = ?1 or x.ciudad = ?2
Is, Equals	findByNombre(nombre) findByNombreIs(nombre) findByNombreEquals(nombre)	… where x.nonmbre = ?1
Between	findByFechaBetween(fecha1, fecha2)	… where x.fecha between ?1 and ?2
LessThan	findByEdadLessThan(edad)	… where x.edad < ?1
LessThanEqual	findByEdadLessThanEqual(edad)	… where x.edad <= ?1
GreaterThan	findByEdadGreaterThan(edad)	… where x.edad > ?1
GreaterThanEqual	findByEdadGreaterThanEqual(edad)	… where x.edad >= ?1
After	findByFechaAfter(fecha)	… where x.fecha > ?1

Palabra:	Ejemplos:	Consulta JPQL generada:
Before	findByFechaBefore(fecha)	… where x.fecha < ?1
IsNull, Null	findByNombreNull() findByNombreIsNull()	… where x.nombe is null
IsNotNull, NotNull	findByNombreNotNull() findByNombreIsNotNull()	… where x.nombre not null
Like	findByNombreLike(nombre)	… where x.nombre like ?1
NotLike	findByNombreNotLike(nombre)	… where x.nombre not like ?1
StartingWith	findByNombreStartingWith(cadena)	… where x.nombre like ?1 (al parámetro se le añade %)
EndingWith	findByNombreEndingWith(cadena)	… where x.nombre like ?1 (al parámetro se le añade %)
Containing	findByNombreContaining	… where x.nombre like ?1 (el parámetro se rodea con %)
OrderBy	findByEdadOrderByNombreDesc(edad)	… where x.edad = ?1 order by x.nombre desc
Not	findByNombreNot(nombre)	… where x.nombre <> ?1
In	findByEdadIn(Collection<Integer> edades)	… where x.edad in ?1
NotIn	findByEdadNotIn(Collection<Integer> edades)	… where x.edad not in ?1
True	findBySocioTrue()	… where x.socio = true
False	findBySocioFalse()	… where x.socio = false
IgnoreCase	findByNombreIgnoreCase(nombre)	… where UPPER(x.nombre) = UPPER(?1)

Consultas personalizadas manuales

Ahora veamos una consulta personalizada que definiremos a través de la anotación @Query:

```
@Query("SELECT count(c) FROM Cliente c WHERE LOWER(c.ciudad) = LOWER(:ciudad)")
int countClientesPara(@Param("ciudad") String ciudad);
```

Esta consulta retorna el número de clientes de una ciudad dada. Como vemos la consulta permite embeber el valor de los parámetros del método precediéndolos con puntos.

Uso de consultas con nombre.

En las propias clases de entidad podemos definir consultas con nombre usando la anotación @NamedQuery. Por ejemplo:

```
@NamedQuery(name = "Cliente.listCiudades",
    query = "select distinct c.ciudad from Cliente c order by c.ciudad")
public class Cliente {
    ...
}
```

La interfaz DAO puede incluir un método con el mismo nombre para asociar la consulta:

```
public interface ClienteRepository extends CrudRepository<Cliente, Integer> {
  List<String> listCiudades();
}
```

3.3.3. Paginación y ordenación.

Al extender la interfaz JpaRepository dispondremos de métodos para realizar consultas con paginación. Y al extender la interfaz PagingAndSortingRepository también dispondremos de opciones de ordenación.

Consultas con paginación.

La técnica de paginación con los métodos que la soportan es la siguiente:

1) Crear y obtener un objeto PageRequest, que es una implementación de la interfaz Pageable.
2) Pasar el objeto PageRequest como argumento al método de repositorio que pretendemos usar.

Podemos crear un objeto PageRequest pasando el número de página solicitado y el número de elementos por página. Aquí el recuento de páginas comienza en cero:

```
Pageable firstPageWithTwoElements = PageRequest.of(0, 2);
Pageable secondPageWithFiveElements = PageRequest.of(1, 5);
```

Una vez que tengamos nuestro objeto PageRequest, podemos pasarlo mientras invocamos el método de nuestro repositorio:

```
Page<Cliente> todos = repo.findAll(firstPageWithTwoElements);
```

```
System.out.println(todos.toList());
```

Una instancia de Page<Cliente>, además de tener la lista de clientes, también conoce el número total de páginas disponibles con getTotalPages(), y sabe si hay más páginas disponibles con hasNext().

Ordenación.

De manera similar, para ordenar los resultados de nuestra consulta pasando una instancia de Sort al método:

```
List<Cliente> todos = repo.findAll(Sort.by("nombre"));
```

Sin embargo, ¿qué pasa si queremos ordenar y paginar nuestros datos? Podemos hacerlo pasando los detalles de ordenación al objeto PageRequest:

```
Pageable sortedByNombre = PageRequest.of(0, 3, Sort.by("nombre"));.
```

3.3.4. Consultas basadas en ejemplos.

Query by Example (QBE) es una técnica de creación de consultas que permite ejecutar consultas basadas en una instancia de entidad de ejemplo. Los campos de la instancia de la entidad deben completarse con los valores de criterios deseados.

En Spring Data JPA se debe usar un repositorio cuya interfaz herede de QueryByExampleExecutor<T>, y se usa una instancia de org.springframework.data.domain.Example como argumento de los métodos count(), exists(), findOne(), findAll() y findBy(). Por ejemplo:

```
Cliente cliente = new Cliente();
clienteEx.setNombre("Erika" );
Example<Cliente> clienteExample = Example.of(cliente);
var resultado = repo.findAll(clienteExample)
```

De forma predeterminada, los campos que tienen valores nulos se ignoran en la consulta subyacente, por lo que este Example será equivalente a la siguiente consulta JPQL:

```
SELECT c FROM Cliente c WHERE c.nombre = 'Erika';
```

Podemos personalizar la instancia de Example para decidir cómo deben coincidir los campos de ejemplo usando un objeto ExampleMatcher, que ofrece los siguientes método estáticos:

- ExampleMatcher matching(): hace coincidir sólo los campos no nulos.
- ExampleMatcher.matchingAny(): hace coincidir uno de los campos no nulos.
- ExampleMatcher.matchingAll(): hace coincidir todos los campos, nulos o no.

El método Example.of() usa por defecto ExampleMatcher matching() para hacer coincidir sólo los campos no nulos. Pero supongamos que debemos retornar aquellos clientes que o bien tengan el nombre "Ana", o bien sean de "Madrid":

```
var cliente = new Cliente();
cliente.setNombre("Ana");
cliente.setCiudad("Madrid");
var example = Example.of(cliente, ExampleMatcher.matchingAny());
```

La interfaz ExampleMatcher también ofrece los siguientes métodos de instancia:

- withIgnorePaths(String... ignoredPaths): permite especificar campos que serán ignorados.
- withStringMatcher(StringMatcher stringMatcher): personaliza la comparación de valores de texto.
- withIgnoreCase(): especifica que se hagan comparaciones ignorando mayúsculas y minúsculas.
- withIgnoreCase(boolean defaultIgnoreCase): activa o desactiva la comparación sensible a mayúsculas y minúsculas.
- withIgnoreCase(String... propertyPaths): aplica comparaciones sensibles solo sobre los campos especificados.
- withIncludeNullValues(): incluye los campos con valores nulos en la consulta.
- withIgnoreNullValues(): ignara los valores nulos.
- withNullHandler(NullHandler nullHandler): especifica un manejador de valores nulos, donde NullHandler es una enumeración con las constantes IGNORE, INCLUDE.
- withMatcher(String propertyPath, MatcherConfigurer<GenericPropertyMatcher> matcherConfigurer): personaliza una consulta sobre un campo especificado.
- withMatcher(String propertyPath, GenericPropertyMatcher genericPropertyMatcher): personaliza una consulta sobre un campo especificado.
- withTransformer(String propertyPath, PropertyValueTransformer propertyValueTransformer): personaliza una consulta sobre un campo especificado.

Veamos ejemplos de uso de estos métodos:

1) Ignorar campos.
Queremos obtener aquellos clientes de nombre "Ana" ignorando el valor de su ID:

```
var cliente = new Cliente();
cliente.setNombre("Ana");
var example = Example.of(cliente, ExampleMatcher.matching().withIgnorePaths("id"));
```

Esta opción es útil cuando trabajamos con campos que no admiten valores nulos.
2) Criterios para casar campos de texto.
Queremos obtener aquellos clientes cuyo nombre comience por "A":

```
var cliente = new Cliente();
cliente.setNombre("A");
var example = Example.of(cliente, ExampleMatcher.matching().withStringMatcher(StringMatcher.STARTING));
```

La enumeración StringMatcher ofrece las siguientes constantes:
DEFAULT: aplica el criterio por defecto.
EXACT, el valor de ejemplo se considera como un literal que debe casar exactamente.
STARTING, el valor de ejemplo se considera el inicio del texto a casar.
ENDING, el valor de ejemplo se considera el final del texto a casar.
CONTAINING, el valor de ejemplo se considera como parte del contenido del texto a casar.
REGEX, el valor de ejemplo se considera como una expresión regular.
3) Transformando valores.
Queremos que los clientes con nombre a valor null sean considerados como strings vacíos:

```
var cliente = new Cliente();
var example = Example.of(
  cliente,
  ExampleMatcher.matchingAny().withTransformer("nombre", n -> n.isPresent()? n : Optional.of(""))
);
```

4) Manejando nulos.
Queremos obtener clientes con el campo nombre a nulo, ignorando los demás campos:

```
var cliente = new Cliente();
var example = Example.of(
  cliente,
  ExampleMatcher.matchingAll().withIgnorePaths("id", "ciudad").withIncludeNullValues()
);
```

3.3.5. Configuración de transacciones

La implementación real del DAO administrado por Spring está oculta, ya que no trabajamos con él directamente. Sin embargo, es una implementación bastante simple, la clase SimpleJpaRepository, que define la semántica de transacciones usando anotaciones.
Usa la anotación @Transactional configurada de solo lectura en el nivel de clase, que luego se anula para los métodos que no son de solo lectura. El resto de la semántica de la transacción es predeterminada, pero se puede anular fácilmente de forma manual según el método.

3.3.6. Configuración del repositorio Spring Data JPA.

Si todas las clase involucradas con JPA son creadas en el paquete o un subpaquete de la clase principal, no será necesario realizar configuraciones adicionales para los repositorios ni las clases de entidad. Si esto no es así, debemos activar el soporte de repositorio Spring JPA usando la anotación @EnableJpaRepositories y especificando el paquete que contiene las interfaces DAO:

```
@EnableJpaRepositories(basePackages = "com.cloudftic.persistence.repository")
public class PersistenceConfig {
    ...
}
```

3.4. Soporte para MongoDb.

Spring Data para MongoDB es parte del proyecto general Spring Data que tiene como objetivo proporcionar un modelo de programación familiar y consistente basado en Spring para las bases de datos de Mongo.
Las dependencias necesarias son:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
  </dependency>
```

3.4.1. Uso de «MongoTemplate».

La clase MongoTemplate sigue el patrón de plantillas estándar de Spring y proporciona una API básica lista para usar para el motor de persistencia subyacente.

Se aconseja crear una configuración basada en Beans, tal como se muestra a continuación:

```
@Configuration
public class SimpleMongoConfig {
  @Bean
  public MongoClient mongo() {
    ConnectionString connectionString = new ConnectionString(
            "mongodb://localhost:27017/mongodbVSCodePlaygroundDB ");
    MongoClientSettings mongoClientSettings = MongoClientSettings.builder()
            .applyConnectionString(connectionString)
            .build();
    return MongoClients.create(mongoClientSettings);
  }
  @Bean
  public MongoTemplate mongoTemplate() throws Exception {
    return new MongoTemplate(mongo(), "mongodbVSCodePlaygroundDB");
  }
}
```

Trabajaremos sobre una base de datos "mongodbVSCodePlaygroundDB" que contendrá una colección "sales" correspondiente a la siguiente clase:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Document
public class Sales {
  @Id
  private int id;
  private String item;
  private double price;
  private int quantity;
  private LocalDateTime date;
  private java.util.List<String> details = new ArrayList<>();
}
```

Inserciones.

Comencemos con la operación de inserción de ventas:

```
Sales venta = new Sales(1, "abc", 10, 5, null, null);
app.mongoTemplate.insert(venta, "sales");
```

En el método insert() debemos indicar el nombre de la colección donde insertar como segundo argumento. Aquellas propiedades con valores nulos no serán incluidas en la inserción.

Guardar o actualizar.

La operación save() permite guardar o actualizar un documento. Si incluimos un valor de ID existente se realiza una actualización y, si no, se realiza una inserción.

```
Sales venta = new Sales(2, "abc", 10, 5, null, null);
app.mongoTemplate.save(venta, "sales");
```

Actualizar el primero.

El método updateFirst() actualiza el primer documento que coincide con la consulta. Por ejemplo, actualizaremos la primera venta de precio 10 para cambiarlo a 100:

```
Query query = new Query();
query.addCriteria(Criteria.where("price").is(10));
Update update = new Update();
update.set("price", 100);
mongoTemplate.updateFirst(query, update, "sales");
```

Varias actualizaciones.

El método UpdateMulti() actualiza todos los documentos que coinciden con la consulta dada.

```
Query query = new Query();
query.addCriteria(Criteria.where("item").is("abc"));
Update update = new Update();
update.set("item", "ABC");
mongoTemplate.updateMulti(query, update, Sales.class);
```

Buscar y modificar

El método findAndModify() es similar a updateMulti(), pero devuelve el documento existente en la colección antes de que se modifique.

El upsert() se puede utilizar para modificar y buscar o para crear: si el documento dado casa con alguno de la colección se actualiza, sino se crea un nuevo documento combinando la consulta y el objeto de actualización.

```
Query query = new Query();
query.addCriteria(Criteria.where("item").is("xyz"));
Update update = new Update();
update.set("price", 50);
mongoTemplate.upsert(query, update, Sales.class);
```

Eliminar.

El método remove() elimina un documento o aquellos documentos que cumplen con una condición. A continuación se elimina las ventas con un precio mayor o igual a 10:

```
Query query = new Query();
query.addCriteria(Criteria.where("price").gte(10));
mongoTemplate.remove(query, "sales");
```

3.4.2. Uso de repositorios.

Los repositorios para Mongo siguen el enfoque centrado en Spring Data y viene con operaciones de API más flexibles y complejas, basadas en los patrones de acceso bien conocidos en todos los proyectos de Spring Data.

Si es necesario, debemos comenzar habilitando el soporte de repositorios de Spring Data Mongo, indicando el paquete donde se encuentran los repositorios:

```
@EnableMongoRepositories(basePackages = "com.cloudftic.repository")
```

Después de la configuración, se crea un repositorio extendiendo la interfaz existente de MongoRepository:

```
@Repository
public interface SalesRepository extends MongoRepository<Sales, Integer> {
  //
}
```

Ahora podemos recuperar el Bean de tipo SalesRepository y usar las operaciones heredadas de MongoRepository o agregar operaciones personalizadas.

En el fichero «application.properties» podemos configurar la conexión a Mongo DB con las siguientes propiedades:

```
# MongoDB Configuration
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=mongodbVSCodePlaygroundDB
```

4. Fundamentos de Aplicaciones Web

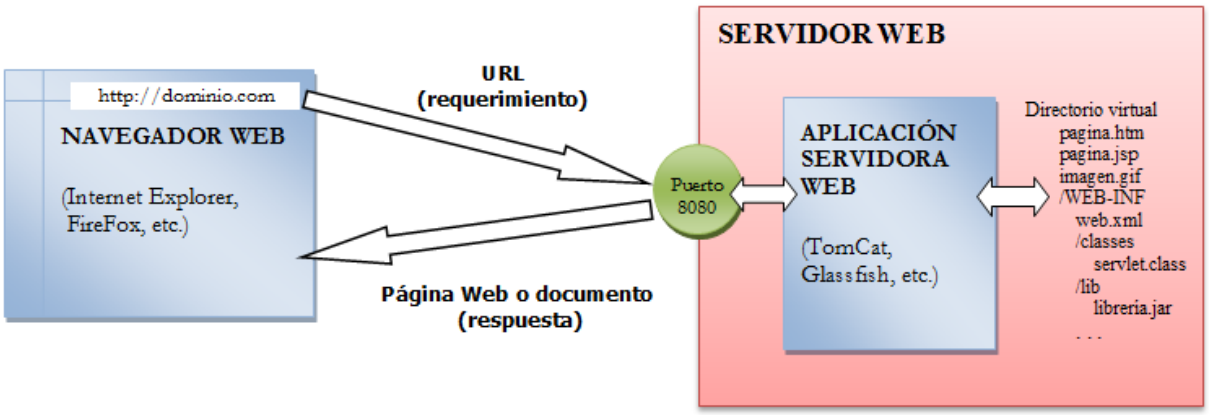
4.1. El modelo cliente/servidor en aplicaciones Web.

Las aplicaciones web de Java se construyen con componentes web que realizan tareas específicas y que exponen sus servicios a través de una red usando el protocolo HTTP. Las aplicaciones Web que usan tecnologías Java se componen de un componente que gestiona las solicitudes entrantes, de clases, de páginas y archivos auxiliares. Todos estos componentes deben coordinarse entre sí para ofrecer un conjunto completo de servicios a los usuarios. En general, las aplicaciones Web residen en una aplicación servidora Web (existiendo en el mercado servidores web como Apache Tomcat, Tomee, JBoss, Glassfish, etc.), mientras que las aplicaciones web con Spring Boot pueden incluir el núcleo de un servidor simple y ejecutarse como microservicios.

En las aplicaciones Web se distinguen dos tipos de recursos:

- Recursos del lado servidor o dinámicos. Incluyen código que es ejecutado por el servidor correspondiente.
- Recursos del lado cliente o estáticos. No incluyen código que deba ser ejecutado por un servidor.

Una aplicación Web no se ejecuta en un único proceso o en una única máquina. En vez de eso, normalmente se hospeda en un servidor Web y es accedida a través de un navegador Web usando el protocolo HTTP. Es necesario tener una comprensión básica de cómo trabajan estos elementos y se comunican entre sí antes de empezar a escribir código.



El proceso de comunicación típico entre un navegador y un servidor se puede generalizar en los siguientes pasos:

1) Un usuario usa un navegador Web (como Internet Explores, Chrome u otro) para iniciar un requerimiento a un recurso de un servidor Web.
2) Se utiliza el protocolo HTTP para enviar un requerimiento de tipo GET al servidor Web.
3) El servidor Web procesa el requerimiento GET sobre el servidor (normalmente localiza el recurso requerido y lo ejecuta).
4) El servidor Web envía entonces una respuesta al navegador Web. Se usa el protocolo HTTP para enviar la respuesta.
5) El navegador Web procesa la respuesta (normalmente ésta llega en formato HTML y JavaScript) y renderiza una página para mostrársela al usuario.
6) El usuario puede introducir datos y realizar acciones como pulsar sobre un botón para enviar datos de regreso al servidor Web para que los procese.
7) Se usa el protocolo HTTP para enviar los datos de regreso al servidor Web, realizando un requerimiento de tipo POST.
8) El servidor Web procesa el requerimiento POST (otra vez, ejecutando algún código).
9) El servidor Web envía una respuesta al navegador Web mediante el protocolo HTTP.
10) El navegador Web vuelve a procesar la respuesta y muestra una página Web al usuario.

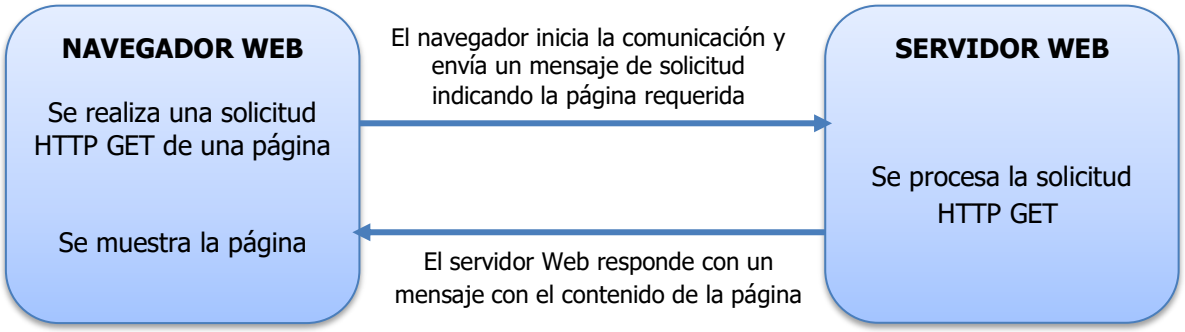
Este proceso se repite una y otra vez durante una sesión típica de navegación por un sitio Web.

4.1.1. El rol del servidor Web.

Los primeros servidores Web eran responsables de recibir y procesar requerimientos de usuario desde navegadores a través de HTTP. El servidor Web gestionaba los requerimientos y enviaba una respuesta de regreso al navegador Web. Hecho esto, el servidor Web entonces cerraba cualquier conexión entre él y el navegador y liberaba todos los recursos involucrados con el requerimiento. Estos recursos eran fácilmente liberados cuando el servidor Web finalizaba de procesar el requerimiento.

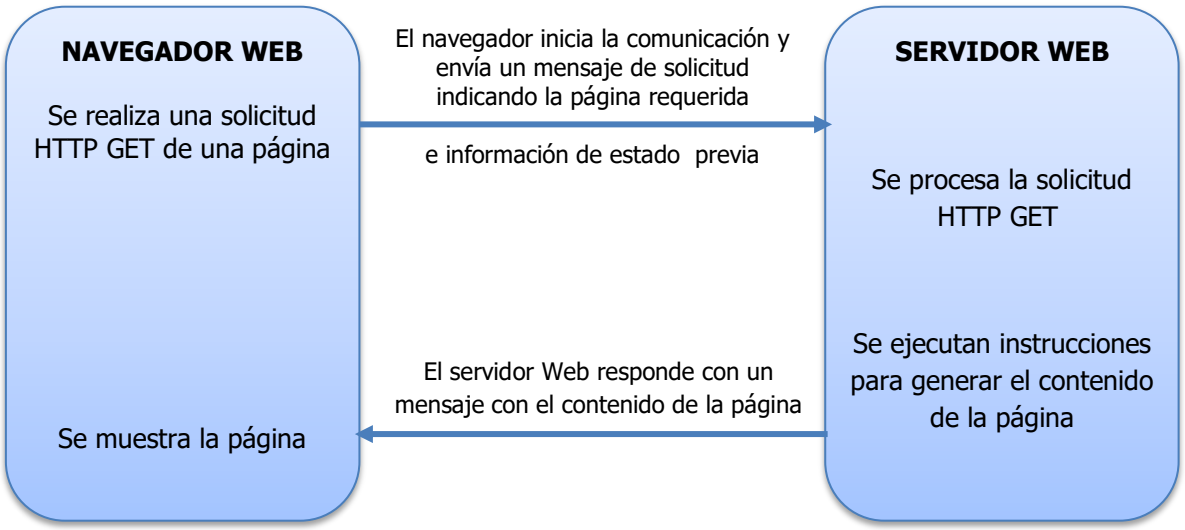
Este tipo de aplicaciones Web eran consideradas sin estado porque los datos no eran conservados por el servidor Web entre requerimientos y las conexiones no se reutilizaban. Estas aplicaciones normalmente involucraban simples página HTML y eran por tanto capaces de gestionar miles de requerimientos por minuto. La siguiente figura muestra un ejemplo de este simple entorno sin estado.

MODELO CLIENTE/SERVIDOR SIN ESTADO



Hoy en día los servidores Web realizan servicios que van más allá de los servidores Web originales. Además de servir archivos HTML estáticos, los modernos servidores Web también gestionan requerimientos a páginas que contienen código que es ejecutado sobre el servidor; el servidor Web ejecuta este código ante el requerimiento y responde con resultados. Estos servidores Web son también capaces de almacenar datos entre los requerimientos. Esto significa que las páginas Web pueden conectarse mediante un formulario a aplicaciones Web que comprenden el estado actual de cada requerimiento individual del usuario. Estos servidores mantienen una conexión abierta con los navegadores durante un periodo de tiempo anticipando requerimientos de páginas adicionales por parte del mismo usuario. Este tipo de interacción se ilustra en la siguiente figura.

MODELO CLIENTE/SERVIDOR CON ESTADO



4.1.2. El rol del navegador Web.

Para que un navegador Web pueda mostrar los datos que recibe desde un servidor de una forma independiente de la plataforma se creó un lenguaje estándar para mostrar contenido. Este lenguaje se concreta en el lenguaje HTML mediante el uso de etiquetas. El lenguaje HTML fue diseñado para ser capaz de renderizar información sobre algún sistema operativo sin tener que poner restricciones sobre el tamaño de la ventana. Esto es así porque las páginas Web se consideran independientes de la plataforma. El HTML fue diseñado para "fluir", para romper el texto si es necesario ajustarlo a los bordes de la ventana del navegador. El navegador Web también muestra imágenes y responde a enlaces a otras páginas. Cada página Web requerida al servidor Web provoca que el navegador Web actualice su contenido para mostrar la nueva información.

Aunque el rol del navegador Web es simplemente presentar información y recolectar datos de los usuarios, muchas nuevas tecnologías del lado cliente permiten hoy en día a los navegadores Web ejecutar código como JavaScript y soportar complementos que aumentan la experiencia del usuario. Tecnologías como AJAX permiten a los navegadores Web realizar refrescos parciales de la página comunicándose con el servidor Web. Estas tecnologías hacen la experiencia del usuario más dinámica e interactiva.

4.2. El protocolo HTTP

HTTP (*Hypertext Transfer Protocol*) es un protocolo de comunicaciones basado en texto que es usado para solicitar páginas Web a un servidor Web y enviar respuestas de retorno al navegador Web. El protocolo HTTP es el que gestiona la navegación por páginas web a través de Internet, y por tanto será el utilizado por las aplicaciones Web JEE.

HTTP es un protocolo orientado al intercambio de mensajes (o transacciones HTTP) y sigue el esquema solicitud-respuesta entre un cliente y un servidor. Al cliente que efectúa la solicitud (habitualmente un navegador web) se lo conoce como "user agent" (agente del usuario). A la información transmitida desde el servidor se la llama recurso y se la identifica mediante un localizador uniforme de recursos (URL). Los recursos pueden ser archivos, el resultado de la ejecución de un programa, una consulta a una base de datos, la traducción automática de un documento, etc.

Actualmente existen dos versiones de este protocolo: HTTP/1.1 y HTTP/2. Respecto a la versión 1.1, la versión 2 introduce las siguientes mejoras:

- Uso de una única conexión para transmisiones simultáneas.
- Compresión de cabeceras.
- El servicio «server push», el cual permite al servidor anticiparse al cliente y enviarle recursos antes de que sean solicitados.

4.2.1. Uso de URLs.

El punto de partida para el inicio de una transacción HTTP es la especificación de una URL (*Uniform Resource Locutor*) por parte de la aplicación cliente. Una URL es una cadena de texto que especifica generalmente un protocolo de red (http o https para solicitudes web), un dominio y la información del recurso solicitado. El formato general es el siguiente:

```
protocolo://dominio/informacion_de_recurso
```

Los protocolos más utilizados son:

- http: para realizar navegaciones por un sitio web. Por ejemplo, para acceder a la página inicial del sitio web de Oracle usaríamos.

  ```
  http://www.oracle.es
  ```

- https: para realizar navegaciones mediante un protocolo seguro por un sitio web. Por ejemplo, las entidades bancarias utilizan habitualmente seguridad SSL para acceder a sus servicios. En estos casos usaríamos:

  ```
  https://www.entidadbancaria.es/cuentas
  ```

- ftp: para acceder al sistema de ficheros del servidor. Algunos sitios web permiten compartir ficheros hospedados en alguna de sus carpetas. Por ejemplo, para acceder a un sitio de publicación de aplicaciones Linux podemos usar:

  ```
  ftp://sunsite.unc.edu/
  ```

- mailto: para acceder al servicio de correo electrónico. Por ejemplo, podemos crear un mensaje de correo electrónico de la siguiente forma.

  ```
  mailto:alguien@example.com?subject=asunto&cc=destinatario&body=mensaje
  ```

- file: para navegar por el sistema de fichero local. Por ejemplo, si queremos acceder a la carpeta raíz de la unidad C: usaríamos:

  ```
  file:///C:/
  ```

Para los protocolos http y https, el dominio es aquella parte de la URL que identifica el sitio web al que queremos hacer una solicitud. Existen dos formatos para especificar el dominio:

- Un formato donde se indica un nombre de host o dirección IP y un puerto.

  Cada dispositivo (host) que forma parte de una red tiene asignado un número que lo identifica de manera lógica y jerárquica. Este número se denomina dirección IP y está formado por 4 segmentos. Por ejemplo, la dirección 127.0.0.1 (o su alias localhost) está reservada para identificar al dispositivo local.

  Además, dentro de cada host pueden ejecutarse varias aplicaciones servidoras accesibles a través de la red. Cada host identifica sus aplicaciones servidoras mediante otro número denominado puerto. La mayoría de las aplicaciones servidoras comerciales utilizan un puerto fijo; por ejemplo, el servidor de base de datos de Oracle utiliza habitualmente el puerto 1521. Por tanto, para especificar un dominio en una URL se utilizará una sintaxis como:

  ```
  http://124.23.0.1:1521
  ```

• Un formato que especifica un nombre de dominio, como por ejemplo http://www.oracle.es.
Para hacer más amigables las URLs, el protocolo HTTP permite especificar un nombre de dominio en vez de una IP y puerto. El nombre de dominio queda registrado en un Sistema de Dominio de Nombres (DNS), donde está asociado con su host y puerto. El protocolo HTTP hace transparente el uso de DNS, y de esa forma es habitual nombres de dominio para navegar por Internet.

La última parte de una URL especifica el recurso solicitado e información adicional. La información del recurso consta de las siguientes partes:

• Una ruta, la cual identifica al recurso dentro de la organización física o virtual del sitio web. Por ejemplo, la ruta "informes/ventas.html" puede hacer referencia a un fichero llamado "ventas.html" ubicado dentro del directorio "informes". Pero como veremos en unidades posteriores, esta ruta puede estar asociada a una aplicación y no a un fichero. Si no se especifica la ruta, habitualmente los sitios web definen una ruta o página por defecto.

• Una cadena de consulta, que especifica datos adicionales. La información de esta cadena está estructurada normalmente con pares "clave=valor". Se utiliza la cadena de consulta como una técnica sencilla para pasar datos adicionales con la solicitud de un recurso. Por ejemplo, si queremos invocar la página de búsqueda de Google para buscar por el término "Oracle" utilizaríamos la siguiente URL:

https://www.google.es/**?q=Oracle**

El comienzo de la cadena de consulta se inicia siempre con el caracter '?'.

4.2.2. Transacciones HTTP mediante mensajes.

Una transacción HTTP está formada por un encabezado seguido, opcionalmente, por una línea en blanco y algún dato. El encabezado especificará cosas como la acción requerida del servidor, o el tipo de dato retornado, o el código de estado.

El uso de campos de encabezados enviados en las transacciones HTTP le da gran flexibilidad al protocolo. Estos campos permiten que se envíe información descriptiva en la transacción, permitiendo así la autenticación, cifrado e identificación de usuario.

Si se reciben líneas de encabezado del cliente, el servidor las coloca en variables de entorno, conocidas como variable CGI, con el prefijo HTTP_ seguido del nombre del encabezado. Cualquier carácter guion "-" del nombre del encabezado se convierte a caracteres "_". Ejemplos de estas variables CGI son HTTP_ACCEPT y HTTP_USER_AGENT:

• La cabecera HTTP_ACCEPT especifica los tipos de contenido (llamados también tipos MIME) de respuesta que el cliente aceptará, dados los encabezados HTTP. Los elementos de esta lista deben estar separados por una coma.

• La cabecera HTTP_USER_AGENT especifica el navegador que utiliza el cliente para realizar la solicitud. El formato general para esta variable es: software/versión biblioteca/versión.

4.2.3. Mensajes de solicitudes HTTP.

Los mensajes de solicitud HTTP son mensajes de texto formados por varias líneas. Su estructura más simple es la siguiente:

```
Línea de encabezado
Una o varias líneas de cabeceras
(Una línea en blanco para separar el contenido opcional)
Cuerpo del mensaje. Un contenido opcional que puede ocupar una o varias líneas.
```

La línea de encabezado consta de tres partes:

- Un verbo de método (GET, HEAD, POST, PUT, DELETE, OPTIONS o TRACE).
- La URI local del recurso requerido.
- La versión HTTP usada.

Un ejemplo de línea de cabecera de solicitud es:

```
GET  /informes/saldos/index.html HTTP/1.0
```

A continuación de la primera línea van líneas de encabezado. Cada línea de encabezado consta de un nombre, dos puntos y el valor. Por ejemplo:

```
Accept-Language: en-us,en;q=0.5
```

Se utilizan estas cabeceras para transmitir cualquier dato de interés desde la aplicación cliente al servidor.

Si después se las cabeceras incluimos un contenido del mensaje debe separarse con una línea en blanco.

4.2.4. Anatomía de una solicitud HTTP GET.

Una solicitud mediante el verbo GET añade la ruta del recurso y cualquier parámetro a la URL en la línea de solicitud.

Para una URL como "http://www.dominio.com/informes/saldos/index.html?n1=valor1&n2=valor2", la estructura de llamada puede ser como la siguiente:

GET /informes/saldos/index.html?n1=valor1&n2=valor2 **HTTP/1.1**	< línea de solicitud
Host: www.dominio.com	< cabeceras …
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/20030624 Netscape/7.1	<
Accept: text/xml,application/xml,application/xhtml+xml,text/html; q=0.9,text/plain; q=0.8,video/x-mng,image/png,image/jpeg,image/gif; q=0.2,*/*;q=0.1	<

En la URL de ejemplo, n1 y n2 son los parámetros y, valor1 y valor2 son los valores de dichos parámetros.

Las solicitudes GET se usan para solicitar recursos pasivos (como páginas HTML) o activos (como páginas JSP).

4.2.5. Anatomía de una solicitud HTTP POST.

Las solicitudes con el verbo POST están diseñadas para ser usadas por navegadores que necesitan realizar una petición más compleja sobre el servidor. Por ejemplo, si un usuario completa un formulario, la aplicación puede querer enviar todos los datos introducidos al servidor para que los almacene en una base de datos. Mediante HTTP POST los datos son enviados al servidor en el cuerpo del mensaje (o *payload*), y pueden ser tan largos como se precise.

La estructura del mensaje para una URL como "http://www.dominio.com/informes/saldos/index.html" que debe enviar dos parámetros (n1=valor1 y n2=valor2") puede ser como la siguiente:

POST /informes/saldos/index.html **HTTP/1.1**	< línea de solicitud
Host: www.dominio.com	< cabeceras...
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/20030624 Netscape/7.1	<
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain; q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1	<
Content-Type: application/x-www-form-urlencoded	<
Content-Length: 27	<
	< línea en blanco
n1=valor1&n2=valor2	< cuerpo mensaje

Cuando se añade un cuerpo al mensaje se debe incluir una cabecera Content-Type que indique el tipo de contenido, y una cabecera Content-Length que indique el tamaño del contenido.

4.2.6. Mensajes de respuestas HTTP.

Cuando un servidor Web responde a una petición de un navegador u otro cliente Web, el mensaje de respuesta consiste típicamente en una línea de estado, algunas cabeceras de respuesta, una línea en blanco, y un cuerpo del mensaje. Aquí tenemos un ejemplo mínimo con la línea de estado por defecto:

HTTP/1.1 200 OK	< Línea de estado
Set-Cookie: JSESSIONID=0AAB6C8DE415E2E5F307CF334BFCA0C1	< Cookie de sesión
Content-Type: text/html	< Cabeceras …
Content-Lenght: 36	<
	< Línea en blanco
<html><body>Hello World</body></html>	< Cuerpo del mensaje

La línea de estado consiste en la versión HTTP, un número entero que se interpreta como el código de estado, y un mensaje muy corto que corresponde con el código de estado. Los códigos de estado son tres dígitos agrupados tal como describe a continuación:

1xx	Información: solitud recibida, procesando.
2xx	Éxito: la acción fue recibida con éxito, atendida y aceptada.

3xx	Comando de redirección: una acción remota debe ser realizada para completar la solicitud.
4xx	Error del cliente: la solicitud tiene un error de sintaxis o el servidor no sabe cómo completar la solicitud.
5xx	Error del servidor: el servidor falló al completar una solicitud que parece ser válida.

Además de grupos de códigos de estado, HTTP/1.1 define códigos de estado únicos y sus razones. Una razón no es nada más que breve descripción del código de estado.
La siguiente tabla muestra los códigos de estado habituales y sus razones.

100	Continue	403	Forbidden
200	OK	404	Not Found
201	Created	407	Proxy Authentication Required
300	Multiple Choices	408	Request Time-out
301	Moved Permanently	413	Request Entity Too Large
302	Found	500	Internal Server Error
400	Bad Request	501	Not Implemented
401	Unauthorized		

La línea de cabecera Content-Type indica el tipo de recurso que será enviado al navegador cliente como parte de la respuesta. En el ejemplo, Content-Type: text/html, indica que se envía un archivo de texto HTML estático.
Un navegador puede administrar varios tipos de archivos, incluyendo documentos PDF, documentos Word, animaciones de Flash, etc.; y mostrarlos directamente en el navegador. Para ello debemos transmitir al navegador el tipo MIME apropiado que describa el contenido del flujo de respuesta. El tipo MIME es una descripción estandarizada de contenidos, independiente de la plataforma, similar al de las extensiones de archivos bajo Windows.
La cabecera Content-Length establece la longitud del cuerpo. De esta manera el navegador sabrá hasta dónde tiene que leer contenido para renderizarlo.

4.2.7. ¿Qué determina en un navegador web el verbo de solicitud HTTP?

Un navegador realiza una petición HTTP GET en los siguientes casos:

- Cuando se introduce directamente una URL en la barra de direcciones del navegador. Por ejemplo:
  http://www.google.es/
- Cuando se pulsa un enlace creado con la etiqueta HTML <a />. Por ejemplo:
  Página de búsqueda
- Cuando se envían datos mediante un formulario que no tiene asignado el atributo method, o lo tiene asignado al valor GET. Por ejemplo:

```
<form action="http://www.google.es" method="GET">
  Texto a buscar: <input type="text" name="q" />
  <input type="submit" value="buscar" />
</form>
```

Los formularios deben incluir habitualmente un botón de posteo (submit). Cuando se pulsa este botón se realiza una solicitud a dirección indicada en el atributo action de la etiqueta <form />.

Nota. Los formularios utilizan por defecto el método de solicitud HTTP GET.

Un navegador realiza una petición HTTP POST en los siguientes casos:

- Cuando se envían datos mediante un formulario que tiene asignado el atributo method al valor POST. Por ejemplo:

```
<form action="http://www.gestionempleado.es" method="POST">
  Nombe empleado: <input type="text" name="nombre" />
  <input type="submit" value="enviar" />
</form>
```

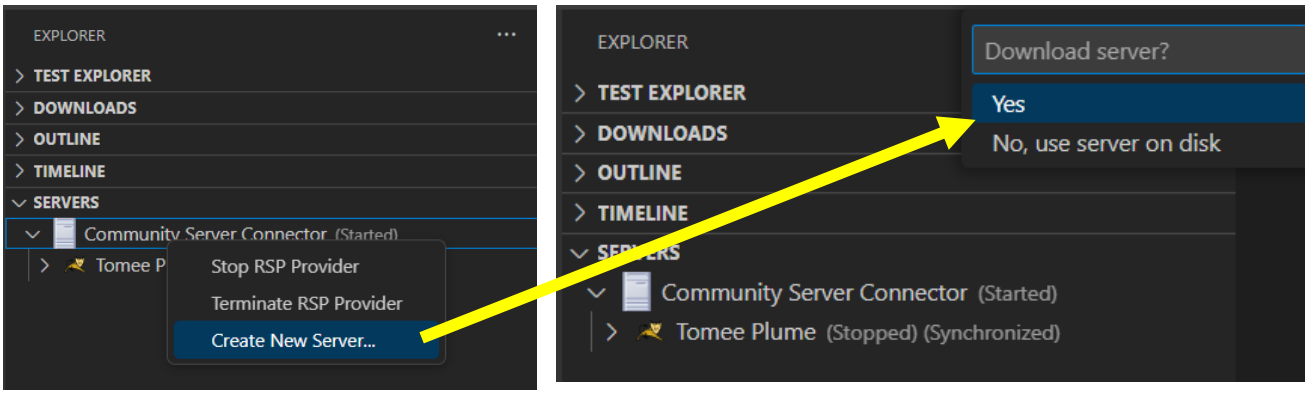
Otros métodos HTTP se aplican cuando se realizan solicitudes mediante código de servidor o código script. Veremos cómo hacer esto más adelante.

4.3. Aplicaciones de servidor web para Java.

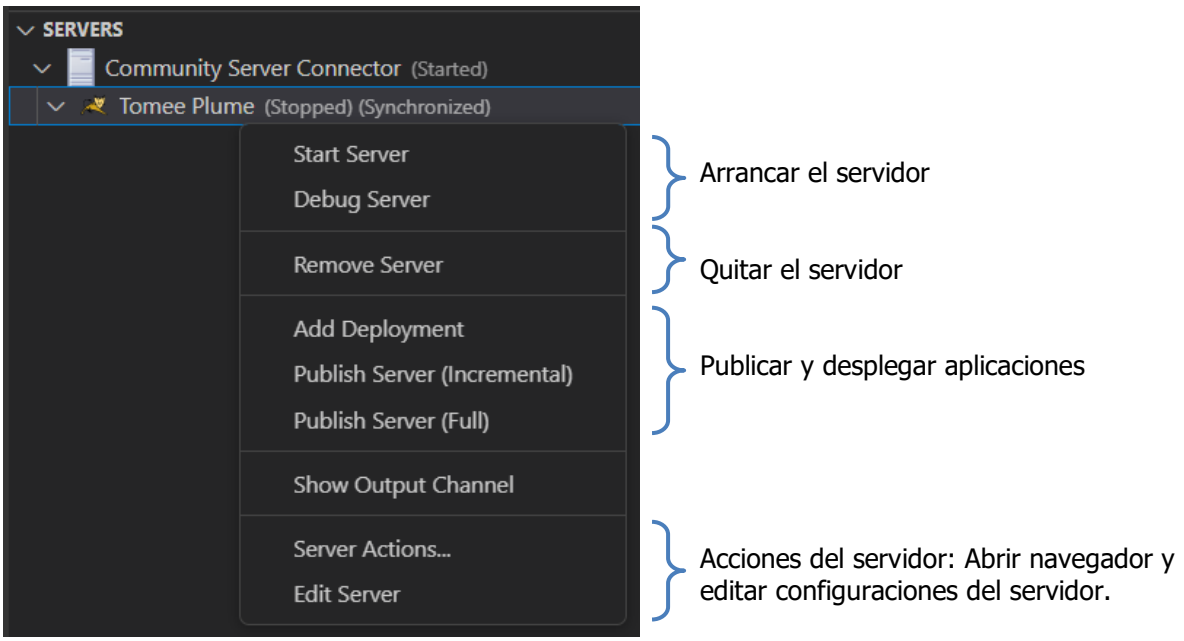
La tecnología Java para Web es soportada por varias aplicaciones servidoras Web. La principal es Apache Tomcat y su variante Apache Tomee.

Si usamos el entorno de desarrollo VS Code podemos instalar la extensión «Community Server Connectors». Esta extensión proporciona un protocolo de conexión a servidores web, con opciones para arrancarlos, pararlos, y publicar aplicaciones en servidores como Apache Felix, Karaf, Tomcat y Tomee.

Una vez instalada la extensión aparecerá una pestaña «SERVERS» con el nodo «Community Server Connector». En su menú contextual podemos pinchar en «Create New Server» para añadir un servidor descargándolo desde la nube o un servidor ya descargado.



Una vez añadido el servidor web en un nodo, podemos mostrar su menú contextual para:



4.3.1. Configuración del servidor Apache Tomcat.

Una vez descargado un servidor Apache Tomcat e instalado en una carpeta (haremos referencia a la misma con «$CATALINA_HOME») podemos configurarla.

En el fichero «$CATALINA_HOME/conf/tomcat-users.xml» se indican los usuarios y roles administrativos. Sería conveniente añadir los roles manager-gui, manager-script y admin-gui:

```
<?xml version="1.0" encoding="UTF-8"?>
<tomcat-users xmlns="http://tomcat.apache.org/xml"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://tomcat.apache.org/xml tomcat-users.xsd"
    version="1.0">
  <user username="tomcat" password="tomcat" roles="manager-gui,manager-script,admin,admin-gui" />
</tomcat-users>
```

4.3.2. Soporte para HTTP/2.

Apache Tomcat da soporte al protocolo HTTP/2 a través del protocolo seguro HTTPS. Para ello es necesario configurar seguridad TLS, y para esto necesitamos un certificado. Si todavía no disponemos de un certificado firmado, podemos utilizar uno auto-firmado para pruebas durante el desarrollo.

Hay varias herramientas para obtener certificados auto-firmados; una de estas herramientas es «keytool», disponible con el JDK de Java. Con el siguiente comando podremos crear un fichero de certificado auto-firmado:

```
> keytool -genkeypair -alias springboot -keyalg RSA -keysize 4096 -storetype JKS -keystore springboot.jks
-validity 3650 -storepass changeit
```

Donde:

-alias: es el nombre con el que haremos referencia al par de claves creado.
-keypass: es la clave con la que podremos acceder a la clave privada del par de claves creado.
-validity: es el tiempo de validez, en días, del certificado.
-keystore: es el nombre del fichero de almacén de claves que se creará.
-storepass: clave para acceder a nuestro almacén de claves.

Durante el proceso se nos pedirá el nombre y apellidos (si estamos generando un certificado para aplicar SSL en un servidor Web de Internet, deberíamos poner el nombre DNS, si el servidor está en intranet deberíamos poner el nombre de la máquina), el nombre de la unidad organizativa, el nombre de la organización, el nombre de la ciudad, el nombre de la provincia, y el código de dos letras del país. (Por simplificación podemos dejar todos estos campos vacíos pulsando la tecla ENTER y confirmando al final del proceso.)

Para ver el contenido actual del almacén podemos ejecutar el siguiente comando:

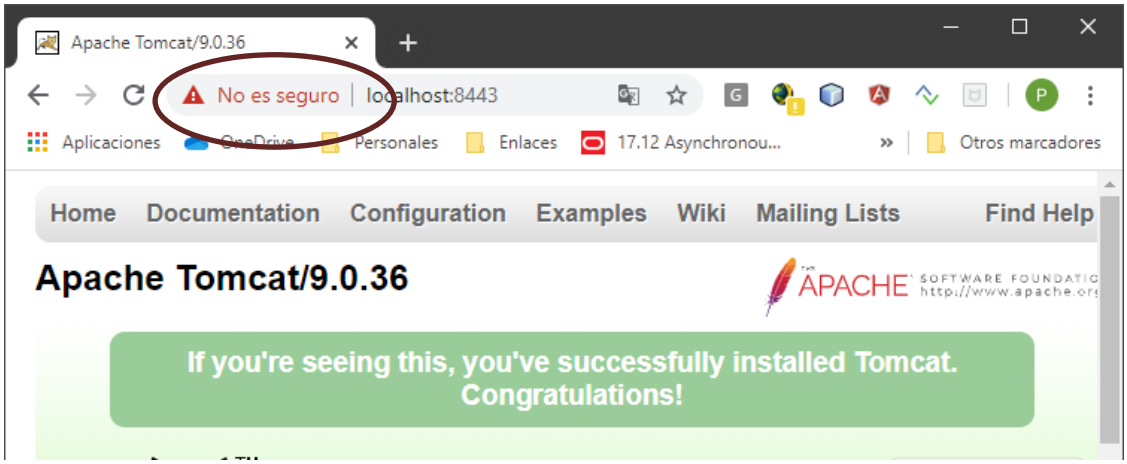
```
> keytool -list -keystore springboot.jks  -storepass changeit
Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

springboot, 12 ene 2023, PrivateKeyEntry,
Certificate fingerprint (SHA-256): B0:3E:51:A1:F2:B8:CE:22:BE:D1:FA:3A:F8:36:B2:E8:0E…

Warning:
The JKS keystore uses a proprietary format. It is recommended to migrate to PKCS12 which is an
industry standard format using "keytool -importkeystore -srckeystore springboot.jks -destkeystore
springboot.jks -deststoretype pkcs12".
```

Por la salida del comando, podemos ver que nuestro almacén sólo contiene el certificado que acabamos de crear. (A partir de este certificado auto-firmado podríamos generar un fichero CRS que se utilizará para obtener un certificado firmado de una autoridad oficial de certificación.)

Ahora debemos añadir a Tomcat, en el fichero «$CATALINA_HOME/conf/server.xml», el siguiente conector (especificando correctamente la ruta del fichero de certificado y la contraseña):

```
<Connector
  protocol="org.apache.coyote.http11.Http11NioProtocol"
  port="8443" maxThreads="200"
  scheme="https" secure="true" SSLEnabled="true"
  keystoreFile=" springboot.jks"
  keystorePass="changeit"
  clientAuth="false" sslProtocol="TLS">
  <UpgradeProtocol className="org.apache.coyote.http2.Http2Protocol" />
</Connector>
```

Para probar la nueva configuración debemos reiniciar el servidor y, en un navegador, ir a la dirección «https://localhost:8443» . Si se muestra la página de inicio de Tomcat es que se ha configurado con éxito TLS. Al usar un certificado auto-firmado, el navegador nos advertirá que estamos accediendo a un sitio no seguro.



5. El framework Spring MVC

Spring es un Framework Java diseñado para agilizar el desarrollo de aplicaciones empresariales, que da soporte a un contenedor de inversión de control (IoC) y es de código abierto. Se puede usar para la programación de aplicaciones web o de escritorio estándar, y cuenta con gran variedad de módulos que nos facilitan el trabajo, como: acceso a datos con JDBC, ORM, JPA, etc…

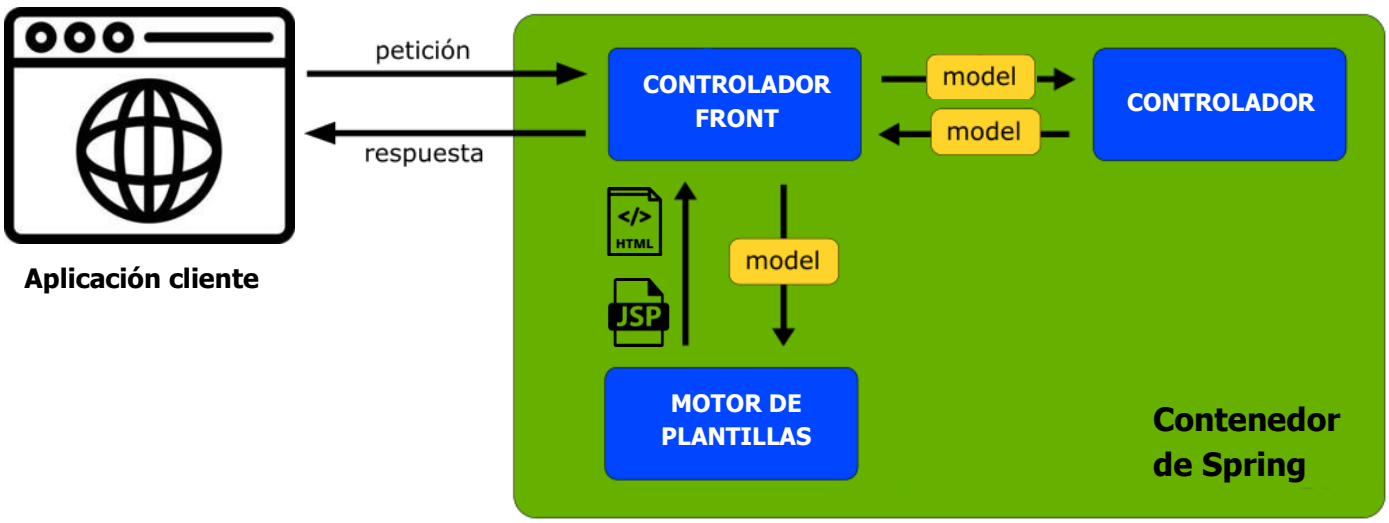
Spring Boot es un sub-proyecto de Spring que simplifica y agiliza el proceso de creación y desarrollo de aplicaciones web o de escritorio que utilicen el Framework Spring. La configuración requerida para iniciar una aplicación, de cualquier tipo, es mínima y automática. Spring Boot se auto-configura analizando el classpath, y dispone de un mecanismo de configuración bastante flexible, permitiéndonos personalizar la configuración siempre que lo necesitemos.

Las aplicaciones web en Spring sigue el patrón de diseño MVC. Este patrón de diseño se caracteriza por una fuerte separación de la lógica de negocio, código de acceso a datos y la interfaz de usuario dentro de Modelos, Controladores y Vistas.

5.1. Fundamentos de MVC.

El Modelo Vista Controlador (MVC) es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador; es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario. 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 y su posterior mantenimiento.

El siguiente diagrama ilustra sobre la arquitectura general de este patrón de diseño:



5.1.1. Modelos.

Cada sitio Web presenta información sobre los diversos tipos de objetos a los visitantes del sitio. Por ejemplo, un sitio web de una editorial puede presentar información sobre los libros y autores. Un libro incluye propiedades como el título, un resumen, y el número de páginas. Un autor puede tener propiedades tales como un nombre y una breve biografía. Cada libro está vinculado a uno o varios autores.

Cuando se escribe una página web MVC para un editor, podemos crear un modelo con una clase para libros y una clase para los autores. Estas clases del modelo incluirían las propiedades descritas y pueden incluir métodos tales como "comprar este libro" o "contactar al autor". Pero no debemos confundir las clases de Modelo con las clases de entidad. Las clases de modelo se configuran para pasar y recoger datos en las páginas web (las vistas), mientras que las clases de entidad se configuran para pasar y recoger datos con una base de datos.

5.1.2. Vistas.

Cada sitio web debe renderizar páginas HTML para que el navegador pueda mostrarlas. Esta renderización es completada por Vistas. Por ejemplo, en el sitio de publicación, una vista puede recuperar los datos del modelo del libro y renderizarlos en una página web para que el usuario pueda ver los detalles completos. En las aplicaciones MVC, las Vistas crean la interfaz de usuario.

En Spring existen varios motores de plantilla (Engine Templates) que se encargan de resolver y renderizar las vistas. Los más usado son **JSP**, que permite usar ficheros ".jsp" que contienen código HTML enriquecido con etiquetas propias, y **Thymeleaf**, que utiliza páginas ".html" enriqueciéndolas con atributos propios.

5.1.3. Controladores.

Cada sitio web debe interactuar con los usuarios cuando hacen clic en los botones y enlaces. Los Controladores responden a las acciones del usuario, construyen los objetos de modelo a partir de datos de la solicitud web y se los pasan a una vista, para que así se pueda renderizar una página web dinámica. Por ejemplo, en el sito de la editorial, cuando el usuario hace doble clic en un libro, él o ella esperan ver los detalles completos de ese libro. Un «Controlador Front» se encarga de aceptar las solicitudes del cliente y redirigirlas a un «Controlador» específico que carga el modelo de libro con el ID del libro, y lo pasa al motor de plantillas, que devuelve la Vista Detalles, lo que hace que una página web muestre el libro. Los Controladores implementan la lógica de entrada y atan los Modelos a las Vistas correctas.

5.2. Creación de aplicaciones web Spring Boot.

Spring Boot permite crear aplicaciones web con dos formatos.

1) Como aplicaciones JAR de escritorio.

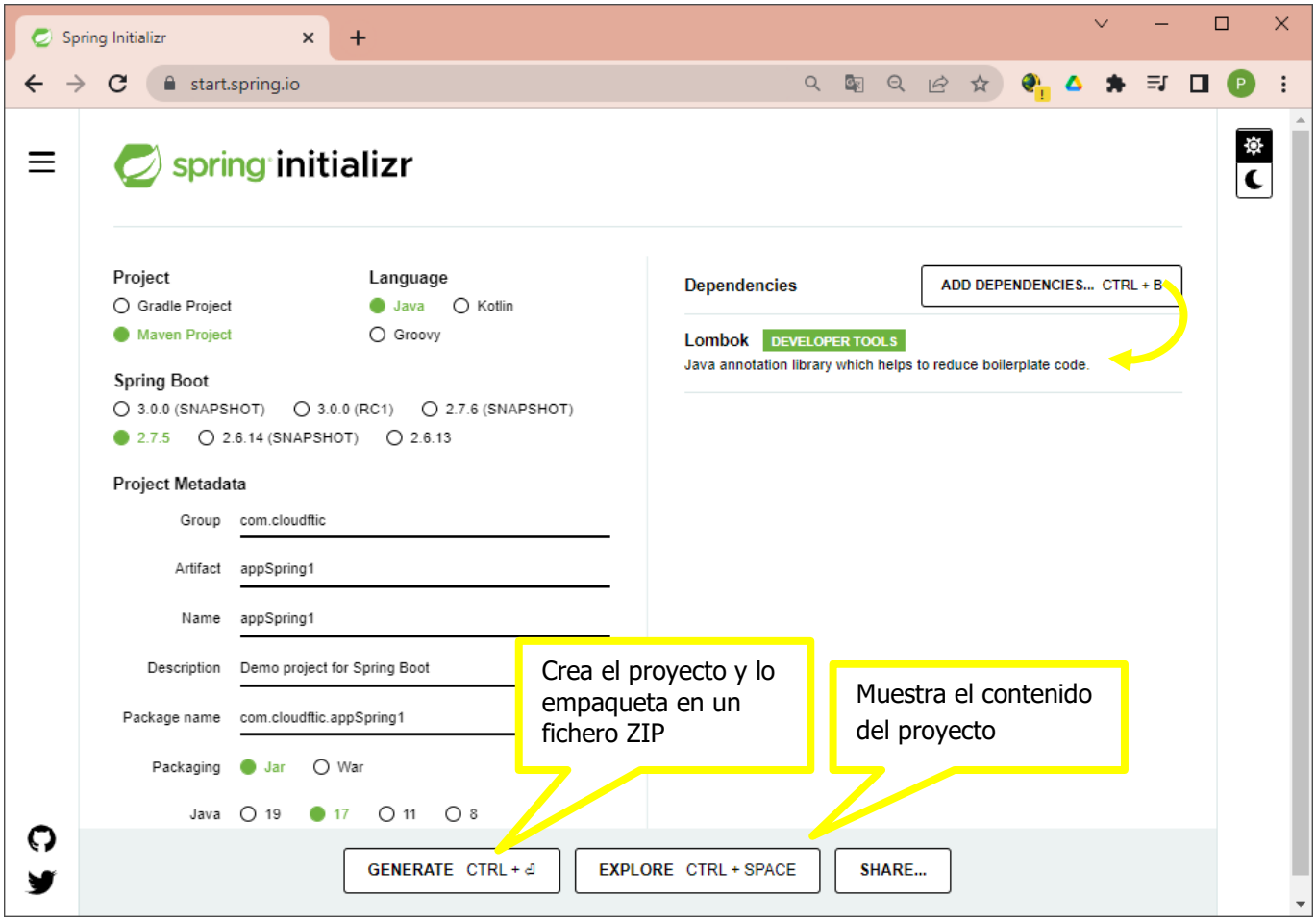
Estas aplicaciones incluyen el núcleo del servidor web Tomcat, de forma que al lanzar la aplicación se crea una instancia del servidor que gestiona el resto de los componentes de la aplicación.

2) Como aplicaciones web WAR.

Los archivos WAR son unos tipos de JAR especiales para empaquetar aplicaciones Web. Contienen un fichero de manifiesto para los metadatos, pero la estructura interna es diferente. Disponen de una carpeta llamada WEB-INF que contiene los archivos de la aplicación Web (aunque algunos pueden estar en la raíz también) y un archivo web.xml con información sobre la misma y con cuestiones como, qué URLs se corresponden con qué elementos de código, las dependencias o ciertas variables entre otras cosas.

Estos archivos WAR deben ser ejecutados por servidores web como Apache Tomcat, Glassfish, etc.

Para crear una aplicación Spring Boot con Maven podemos acceder a la página «https://start.spring.io/» para configurar un proyecto y descargarlo:

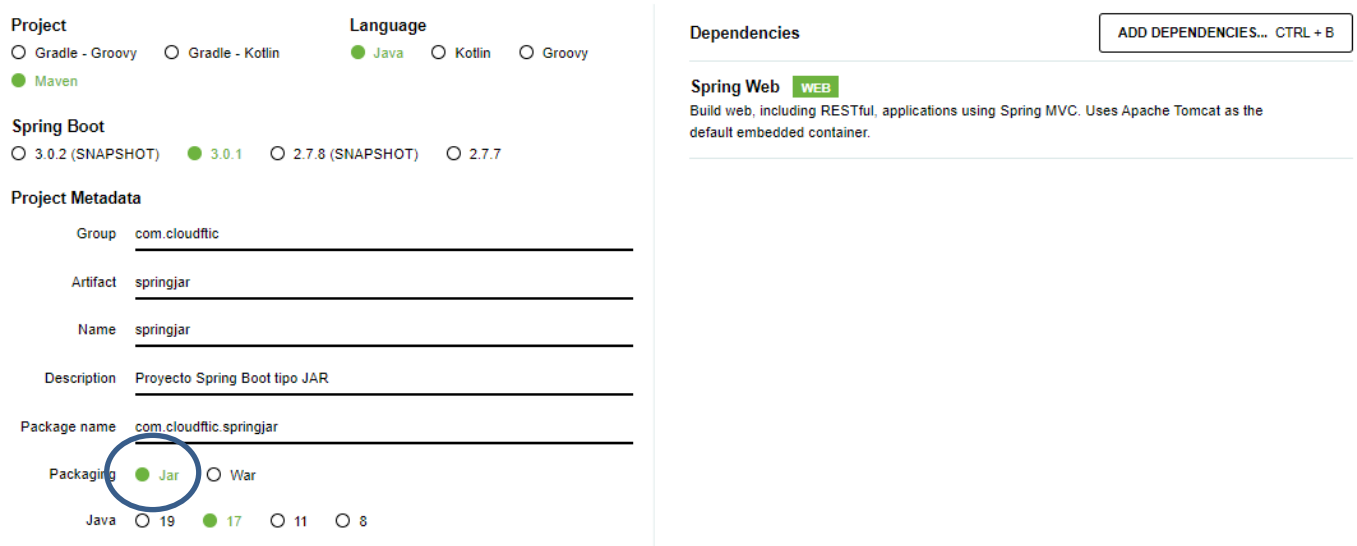


Tras seleccionar las opciones adecuadas hay que pulsar el botón [Generate] y se descargará un fichero ZIP que contiene el proyecto.

En VS Code podemos instalar la extensión «Spring Initializr Java Support» para hacer lo mismo. En la paleta de comandos debemos seleccionar «Spring Initializr: Create Maven Project» y seguir los pasos de configuración hasta crear el nuevo proyecto.

5.2.1. Cómo crear una aplicación JAR de tipo Web.

Para una aplicación de tipo JAR debemos seleccionar las siguientes opciones:



Project: Maven
Language: Java

Packaging: Jar

Y la dependencia mínima será:

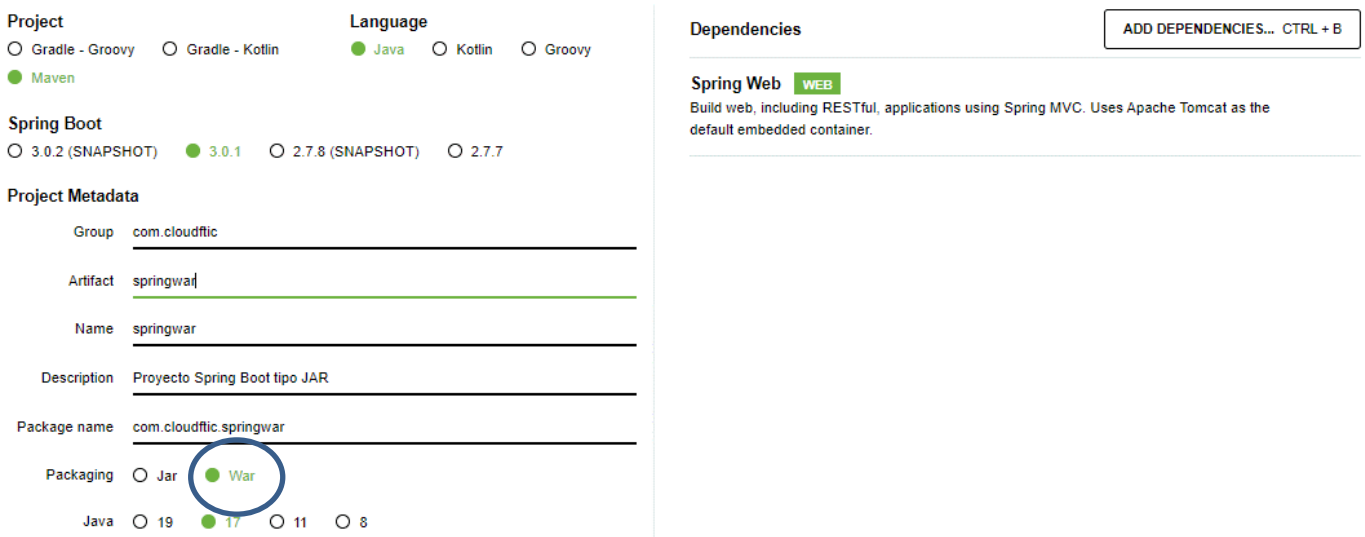
```
<!-- Spring Web -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Para probar la aplicación, añadiremos una página «index.html» en la carpeta «src/main/resources/static»:

Si compilamos y ejecutamos la aplicación se iniciará un servidor sencillo a través del puerto 8080, por defecto, esperando a recibir solicitudes. En un navegador web simplemente debemos ir a la dirección «http://localhost:8080/index.html» y se debe mostrar el texto ESTO FUNCIONA.

5.2.2. Cómo crear una aplicación WAR.

Las aplicaciones WAR se crean como las JAR, pero seleccionando el packaging War:



Project: Maven

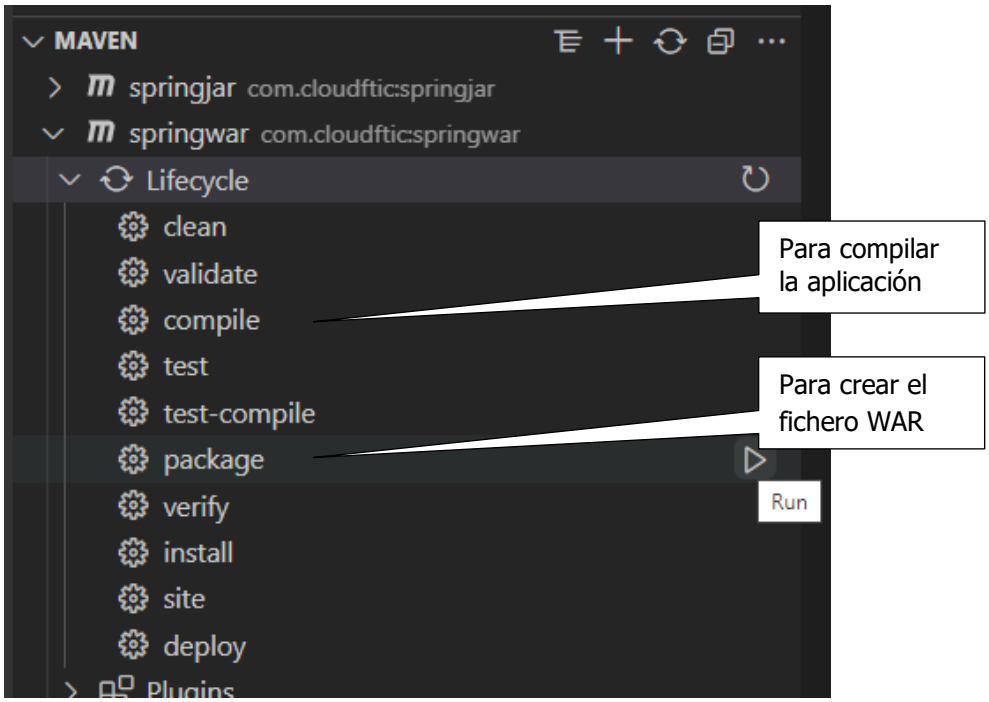
Language: Java

Packaging: War

Al seleccionar la opción War se añade automáticamente una nueva dependencia para poder ejecutar la aplicación en un servidor Apache Tomcat externo:

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

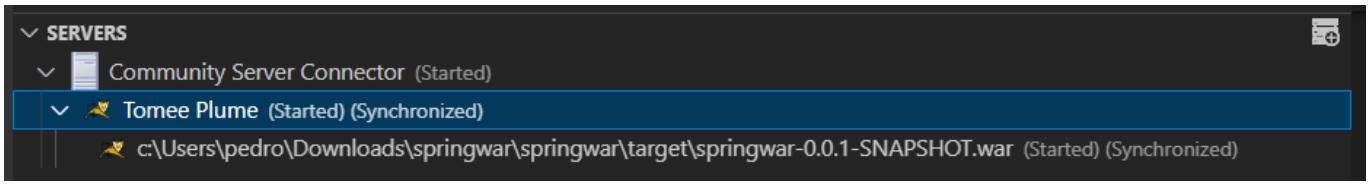
Para probar la aplicación también añadiremos la página «src/main/resources/static/index.html». Pero ahora solamente debemos compilar la aplicación y generar el WAR usando los comandos de Maven:



Con esto el fichero «springwar-0.0.1-SNAPSHOT.war» será creado en la carpeta «target». (El nombre del fichero WAR dependerá del nombre y la versión del proyecto Maven.)

Para probar la aplicación necesitamos tener instalado un servidor web compatible, como Apache Tomcat o Apache Tomee. Como se ha visto previamente en el apartado **1.3.**, si usamos el entorno de desarrollo VS Code podemos hacer uso de la extensión "Community Server Connectors".

Primero debemos arrancar el servidor con el menú «Start Server», y después desplegaremos nuestro fichero WAR. Podemos hacerlo de dos formas: usando el menú «Add Deployment» del servidor, o el menú «Run on Server» del propio fichero WAR. A continuación se mostrará un subnodo colgando del nodo del servidor:



Ahora podemos usar la opción «Server Actions» y «Show in Browser...» para mostrar la url «http://localhost:8080/springwar-0.0.1-SNAPSHOT/». Esta url debería mostrar el contenido del fichero «index.html», puesto que "index" es un nombre para mostrar la pagina por defecto de la aplicación.

5.2.3. Configuración de la aplicación.

El archivo «application.properties» permite configurar fácilmente algunos aspectos de la aplicación web. Ese archivo se encuentra en la carpeta «src/main/resources». Un ejemplo de propiedades en este archivo permite ver cómo cambiar el puerto y nombre de la aplicación web:

```
server.port = 9090
spring.application.name = springjar
```

Como alternativa al archivo «application.properties» se puede crear un archivo «application.yaml». En este archivo los nombres de propiedades se organizan jerárquicamente:

```
server:
  port: 9090
spring:
  application:
  name: springjar
```

La ruta raíz de la aplicación.

Las aplicaciones web de Spring Boot, por defecto usan la barra "/" para identificar el acceso a sus recursos. De forma que la url «http://localhost:9090/» realiza una solicitud sobre el recurso por defecto de la aplicación.

Podemos asignar una ruta raíz personalizada (un context path) para nuestra aplicación con la siguiente propiedad:

```
server.servlet.context-path=/cloudftic
```

Con lo cual, ahora tendremos que usar la url «http://localhost:9090/cloudftic» para realizar una solicitud sobre el recurso por defecto de la aplicación.

Habilitar HTTPS.

Las aplicaciones JAR de Spring Boot se configuran por defecto con el protocolo HTTP. Si queremos hablitar el protocolo seguro HTTPS debemos obtener primero un almacén de certificados y a continuación configurar las siguientes propiedades:

```
# Propiedades para habitar HTTPS con el certificado "springboot.jsk":
server.ssl.enabled=true
server.ssl.key-store: classpath:springboot.jks
server.ssl.key-store-password: changeit
server.ssl.key-store-type: JKS
server.ssl.key-alias: springboot
server.ssl.key-password: changeit
```

En este ejemplo se supone que el fichero «springboot.jks» se encuentra en la ruta del classpath de la aplicación. Si no es así, se debe proporcionar la ruta completa del fichero.

5.3. Uso de controladores.

Ahora veremos cómo crear un controlador específico para gestionar las solicitudes web realizadas desde aplicaciones cliente. Para dar soporte a la redirección de páginas HTML añadiremos al fichero POM la dependencia del motor de plantillas Thymeleaf:

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

Con Thymeleaf, debemos ubicar las páginas HTML en la carpeta «src/main/resources/templates».

Para que Spring Framework reconozca clases controladoras hay que crearlas en el paquete o un subpaquete de la clase principal. En otro caso tendremos que especificar los paquetes que contienen controladores decorando la clase principal:

```
@SpringBootApplication(scanBasePackages ={"controllers"} )
public class SpringjarApplication {
}
```

5.3.1. Definiendo un controlador con @Controller.

La anotación @Controller convierte a una clase en un controlador, mientras que la anotación @RequestMapping asocia este controlador con un segmento de recurso:

```
@Controller
@RequestMapping("/test")
public class TestController {

}
```

Aquí, el controlador TestController gestionará solicitudes que comiencen con «http://localhost:9090/test/». Si no se utiliza la anotación @RequestMapping sobre la clase, el controlador quedará asociado con el segmento «/» después del context path.

Ahora, cada método dentro de esta clase deberá quedar asociado con la uri de un recurso usando también la anotación @RequestMapping, o sus variantes @GetMapping, @PostMapping, @PutMapping y @DeleteMapping.

```
@RequestMapping(path="index", method = {RequestMethod.GET})
public String index() {
  return "index";
}
```

Un controlador puede gestionar solicitudes para cada acción HTTP; en este caso se crea un método de acción llamado index() para que sea el encargado de procesar las solicitudes HTTP GET «http://localhost:9090/test/index».

El string que retorna el método siempre será interpretado como un identificar de vista. En este caso, al usar el motor de plantillas Thymeleaf, se interpreta que es el nombre de un archivo con extensión ".html" ubicado en la carpeta «templates» del classpath.

5.3.2. Mapeado de solicitudes con @RequestMapping.

Se usan anotaciones @RequestMapping para asociar URIs a una clase o a un método concreto. Normalmente la anotación a nivel de clase especifica la ruta o patrón de la solicitud que gestionará el controlador, mientras que las anotaciones a nivel de método especificarán el método HTTP de la solicitud (GET, POST, PUT, DELETE, ...), o especificarán un parámetro de solicitud HTTP.

El siguiente ejemplo muestra un controlador que usa esta anotación:

```
@Controller
@RequestMapping("/clientes")
public class ClientesController {
  @GetMapping(path="/consulta")
  public String consulta() {
    return "listado_clientes";
  }
  @PostMapping(path="/insertar")
  public String insertar(@RequestBody Cliente cliente) {
    // Aquí instrucciones para procesar e insertar el objeto cliente
    return "redirect: consulta";
  }
  @PutMapping(path="/actualizar")
  public String actualizar((@RequestBody Cliente cliente) {
    // Aquí instrucciones para procesar y actualizar todos los datos cliente
    return "redirect: consulta";
  }
  @PatchMapping(path="/cambiar")
  public String cambiar(@RequestBody DatosParciales datosParciales) {
    // Aquí instrucciones para cambiar algunos datos de cliente
    return "redirect: consulta";
  }
  @DeleteMapping(path="/eliminar/{id}")
  public String eliminar(@PathVariable String id) {
    // Aquí instrucciones para eliminar un ciente de id dado
    return "redirect: consulta";
  }
  @RequestMapping(path="/operacion", method = RequestMethod.TRACE)
  public String trazar() {
    // Aquí instrucciones para realizar alguna operación de trazado
    return "redirect: consulta";
  }
}
```

El método consulta() se encargará de procesar solicitudes GET sobre la url «.../clientes/consulta», y retorna la vista «listado_clientes.html».

El método insertar() procesa solicitudes POST, y se encargará de insertar un nuevo cliente. El parámetro que recibe será instanciado a partir de los datos en el cuerpo de la solicitud. Con el valor de retorno "redirect: consulta" se indica que en vez de retornar el identificador de una vista, debe realizarse una redirección a la url relativa «clientes/consulta».

El método actualizar() procesa solicitudes PUT, y se encargará de actualizar todos los campos de cliente. Mientras que el método cambiar() procesa solicitudes PATCH, y se encargará de actualizar sólo determinados campos de cliente. Éste es el uso habitual de los verbos PUT y PATCH.

El método eliminar() procesa solicitudes DELETE, y se encargará de eliminar un cliente. Las urls que gestionar serán del tipo «.../clientes/eliminar/5», donde el último segmento será el ID de cliente. Con la sintaxis {id} y la anotación @PathVariable en el mapeado se asocia el valor de último segmento con el parámetro del método.

El método trazar() procesa solicitudes TRACE, y se encargará de realizar alguna operación de trazado.

5.3.3. Patrones URI para mapear variables de rutas.

Para acceder a partes de una URI solicitada en los métodos de acción se usan patrones URI en el parámetro path de la anotación @RequestMapping.

Un patrón URI es un string que contiene uno o más segmentos de ruta separados por /. Su propósito es definir una URI parametrizada. Por ejemplo, la plantilla URI «/cliente/eliminar/{idcliente}» contiene el identificador idcliente entre llaves. Si queremos asignar a esta variable el valor "C143", la solicitud será con la siguiente URI:

```
http://localhost:9090/clientes/eliminar/C143
```

Durante el proceso de la solicitud, la URL se comparará con el patrón URI esperado para extraer su colección de segmentos y variables de ruta.

Se usa la anotación @PathVariable sobre un parámetro del método de acción para indicar que el parámetro se debe enlazar con el valor de la variable del patrón. El siguiente código muestra cómo usar esta anotación:

```
@GettMapping(path="/cliente/eliminar/{idcliente}")
public String eliminar(@PathVariable("idcliente") String idCliente) {
  .......
}
```

El patrón URI «/cliente/eliminar/{idcliente}" especifica la variable de ruta llamada idcliente. Cuando el controlador asocia esta solicitud, el valor de idcliente es asignado al parámetro anotado con el mismo nombre.

Se pueden usar varias anotaciones @PathVariable para enlazar varias variables de la plantilla URI. Los parámetros de métodos decorados con la anotación @PathVariable pueden ser de cualquier tipo simple como int, long, Date, etc. Spring automáticamente los convertirá al tipo apropiado, y lanzará una TypeMismatchException si el tipo no es correcto. Pero además, podremos especificar el formato del segmento asociado con la variable de ruta usando expresiones regulares. Para este ejemplo, si el ID de cliente es un número entero podemos usar la siguiente expresión regular:

```
@GettMapping(path="/cliente/eliminar/{idcliente:\\d+}")
public String eliminar(@PathVariable("idcliente") int idCliente) {
  .......
}
```

Al garantizar que el valor casará con un número entero, podemos declarar el parámetro de método idCliente como de tipo int sin ningún problema. Si la URL contiene un valor que no case con la expresión regular no se asociará la ruta con este método.

5.3.4. Filtrado de rutas.

Los patrones URI también soportan opciones de filtrado para que la ruta case si se incluye n un determinado parámetro de solicitud. Por ejemplo, podemos forzar que sólo se gestione una URI que contenga la cadena de consulta "activo=true":

```
@GetMapping(path="/consulta", params="activo=true")
public String consulta() {
    .........
}
```

Por ejemplo, el método consulta() podría ser invocado por la URI: .../consulta?activo=true

Pero no por la URI: .../consulta?activo=false

Podemos simplemente evaluar que exista el parámetro de solicitud "activo" independientemente de su valor. O que no exista dicho parámetro de solicitud con "!activo". De forma similar podemos mapear la existencia de cabeceras con la solicitud:

```
@GetMapping(value = "/consulta", headers="content-type=text/*")
public String consulta(@RequestBody String contenido) {
    .........
}
```

En este ejemplo, el método consulta() solo será invocado si el tipo MIME del contenido del mensaje casa con el patrón text/*; como por ejemplo, "text/plain".

5.3.5. Soporte de varios tipos de argumentos y tipos de retorno.

Los métodos que están decorados con @RequestMapping pueden tener firmas muy flexibles permitiendo retornar varios tipos de datos, y permitiendo inyectar varios tipos de parámetros.

Tipo de parámetros.

Se admiten los siguientes tipos de parámetros en los métodos de un controlador:

• Objetos request y response (Servlet API). Se puede elegir un tipo específico, como ServletRequest/ServelResponse o HttpServletRequest/HttpServelResponse.
• Objetos de sesión (Servlet API) de tipo HttpSession. Este argumento nunca debe ser nulo.
• Objetos de tipo WebRequest o NativeWebRequest (del paquete org.springframework.web.context.request). Permiten acceso a parámetros de solicitud así como a atributos de solicitud o sesión.
• Objetos java.util.Locale, para la localización de la solicitud.
• Objetos java.io.InputStream/java.io.Reader, para poder acceder al contenido de la solicitud.
• Objetos java.io.OutputStream/java.io.Writer, para poder generar el contenido de la respuesta.
• Objetos java.security.Principal, el cual contiene al usuario autentificado.
• Parámetros anotados con @PathVariable, para acceder a las variable de patrón URI.
• Parámetros anotados con @RequestParam, para acceder a un parámetros de la cadena de consulta.
• Parámetros anotados con @RequestHeader, para acceder a una cabecera HTTP específica.
• Parámetros anotados con @RequestBody, para acceder al cuerpo de la solicitud HTTP.
• Objetos de tipo HttpEntity<?>, para acceder a cabeceras de la solicitud.
• Objetos java.util.Map/org.springframework.ui.Model/org.springframework.ui.ModelMap, para inyectar atributos al modelo implícito que se expone en las vistas.
• Objetos anotados con @ModelAttribute para enlazar objetos de modelo que habitualmente contendrán los datos del formulario procesado por la solicitud.
• Objetos org.springframework.validation.Errors/org.springframework.validation.BindingResult, para validar los datos recibidos en objeto de modelo. Estos parámetros deben ir a continuación de un parámetro correspondiente a un objeto de modelo.
• Objetos de estado org.springframework.web.bind.support.SessionStatus, que permiten la limpieza de atributos de sesión.

Tipos de retorno.

Los siguientes tipos de retorno son soportados por los métodos de un controlador:

• Un objeto ModelAndView, el cual permite especificar atributos con datos para inyectar en las vistas, y la propia vista.
• Un objeto Map para exponer un modelo de datos, con el nombre implícito de la vista determinado por un RequestToViewNameTranslator y el modelo implícito enriquecido con el objeto de comando y los resultados de las anotaciones @ModelAttribute.
• Un objeto View, con el modelo implícito determinado mediante el objeto de comando y las anotaciones @ModelAttribute. El método puede también enriquecer programáticamente el modelo de datos declarando un paramétro Model.
• Un String que, por defecto, es interpretado como el nombre lógico de la vista, con el modelo implícito determinado a través de objetos de comando y anotaciones @ModelAttribute. El método de acción puede también enriquecer programáticamente el modelo declarando un parámetro Model.
• Si se retorna void el método gestiona la respuesta por sí mismo (escribiendo el contenido de respuesta directamente, mediante un objeto ServletResponse/HttpServletResponse declarado como parámetro), o si el nombre de la vista está implícitamente determinado por un RequestToViewNameTranslator (sin declarar un argumento response en la firma del método).
• Si el método es anotado con @ResponseBody, el dato retornado es escrito al cuerpo de la respuesta HTTP. El valor retornado será convertido al tipo de argumento declarado usando HttpMessageConverters.
• Un objeto HttpEntity<?> o ResponseEntity<?>, para proporcionar acceso a las cabeceras de respuesta del servlet y a su contenido. El cuerpo será convertido para la respuesta usando HttpMessageConverters.
• Cualquier otro tipo es considerado como un atributo del modelo simple que será expuesto en la vista, usando el atributo especificado mediante @ModelAttribute al nivel del método (o el nombre del atributo por defecto basado en el nombre de la clase del tipo retornado). El modelo es implícitamente enriquecido con objetos de comando y el resultado de anotaciones @ModelAttribute.

Técnicas para retornar vistas.

Habitualmente se utilizan dos enfoques en los métodos que deben retornar vistas:

1) El método retorna un string con el nombre de la vista y declara un parámetro de tipo Model para inyectar atributos a la vista:

```
public String consulta(Model model) {
  model.addAttribute("un_dato", "un_valor");
  return "una_vista";
}
```

2) El método retorna un ModelAndView, que incluye la vista y los atributos:

```
public ModelAndView consulta() {
  return new ModelAndView("una_vista").addObject("un_dato", "un_valor");
}
```

Técnicas para retornar URLs de redirección.

Para los controladores definidos con @Controller, el Framework Spring espera que sus métodos retornen el nombre de una vista, pero también podremos retornar una URL para provocar redirecciones.

Existen dos formas de redirección:

1) Externas: en este caso el servidor envía como respuesta una cabécara "location" asignada con una url. Esta cabecera le dice al navegador cliente que se redirecciones a la url especificada.

2) Internas: se producen dentro del propio servidor sin que el navegador cliente sea consciente.

Para redirecciones externas se utiliza la sintaxis "redirect:url" cuando el método retorna un string. Si la url especificada comienza por / será relativa al contexto raíz de la aplicación, si comienza por http:// o https:// será una url absoluta.

Para redirecciones internas se utiliza la sintaxis "forward:url" cuando el método retorna un string. En este caso la url especificada debe ser sobre una ruta interna del servidor, no admite rutas absolutas.

Nota: Cuando se realiza una redirección interna debemos tener en cuenta que se conservan los datos de solicitud original, incluido el método de solicitud (GET, POST, …). Mientras que las redireccioens externas no conservan los datos de la solicitud original y provocan una nueva solicitud GET al redireccionar.

Cuando el método retorna un ModelAndView, tendremos que usar instancias de View para establecer redirecciones. La clase RedirectView permite especificar una redirección externa:

```
return new ModelAndView(new RedirectView(url));
```

5.3.6. Enlace de parámetros en los métodos de acción.

Se usa la anotación **@RequestParam** para enlazar parámetros de la solicitud (parámetros de la cadena de consulta o parámetros de formulario) a parámetros de métodos del controlador. El siguiente código muestra cómo usar esta anotación:

```
@GetMapping()
public void getParam(@RequestParam("id") int id, Writer writer) throws IOException {
  writer.write("ID = " + id);
}
```

Los parámetros que usan esta anotación son requeridos por defecto, pero podemos especificar que es opcional asignando el atributo required a false. Por ejemplo, @RequestParam(name="id", required=false, defaultValue = "1").

La anotación **@RequestHeader** indica que un parámetro del método debería ser enlazado al valor de una cabecera de la solicitud HTTP. Por ejemplo:

```
@GetMapping()
public void getHeader(@RequestHeader("User-Agent") String header, Writer writer) throws IOException {
  writer.write(header);
}
```

La anotación **@CookieValue** permite asociar un parámetro del método de acción con el valor de una cookie HTTP. Si consideramos la siguiente cookie recibida con la solicitud:

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

El siguiente código muestra cómo obtener este valor:

```
@GettMapping()
public void getCookie(@CookieValue("JSESSIONID") String cookie, Writer writer) throws IOException {
  writer.write(cookie);
}
```

La anotación **@RequestBody** indica que un parámetro del método de acción debería ser enlazado al valor del cuerpo de la solicitud HTTP. Por ejemplo:

```
@PostMapping()
public void getBody(@RequestBody String body, Writer writer) throws IOException {
```

```
    writer.write(body);
  }
```

5.3.7. Controladores REST.

La anotación **@ResponseBody** se utiliza sobre un método de controlador para indicar que lo retornado no debe ser interpretado como una vista, sino como datos que deben ser escritos en el cuerpo de la respuesta HTTP.

```
@Controller
public class TestController {
  @GetMapping("/horalocal")
  @ResponseBody
  public String horaLocal() {
    return LocalDateTime.now().toString();
  }
}
```

Si en un controlador todos sus métodos serán usados para retornar datos, podemos utilizar la anotación @RestController en vez de @Controller.

```
@RestController
public class TestController {
  @GetMapping("/horalocal")
  public String horaLocal() {
    return LocalDateTime.now().toString();
  }
}
```

En este caso, todos los métodos del controlador aplicarán automáticamente un @ResponseBody.

El término REST se ha convertido rápidamente en el estándar de facto para crear servicios web en la web porque son fáciles de crear y consumir. En ese sentido, podemos utilizar un RestController para gestionar operaciones con bases de datos usando los diversos verbos Http y las API de datos de Spring.

Como ejemplo, supongamos que disponemos de una clase de entidad Cliente y un repositorio JPA ClienteRepository. El siguiente controlador permite realizar las operaciones CRUD básicas:

```
@RestController
@RequestMapping("/clientes")
class ClienteController {
  @Autowired
  private ClienteRepository repo;

  /*
   * Retorna el cliente de ID solicitado, o una respuesta erronea con el
   * código de estado 404.
   */
  @GetMapping(path = "/{id:\\d+}")
  public ResponseEntity<Cliente> read(@PathVariable("id") int id) {
    Optional<Cliente> result = repo.findById(id);
    return result.isPresent() ? ResponseEntity.ok(result.get()) : ResponseEntity.notFound().build();
  }

  /*
   * Retorna todos los clientes
   */
  @GetMapping(path = "/")
  public List<Cliente> read() {
    return repo.findAll();
  }

  /*
   * Inserta un nuevo cliente
   */
  @PostMapping(path = "/")
  public void create(@RequestBody Cliente cliente) {
```

```
    repo.save(cliente);
  }

  /*
  * Actualiza todos los datos de un cliente.
  */
  @PutMapping(path = "/")
  public void update(@RequestBody Cliente cliente) throws IOException {
    if (! repo.existsById(cliente.getId())) {
      throw new IOException("El registro no existe.");
    }
    repo.save(cliente);
  }

  /*
  * Actualiza solo el nombre de un cliente.
  */
  @PatchMapping(path = "/update/nombre")
  public void updateNombre(@RequestBody Cliente cliente) throws IOException {
    Optional<Cliente> result = repo.findById(cliente.getId());
    if (! result.isPresent()) {
      throw new IOException("El registro no existe.");
    }
    Cliente c = result.get();
    c.setNombre(cliente.getNombre());
    repo.save(c);
  }

  /*
   * Elimina el cliente de ID dado.
   */
  @DeleteMapping(path = "/{id:\\d+}")
  public void delete(@PathVariable("id") int id) {
    repo.deleteById(id);
  }
}
```

5.4. Repositorios con Spring Data Rest.

El API Spring Data Rest combina las funcionalidades de Spring Data JPA para crear repositorios con las de un controlador Rest.

La dependencia para instalarlo es:

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

Si hemos configurado el contexto para JPA y vamos a trabajar con una clase de entidad Cliente:

```
@Entity
@Data
public class Cliente {
  @Id
  private int id;
  private String nombre;
}
```

Entonces crearemos un repositorio que a la vez será un RestController de la siguiente manera:

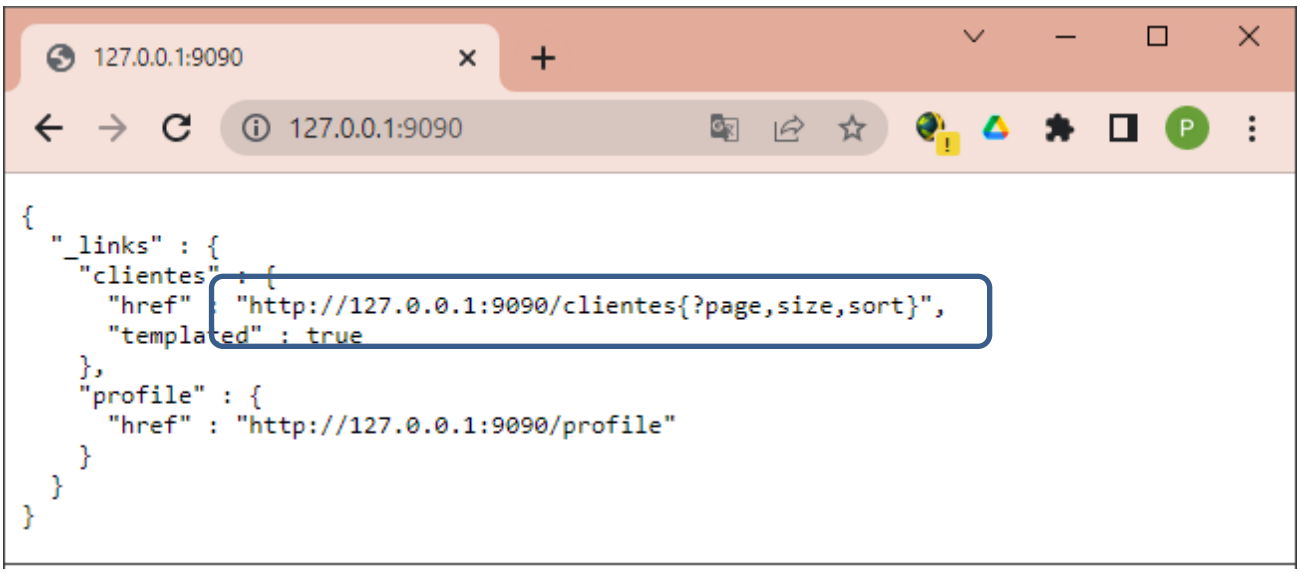
```
@RepositoryRestResource(path="clientes", collectionResourceRel = "clientes")
public interface ClienteRepository extends JpaRepository<Usuario, Integer> {
  List<Cliente> findByNombre(@Param("nombre") String nombre);
}
```

El parámetro path establece la uri asociada al RestController. El parámetro collectionResourceRel establece el nombre del atributo donde quedarán los resultados de consultas, tal como veremos a continuación.
Si es necesario, debemos informar a Spring dónde se ubican los repositorios en la clase principal:

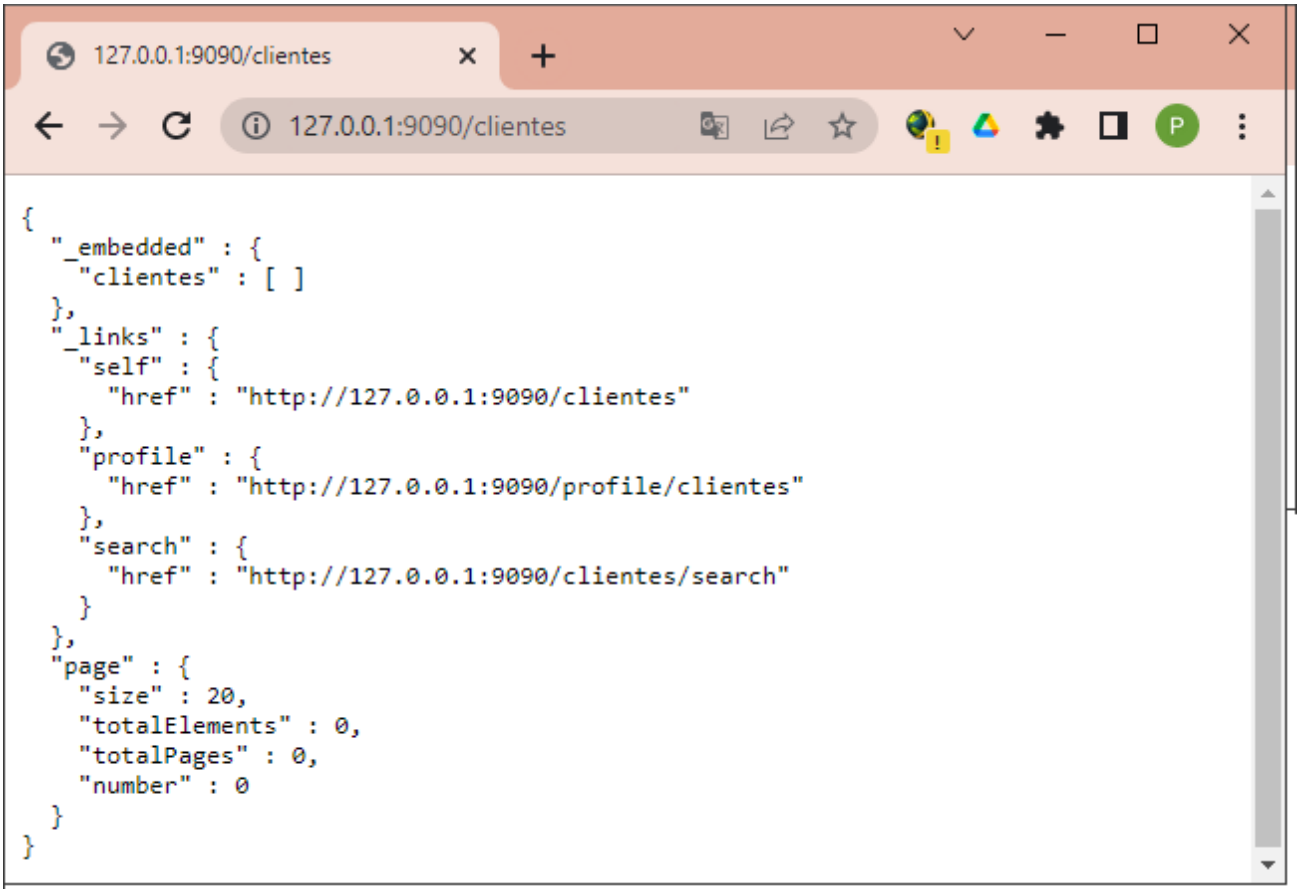
```
@EnableJpaRepositories
@SpringBootApplication(scanBasePackages = "com.cloudftic.repositories" )
public class SpringjarApplication {
  public static void main(String[] args) {
    SpringApplication.run(SpringjarApplication.class, args);
  }
}
```

5.4.1. Consultas sobre el RestRepository.

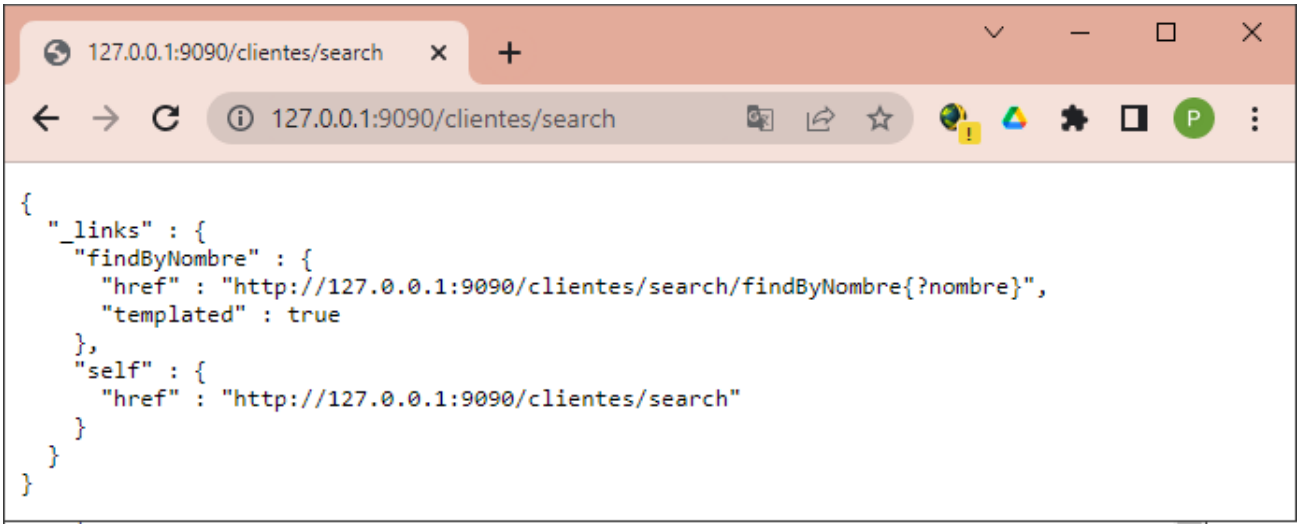
Si la url base no está asociada con ningún otro recuso, y la solicitamos, podemos realizar la siguiente consulta en un navegador:



Spring Data Rest se adhiere al principio de HATEOAS (Hypertext as the Engine of Application State). En términos generales, el principio implica que la API debe guiar al cliente a través de la aplicación devolviendo información relevante sobre los próximos pasos potenciales, junto con cada respuesta. Por eso, en la respuesta se añaden enlaces que informan al cliente.
El enlace enmarcado en la imagen previa nos permite acceder a los clientes almacenados. Aunque en este momento no tenemos ninguno. La consulta nos permitirá especificar opciones para tamaño de paginación, número de página y ordenación.



En la parte inferior se observan los datos de paginación como el tamaño de página, el total de elementos, el total de páginas y el número de la página actual.
Podemos ver también una lista de todos los enlaces de consulta creados por la interfaz ClientesRepository. En este ejemplo únicamente el enlace para la consulta findByNombre(@Param("nombre") String nombre), de forma que el valor asignado en la anotación @Param se usará como parámetro de consulta.



El enlace «http://127.0.0.1:9090/clientes/search/findByNombre{?nombre}» nos permite buscar un cliente por su nombre, como por ejemplo:

http://127.0.0.1:9090/clientes/search/findByNombre?nombre=Juan

de forma que podríamos obtener:

```
{
  "_embedded" : {
    "clientes" : [ {
```

```
        "nombre" : "Juan",
        "_links" : {
          "self" : {"href" : "http://127.0.0.1:9090/clientes/1"},
          "cliente" : {"href" : "http://127.0.0.1:9090/clientes/1"}
        }
      } ]
    },
    "_links" : {
      "self" : {"href" : "http://127.0.0.1:9090/clientes/search/findByNombre?nombre=Juan"}
    }
  }
```

Como el resultado está en formato JSON, sólo debemos parsearlo y acceder al atributo clientes para obtener una lista con el resultado. Como vemos, el resultado no incluye el ID de cliente como atributo, sino que es incluido como segmento final de las urls asignadas en lo atributos self y cliente. Sin embargo, siempre podremos realizar, por defecto, consultas filtrando por el ID:

```
http://127.0.0.1:9090/clientes/1
```

5.4.2. Operaciones de actualización.

El RestRepository también ofrece soporte para operaciones POST, PUT y DELETE.

Una operación POST permitirá insertar un nuevo registro, cuyos datos deben ser incluidos en formato Json en el cuerpo de la solicitud.

```
POST: http://127.0.0.1:9090/clientes
BODY: {"id": 3, "nombre": "Miguel"}
```

Una operación PUT o PATCH permitirán actualizar los datos proporcionados para un registro. En el segmento final de la ruta se debe indicar el ID del registro, y en el cuerpo un objeto Json con los datos para actualizar:

```
PUT: http://127.0.0.1:9090/clientes/3
BODY: {"nombre": "Miguel Ángel"}
```

Una operación DELETE permitirá eliminar un registro. En el segmento final de la ruta se debe indicar el ID del registro:

```
DELETE: http://127.0.0.1:9090/clientes/3
```

5.5. Spring HATEOAS.

Los servicios REST forman parte de los paradigmas de programación más importantes en el desarrollo de aplicaciones web. Este principio de arquitectura tiene como objetivo principal adaptar las aplicaciones web lo mejor posible a los requisitos de la Web moderna. Los resultados de este esfuerzo, sin embargo, no son siempre realmente REST (RESTful), porque a menudo ciertos principios o propiedades como HATEOAS no se implementan correctamente.

El término HATEOAS es el acrónimo de *Hypermedia as the engine of applicacion state* (en castellano, hipermedia como el motor del estado de la aplicación) y describe una de las propiedades más significativas de REST: el de que debe ofrecer una interfaz universal, de forma que el cliente pueda moverse por la aplicación web únicamente siguiendo a los identificadores únicos URI en formato hipermedia. Cuando se aplica este principio, el cliente, aparte de una comprensión básica de los hipermedias, no necesita más información para poder interactuar con la aplicación y el servidor.

Una aplicación que siga el principio HATEOAS permite que la interfaz de un servicio REST pueda modificarse siempre que se requiera, lo que constituye una ventaja fundamental de esta arquitectura frente a otras.

Como se ha visto Spring Data Rest sigue este principio, y si queremos aplicarlo en controladores REST personalizados podemos hacer uso del API Spring HATEOAS. Para utilizarlo debemos agregar la siguiente dependencia:

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

5.5.1. Añadiendo compatibilidad con HATEOAS.

Spring HATEOAS ofrece tres abstracciones para crear las URI que añadiremos a los datos:

- La superclase RepresentationModel, se usa para dar soporte a las clases de recursos de datos.
- La clase Link, se usa para crear enlaces simples.

- La clase WebMvcLinkBuilder, para crear enlaces a partir de controladores y métodos.

Añadiendo soporte hipermedia a un recurso.

Se utiliza la clase genérica org.springframework.hateoas.RepresentationModel para que herede de ella una clase de recursos de datos:

```
@Data
public class Book extends RepresentationModel<Book> {
  private int id;
  private String title;
}
```

La clase RepresentationModel proporciona mediante herencia un método add(). Una vez creado un enlace podemos usar esté método para añadirlo a la respuesta de REST.

Creación de enlaces simples.

La clase org.springframework.hateoas.Link permite crear un enlace que almacene los metadatos. La forma más simple es proporcionar un enlace fijo manualmente:

```
Link link = Link.of("http://localhost:8080/books/10");

var book1 = new Book(10, "A").add(link);
```

El objeto Link aplica la sintaxis de enlace Atom y añade un atributo que identifica la relación con el recurso de datos y el atributo href, que es el propio enlace.

Así es como se verá el recurso Book ahora que contiene el nuevo vínculo:

```
{
  "id": 10,
  "title": "A",
  "_links": {
    "self": {
      "href": "http://localhost:8080/books/10"
    }
  }
}
```

La URI asociada a la respuesta se califica como un vínculo propio ("self").

También se puede crear un enlace que actúe de plantilla:

```
var link = Link.of("http://localhost:8080/{rest}/{?id}");
// Se asigna valor a los parámetros:
var link1 = link.expand("books", 10);
// o bien se usa un mapa para asociar cada parámetro:
// var link1 = link.expand(Map.of("rest", "book", "id", 10));
var book1 = new Book(10, "A").add(link1);
```

Obteniendo el siguiente enlace: "http://localhost:8080/book/?id=10"

En la plantilla de la URL se establecen parámetros de ruta con la sintaxis {param}, y parámetros de la cadena de consulta con la sintaxis {?param}.

Creando mejores enlaces.

Con la clase fabricadora WebMvcLinkBuilder se simplifica la creación de URIs. Ofrece métodos estáticos que podemos encadenar para ir formando un enlace. Los métodos estáticos que ofrece son:

.linkTo(class), para inspeccionar la clase de un controlador y obtener su URI raíz.

.slash(value), para agregar un valor de ruta al enlace.

.withRel(name), para asignar el nombre del atributo de relación.

.withSelfMethod(), para califica la relación como un autoenlace (con el nombre "self").

Veamos un ejemplo de uso:

```
@RestController
@RequestMapping("/books")
public class BookController {
  @GetMapping("{bookId}")
  public Book getBook(@PathVariable int bookId) {
    var book = new Book(bookId, "A");
    var link = WebMvcLinkBuilder
```

```
                .linkTo(BookController.class)
                .slash(book.getId())
                .withSelfRel();
        return book.add(link);
      }
    }
```

De forma que se obtiene:

```
{
  "id": 10,
  "title": "A",
  "_links": {
    "self": {
      "href": "http://localhost:8080/books/10"
    }
  }
}
```

Añadiendo relaciones.

En la sección anterior, hemos mostrado una relación autorreferencial. Sin embargo, los sistemas más complejos pueden implicar también otras relaciones.

Por ejemplo, un libro (book) puede tener una relación con sus capítulos (chapter). Vamos a modelar la clase Chapter como un recurso de datos:

```
@Data
@AllArgsConstructor
public class Chapter extends RepresentationModel<Chapter> {
  private int id;
  private String title;
  private String content;
}
```

Ahora podemos agregar al controlador BookController un método que devuelva un capítulo por id (getChapter), y un método que devuelva los capítulos de un libro concreto (getChaptersOfBook):

```
@RestController
@RequestMapping("/books")
public class BookController {
  ...

  @GetMapping(value = "/chapter/{chapterId}")
  public Chapter getChapter(@PathVariable int chapterId) {
    return new Chapter(chapterId, "A", "B");
  }

  @GetMapping(value = "/{bookId}/chapters")
  public CollectionModel<Chapter> getChaptersOfBook(@PathVariable int bookId) {
    // El método inventado findChapters() recupera los capítulos de un libro:
    List<Chapter> chapters = findChapters(bookId);

    // Añadimos enlaces a cada capítulo basados en el método getChapter()
    chapters.forEach(chapter -> {
        var link = WebMvcLinkBuilder.linkTo(
           WebMvcLinkBuilder.methodOn(BookController.class).getChapter(chapter.getId())
        ).withSelfRel();
        chapter.add(link);
      });

    // Añadimos el enlace propio a este método:
    var link = WebMvcLinkBuilder.linkTo(
           WebMvcLinkBuilder.methodOn(BookController.class).getChaptersOfBook(bookId)
        ).withSelfRel();
```

```
        return CollectionModel.of(chapters, link);
      }
    }
```

El método getChaptersOfBook() retorna un objeto CollectionModel para cumplir con el estándar, así como un enlace "_self" para cada uno de los capítulos y la lista completa:

```
    {
      "_embedded": {
        "chapterList": [
          {
            "id": 1,
            "title": "Uno",
            "content": "abc",
            "_links": {
              "self": {
                "href": "http://localhost:8080/books/chapter/1"
              }
            }
          }
        ]
      },
      "_links": {
        "self": {
          "href": "http://localhost:8080/books/1/chapters"
        }
      }
    }
```

El método WebMvcLinkBuilder.methodOn() permite obtener la asignación de métodos de controlador realizando una invocación ficticia del método.

5.6. Gestión de errores.

Cuando un método de controlador lanza una excepción no gestionada, provoca una respuesta inmediata al cliente con el código HTT 500 (error interno del servidor).

Veremos primero cómo el framework Spring permite automatizar la gestión de excepciones.

5.6.1. Uso de la anotación @ExceptionHandler a nivel de controlador.

Uno de los enfoques iniciales de Spring para gestionar excepciones es declarar en el controlador un método de gestión de errores anotado con @ExceptionHandler.

```
    public class AlgunController{
      ...
      @ExceptionHandler({Exception.class})
      public void handleException() {
        ...
      }
    }
```

Esto permite que cualquier (o determinadas excepciones) lanzadas por métodos de solicitud sean derivados a la ejecución del método de gestión de errores.

El método handleException() puede ser personalizado, al igual que los método de solicitud, para recibir determinados argumentos o retornar datos o vistas.

Este enfoque tiene un inconveniente importante: el método anotado con @ExceptionHandler solo está activo para su controlador, no globalmente para toda la aplicación. Por supuesto, agregar esto a cada controlador hace que no sea adecuado para un mecanismo general de manejo de excepciones.

Podemos solucionar esta limitación haciendo que todos los controladores hereden de una clase de controlador base que proporcione el método de gestión de errores.

5.6.2. Gestión global de errores con HandlerExceptionResolver.

A continuación, veremos otra forma de resolver el problema del manejo de excepciones, una que sea global y no incluya ningún cambio en los artefactos existentes, como los controladores.

Debemos definir una subclase que herede de HandlerExceptionResolver y dejarla disponible como un Bean. Pero antes veamos las implementaciones predefinidas de HandlerExceptionResolver en Spring:

1) ExceptionHandlerExceptionResolver

Se introdujo en Spring 3.1 y es la que usa por defecto el mecanismo @ExceptionHandler.

2) DefaultHandlerExceptionResolver

Se introdujo en Spring 3.0 y es la que resuelve las excepciones estándar de Spring a los códigos de estado HTTP correspondientes, es decir, los códigos de estado de error de cliente 4xx y error de servidor 5xx. Pero no añade nada al cuerpo de la respuesta.

3) ResponseStatusExceptionResolver

Esta clase se encarga de utilizar la anotación @ResponseStatus disponible en excepciones personalizadas y asignar estas excepciones a códigos de estado HTTP. Una excepción personalizada puede verse así:

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class MyResourceNotFoundException extends RuntimeException {
  ...
}
```

Si lanzamos este tipo de excepción en un método, Spring retornará una respuesta con el código de estado 404 (correspondiente a NOT FOUND). Pero, al igual que DefaultHandlerExceptionResolver, este solucionador está limitado en la forma en que trata el cuerpo de la respuesta: asigna el código de estado en la respuesta, pero el cuerpo sigue siendo nulo.

Si estos manejadores predefinidos no bastan para nuestras necesidades implementaremos nuestra propia solución. Por ejemplo:

```
@Component
public class MiResponseExceptionResolver extends DefaultHandlerExceptionResolver {
  @Override
  protected ModelAndView doResolveException(
        HttpServletRequest request, HttpServletResponse response,
        Object handler, Exception ex) {
    try {
      response.setStatus(HttpStatus.CONFLICT.value());
      response.getWriter().append("Se produjo un error.").close();
    } catch (IOException e) {
      e.printStackTrace();
    }
    return new ModelAndView();
  }
}
```

Reescribiendo el método doResolveException() podemos personalizar totalmente la respuesta usando el objeto de bajo nivel de tipo HttpServelResponse, además de disponer de la excepción que produjo el error. El objeto de tipo ModelAndView retornado también permite especificar una vista que podemos personalizar.

5.6.3. Gestión de errores con @ControllerAdvice.

Para la gestión global de excepciones es preferible usar la anotación @ControllerAdvice. Esto habilita un mecanismo que rompe con el modelo MVC y hace uso de objetos ResponseEntity junto con la seguridad y flexibilidad de tipos de @ExceptionHandler:

```
@ControllerAdvice
public class RestResponseEntityExceptionHandler {
  @ExceptionHandler(value={ MiExcepcionNoFound.class, IllegalStateException.class})
  public ResponseEntity<Object> handleConflict(RuntimeException ex, WebRequest request) {
    String bodyResponse = "Error: "+ ex;
    return ResponseEntity.internalServerError().body(bodyResponse);
  }
}
```

La anotación @ControllerAdvice permite consolidar nuestros múltiples @ExceptionHandler dispersos de antes en un solo componente de manejo de errores global.

Debemos procurar que los tipos de excepción especificados en @ExceptionHandler casen con la excepción utilizada como argumento del método.

5.6.4. Gestión de errores con ResponseStatusException.

Spring 5 introdujo la clase ResponseStatusException como tipo de excepción que admite como argumento un objeto HttpStatus y, opcionalmente, un motivo y una causa. Por ejemplo:

```
@GetMapping(path = "/test")
public String test() {
  // ...
  throw new ResponseStatusException(
    HttpStatusCode.valueOf(HttpStatus.BAD_REQUEST.value()),
    "Razón: mala solicitud",
    new IOException("La causa del error"));
}
```

Esta técnica es la mejor para la creación de prototipos, ya que podemos implementar una solución básica bastante rápido. Permite que un mismo tipo de excepción dé lugar a varias respuestas diferentes. Esto reduce el acoplamiento estrecho en comparación con @ExceptionHandler, y no tendremos que crear tantas clases de excepción personalizadas.

De esta forma tenemos mayor control sobre la gestión de excepciones, ya que las éstas se pueden crear mediante programación.

Por el contrario, no permite una forma unificada de gestionar wxcepciones: es más difícil hacer cumplir algunas convenciones de toda la aplicación en comparación con @ControllerAdvice, que proporciona un enfoque global.

También debemos tener en cuenta que es posible combinar diferentes enfoques dentro de una aplicación. Por ejemplo, podemos implementar un @ControllerAdvice globalmente pero también ResponseStatusException localmente.

5.6.5. Gestión de errores en Spring Boot.

Spring Boot proporciona una implementación de ErrorController para gestionar los errores de manera unificada. Para navegadores que realizan solicitudes HTML sirve una página de error de etiqueta blanca, y para clientes que realizan solicitudes REST envía una respuesta Json como la siguiente:

```
{
  "timestamp": "2023-01-17T16:12:45.977+0000",
  "status": 500,
  "error": "Internal Server Error",
  "message": "Error processing the request!",
  "path": "/my-endpoint-with-exceptions"
}
```

Si usamos como motor de plantillas Tymeleaf, también podemos crear una página «error.html» en la carpeta «templates», y será utilizada automáticamente por el gestor de errores. En esta página podemos recuperar toda la información de error y mostrarla de manera adecuada al usuario. A continuación se muestra un ejemplo:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
</head>
<body>
  <h1>Página de error de Thymeleaf</h1>
  <p>Fecha: <label th:text="${timestamp}"></label></p>
  <p>Ruta: <label th:text="${path}"></label></p>
  <p>Error: <label th:text="${error}"></label></p>
  <p>Código de error: <label th:text="${status}"></label></p>
  <p>Mensaje: <label th:text="${message}"></label></p>
  <p>Excepción: <label th:text="${exception}"></label></p>
  <p>Traza: <label th:text="${trace}"></label></p>
</body>
</html>
```

También podemos crear páginas para asociarlas con código de error HTTP específicos. Por ejemplo, la página «/templates/error/404.html» será usada para gestionar errores 404.

Como es habitual, Spring Boot permite configurar estas características con propiedades:

• server.error.include-stacktrace: con el valor always incluye el valor de traza en la respuesta (en la página de error de Thymeleaf se recupera con ${trace}).
• server.error.include-message: Spring Boot oculta el campo message en la respuesta para evitar la filtración de información confidencial; podemos usar esta propiedad con un valor always para habilitarla.
• server.error.whitelabel.enabled: se puede usar para deshabilitar la página de error de etiqueta blanca y confiar en el contenedor de servlet para proporcionar un mensaje de error HTML.
• server.error.path: permite asignar la URI que gestionará los errores (por defecto "/error").

Por tanto, para poder visualizar todos la información en la página de error debemos habilitar las siguientes propiedades:

```
server.error.include-stacktrace = always
server.error.include-message = always
```

5.7. Websocket.

Las páginas web solicitan datos bajo demanda a un servidor web enviando solicitudes HTTP. Este modelo es ideal para crear aplicaciones interactivas, donde la funcionalidad es dirigida por las acciones del usuario. Sin embargo, en una aplicación que necesita mostrar información que cambia constantemente, este mecanismo es menos útil. Por ejemplo, una página de valores financieros es inútil si muestra precios que quedan obsoletos en pocos minutos, y no podemos esperar que el usuario tenga que refrescar la página continuamente. Es aquí donde los sockets web son útiles. El API Web Sockets proporciona un mecanismo para implementar comunicaciones en tiempo real de dos vías entre el servidor web y el navegador.

5.7.1. ¿Cómo funcionan los sockets web?

Los sockets son los componentes básicos para las comunicaciones en red. El protocolo de sockets permite a un servidor escuchar solicitudes de conexión desde una dirección establecida. Cuando un cliente se conecta a esa dirección, ocurre una negociación y las dos partes establecen una comunicación privada. La aplicación cliente y el servidor intercambian mensajes a través de un canal privado, y cuando la conversación se completa, cada parte cierra la conexión.

El API Web Sockets permite a las páginas web y al servidor web explotar el protocolo de sockets. Los sockets web proporcionan un método simple y ligero de realizar comunicaciones full-dúplex, en tiempo real, entre un cliente y un servidor sin utilizar HTTP.

El protocolo de sockets web establece que hay cuatro pasos para intercambiar datos entre un cliente y un servidor:

1) El cliente solicita una conexión al servidor sobre HTTP o HTTPS.
2) Si el servidor responde positivamente, tanto el cliente como el servidor cambian al protocolo sockets web (conocido como WS) o WSS (una variante segura de WS), y se crea una conexión persistente de socket bidireccional entre ambos.
3) El cliente y servidor envían y reciben mensajes sobre la conexión abierta. El formato de los datos en los mensajes es establecido por el cliente y el servidor; el cliente debe crear mensajes en un formato que el servidor entienda y viceversa.
4) El cliente y el servidor cierran la conexión explícitamente, o bien se cierra automáticamente al pasar cierto tiempo de rechazo.

Estos cuatro pasos se realizan de manera transparente para el navegador en respuesta a los métodos implementados por el API Web Sockets.

5.7.2. Usando el API WebSocket en el lado cliente.

En las páginas web, se usa código JavaScript para crear objetos WebSocket; estos objetos se encargan de implementar la comunicación mediante el protocolo Web Socket. En el lado servidor, tal como veremos posteriormente, se utilizan clases configuradas para este protocolo.

El objeto WebSocket proporciona los métodos que usará el cliente para conectarse al servidor, y enviar y recibir mensajes. El objeto WebSocket también contiene varias propiedades que mantienen información sobre el estado de la conexión actual.

Plantearemos el siguiente ejemplo para mostrar cómo funciona este protocolo: en una página HTML incluiremos un campo de texto, un botón y un panel. El usuario podrá escribir mensajes en el campo de texto, y al pulsar el botón el mensaje se enviará al servidor mediante un web socket. El servidor reenviará el mensaje a todos los clientes conectados. Los mensajes recibidos se mostrarán en el panel.

Usaremos la siguiente plantilla de una página HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body onload="init()">
    <script>
      // Implementación del websocket
      const uri = "ws://localhost:9090/chat";
      let socket;

      function init() {
        // Inicialización del socket aquí
        socket = new WebSocket(uri);
        socket.onopen = function(evt) {
              // Servidor Web Socket está conectado
              document.getElementById("btnenviar").disabled=false;
            };
        socket.onerror = function(event) {
              alert("Ha ocurrido el error: " + event.data);
            };
        socket.onmessage = onMessage;
      }
      function cerrar() {
        // Finaliza la conexión con el servidor
        socket.close(1000, "No error. Comunicación finalizada.");
        document.getElementById("btnenviar").disabled=true;
      }
      function enviarMensaje() {
        // Procesa el mensaje escrito y lo envía al servidor
        const inputMensaje = document.getElementById("mensaje");
        const mensaje = inputMensaje.value;
        inputMensaje.value = "";
        socket.send(mensaje);
      }
      function onMessage(event) {
        // Procesa los mensajes recibidos desde el servidor
        const mensaje = event.data;
        document.getElementById("content").innerHTML += "<p>"+mensaje+"</p>";
      }
    </script>

    <div>
      <label>Mensaje:</label>
      <input type="text" id="mensaje">
    </div>
    <input id="btnenviar" type="button" disabled="disabled" value="enviar" onclick="enviarMensaje()">
    <input type="button" value="Cerrar" onclick="cerrar()">
    <h3>Mensaje rebibido:</h3>
    <div id="content"></div>
  </body>
</html>
```

Abriendo la conexión.

El comienzo de toda comunicación con un socket web de servidor es mediante un abrazo sobre HTTP entre el código cliente que se ejecuta en una página web y el servidor. El constructor WebSocket() permite crear una nueva

conexión y especificar la URL del servidor al que nos conectamos. Esta URL usa el protocolo ws o wss para indicar que es una dirección de socket web:

```
function init() {
  socket = new WebSocket(uri);
  ....
}
```

El abrazo inicial sobre HTTP es realizado automáticamente, y si el servidor acepta la solicitud desde el cliente se establece una nueva conexión usando el protocolo de transporte de socket web.

El API WebSocket es asíncrono. Esto es así porque puede llevar tiempo establecer una conexión, y después de que la conexión ha sido abierta, los mensajes pueden ser recibidos en cualquier momento. Después de crear un objeto WebSocket, deberíamos no intentar usarlo hasta que esté listo para la comunicación. Por ello debemos detectar cuándo una conexión está disponible manejando el evento open del objeto WebSocket. En este punto podemos empezar a enviar y recibir mensajes sobre la conexión.

```
function init() {
  socket = new WebSocket(uri);
  socket.onopen = function(evt) {
    // Servidor Web Socket está conectado
    document.getElementById("btnenviar").disabled=false;
  };
  socket.onerror = function(event) {
    alert("Ha ocurrido el error: " + event.data);
  };
  ...
}
```

Al confirmarse la conexión habilitaremos el botón para postear el formulario. Si ocurre un error mientras estamos conectados al servidor, se lanza el evento error (este evento también es lanzado si ocurre un error mientras se desconecta, o envía un mensaje, el servidor). El mensaje de error está disponible en la propiedad data del parámetro recibido por el manejador del evento. Podemos enlazar este evento usando el retorno de llamada onerror.

Cerrando una conexión.

Para cerrar la conexión con el servidor se llama a la función close() del objeto WebSocket. Esta función toma dos parámetros opcionales, code y reason, que permiten enviar al servidor un código de estado de cierre y un texto razonando el cierre de la conexión.

```
function cerrar() {
  socket.close(1000, "No error. Comunicación finalizada.");
  ...
}
```

Si es el servidor quien cierra la conexión, podemos gestionar el evento close para responder a este hecho:

```
socket.onclose = function(event) {
  // La conexión ha sido cerrada
  if (event.wasClean) {
    alert("Conexión cerrada correctamente");
  } else {
    alert("Conexión cerrada con problemas. Código " + event.code);
  }
};
```

El parámetro que recibe la función manejadora tiene tres propiedades:

- wasClean, que es un valor booleano que indica si la conexión fue cerrada limpiamente (true) o por algún problema (false).
- code, que es el código de estado de cierre.
- reason, que es el texto razonando el cierre.

Enviando mensajes al socket web.

Después de tener establecida una conexión con un servidor, podemos enviar mensaje al servidor usando la función send() del objeto WebSocket, de la siguiente manera:

```
var message = "información del mensaje";
socket.send(message);
```

Cuando la función send() es llamada, los datos del mensaje se ponen en un búfer y se transmiten asíncronamente. Para nuestro ejemplo haremos esto en la función enviarMensaje():

```
function enviarMensaje() {
  const inputMensaje = document.getElementById("mensaje");
  const mensaje = inputMensaje.value;
  inputMensaje.value = "";
  socket.send(mensaje);
}
```

Recibiendo mensajes desde un socket web.

El protocolo WebSocket es bidireccional, y el servidor que está conectado puede enviarnos mensajes en cualquier momento. Para nuestro ejemplo, el servidor nos debe enviar cualquier mensaje que reciba de vuelta.

Se lanza el evento message cuando se recibe un mensaje desde el servidor, dándonos la posibilidad de recibir y procesar el mensaje. El parámetro recibido por el manejador de este evento tiene dos propiedades:

- type, que indica el mensaje recibido es de texto o binario.
- data, que contiene el mensaje.

Para nuestro ejemplo el código será el siguiente:

```
function init() {
  socket = new WebSocket(uri);
  socket.onopen = function(evt) {
    // Servidor Web Socket está conectado
    document.getElementById("buttonSubmit").disabled=false;
  };
  socket.onerror = function(event) {
    alert("Ha ocurrido el error: " + event.data);
  };
  socket.onmessage = onMessage;
}

function onMessage(event) {
  // Procesa los mensajes recibidos desde el servidor
  const mensaje = event.data;
  document.getElementById("content").innerHTML += "<p>"+mensaje+"</p>";
}
```

5.7.3. Usando el API WebSocket en el lado servidor.

Para implementar WebSocket en Spring Boot debemos añadir la siguiente dependencia:

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

En el extremo del servidor, recibiremos los mensajes y responderemos al cliente. En Spring podemos crear un controlador personalizado usando TextWebSocketHandler (para datos de texto) o BinaryWebSocketHandler (para manejar tipos de datos más enriquecidos, como imágenes). En nuestro caso, dado que solo necesitamos manejar texto, usaremos TextWebSocketHandler.

```
@Component
public class SocketTextHandler extends TextWebSocketHandler {
  @Override
  public void handleTextMessage(WebSocketSession session, TextMessage message)
    throws InterruptedException, IOException {
    // se recupera el mensaje
    String payload = message.getPayload();
    // se reenvía de vuelta
    session.sendMessage(new TextMessage("Recibido: " + payload));
  }
}
```

El método handleTextMessage() recibe como argumento el objeto que mantiene la sesión con el cliente, y un objeto que encapsula el mensaje.

Para decirle a Spring que utilice este controlador cuando se reciba una conexión websocket de un cliente, debemos registrarlo y asociarlo a un punto final.

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {
  @Override
  public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
    registry.addHandler(new SocketTextHandler(), "/chat");
  }
}
```

Con la anotación @EnableWebSocket se WebSocket en nuestra aplicación. Y con la instrucción registry.addHandler(new SocketTextHandler(), "/chat") asociado nuestro controlador con la uri "/chat". A partir de este momento, cualquier cliente podrá conectarse mediante websocket.

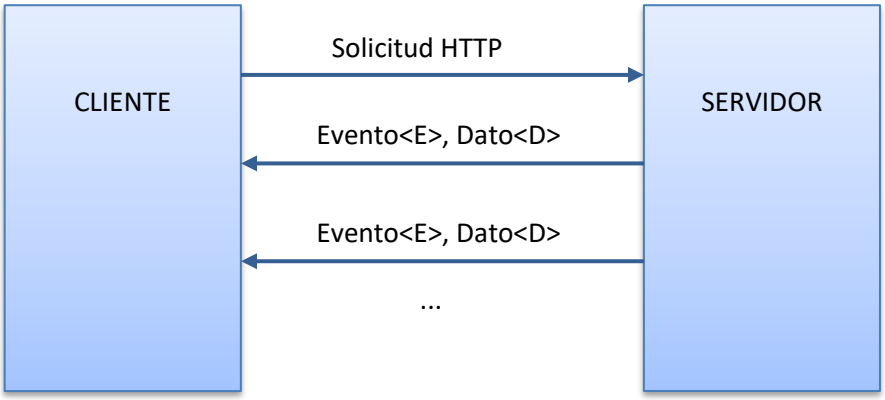
En este ejemplo simplemente recibimos un mensaje de un cliente y se lo devolvemos. Si queremos reenviar dicho mensaje a todos los clientes conectados por WebSocket podemos guardar los objetos de sesión cada vez que se establece una conexión:

```
@Component
public class SocketTextHandler extends TextWebSocketHandler {
  private List<WebSocketSession> sesiones = new ArrayList<>();

  @Override
  public void afterConnectionEstablished(WebSocketSession session) throws Exception {
    super.afterConnectionEstablished(session);
    // Cada vez que se recibe una conexión guardamos la sesión.
    sesiones.add(session);
  }

  @Override
  public void afterConnectionClosed(WebSocketSession session, CloseStatus status) throws Exception {
    super.afterConnectionClosed(session, status);
    // Cada vez que un cliente se desconecta lo removemos.
    sesiones.remove(session);
  }

  @Override
  public void handleTextMessage(WebSocketSession session, TextMessage message)
  throws InterruptedException, IOException {
    String payload = message.getPayload();
    for (WebSocketSession sesion : sesiones) {
      sesion.sendMessage(new TextMessage(payload));
    }
  }
}
```

5.8. Eventos enviados por el servidor (SSE).

SSE es un API de de comunicación entre cliente y servidor, cuyo flujo de comunicación sólo va desde el servidor hacia el cliente (no es bidireccional como los WebSockets). La idea consiste en que el cliente crea una conexión con el servidor una sola vez y éste le va enviando información cuando hay nuevos datos (por ejemplo, ha detectado un cambio en una BD y le manda los nuevos datos). Ya que está orientado a cambios que detecte el servidor para informar al cliente, a este sistema se le ha denominado Server Sent Events (Eventos Enviados por el Servidor).

La comunicación se realiza a través de HTTP, tal como muestra la siguiente figura:



El API Server-Sent Events permite mantener una comunicación abierta entre la aplicación web y un cliente de forma que la aplicación irá transmitiendo eventos hacia el cliente. El «avance» obtenido con este método es que los datos se proporcionan desde el servidor sin tener que realizar una petición mediante AJAX cada varios segundos. Este tipo de conexión se considera como «contenido en streaming»: el servidor no tiene que esperar a tener todo el buffer de salida preparado para mandárselo todo al cliente tras terminar su ejecución; en lugar de ello, el servidor va escribiendo datos en streaming que le llegan al cliente sin que, para ello, tenga que terminar la ejecución del script de la parte servidora.

5.8.1. Enviando eventos desde el servidor.

El servidor simplemente tendrá que usar el tipo MIME "text/event-stream" para implementar eventos. Cada evento se envía con un bloque de texto terminado en un par de saltos de línea. Lo habitual es que los eventos comiencen por una palabra clave seguida de dos puntos y el contenido del evento. Para enviar simplemente datos debemos usar la palabra clave data.

En un controlador habilitaremos la funcionalidad asíncrona para enviar datos cada segundo:

```
@EnableAsync
@ResController
public class Controlador {
  @Async()
  @GetMapping(path = "/stream", produces = "text/event-stream")
  public CompletableFuture<Void> streamsse(Writer response) {
    return CompletableFuture.runAsync(() -> {
      try {
        for(int i = 0; i < 10; i++) {
          response.write("data: item " + i + "\n\n");
          response.flush();
          Thread.sleep(1000);
        }
        response.close();
      } catch (Exception e) {
        e.printStackTrace();
      }
    });
  }
}
```

Spring Boot proporciona la clase SseEmiter para simplificar el emisión de eventos. El controlador quedaría así:

```
@ResController
public class Controlador {
  @GetMapping(path = "/stream")
  public ResponseEntity<SseEmitter> stream() {
    var sse = new SseEmitter();
    CompletableFuture.runAsync(() -> {
      try {
        for (int i = 0; i < 10; i++) {
          sse.send(SseEmitter.event().data("item " + i));
```

```
            Thread.sleep(1000);
          }
        } catch (Exception e) {
        }
        sse.complete();
      });
      return ResponseEntity.ok().body(sse);
    }
  }
```

5.8.2. Un cliente de eventos.

Si en un navegador realizamos la consulta recibiremos un línea de datos cada segundo.

Para utilizar una página HTML como cliente, podemos usar el objeto EventSource de JavaScript. El siguiente código consulta la aplicación de Spring Boot, recupera los eventos y los añade a un panel:

```
<body>
  <div id="panel"></div>
  <script>
    var panel = document.getElementById("panel");
    var evs = new EventSource("stream");
    evs.onopen=function(evt) {
      console.log("Conexión establecida.");
    };
    evs.onmessage = function(evt) {
      console.log("Se ha recibido un evento.");
      panel.innerHTML += evt.data + "<br />";
    };
    evs.onerror = function(evt) {
      console.log("Se produjo un error en la conexión.");
      evs.close();
    }
  </script>
</body>
```

En este caso, en el manejador onmessage, el parámetro permite recuperar el contenido del evento con su atributo data.

5.8.3. Formato de flujo de eventos (formato stream).

Los eventos SSE son un flujo de datos de texto. Los mensajes en este flujo de eventos se separan con un par de caracteres de salto de línea. Si hay un símbolo de dos puntos como primer caracter de una línea, se entiende que es un comentario y es ignorado.

Cada mensaje consiste en una o más líneas de texto que enumeran los campos para ese mensaje. Cada campo está representado por el nombre del campo, seguido por los datos de texto para el valor de ese campo.

Los campos predefinidos por la especificación son:

- event: el tipo de evento. Si se especifica, se enviará un evento al navegador de escucha para el nombre del evento especificado, y con JavaScript usaríamos addEventListener() para escuchar eventos nombrados, y onmessage si no se especifica el nombre del evento para un mensaje.

  Por ejemplo, si enviamos eventos encabezados con evento1:

  ```
  "event:evento1"
  ```

  Debemos gestionar lo siguiente con JavaScript:

  ```
  var evs = new EventSource("testsse");
  evs.addEventListener("evento1", function(evt) { } );
  ```

- data: el campo de datos para el mensaje.
- id: el ID del evento que establecerá el último ID del objeto EventSource.
- Retry: es el tiempo usado de reconexión para intentar enviar el evento. Es un número entero, que especifica el tiempo de reconexión en milisegundos.

Normalmente, los navegadores se reconectan al origen de eventos cuando la conexión se cierra, pero este comportamiento se puede cancelar utilizando el método cancel() sobre el objeto EventSource. Para cancelar desde

el servidor, debe responder con un tipo de contenido distinto de "text/event-stream" o retornando un código de error (por ejemplo, 400).

6. El API Http Client

El API Http Client es un nuevo estándar para implementar solicitudes HTTP sacando provecho de las nuevas características del protocolo HTTP/2, y los nuevos modelos de flujos de datos incorporados en Java.
Con HTTP/2 se introdujeron características como:

- Compresión de las cabeceras.
- Conexiones con capacidad de multiplexar varias respuestas.
- Compromiso de devolución (modelo Push Promise)

Con Http Client podremos configurar los mensajes de solicitud y realizar conexiones HTTP y WebSocket.
El paquete java.net.http incluye los siguientes tipos principales:

- HttpClient: es el tipo principal de este API. Representa un objeto cliente que quiere realizar solicitudes a un servidor. Soporta envío y recepción de mensajes tanto síncrona como asíncronamente.
- HttpRequest: encapsula la información de solicitud para construir los mensajes de HTTP, los cuales incluyen la acción (GET, POST, etc.), la URI destino, la versión HTTP, cabeceras y cuerpo del mensaje. Estos objetos pueden reutilizarse las veces que queramos para realizar las mismas solicitudes.
- HttpResponse: encapsula la respuesta HTTP, incluyendo el código de estado, las cabeceras de respuesta, y el cuerpo del mensaje, si lo hay.

Además, se incluyen tipos auxiliares para configurar y procesar la solicitud y la respuesta:

- HttpRequest.BodyPublisher: si la solicitud tendrá cuerpo, se utiliza un objeto BodyPublisher para poblarlo desde un origen de datos, por ejemplo, desde un string, un fichero, etc.
- HttpResponse.BodyHandler: si la respuesta consta de un cuerpo, los objetos BodyHandler permiten recuperar su contenido mediante diversas técnicas.
- HttpResponse.BodySubscriber: son objetos que se suscriben para consumir la respuesta de la solicitud.

Los BodyPublisher y BodySubscriber siguen el modelo del API Flow de Java. Esto significa que podremos utilizarlos para alinear la respuesta mediante flujos reactivos cuando se envíen respuestas asíncronas usando HTTP/2.

6.1. Creación del objeto HttpClient.

Para realizar solicitudes primero debemos crear un objeto HttpClient. Se puede crear de dos formas:

1) Con el método estático HttpClient.newBuilder(). Este método aplica el patrón Builder y nos permite configurar varios aspectos de la conexión:

```
HttpClient client = HttpClient.newBuilder()
  .authenticator(Authenticator.getDefault())      // Autentificación
  .connectTimeout(Duration.ofSeconds(30))         // Máximo tiempo de espera
  .cookieHandler(CookieHandler.getDefault())      // Manejador de cookies
  .executor(Executors.newFixedThreadPool(2))      // Executor para solicitudes asíncronas
  .followRedirects(Redirect.NEVER)                // Redirección
  .priority(1)                                    // HTTP/2 priority
  .proxy(ProxySelector.getDefault())              // Conexión a través de un proxy
  .sslContext(SSLContext.getDefault())            // Contexto SSL
  .version(Version.HTTP_2)                        // Versión del protocolo HTTP
  .sslParameters(new SSLParameters())             // Parámetros SSL
  .build();
```

2) Con el método estático HttpClient.newHttpClient():

```
HttpClient client = HttpClient.newHttpClient();
```

Con este método se crea un objeto con configuraciones por defecto. Estas configuraciones incluyen: el protocolo HTTP/2, tiempo de espera ilimitado, sin manejador de cookies, sin autentificación, uso del pool por defecto para hilos, selector de proxy por defecto, y contexto SSL por defecto.

Una vez creada, la instancia HttpClient es inmutable, es de hilos seguros, y permite enviar varias solicitudes. Por defecto, el cliente intenta abrir conexiones HTTP/2.

6.1.1. Especificando la versión del protocolo HTTP.

Aunque el cliente HTTP intenta usar por defecto el protocolo HTTP/2, podemos forzarlo:

```
HttpClient httpClient1 = HttpClient.newBuilder()
```

```
    .version(Version.HTTP_1_1)
    .build();

HttpClient httpClient2 = HttpClient.newBuilder()
    .version(Version.HTTP_2)  // configuración por defecto
    .build();
```

Al especificar HTTP/2, la primera solicitud a un servidor intentará usar este protocolo. Si el servidor no lo soporta usarán HTTP/1.1.

6.1.2. Especificando un proxy.

Para realizar las conexiones a través de un proxy debemos especificar su configuración con el método proxy(). Por ejemplo:

```
HttpClient httpClient = HttpClient.newBuilder()
    .proxy(ProxySelector.of(new InetSocketAddress("host", 7070)))
    .build();
```

Mediante un objeto java.net.InetSocketAddress podemos especificar el host y puerto del servidor proxy.

6.1.3. Redirección a otras direcciones.

Algunas veces la dirección a la que queremos acceder ha sido cambiada por otra. En estos casos, la respuesta HTTP enviará un código 3xx, con la información sobre la nueva dirección. HttpClient puede redirigirnos automáticamente a la nueva dirección si asignamos la política de redirección adecuada.

Podemos hacer esto con el método followRedirects():

```
HttpClient httpClient = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build()
```

La enumeración HttpClient.Redirect dispone de las constantes ALWAYS (redirigir siempre), NEVER (nunca) y NORMAL (redirigir excepto de HTTPS a HTTP).

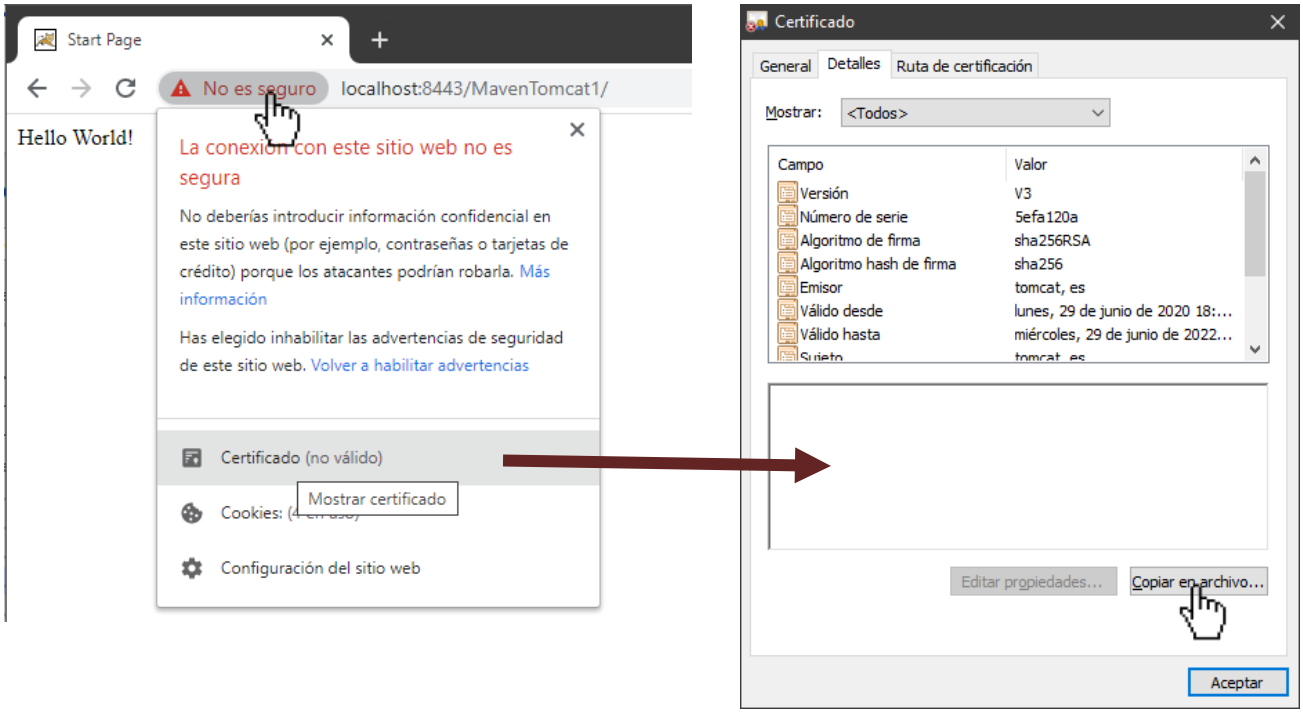
6.1.4. Asignación de autentificación.

Los objetos Authenticator permiten negociar las credenciales de autentificación HTTP para una conexión. Soportan varios esquemas de autentificación (como el basic o digest), y normalmente requieren de unas credenciales.

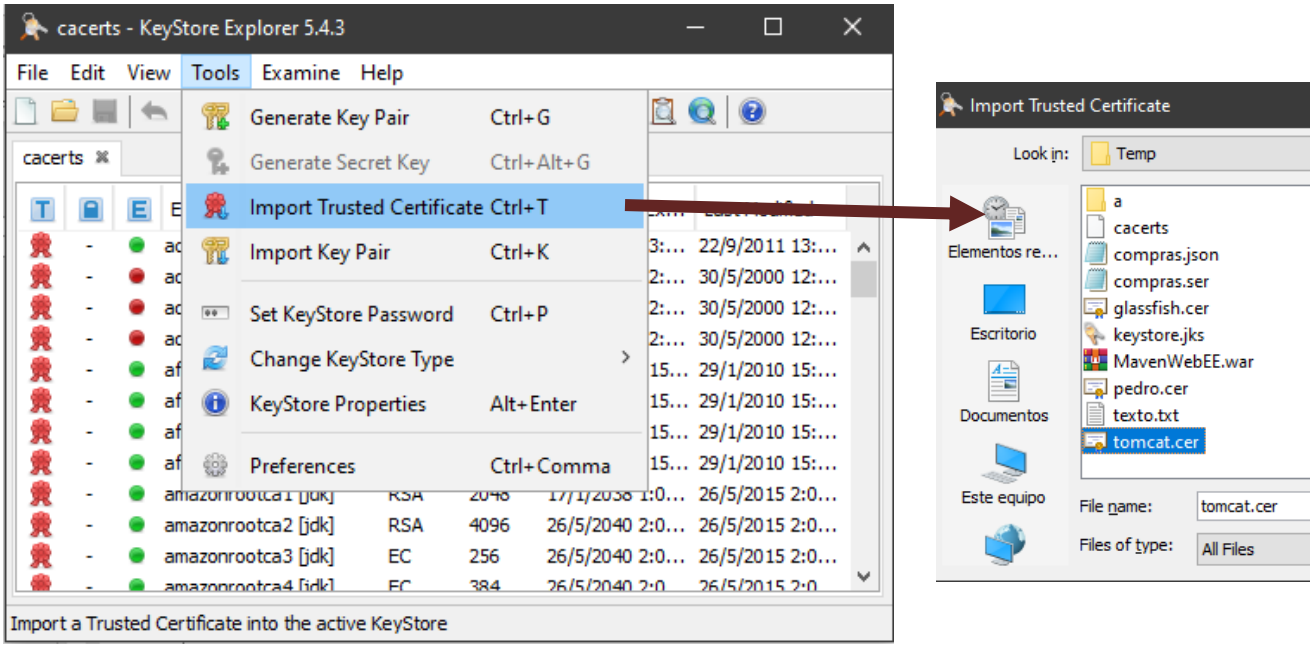
Podemos usar la clase PasswordAuthentication para encapsular estas credenciales:

```
HttpClient httpClient = HttpClient.newBuilder()
    .authenticator(new Authenticator() {
        @Override
        protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication("username", "password".toCharArray());
        }
    })
    .build();
```

Si la solicitud es de tipo HHTPS necesitaremos aportar el certificado desde el lado cliente. Si nos conectamos al sitio web desde un navegador podemos descargar el certificado que se utiliza. Debemos hacer clic sobre el icono indicador de sitio seguro que aparece a la izquierda de la URL, y pulsar en la opción «Certificado». Se abrirá un cuadro de diálogo que nos permitirá descargar el certificado y guardarlo en nuestro disco.



Supongamos que hemos guardado el certificado en un fichero llamado «tomcat.cer». Ahora debemos añadirlo al almacén de confianza que utiliza Java. Dentro de la carpeta «$Java_Home/lib/security» se incluye el fichero «cacerts», el cual contiene los certificados de confianza para las aplicaciones de Java. Podemos crear una copia de este fichero para personalizarlo para nuestra aplicación, y abrirlo con «KeyStore Explorer». Su contraseña por defecto es changeit.



Al agregar el certificado al almacén debemos asignarle un alias. Ahora indicaremos en el programa dónde está el almacén de confianza usando propiedades del sistema:

```
// Configuración para SSL: establecemos el almacén de confianza con los certificados
System.setProperty("javax.net.ssl.trustStore", "C:/Temp/cacerts"); // ubicación física
System.setProperty("javax.net.ssl.trustStorePassword", "changeit"); // contraseña
System.setProperty("javax.net.ssl.trustStoreType", "JKS"); // tipo
// Si usamos un certificado auto-firmado deshabilitamos la conprobación del host
System.setProperty("jdk.internal.httpclient.disableHostnameVerification", "true");
```

Con esta configuración podremos realizar las solicitudes HTTPS sin problemas.

6.1.5. Asignación de un administrador de cookies.

Las cookies son administradas automáticamente por navegadores y servidores, pero en nuestras solicitudes podemos controlarlas definiendo un CookieManager:

```
HttpClient httpClient = HttpClient.newBuilder()
  .cookieHandler(new CookieManager(null, CookiePolicy.ACCEPT_NONE))
  .build();
```

En este ejemplo no se aceptan cookies para ninguna de las solicitudes.

El constructor de CookieManager requiere dos argumentos: un CookieStore y un valor de la enumeración CookiePolicy con la política que se aplica. Un CookieStore representa un almacén que gestiona las cookies, un valor null provoca que se cree un almacén por defecto en memoria.

6.2. Creación del objeto HttpRequest.

Los objetos HttpRequest nos permitirán configurar los mensajes de solicitud HTTP, los cuales se usarán para realizar la solicitud a través del HttpClient.

Mediante un objeto HttpRequest podremos configurar el verbo, uri, cabeceras y cuerpo del mensaje:

```
HttpRequest request = HttpRequest.newBuilder()
  .uri(URI.create("http://www.negocio.informes.info"))
  .timeout(Duration.ofMinutes(2))
  .header("Content-Type", "application/x-www-form-urlencoded")
  .POST(BodyPublishers.ofString("n1=valor1&n2=valor2"))
  .build();
```

Los objetos HttpRequest son inmutables y pueden utilizarse varias veces para diversas solicitudes. Se realiza la solicitud usando el método send() o sendAsync() del HttpClient:

```
HttpClient httpClient = HttpClient.newBuilder().build();
HttpRequest request = HttpRequest.newBuilder()
  .uri(URI.create("http://www.negocio.informes.info"))
  .GET()
  .build();

HttpResponse<Void> response1 = client.send(request, BodyHandlers.discarding());
CompletableFuture<HttpResponse<Void>> response2 = client.sendAsync(request, BodyHandlers.discarding());
```

6.2.1. Asignación de la URI.

Podemos especificar la URL de la solicitud tanto con el propio método newBuilder():

```
HttpRequest request = HttpRequest.newBuilder(URI.create("http://www.negocio.informes.info"))
  ...
  .build();
```

Como con el método enlazado uri():

```
HttpRequest request = HttpRequest.newBuilder()
  .uri(URI.create("http://www.negocio.informes.info"))
  ...
  .build();
```

Sendos métodos solicitan un objeto java.net.URI.

6.2.2. Asignación del verbo HTTP.

El cliente soporta todos los verbos HTTP, pero el constructor de la solicitud sólo contiene los verbos predefinidos mediante los métodos: GET(), POST(), DELETE() y PUT(). Para crear una solicitud con un verbo diferente se debe utilizar el método method():

```
HttpRequest request = HttpRequest.newBuilder()
  .method("HEAD", BodyPublishers.noBody())
  ...
  .build();
```

Con BodyPublishers.noBody() se especifica que el mensaje no dispondrá de cuerpo.

6.2.3. Asignación de tiempos de espera.

Podemos especificar un tipo de espera máximo para que se complete una solicitud. En el siguiente ejemplo:

```
HttpRequest request = HttpRequest.newBuilder()
  .timeout(Duration.ofMinutes(2))
  ...
```

```
        .build();
```

Especificamos que, si pasados 2 minutos, no se completa la solicitud de debe lanzar una excepción del tipo HttpTimeoutException desde el método que realiza la solicitud: send() o sendAsync()

6.2.4. Reutilización de los objetos de solicitud.

Podemos realizar copias de los objetos HttpRequest, de forma que podemos personalizarlos para varias configuraciones; por ejemplo, asignándoles cabeceras diferentes.

```
var builder = HttpRequest.newBuilder()
        .GET()
        .uri(URI.create("http://www.negocio.informes.info"));
var request1 = builder.copy().setHeader("X-Counter", "1").build();
var request2 = builder.copy().setHeader("X-Counter", "2").build();
```

6.2.5. Publicación del cuerpo de la solicitud.

En solicitudes POST y PUT es habitual enviar información en el cuerpo de la solicitud. Los métodos POST(), PUT() y method(), solicitan objetos de tipo BodyPublisher para poblar el cuerpo del mensaje desde diversos orígenes. HttpRequest.BodyPublisher es una interfaz que hereda de Flow.Publisher, y que por tanto permite aplicar un modelo de flujo reactivo para proporcionar el contenido del cuerpo.

Para la mayoría de los escenarios, la clase BodyPublishers ofrece métodos estáticos que permiten poblar el cuerpo del mensaje desde diversos orígenes:

1) Si no hay cuerpo.

Se utiliza el método noBody().

```
HttpRequest request = HttpRequest.newBuilder()
  .POST(BodyPublishers.noBody())
  ...
  .build();
```

2) Cuando el origen es un string:

En este caso podemos usar los métodos ofString(String) y ofString(String, Charset). Simplemente debemos proporcionar un texto con el contenido del cuerpo y opcionalmente el charset.

```
HttpRequest request = HttpRequest.newBuilder()
  .header("Content-Type", "application/x-www-form-urlencoded")
  .POST(BodyPublishers.ofString("n1=valor1&n2=valor2", Charset.defaultCharset()))
  ...
  .build();
```

3) Cuando el origen es un array de bytes.

En este caso usaremos el método ofByteArray(byte[]) o ofByteArray(byte[], int, int), donde este segundo método permite especificar el desplazamiento en el array y número de bytes a consumir.

```
HttpRequest request = HttpRequest.newBuilder()
  .POST(BodyPublishers.ofByteArray("Contenido".getBytes(), 0, 5))
  ...
  .build();
```

Para procesar varios arrays de bytes se utiliza el método ofByteArrays(Iterable<byte[]>).

4) Cuando el origen es un fichero.

En este caso podemos usar el método ofFile(Path). Simplemente debemos proporcionar la ruta del fichero como un objeto Path.

```
HttpRequest request = HttpRequest.newBuilder()
  .header("Content-Type", "application/x-www-form-urlencoded")
  .POST(BodyPublishers.ofFile( Path.of("datos.txt") ))
  ...
  .build();
```

5) Cuando el origen se conecta mediante un InputStream.

En este caso usaremos el método ofInputStream(Supplier<InputStream>). Este método recibirá el canal de lectura mediante un objeto Supplier.

```
HttpRequest request = HttpRequest.newBuilder()
  .POST(BodyPublishers.ofInputStream(()->new FileInputStream("datos.txt")))
  ...
  .build();
```

6) Cuando los datos son obtenidos desde un publicador reactivo.

En este caso podemos usar los métodos fromPublisher(Flow.Publisher<ByteBuffer>) y fromPublisher(Flow.Publisher<ByteBuffer>, in), donde este segundo método debe especificar cuántos bytes va a proporcionar el publicador.

Podemos usar fromPublisher(Flow.Publisher<ByteBuffer>) cuando el tamaño del cuerpo es desconocido de antemano, ya que el publicador irá poblando el cuerpo con una cantidad indeterminada de datos.

En el siguiente ejemplo se crea un publicador que ofrece parámetros de formulario:

```
class Publicador extends SubmissionPublisher<ByteBuffer> {
  public Publicador() {
    super.getExecutor().execute( () -> {
      // Espera a que se registre algún suscriptor
      while (this.hasSubscribers()==false);
      // Se ofrecen los datos
      List.of("a=uno&", "b=dos&", "c=tres&", "d=cuatro")
        .stream()
        .map(s -> s.getBytes())            // convierte string a byte[]
        .map(ByteBuffer::wrap)             // crea ByteBuffer a partir de byte[]
        .forEach(bb-> this.submit(bb));    // publica el ByteBuffer
      this.close();
    });
  }
}
```

El cliente poblará el cuerpo del mensaje con lo que ofrece el publicador:

```
HttpClient client = HttpClient.newBuilder().build();
HttpRequest request = HttpRequest.newBuilder()
  .header("Content-Type", "application/x-www-form-urlencoded")
  .POST(BodyPublishers.fromPublisher(new Publicador()))
  .build();
```

6.3. Creación de objetos HttpResponse.

Los objetos HttpResponse son el resultado de la respuesta de una solicitud realizada con los métodos send() o sendAsync() del HttpClient.

Un proceso de solicitud que se completa puede ser el siguiente:

```
// Creación del objeto cliente
HttpClient client = HttpClient.newBuilder()
    .build();
// Creación del objeto de solicitud
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("http://localhost:8084/webapi/facturas"))
    .GET()
    .build();
// Ejecución de la solicitud
HttpResponse<String> response = client.send(request1, BodyHandlers.ofString());
// El objeto de retorno encapula los datos del mensaje de respuesta
System.out.println(response.body());
```

Los objetos HttpResponse encapsulan toda la información de respuesta: el protocolo, código de respuesta, cabeceras y cuerpo del mensaje. Y para recuperarla, ofrece los siguientes métodos:

- body(): recupera el cuerpo del mensaje según el formato especificado al crear el HttpResponse.
- headers(): retorna un objeto HttpHeaders que permite recuperar las cabeceras de respuesta.
- request(): retorna el HttpRequest que provocó la respuesta.
- statusCode(): retorna el código HTTP de estado. Normalmente será el código 200 para indicar que la solicitud fue correcta.
- uri(): retorna la URI de la solicitud.
- version(): retorna un HttpClient.Version con la versión del protocolo HTTP.

6.3.1. Solicitudes síncronas.

Una solicitud síncrona se realiza con el método send() del objeto HttpClient. Este método solicita dos argumentos:

1) Un HttpRequest: el objeto de solicitud.

2) Un BodyHandler<T>: un objeto genérico que determinará cómo debe ser procesado el cuerpo de la respuesta. El genérico T indica el formato del cuerpo del mensaje.

La clase BodyHandlers proporciona varios métodos estáticos que devuelven un BodyHandler:

1) Si no hay cuerpo.

Se utiliza el método discarting().

```
HttpResponse<Void> response = client.send(request, BodyHandlers.discarting());
```

2) Si el cuerpo debe ser consumido como un string.

Se utilizan los métodos ofString() u ofString(Charset).

```
HttpResponse<String> response = client.send(request, BodyHandlers.ofString());
```

3) Si el cuerpo debe ser consumido como un array de bytes.

Se utiliza el método ofByteArray().

```
HttpResponse<byte[]> response = client.send(request, BodyHandlers.ofByteArray());
```

Con el método ofByteArrayConsumer(Consumer<Optional<byte[]>) podemos consumir rápidamente el array de bytes, si existe.

```
HttpResponse<Void> response = client.send(
    request,
    BodyHandlers.ofByteArrayConsumer(optionalBytes -> {
        String contenido = new String(optionalBytes.orElse(new byte[0]));
        // ...
      }
    )
  );
```

4) Si el cuerpo debe ser enviado directamente a un fichero.

Se utilizan los métodos ofFile(Path), ofFile(Path, OpenOption...) y ofFileDownload(Path, OpenOption...). En su forma más simple basta con indicar una ruta de fichero:

```
Path ruta = Pahts.get("destino.json");
HttpResponse<Path> response = client.send(request, BodyHandlers.ofFile(ruta));
```

5) Si queremos procesar el cuerpo con un InputStream.

Se utiliza el método ofInputStream().

```
byte[] respuesta = client.send(request, HttpResponse.BodyHandlers.ofInputStream())
    .body().readAllBytes();
```

6) Si el cuerpo se compone de líneas.

Podemos usar el método ofLines() para consumirlo mediante un Stream:

```
client.send(request, HttpResponse.BodyHandlers.ofLines())
  .body().forEach(System.out::println);
```

O el método fromLineSubscriber() para consumir las líneas mediante un suscriptor:

```
Flow.Subscriber<String> suscriptor = new Flow.Subscriber<String>() {
  private Flow.Subscription subscription;
  @Override
  public void onSubscribe(Flow.Subscription subscription) {
    this.subscription = subscription;
    subscription.request(1);
  }
  @Override
  public void onNext(String item) {
    System.out.println(item);
    subscription.request(1);
  }
  @Override
  public void onError(Throwable throwable) {
  }
  @Override
  public void onComplete() {
  }
};
...
```

```
client.send(request, HttpResponse.BodyHandlers.fromLineSubscriber(suscriptor));
```

7) Si queremos consumir el cuerpo mediante un suscriptor.
Disponemos del método fromSubscriber().

```
Flow.Subscriber<List<ByteBuffer>> suscriptor = new Flow.Subscriber<>() {
  private Flow.Subscription subscription;
  @Override
  public void onSubscribe(Flow.Subscription subscription) {
    this.subscription = subscription;
    subscription.request(1);
  }
  @Override
  public void onNext(List<ByteBuffer> items) {
    items.forEach(item -> {
      byte[] data = new byte[item.limit()];
      item.get(data);
      System.out.println(new String(data));
    });
    subscription.request(1);
  }
  @Override
  public void onError(Throwable throwable) {
  }
  @Override
  public void onComplete() {
  }
};
...
client.send(request, HttpResponse.BodyHandlers.fromSubscriber(suscriptor));
```

Los objetos BytBuffer que consume el suscriptor son de sólo lectura.

6.3.2. Solicitudes asíncronas.

Para solicitudes síncronas se utiliza el método sendAsync() del objeto HttpClient. Este método retorna inmediatamente un objeto CompletableFuture<HttpResponse> que se completa cuando el objeto HttpResponse queda disponible. Con los objetos CompletableFuture podemos aplicar un modelo de flujo para procesar la respuesta.

```
client.sendAsync(request, BodyHandlers.ofString())
  .thenApply(response -> {
    System.out.println(response.statusCode());
    return response;
  })
  .thenApply(HttpResponse::body)
  .thenAccept(System.out::println);
```

6.4. Cliente WebSocket.

Podemos usar HttpClient para crear un cliente WebSocket que se comunique con un componente del lado servidor que implemente WebSocket. Supongamos creado el componente WebSocket del ejemplo del apartado 3.6.3.
Con el método HttpClient.newWebSocketBuilder() podremos crear el componente cliente:

```
// El cliente será un objeto de tipo Listener
WebSocket.Listener oyente = new WebSocket.Listener() {
  // Método que recibe los mensaje de texto desde el servidor
  @Override
  public CompletionStage<?> onText(WebSocket webSocket, CharSequence data, boolean last) {
    System.out.println(data);
    return WebSocket.Listener.super.onText(webSocket, data, last);
  }
  // Los demás métodos tienen una implementación por defecto
  @Override
  public void onError(WebSocket webSocket, Throwable error) {
```

```
        WebSocket.Listener.super.onError(webSocket, error);
      }
      @Override
      public CompletionStage<?> onClose(WebSocket webSocket, int statusCode, String reason) {
        return WebSocket.Listener.super.onClose(webSocket, statusCode, reason);
      }
      @Override
      public CompletionStage<?> onPong(WebSocket webSocket, ByteBuffer message) {
        System.out.println("PING: " + new String(message.array()));
        return WebSocket.Listener.super.onPong(webSocket, message);
      }
      // @Override
      // public CompletionStage<?> onPing(WebSocket webSocket, ByteBuffer message) {
      //   return WebSocket.Listener.super.onPing(webSocket, message);
      // }
      @Override
      public CompletionStage<?> onBinary(WebSocket webSocket, ByteBuffer data, boolean last) {
        return WebSocket.Listener.super.onBinary(webSocket, data, last);
      }
      @Override
      public void onOpen(WebSocket webSocket) {
        WebSocket.Listener.super.onOpen(webSocket);
      }
    };

    URI  uri = new URI("ws://localhost:9090/chat");
    HttpClient httpClient = HttpClient.newBuilder().build();
    WebSocket webSocket = httpClient.newWebSocketBuilder()
          .buildAsync(uri, oyente)
          .join();
    // Enviamos un mensaje
    webSocket.sendText("Hola", true);
    // La respuesta será recibida en el método oyente.onTex()
```

La interfaz java.net.http.WebSocket.Listener define métodos default que si reescribimos debemos invocar para que se conserve el funcionamiento esperado del WebSocket.
Los métodos onText() y onBinary() reciben los mensajes desde el servidor según su formato, de texto o binario.
El método onPing() no es necesario reescribirlo. Se utiliza el método sendPing() del web socket para realizar una prueba de conexión, de forma que la respuesta se recibe en el método onPong().

```
webSocket.sendPing(ByteBuffer.wrap("Hola".getBytes()));
```

7. Motor de plantillas Thymeleaf.

El marco web de Spring se basa en el patrón MVC (Modelo-Vista-Controlador), lo que facilita la separación de los diversos aspectos en una aplicación. Esto permite la posibilidad de utilizar diferentes tecnologías de visualización, desde la tecnología JSP bien establecida hasta una variedad de motores de plantillas o vistas.
Dado que los aspectos de una aplicación Spring están claramente separados, cambiar de una tecnología de vistas a otra es principalmente una cuestión de configuración. Para utilizar un motor de vistas necesitamos definir un bean de tipo ViewResolver correspondiente a cada tecnología. Esto permitirá que los métodos de un controlador pueden retornar los nombres de vistas independientemente del motor de vistas que utilicemos.
En este apartado veremos cómo usar JSP y el motor de plantillas Thymeleaf.

7.1. Configuración de Thymeleaf.

Thymeleaf es un motor de plantillas Java que puede procesar archivos HTML, XML, texto, JavaScript o CSS. A diferencia de otros motores de plantillas, Thymeleaf permite usar plantillas como prototipos, lo que significa que se pueden ver como archivos estáticos.
Para integrar Thymeleaf con Spring Boot, necesitamos agregar las siguientes dependencias:

```
<dependency>
```

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

  <!-- Dependencia para utilizar plantillas dialectos de diseño -->
  <dependency>
    <groupId>nz.net.ultraq.thymeleaf</groupId>
    <artifactId>thymeleaf-layout-dialect</artifactId>
  </dependency>
```

De forma predeterminada, los archivos HTML deben colocarse en la ubicación de «resources/templates», aunque podemos cambiar varias configuraciones en el fichero «application.properties»:

```
# Ubicación de las páginas
spring.thymeleaf.prefix=classpath:/templates/
# Extensión de las páginas
spring.thymeleaf.suffix=.html
# Versión de las páginas
spring.thymeleaf.mode=HTML5
# Codificación de las páginas
spring.thymeleaf.encoding=UTF-8
# Tipo MIME de las páginas
spring.thymeleaf.content-type=text/html
# Soporte para caché
spring.thymeleaf.cache=true
```

Las páginas HTML actúan como plantillas Thymeleaf si añadimos la siguiente configuración:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    .........
  </head>
  <body>
    .........
  </body>
</html>
```

El atributo xmlns:th configura un espacio de nombres para utilizar procesadores de atributos th:* en las etiqueta HTML, tal como veremos a continuación:

```
<body>
  <p th:text="Hola"></p>
</body>
```

El atributo th:text asigna el contenido para la etiqueta donde se utiliza, en este caso la etiqueta <p> define un párrafo.

Algunas de las características que están disponibles cuando se usa Thymeleaf son:

- Compatibilidad con la definición del comportamiento de los formularios.
- Enlazar controles de formulario a modelos de datos.
- Validación de los controles de formulario
- Mostrar valores de mensajes de propiedades.
- Representación de fragmentos de plantilla.

7.2. Visualización de mensajes de propiedades.

En Spring Boot, la ubicación por defecto para propiedades de mensajes es el fichero «resources/messages.properties», aunque podemos utilizar otro nombre y ubicación si lo especificamos en el fichero de propiedades de la aplicación:

```
spring.messages.basename=messages
```

Supongamos los siguientes mensajes, donde el último utiliza un parámetro:

Fichero «resources/messages.properties»
user.name=Pedro title.message=Presentación

```
welcome.message=Saludos, {0}
```

Podemos visualizar el primer mensaje en las vistas usando la sintaxis #{key}. Por ejemplo:

```
<h1 th:text="#{title.message}"></h1>
```

Cuando el mensaje define parámetros se debe usar la sintaxis #{key(param1, param2,...}. Por ejemplo:

```
<p th:text="#{welcome.message(#{user.name})}"></h1>
```

El valor para el parámetro podrá se obtenido desde otro mensaje o un objeto de modelo, tal como iremos viendo.

7.3. Visualización de atributos del modelo.

Recordemos que los controladores permiten pasar atributos del modelo usando objetos Model o ModelAndView. Por ejemplo, supongamos el siguiente método de controlador:

```
@GetMapping("/fecha")
  public String fecha(Model model) {
    model.addAttribute("fecha", LocalDate.now());
    return "test";
  }
```

7.3.1. Atributos simples

Podemos usar la sintaxis ${atributo} para recuperar atributos del modelo en las vistas Thymeleaf. Recuperaremos el atributo fecha en la página «test.html» para asignar el valor inicial de un control de calendario:

```
<input type="date" th:value="${fecha}">
```

Se utiliza el atributo th:value para asignar el valor de un campo de edición.

7.3.2. Atributos de colección.

Si el atributo de modelo es una colección de objetos, podemos usar el atributo de etiqueta th:each para iterar sobre él. Por ejemplo, si disponemos de una clase UserForm:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class UserForm {
  private String name;
  private String email;
  private String password;
  private boolean admin;
}
```

Desde el controlador pasaremos una lista de usuarios como atributo:

```
@GetMapping("/usuarios")
public String usuarios(Model model) {
  model.addAttribute("users", List.of(
        new UserForm("Pedro", "pedro@com","", true),
        new UserForm("Ana", "ana@com","", false)
        ));
  return "test";
}
```

Finalmente, en la vista iteraremos sobre el nombre y correo de los usuarios para mostrarlos en forma de tabla:

```
<table>
  <thead>
    <tr><th>Nombre</th><th>Correo</th></tr>
  </thead>
  <tbody>
    <tr th:each="user: ${users}">
      <td th:text="${user.name}"></td>
      <td th:text="${user.email}"></td>
    </tr>
  </tbody>
</table>
```

El valor de th:each es similar a la sintaxis de un buble for(:) de Java. Primero se incluye el nombre de una variable de iteración, seguido de dos puntos y del objeto iterable. La variable de iteración podrá ser recuperada como

cualquier otro atributo con la sintaxis ${variable}. Como en este ejemplo son objetos de tipo UserForm podemos usar la sintaxis del punto para recuperar una de sus propiedades.

7.3.3. Definir variables.

En las propias plantillas de Thymeleaf podemos definir variables como si fuese atributos del modelo de dos formas. La primera forma ya la hemos visto, consiste en crear una variable de iteración sobre una colección en el atributo th:each:

```
<div th:each="user : ${users}">
  <a th:text="${user.name}" th:href="${user.email}"></a>
</div>
```

Otra forma es definir una nueva variable basada en otra utilizando th:with. Por ejemplo, podemos tomar el primer usuario de la colección de usuarios:

```
<div th:with="firstUser=${users[0]}">
  <a th:text="${firstUser.name}" th:href="${firstUser.email}"></a>
</div>
```

O podemos crear una nueva variable que contenga solo el nombre del usuario:

```
<div th:each="user : ${users}" th:with="userName=${user.name}">
  <a th:text="${userName}" th:href="${user.email}"></a>
</div>
```

También es posible definir múltiples variables. Por ejemplo, podemos crear dos variables separadas para contener el nombre del usuario y su correo:

```
<div th:each="user : ${users}" th:with="userName=${user.name}, userEmail=${user.email}">
  <a th:text="${userName}" th:href="${userEmail}"></a>
</div>
```

Nota: hay que tener en cuenta que las variables definidas en la plantilla tienen un alcance local. Solo se pueden usar dentro del rango del elemento en el que se definieron.

7.3.4. Cambiar el valor de una variable.

También es posible sobrescribir el valor de una variable en un ámbito determinado. En el siguiente ejemplo, redefiniremos el atributo users del modelo para que contenga sólo los dos primeros usuarios:

```
<div th:with="users = ${ { users[0], users[1] } }">
  <div th:each="user : ${users}">
    <a th:text="${user.name}" th:href="${user.email}"></a>
  </div>
</div>
```

Fuera del DIV, la variable users aún tendrá su valor original pasado desde el controlador.

7.4. Evaluación condicional.

Thymeleaf incluye el uso de condicionales similares al if y switch de Java.

7.4.1. Condicionales if y unless.

Se usa el atributo th:if="${condition}" para mostrar un elemento de la vista si se cumple una condición. Y se usa el atributo th:unless="${condition}" para mostrar un elemento de la vista si no se cumple una condición.

Utilizaremos la propiedad isAdmin de los objetos de modelo UserForm para ilustrar un ejemplo. Si el usuario es administrador se mostrará una tercera celda con una marca:

```
<table>
  <thead>
    <tr><th>Nombre</th><th>Correo</th><th>Admin</th></tr>
  </thead>
  <tbody>
    <tr th:each="user: ${users}">
      <td th:text="${user.name}"></td>
      <td th:text="${user.email}"></td>
      <td th:if="${user.admin}" th:text="X"></td>
    </tr>
  </tbody>
</table>
```

El propio valor de th:if y th:unless admiten expresiones de comparación. A continuación se ilustran algunos ejemplos:

```
<td th:if="${user.name=='Pedro'}" th:text="Eres Pedro"></td>
<td th:if="${user.name} >='A' and ${user.name}<'Q'" th:text="X"></td>
<td th:if="${user.name} =='Ana' or ${user.name}=='Pedro'" th:text="X"></td>
```

7.4.2. Condicionales switch y case.

Usaremos los atributos th:switch y th:case para mostrar contenido condicionalmente usando una estructura similar a la instrucción switch de Java.

Reescribiremos el ejemplo previo para indicar en una tercera celda si el usuario es administrador o no:

```
<tr th:each="user: ${users}">
  <td th:text="${user.name}"></td>
  <td th:text="${user.email}"></td>
  <td th:switch="${user.admin}">
    <span th:case="true" th:text="SI" />
    <span th:case="false" th:text="NO" />
  </td>
</tr>
```

7.5. Sintaxis para URLs.

Thymeleaf ofrece una forma de crear URLs mediante la sintaxis @{url}. En las páginas se utilizan urls en las etiquetas de hiperlink (<a href="url">), formularios (<form action="url">) imágenes (<img src="url">), etiquetas de cabecera (<link href="url">), etc.

7.5.1. URLs absolutas.

Las URLs absolutas le permiten crear enlaces a otros servidores. Comienzan especificando un nombre de protocolo: http:// o https://

Por ejemplo:

```
<a th:href="@{http://www.thymeleaf/documentation.html}">
```

Genera el siguiente elemento HTML:

```
<a href="http://www.thymeleaf/documentation.html">
```

7.5.2. URLs relativas al contexto.

El tipo de URL más utilizado son las relativas al contexto. Estas son URLs que se supone que son relativas a la raíz de la aplicación web una vez que se instala en el servidor. Por ejemplo, si hemos configurado las siguientes propiedades en el fichero «application.properties» de una aplicación de Spring Boot:

```
server.port: 9090
server.servlet.context-path=/cloudftic
```

Entonces nuestra aplicación podrá ser accedida localmente con http://localhost:8080/cloudftic, donde cloudftic será el nombre de contexto.

Las URLs relativas al contexto comienzan con /:

```
<a th:href="@{/pedidos/listado}">
```

Lo cual generará el siguiente elemento HTML:

```
<a href="/cloudftic/pedidos/listado">
```

7.5.3. URL relativas al servidor.

Las URL relativas al servidor son similares a las URL relativas al contexto , excepto que no asumen que la URL se vincule a un recurso dentro del contexto de la aplicación y, por lo tanto, permiten vincular un contexto diferente en el mismo servidor:

```
<a th:href="@{~/otra-app/mostrarDetalles.htm}">
```

El contexto de la aplicación actual se ignorará, por lo tanto, aunque nuestra aplicación se implemente en http://localhost:8080/cloudftic, se generará la siguiente URL:

```
<a href="/otra-app/mostrarDetalles.htm">
```

7.5.4. URLs relativas al protocolo.

Las URLs relativas al protocolo son, de hecho, URL absolutas que mantendrán el protocolo (HTTP, HTTPS) que se utiliza para mostrar la página actual. Suelen utilizarse para incluir recursos externos como estilos, scripts, etc.:

```
<script th:src="@{//scriptserver.example.net/myscript.js}">...</script>
```

…que se renderizará sin modificaciones (excepto para la reescritura de URL), como:

```
<script src="//scriptserver.example.net/myscript.js">...</script>
```

7.5.5. URLs con parámetros.

Podemos añadir parámetros a las URLs de dos formas: como una variable de ruta ("/pedido/4"), o como parámetro de la cadena de consulta ("/pedido?id=4").

Añadir un parámetro de cadena de consulta es tan sencillo como se muestra a continuación:

```
<a th:href="@{/pedido/detalles(id=3)}">
```

Lo que daría como resultado:

```
<a href="/pedido/detalles?id=3">
```

Podemos agregar varios parámetros separándolos con comas:

```
<a th:href="@{/pedido/detalles(id=3, action='show_all')}">
```

Lo que daría como resultado:

```
<!-- El ampersand (&) debe ser escapado mediante &amp; -->
<a href="/pedido/detalles?id=3&amp;action=show_all">
```

Podemos incluir variables de ruta de forma similar especificando un marcador de posición dentro de la ruta de la URL:

```
<a th:href="@{/pedido/{id}/detalles(id=3,action='show_all')}">
```

Lo que daría como resultado:

```
<a href="/pedido/3/detalless?action=show_all">
```

7.5.6. Identificadores de fragmentos de URL.

En HTML, los identificadores de fragmentos permiten localizar una posición dentro de la página a la que apunta la URL. Se usa #identificador al final de la URL para apuntar al id de un elemento dentro de la página.

Con Thymeleaf se pueden incluir identificadores con # en las URL, con y sin parámetros. Siempre se incluirán en la base de la URL, de forma que:

```
<a th:href="@{/home#all_info(action='show')}">
```

Dará como resultado:

```
<a href="/home?action=show#all_info">
```

7.5.7. Reescritura de URL.

La reescritura URL es una técnica que permite concatenar el identificador de sesión como parte de una URL en vez de usar una cookie, como es habitual.

Mediante instrucciones de Java se utiliza el método encondeURL() de la clase javax.servlet.http.HttpServletResponse para realizar esto.

Thymeleaf permite configurar filtros de reescritura de URL llamando al método encodeURL(...) de la clase javax.servlet.http.HttpServletResponse para cada URL de la plantilla. Esta es la forma estándar de admitir operaciones de reescritura de URL en aplicaciones web Java y permite que las URL:

- Detecten automáticamente si el usuario tiene habilitadas las cookies o no, y agregar el fragmento ;jsessionid=... a la URL si no es así, o si es la primera solicitud y aún se desconoce la configuración de las cookies.
- Aplique automáticamente la configuración del proxy a las URL cuando sea necesario.
- Hacer uso (si así está configurado) de diferentes configuraciones de CDN (Content Delivery Network), para vincular contenido distribuido entre varios servidores.

7.5.8. Uso de expresiones en URL.

La sintaxis @{} admite el uso de expresiones para completar la URL. Por ejemplo, tenemos que construir un enlace como el siguiente:

```
<a th:href="@{/pedido/detalles(id=3,action='show_all')}">
```

Pero el valor de los parámetros id y action sólo se conocen en tiempo de ejecución y están disponibles como atributos de modelo.

¡No hay problema! Cada valor de parámetro de URL es, de hecho, una expresión, por lo que puede sustituir fácilmente sus literales con cualquier otra expresión:

```
<a th:href="@{/pedido/detalles(id=${user.id},action=(${user.admin}? 'show_all' : 'show_public'))}">
```

Es más, una expresión URL como:

```
<a th:href="@{/pedido/detalles(id=${user.id})}">
```

…es de hecho un atajo para:

```
<a th:href="@{'/pedido/detalles'(id=${user.id})}">
```

Lo que significa que la propia URL base se puede especificar como una expresión, por ejemplo, una expresión variable:

```
<a th:href="@{${detallesURL}(id=${user.id})}">
```

…o un texto externalizado/internacionalizado:

```
<a th:href="@{#{pedidos.detalles.url}(id=${user.id})}">
```

…incluso se pueden usar expresiones complejas, incluyendo condicionales, por ejemplo:

```
<a th:href="@{(${user.admin}? '/admin/home' : ${user.homeUrl})(id=${user.id})}">
```

O más simple, con th:with:

```
<a th:with="baseUrl=(${user.admin}? '/admin/home' : ${user.homeUrl})"
   th:href="@{${baseUrl}(id=${order.id})}">
```

…o…

```
<div th:with="baseUrl=(${user.admin}? '/admin/home' : ${user.homeUrl})">
  <a th:href="@{${baseUrl}(id=${usr.id})}">...</a>
</div>
```

7.6. Gestión de formularios.

Podemos gestionar la edición de un formulario HTML usando los atributos th:action="@{url}" y th:object="${object}".

El atributo th:action se utiliza para proporcionar la URL de la acción del formulario (a dónde se deben postear los datos editados) y el atributo th:object se utiliza para especificar un objeto al que se enlazarán los datos del formulario enviados.

Los campos individuales del formulario se asignan mediante atributos th:field="*{name}", donde name es la propiedad coincidente del objeto especificado en th:object.

Crearemos un formulario para editar los datos de un usuario (un objeto de la clase de modelo UserForm).

```
<form method="POST" th:action="@{/users/save}" th:object="${user}">
  <div>
    <label th:text="Nombre"></label>
    <input th:field="*{name}" type="text" />
  </div>
  <div>
    <label th:text="Correo"></label>
    <input th:field="*{email}" type="email" />
  </div>
  <div>
    <label th:text="Contraseña"></label>
    <input th:field="*{password}" type="password" />
  </div>
  <div>
    <label th:text="'Es administrador'"></label>
    <input th:field="*{admin}"  th:checked="${user.admin}" type="checkbox" />
  </div>
  <input type="submit" value="Submit" />
</form>
```

Para editar la propiedad admin se usa un campo de tipo Checkbox, donde el atributo th:checked establece el estado de marcado de la casilla. Cuando se realiza el posteo del formulario, si el checkbox está marcado se enviará el valor asociado (en este caso true), y si no está marcado no se enviará nada.

En el controlador crearemos un método GET para retornar la vista, y un método POST para procesar los datos enviados desde el formulario:

```
@Controller
@RequestMapping("/users")
public class UserController {
  @GetMapping("/save")
  public String save(Model model) {
    model.addAttribute("user", new UserForm());
    return "usersave";
  }
  @PostMapping("/save")
  public String save(@ModelAttribute("user") UserForm user, BindingResult errors) {
```

```
        // lógica para almacenar el usuario
        return "redirect:save";
    }
```

La solicitud para mostrar el formulario será "users/save", la misma que para postear los datos. Por eso, en este ejemplo, no será necesario asignar el atributo th:action.
La vista recibe como objeto una nueva instancia de UserForm cada vez que se solicita. En el método save() la anotación @ModelAttribute enlaza los campos de formulario al objeto UserForm.

Nota: Para que el mecanismo de inyección y validación de modelo funcione correctamente, es necesario que la anotación @ModelAttribute indique el nombre que identifica el objeto de modelo.

7.7. Visualización de errores de validación.

Podemos usar la función #fields.hasErrors() para comprobar si un campo tiene algún error de validación. Y usaremos la función #fields.errors() para mostrar errores para un campo en particular. El nombre del campo es el parámetro de entrada para ambas funciones (o bien '*' para indicar cualquier campo).
Veamos cómo iterar y mostrar los errores para cada uno de los campos del formulario:

```
<ul>
  <li th:each="err : ${#fields.errors('name')}" th:text="${err}" />
  <li th:each="err : ${#fields.errors('password')}" th:text="${err}" />
</ul>
```

En lugar del nombre del campo, las funciones anteriores aceptan el carácter comodín '*' o la constante 'all' para indicar todos los campos. Se utiliza el atributo th:each para iterar sobre los múltiples errores que pueden estar presentes para cada uno de los campos.
Aquí está el código HTML anterior reescrito usando el comodín *:

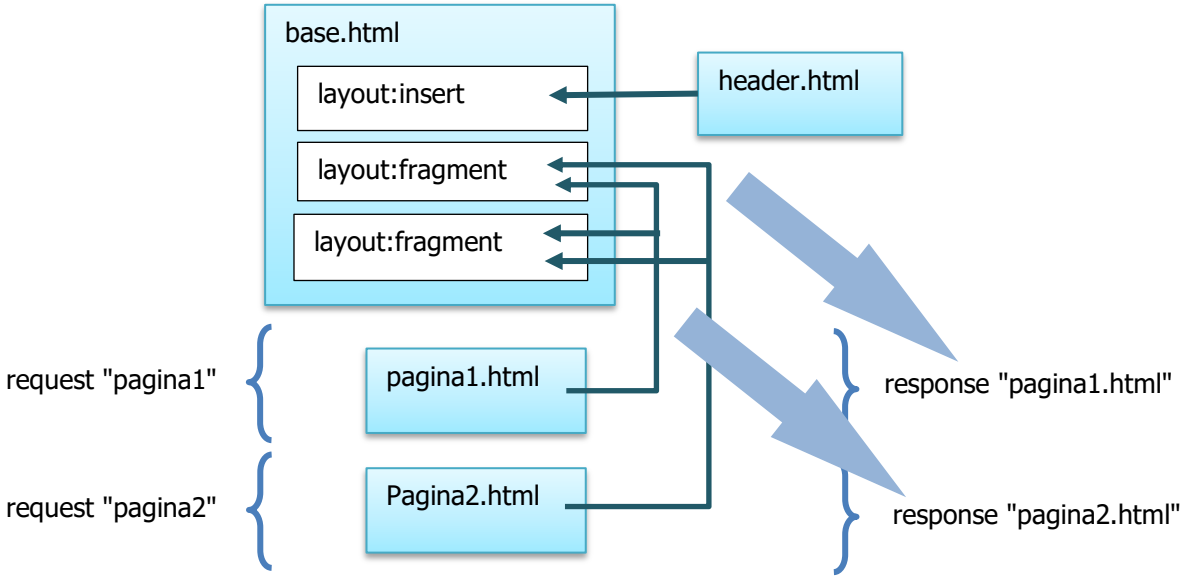
```
<ul th:if="th:text="${#fields.hasErrors('*')}">
  <li th:each="err : ${#fields.errors('*')}" th:text="${err}" />
</ul>
```

Del mismo modo, podemos mostrar errores globales en Spring usando la constante 'global'.
Además, se utiliza el atributo th:errors para mostrar mensajes de error específicos. El código anterior para mostrar errores en el formulario se puede reescribir usando este atributo:

```
<ul>
  <li th:errors="*{name}" />
  <li th:errors="*{password}" />
</ul>
```

7.8. Dialecto de diseño.

En la mayoría de diseño de plantillas, las páginas deben compartir componentes comunes como el encabezado, el pie de página, el menú y potencialmente mucho más.
Thymeleaf aborda eso con Layout Dialect: podemos crear diseños utilizando el sistema de diseño estándar de Thymeleaf o el dialecto de diseño , que utiliza el patrón Decorator para trabajar con los archivos de diseño.
La idea es crear una página base o maestra que establezca el diseño general de la páginas y que defina secciones que deben ser rellenadas desde páginas de contenido. Cuando desde un controlador se devuelva una vista de contenido, ésta se fusionará con la página base para crear el contenido completo que se enviará el cliente.



Para integrar Thymeleaf con Spring y su dialecto de diseño son necesarias estas dos dependencias:

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

<!-- Dependencia para utilizar plantillas dialectos de diseño -->
<dependency>
  <groupId>nz.net.ultraq.thymeleaf</groupId>
  <artifactId>thymeleaf-layout-dialect</artifactId>
</dependency>
```

Con estas dependencias, Spring Boot se encarga de configurarlo todo para usar Layout Dialect.

7.8.1. Procesadores de atributos y espacios de nombres

Una vez configurado, podemos comenzar a usar el espacio de nombres de diseño y cinco nuevos atributos: decorate, title-pattern, insert, replace y fragment.

Como ejemplo de plantilla de diseño para nuestros archivos HTML, creamos el siguiente archivo, llamado «template.html»:

```
<!DOCTYPE html>
<html xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
  ...
</html>
```

Como podemos ver, cambiamos el espacio de nombres del estándar xmlns:th="http://www.thymeleaf.org" a xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout".

Ahora podemos empezar a trabajar con procesadores de atributos en nuestros archivos HTML.

7.8.2. Fragmentos: «layout:fragment»

Para crear secciones personalizadas (fragmentos) en nuestra plantilla «template.html» reutilizable que puedan ser rellenadas por otras plantillas, se usa el atributo fragment dentro de nuestro:

```
<body>
  <header>
    <h1>Ejemplo de Layout Dialect</h1>
  </header>
  <section layout:fragment="content">
    <p>El contenido de la página va aquí</p>
  </section>
  <footer>
    <p>El pie de página personlaizado</p>
    <p layout:fragment="footer">El pie de página va aquí</p>
  </footer>
```

```
</body>
```

El contenido de los dos elementos (o fragmentos) con el atributo layout:fragment será reemplazado por páginas personalizadas. A continuación crearemos la página «content.html» que proporcionará contenido a los fragmentos "content" y "footer".

También es importante escribir HTML predeterminado que se utilizará si no se encuentra alguno de los fragmentos.

7.8.3. Decorador: «layout:decorate»

El siguiente paso es crear páginas de contenido que proporcionen el HTML que rellenará los fragmentos definidos en «template.html».

Un archivo «content.html» puede ser así:

```
<!DOCTYPE html>
<html xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorate="~{template.html}">
  <head>
    <title layout:title-pattern="$LAYOUT_TITLE : $CONTENT_TITLE">Contenido</title>
  </head>
  <body>
    <section layout:fragment="content">
      <p>Este el el contenido principal de la página</p>
    </section>
    <footer layout:fragment="footer">
      <p>Algún contenido para el pie de página</p>
    </footer>
  </body>
</html>
```

Se utiliza layout:decorate para asociar esta página «content.html» con la plantilla maestra «template.html». Todos los fragmentos del archivo de diseño que coincidan con fragmentos en un archivo de contenido serán reemplazados por su implementación personalizada.

7.8.4. Patrón de título: «layout:title-pattern»

Dado que Dialect Layout anula automáticamente el título de la página de diseño con el que se encuentra en la plantilla de contenido, podemos conservar parte del título que se encuentra en el diseño.

El atributo layout:title-pattern permite especificar en su valor dos constantes:

$LAYOUT_TITLE: recupera el título de la página de diseño

$CONTENT_TITLE: recupera el título de la página de contenido

Por lo tanto, en nuestras páginas de ejemplo, teniendo:

```
<title layout:title-pattern="$LAYOUT_TITLE : $CONTENT_TITLE">Contenido</title>
```

Obtendremos el siguiente elemento de título:

```
<title>Ejemplo de Layout Dialect : Contenido</title>
```

7.8.5. Insertar o reemplazar: «layout:insert»/«layout:replace»

El atributo layout:insert permite insertar directamente el contenido de una página dentro de una sección de la plantilla de diseño. Esto es muy útil si tenemos páginas con un contenido que podemos reutilizar.

El atributo layout:replace, es similar al primero, pero con la diferencia de que sustituirá la propia etiqueta donde se utiliza th:replace por el código de la página referenciada.

Por ejemplo, supongamos que en el fichero «header.html» se presenta una cabecera común para todas las páginas de la aplicación. Y que parte del contenido de esta cabecera dependerá de atributos de modelo que deben ser pasados desde el controlador.

Entonces, la página «header.html» contendrá lo siguiente:

```
<section layout:fragment="headerFrag(texto)">
  <img src="logo.jpg" /><h1 th:text="${texto}"></h1>
</section>
```

En el atributo layout:fragment se establece un identificador headerFrag y, con sintaxis de función, un parámetro texto. Este parámetro se utilizará para recibir un dato de modelo a través de la página de diseño, y como vemos, se comporta como un atributo cuyo valor podemos recuperar con ${texto}.

Ahora, en la página de diseño «template.html» insertaremos «header.html»:

```
<body>
  <header>
    <div layout:insert="~{/header.html :: headerFrag(texto='UN LOGO')}"></div>
  </header>
  ......
</body>
```

En el atributo layout:insert se indica la ruta de la página que queremos insertar, y con el separador :: indicamos el identificador de fragmento para asignar un valor al parámetro texto. En este caso se asigna un valor literal, pero también puede ser obtenido desde un atributo de modelo:

```
layout:insert="~{/header.html :: headerFrag(texto=${headerText})}"
```

8. Gestión de validación en Spring Boot.

Spring Boot da soporte para la validación de los objetos de modelo mediante la dependencia Bean Validation:

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

8.1. Conceptos básicos de validación.

Básicamente, Bean Validation funciona definiendo restricciones a los campos de una clase mediante anotaciones.

8.1.1. Anotaciones de validación comunes.

Las anotaciones de validación pertenecen al paquete jakarta.validation.constraints, y algunas de las más comunes son:

- @NotNull: indica que un campo no debe ser nulo.
- @NotEmpty: indica que un campo de lista no debe estar vacío.
- @NotBlank: indica que un campo de texto no debe estar vacío (es decir, debe tener al menos un carácter).
- @Min y @Max: indica que un campo numérico solo es válido cuando su valor está por encima o por debajo de un cierto valor.
- @Pattern: indica que un campo de texto solo es válido cuando coincide con una determinada expresión regular.
- @Email: indica que un campo de texto debe ser una dirección de correo electrónico válida.
- @Size: permite especificar un rango de valores para campos de texto (su longitud), listas, mapas y arrays (su número de elementos).

Un ejemplo de uso sería el siguiente:

```
public class UserForm {
  // El nombre no puede estar vacío y debe comenzar por una letra en mayúsculas.
  @NotBlank
  @Pattern(regexp = "[A-Z]\\w*")
  private String name;

  // Debe ser un correo electrónico válido
  @Email
  private String email;

  // La contraseña no puede estar vacía y debe tener al mínimo 8 caracteres
  @NotBlank
  @Size(min = 8)
  private String password;

  private boolean admin;
  .........
}
```

8.1.2. Validadores.

Si no queremos usar anotaciones de validación, para decidir si un objeto es válido podemos usar una instancia de org.springframework.validation.Validator, la cual comprueba si se cumplen ciertas restricciones.

Por ejemplo, la siguiente clase se puede usar para validar objetos UserForm:

```
@Component
public class UserFormValidator  implements Validator {
  // Determina que tipo de objetos valida esta clase
  @Override
  public boolean supports(Class<?> clazz) {
    return clazz == UserForm.class;
  }

  // Valida un objeto
  @Override
  public void validate(Object target, Errors errors) {
    var user = (UserForm) target;
    // Compobamos que el nombre no esté vacío.
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name", "error.name.blank");
    // Comprobamos que el nombre casa con una expresión regular
    if (!user.getName().matches("[A-Z]\\w*")) {
      errors.rejectValue("name", "error.name.pattern");
    }
    // Compobamos que la contraseña no esté vacía.
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "password", "error.password.blank");
    // Comprobamos que la longitud de la contraseña tiene al menos 8 caracteres
    if (user.getPassword()!=null && user.getPassword().length()<8) {
      errors.rejectValue("password", "error.password.length");
    }
    // Comprobamos que el correo sea válido
    if (user.getPassword()!= null && user.getPassword().matches("[^@]+@[^@]+")) {
      errors.rejectValue("email", "error.email.valid");
    }
  }
}
```

En un método de controlador que deba procesar un objeto UserForm, haríamos uso del validador de la siguiente manera:

```
@Controller
@RequestMapping("/users")
public class UserController {
  @Autowired
  private UserFormValidator userFormValidator;
  .....
  @PostMapping("/save")
  public String save(Model model, @Valid  @ModelAttribute("user") UserForm user, BindingResult errors) {
    userFormValidator.validate(user, errors);
    if (errors.hasErrors()) {
      model.addAttribute("user", user);
      return "usersave";
    }
    .....
  }
}
```

8.1.3. @Validated y @Valid

Usando anotaciones de validación, Spring hace la validación por nosotros. Ni siquiera necesitamos crear un objeto Validador nosotros mismos. En su lugar, podemos hacerle saber a Spring que queremos validar un determinado objeto. Esto funciona utilizando las anotaciones @Validated y @Valid en el controlador.

La anotación @Validated es una anotación de nivel de clase que podemos usar para indicar a Spring que valide los parámetros que se pasan a un método del controlador.

Por su parte, podemos poner la anotación @Valid en los parámetros y campos del método para decirle a Spring que queremos que se valide.

8.2. Validación de las entradas en un controlador.

Supongamos que hemos implementado un controlador y queremos validar la entrada que pasa un usuario. Hay tres cosas que podemos validar para cualquier solicitud HTTP entrante:

1) el cuerpo de la solicitud,
2) variables dentro de la ruta, y
3) parámetros de consulta.

Veamos cada uno de ellos con más detalle.

8.2.1. Validación del cuerpo de solicitud.

En las solicitudes POST y PUT, es común pasar un objeto Json dentro del cuerpo de la solicitud. Spring Boot se encarga de mapear los datos del objeto Json con un objeto Java. Pero ahora queremos comprobar si el objeto Java entrante cumple con nuestros requisitos.

Para validar el cuerpo de la solicitud (con los datos de un usuario) de una solicitud HTTP entrante, anotaremos el parámetro con @Valid:

```
@RestController
@RequestMapping("/users")
public class UserController {
  .........
  @PostMapping("/save")
  public ResponseEntity<String> save(@Valid  @RequestBody UserForm user) {
    return ResponseEntity.ok("valid");
  }
}
```

Simplemente hemos agregado la anotación @Valid al parámetro, que también está anotado con @RequestBody para marcar que debe leerse desde el cuerpo de la solicitud. Al hacer esto, le estamos diciendo a Spring que pase el objeto a un Validator antes de hacer cualquier otra cosa.

> **Nota:** Si la clase contiene un campo con otro tipo complejo que debe validarse, este campo también necesita para ser anotado con @Valid.

Si la validación falla se lanzará un MethodArgumentNotValidException, y de forma predeterminada, Spring traducirá esta excepción a un estado HTTP 400 (solicitud incorrecta).

8.2.2. Validación de variables de ruta y parámetros de solicitud

La validación de variables de ruta y parámetros de solicitud funciona de manera un poco diferente. No estamos validando objetos Java complejos en este caso, ya que las variables de ruta y los parámetros de solicitud son tipos primitivos o clases como Integer o String.

En lugar de anotar un campo de clase como hemos visto, añadiremos una anotación de restricción directamente al parámetro en el método del controlador. Por ejemplo:

```
@RestController
@RequestMapping("/users")
@Validated
public class UserController {
  .........
  @GetMapping("/validatePathVariable/{id}")
  public ResponseEntity<String> validatePathVariable(@PathVariable("id") @Min(5) int id) {
    return ResponseEntity.ok("valid");
  }
  @GetMapping("/validateRequestParameter")
  public ResponseEntity<String> validateRequestParameter(@RequestParam("param") @Min(5) int param) {
    return ResponseEntity.ok("valid");
  }
  .........
}
```

Tenemos que añadir la anotación @Validated a nivel de la clase para decirle a Spring que evalúe las anotaciones de restricción en los parámetros del método.

En este caso, si la validación falla, se lanzará una ConstraintViolationException, que provocará una respuesta con el estado HTTP 500 (Error interno del servidor).

Si queremos devolver un estado HTTP 400 en su lugar (lo cual tiene sentido, ya que el cliente proporcionó un parámetro no válido, convirtiéndolo en una mala solicitud), podemos agregar un controlador de excepciones personalizado al controlador.

```
@RestController
@RequestMapping("/users")
@Validated
public class UserController {
  .........
  @ExceptionHandler(ConstraintViolationException.class)
  @ResponseStatus(HttpStatus.BAD_REQUEST)
  ResponseEntity<String> handleConstraintViolationException(ConstraintViolationException e) {
    return new ResponseEntity<>(e.getMessage(), HttpStatus.BAD_REQUEST);
  }
}
```

8.2.3. Validar datos en métodos de servicio.

En vez de (o adicionalmente) validar los datos en el nivel del controlador, también podemos validarlos en cualquier componente de Spring. Para ello, utilizamos una combinación de las anotaciones @Validated y @Valid.

```
@Service
@Validated
class ValidatingService{
  public void validateInput(@Valid UserForm user) {
    // hacer algo
  }
}
```

Una vez más, la anotación @Validated solo se evalúa a nivel de clase, así que no hay que colocarla en un método.

8.2.4. Validación de entidades JPA.

La última línea de defensa para la validación es la capa de persistencia. De forma predeterminada, Spring Data usa Hibernate por debajo, que admite la validación de beans. Aunque deberíamos evitar este tipo de validación, puesto que significa que en la capa de la lógica de la aplicación se han transmitido datos potencialmente no válidos que pueden dar lugar a errores imprevistos.

Partamos de una clase de entidad:

```
@Entity
public class Input {
  @Id
  @GeneratedValue
  private Long id;

  @Min(1)
  @Max(10)
  private int nota;

  @Pattern(regexp = "^[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}$")
  private String direccion;

  // ...
}
```

Y creamos un repositorio:

```
public interface ValidatingRepository extends CrudRepository<Input, Long> {
}
```

De forma predeterminada, cada vez que usemos el repositorio para almacenar un objeto cuyas anotaciones de restricción se violan, se lanzará una InputConstraintViolationException.

Hay que tener en cuenta que la validación de Hibernate sólo se activa cuando el EntiyManager realiza un flush. Si por alguna razón queremos deshabilitar Bean Validation en los repositorios de Spring Data, podemos configurar la propiedad spring.jpa.properties.javax.persistence.validation.mode a valor none.

8.3. Un validador personalizado con Spring Boot.

Si las anotaciones de restricción disponibles no son suficientes para nuestros casos de uso, es posible que deseemos crear una nosotros mismos.

Vemos como crear un ejemplo trivial que valida un campo entero para que se encuentre dentro de un rango de valores.

```
@Target({ FIELD })
@Retention(RUNTIME)
@Constraint(validatedBy = NumberInRangeValidator.class)
@Documented
public @interface NumberInRange {
  String message() default "{NumberInRange.invalid}";
  Class<?>[] groups() default { };
  Class<? extends Payload>[] payload() default { };
  int min() default 0;
  int max() default Integer.MAX_VALUE;
}
```

Una anotación de restricción personalizada necesita todo lo siguiente:

- El parámetro message, que referencia a una clave de propiedad en ValidationMessages.properties, que se utiliza para resolver un mensaje en caso de violación.
- El parámetro groups, que permite definir en qué circunstancias se activará esta validación.
- El parámetro payload, que permite definir una carga útil que se pasará con esta validación (esto es una característica que rara vez se usa)
- Un anotación @Constraint que apunta a una implementación de la interfaz ConstraintValidator.

La implementación del validador será como sigue:

```
class NumberInRangeValidator implements ConstraintValidator<NumberInRange, String> {
  private NumberInRange constraintAnnotation;
  @Override
  public void initialize(NumberInRange constraintAnnotation) {
    this.constraintAnnotation = constraintAnnotation;
  }

  @Override
  public boolean isValid(String value, ConstraintValidatorContext context) {
    Integer num = Integer.parseInt(value);
    return num >= constraintAnnotation.min() && num <= constraintAnnotation.max();
  }
}
```

Ahora podemos usar la anotación de la misma forma que cualquier otra:

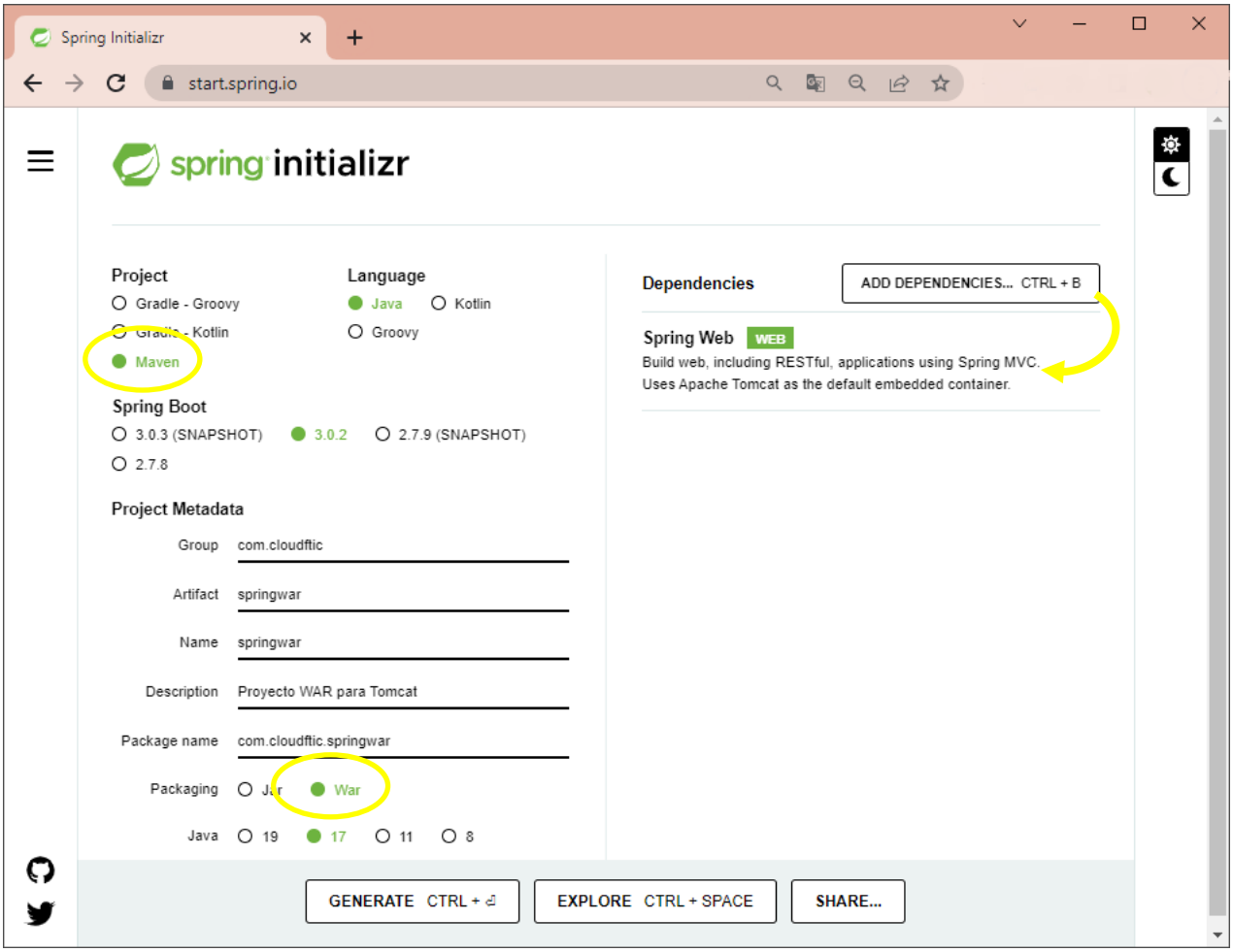
```
@NumberInRange(min=5, max= 10)
private int valor;
```

9. Aplicaciones WAR y JSP

En las aplicaciones Spring Boot, el empaquetado predeterminado es JAR, de forma que la aplicación se implementa utilizando un servidor Tomcat básico integrado. Para generar aplicaciones que se ejecuten a través de servidores web externos (como Jboss, Weblogic o Tomcat) debemos generar un fichero WAR.

9.1. ¿Cómo crear aplicaciones WAR?

Para poder crear una aplicación WAR con Spring Boot debemos seleccionar las opciones adecuadas al generar el proyecto.



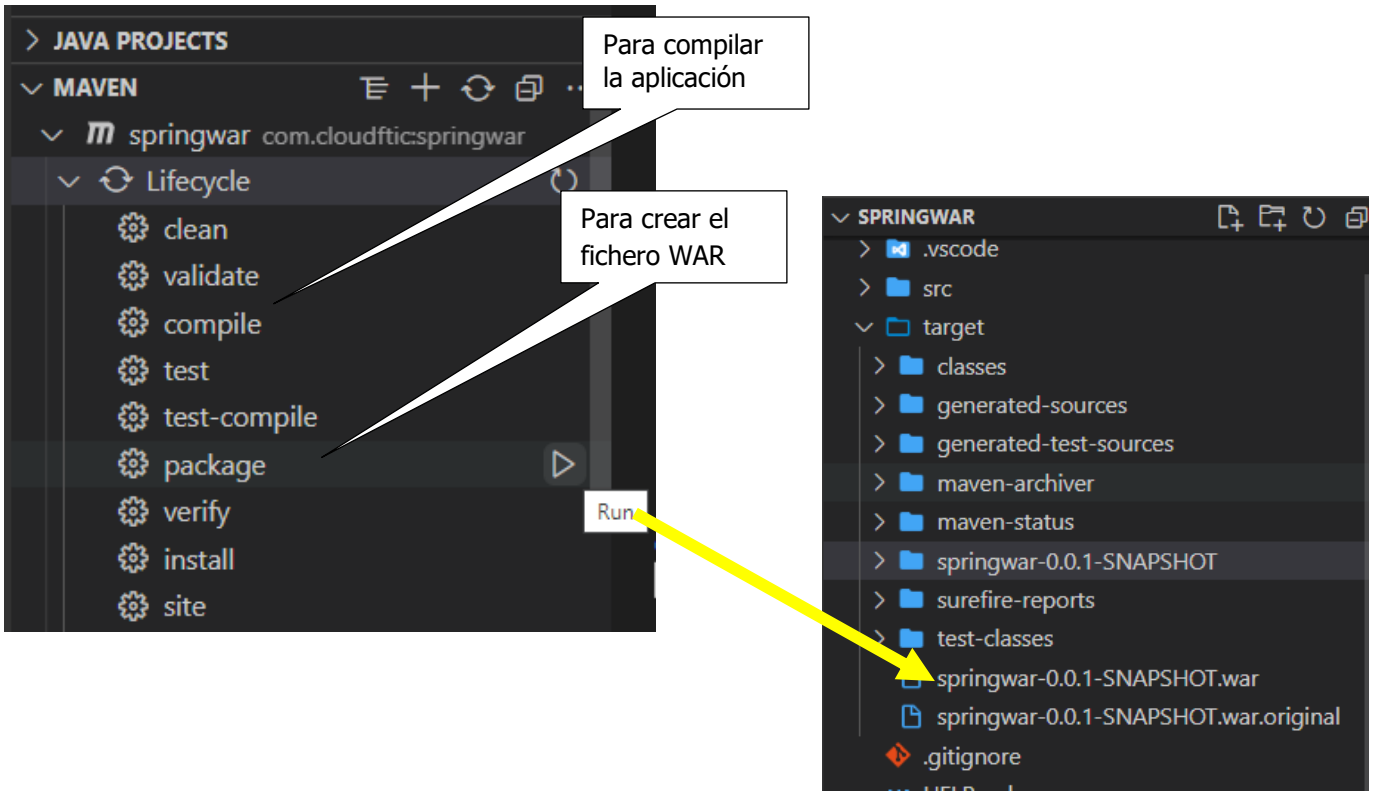
En la opción de empaquetado debemos seleccionar (war). Con esto se generarán las siguientes dependencias:

```
<packaging>war</packaging>

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

El scope o ámbito provided indica que espera que el JDK o un contenedor proporcione la dependencia en tiempo de ejecución. Este alcance solo está disponible en el classpath de compilación y prueba, y no es transitivo.
En la aplicación se habrá creado una clase ServletInitializer:

```
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

public class ServletInitializer extends SpringBootServletInitializer {
  @Override
  protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
    return application.sources(SpringwarApplication.class);
  }
}
```

Esta clase inicializa el contexto de Servlet, lo cual permite ejecutar nuestra aplicación en un contenedor Web con la tecnología JEE de Java, que es la que soportan los servidores web como Tomcat.
Ahora, al empaquetar la aplicación en un fichero WAR, podremos ejecutarla como cualquier contenedor web JEE de forma tradicional.
Si usamos VS Code como entorno de desarrollo podemos usar los comando de Maven para compilar y empaquetar el proyecto:



En la carpeta «target» se habrá creado el fichero «springwar-0.0.1-SNAPSHOT.war». (El nombre del fichero WAR dependerá del nombre y la versión del proyecto Maven.)

9.2. Java Server Pages (JSP).

JSP fue la tecnología de vista más popular para aplicaciones Java y también es compatible con Spring. JSP, a su vez, está basado en los componentes Servlets del Framework JEE.
Los servlets son clases de Java que se ejecutan en un servidor como respuesta a una solicitud HTTP desde un cliente, y que (aparte de otros procesos) normalmente generan una página web como respuesta. De hecho, los controladores de Spring son corridos por un servlet a bajo nivel.
Un contenedor web (o también llamado contenedor de servlets) es un módulo que permite mantener la arquitectura multicapa de Java EE. Aquellos servidores web que soportan la plataforma Java EE, como Apache Tomcat, implementan este módulo.
Este módulo especifica, dentro de un servidor, un entorno de ejecución para componentes web que incluye seguridad, concurrencia, gestión del ciclo de vida, procesamiento de transacciones, despliegue y otros servicios. Un contenedor web da soporte a páginas JSP así como de otros tipos de recursos de la plataforma J2EE.
Un modelo muy simplificado de cómo funciona un contendor web viene dado por el siguiente escenario:

1) Una página web, que contiene un formulario, es mostrada en un navegador cliente.
2) El usuario rellena el formulario y pulsa el botón para postear los datos. Los datos son posteados mediante una solicitud Post o Get, y enviados a un recurso especificado por la ruta de la solicitud.
3) En el servidor web, el contenedor web determina el servlet asociado a la ruta solicitada.
4) El servlet recupera los datos posteados, los procesa y genera una página web de respuesta hacia el navegador cliente.

Las páginas JSP surgieron como una simplificación de este uso de los Servlets, puesto que incluyen el código HTMl de la vista, y código Java oculto de la lógica que aplicaría un servlet.

9.2.1. Configuración básica.

Para representar archivos JSP, el tipo de bean ViewResolver más utilizado es InternalResourceViewResolver:

```
@Configuration
public class ApplicationConfiguration {
  @Bean
  public ViewResolver jspViewResolver() {
    InternalResourceViewResolver bean = new InternalResourceViewResolver();
    bean.setPrefix("/WEB-INF/jsp/");       // Ubicación de las vistas respecto a la raíz de la aplicación
    bean.setSuffix(".jsp");                // Extensión de las páginas
    return bean;
  }
}
```

Esta clase de configuración puede ser sustituida por propiedades en «application.properties»:

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

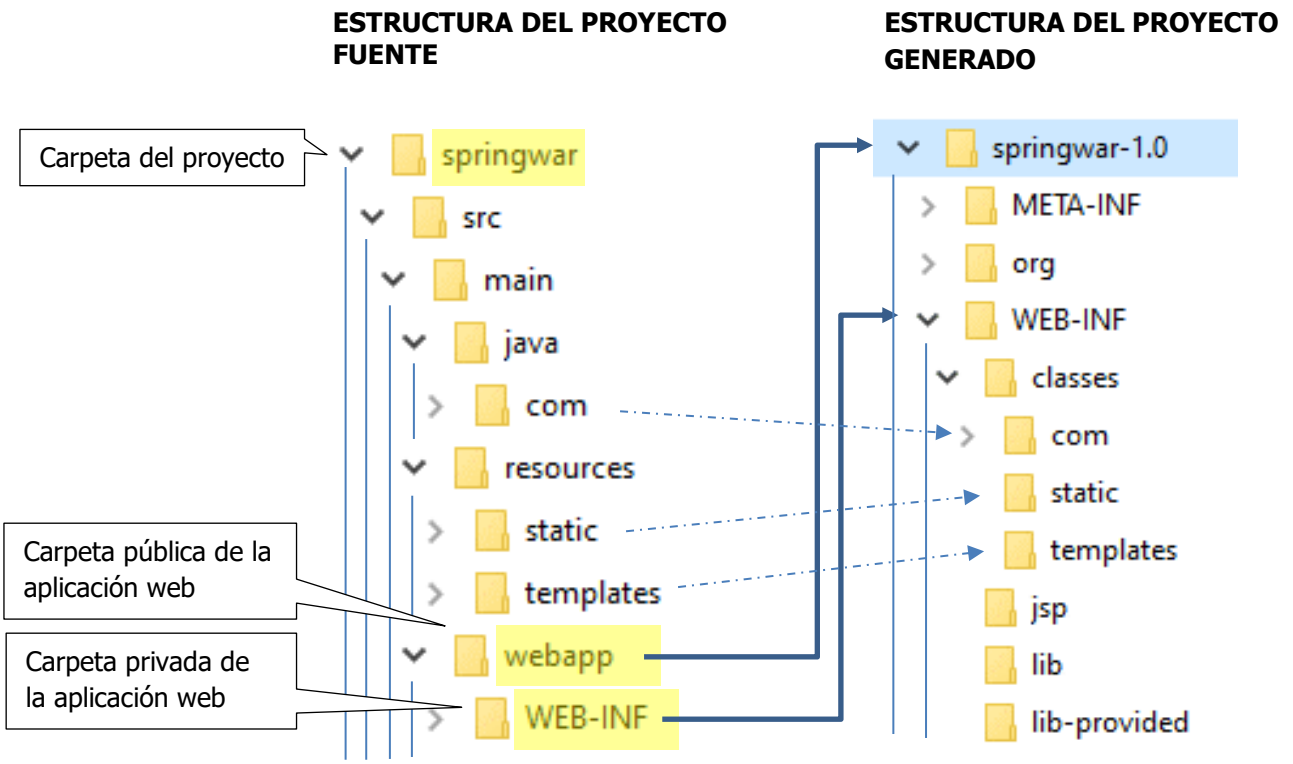
Al igual que las plantillas Thymeleaf, las páginas JSP están basadas en HTML pero, para enriquecerlas, se utilizan etiquetas definidas en ficheros TLD. Las etiquetas más básicas se denominan JSTL, y las dependencias necesarias para usarlas con Spring Boot 3.0 son:

```
<dependency>
  <groupId>jakarta.servlet.jsp.jstl</groupId>
  <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
  <version>3.0.0</version>
</dependency>
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>apache-jstl</artifactId>
  <version>11.0.0</version>
</dependency>
```

Nota: esta combinación de dependencias puede que no funcione para otras versiones de Spring Boot.

9.2.2. Arquitectura de la aplicación.

Las aplicaciones web basadas en JSP requieren una estructura de capetas determinadas, tal como se muestra en la siguiente imagen:



La carpeta fuente «webapap» se corresponde con el contenido raíz de la aplicación web que se va a generar. En esta carpeta se pueden ubicar páginas JSP o cualquier recurso estático que queramos que sean públicos. La subcarpeta «webapp/WEB-INF» es opcional en el proyecto fuente, pero siempre se incluye en el proyecto generado. Esta carpeta incluye contenido privado de la aplicación:

- Se recomienda que los JSP se ubiquen dentro de esta carpeta (o en una subcarpeta) para que sean privados. Su acceso se producirá a través de un servlet o un controlador.
- Opcionalmente puede contener un fichero «web.xml» (denominado archivo descriptor de la aplicación) que incluye configuraciones de inicialización del contenedor web.
- En una subcarpeta «classess» contiene los paquetes y ficheros .class de la aplicación.
- En una subcarpeta «lib» contiene los ficheros .jar correspondientes a librerías y dependencias incluidas en el proyecto.

9.3. Servlets.

Aunque las aplicaciones de Spring Boot no requieren de servlets, puesto que usan clases controladoras, veremos tres ejemplos de su uso. Los servlets, al ser clases de Java se crean dentro de la carpeta fuente «src/main/java/».
En este primer ejemplo, usaremos un servlet para que retorne directamente un contenido:

```
package com.cloudftic.springwar.servlets;

import java.io.IOException;
import java.io.PrintWriter;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns = "/prueba1")
public class Servlet1 extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException {
    resp.setContentType("text/plain");
    PrintWriter out = resp.getWriter();
    out.append("Este servlet funciona en la ruta:")
      .append(req.getServletContext().getContextPath())
      .close();
  }
}
```

La anotación @WebServlet registra la clase Servlet1 como un servlet y queda asociado a la uri "prueba1". La clase debe heredar de la clase HttpServlet, que proporciona métodos doGet(), doPost(), doDelete(), etc. para gestionar cada método de solicitud HTTP.
Estos métodos siempre reciben como argumento un objeto HttpServletRequest que nos dará acceso a toda la información de solicitud y un objeto HttpServletResponse que permite generar una respuesta.
En el siguiente ejemplo, se utiliza el servlet para redireccionar a una página JSP, según el valor de un parámetro de consulta:

```
@WebServlet(urlPatterns = "/prueba2")
public class Servlet2 extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException {
    // Se recupera un parámetro de solicitud llamado 'estado'
    String estado = req.getParameter("estado");
    // si no existe parámetro se lanza una excepción
    if (estado == null)
      throw new ServletException("Falta el parámetro 'estado'.");
    // Obtenemos la url a la que hay que redireccionar:
```

```
        String href = req.getContextPath(); // la url base
        href += estado=="1"? "/estado1.jsp" : "/estado2.jsp";
        // Respuesta de redirección:
        resp.sendRedirect(href);
      }
    }
```

Por último redireccionaremos internamente a una página JSP, en la cual se inyectan datos del modelo:

```
    @WebServlet(urlPatterns = "/prueba3")
    public class Servlet1 extends HttpServlet {
      @Override
      protected void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
        // Se pasan atributos al modelo de datos de la página
        req.setAttribute("items", List.of("Uno", "Dos", "Tres"));
        // Redirección interna:
        req.getRequestDispatcher("/pagina.jsp").forward(req, resp);
      }
    }
```

9.4. Diseño de páginas JSP.

A continuación, podemos ver la plantilla de una página JSP:

Fichero «pagina.jsp»

```
<!DOCTYPE html>
<%@page contentType="text/html" pageEncoding="UTF-8" language="java" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  </head>
  <body>
    <%-- Comentario: Creación de un atributo local --%>
    <c:set var="x" value="3" />

    <%-- Comentario: Se recuperan atributos de modelo con la sinxtaxis ${modelo} --%>
    <p>Dato del modelo ${dato}</p>

    <%-- Comentario: Ejemplos de condicionales --%>
    <c:if test="${dato eq true}">
      <p>Se cumple la condición</p>
    </c:if>

    <c:choose>
      <c:when test='${x le 3}'>
        <p>La X es menor o igual que 3</p>
      </c:when>
      <c:when test='${x le 6}'>
        <p>La X es mayor que 3 y menor o igual que 6</p>
      </c:when>
      <c:otherwise>
        <p>La X es mayor 6</p>
      </c:otherwise>
    </c:choose>

    <%-- Comentario: Ejemplo de bucle --%>
```

```
    <ul>
      <c:forEach var="item" items="${items}">
        <li>${item}</li>
      </c:forEach>
    </ul>
  </body>
</html>
```

9.4.1. Directivas.

Las etiquetas que comienza por <%@ se denominan directivas, y le dicen al motor de plantillas JSP que debe ejecutar determinadas instrucciones.

La directiva <%@page %> informa al motor de plantillas JSP acerca de varias propiedades de la página. Los atributos que podemos usar dentro de esta etiqueta son los siguientes:

import="java.util, java.util.*"	Permite importar clases. Es el equivalente a una instrucción import de un fichero .java.
contentType="text/html"	Indica el tipo de contenido que debe generar el fichero.
session="true|false"	Indica si deben crearse sesiones automáticamente al invocar la página.
buffer="100|none"	Indica el tamaño del buffer (en KBytes) de la salida. El valor mínimo debería ser 8kb.
autoflush="true|false"	El valor de true (por defecto) indica que el buffer de salida debe descargase cuando esté lleno. Un valor de false, raramente utilizado, indica que se debe lanzar una excepción cuando el buffer se sobrecargue.
extends="package.class"	Indica una superclase de la que debe heredar esta página.
info="message"	Define un string que puede ser recuperado mediante el método getServletInfo().
errorPage="url"	Indica una página JSP que se debería mostrar si se produce cualquier excepción o error no capturado en la página actual.
isErrorPage="true|false"	Indica si la página actual actúa o no como página de error de otra página JSP. El valor por defecto es false.
language="java"	Especifica el lenguaje para instrucciones backend (habitualmente java).

Otras directivas son las siguientes:

- <%@include %>, permite insertar el contenido de un fichero. Su sintaxis completa es:
  <%@include file="url relativa" %>

  La dirección URL es relativa a la página JSP de referencia. Los contenidos del fichero incluido son analizados como texto normal JSP, y así pueden incluir código HTML, elementos de script, directivas y acciones.
- <%@taglib %>, permite referencias a librerías de etiquetas personalizadas (TLD). El conjunto de librerías JSTL incluye los siguientes TLDs:

  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  Proporciona etiquetas como <c:if>, <c:forEach>, <c:set>, etc. correspondientes a instrucciones habituales de programación.

  <%@taglib prefix="xml" uri="http://java.sun.com/jsp/jstl/xml" %>
  Proporciona etiquetas para manipular documentos XML.

  <%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
  Proporciona etiquetas para aplicar formatos a números y fechas, y etiquetas para localizar la aplicación mediante ficheros de propiedades de idiomas.

  <%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
  Proporciona etiquetas para manipular bases de datos SQL.

9.4.2. Etiquetas JSP.

El motor JSP incluye una librería de etiquetas integradas con el prefijo <jsp: >. Estas etiquetas son comandos que obligan al motor JSP a realizar ciertas tareas durante la ejecución de la página. A continuación se resumen:

- <jsp:include page="url" />
  Permite incluir el contenido de un fichero en tiempo de ejecución. Se puede usar para incluir trozos de HTML y JSP de otras páginas.
- <jsp:forward page="url" />

Permite redireccionar a otra URL. Por ejemplo, podemos basarnos en una parámetro de solicitud para tomar decisiones de redirección:

```
<body>
  <c:if test="${param.status eq 0}>
    <jsp:forward page="estado0.jsp" />
  </c:if>
  <c:if test="${param.status eq 1}>
    <jsp:forward page="estado1.jsp" />
  </c:if>
</body>
```

- <jsp:useBean>

  Encuentra o crea un objeto JavaBean como atributo del modelo. Por ejemplo, la siguiente etiqueta:

  ```
  <jsp:useBean id="fecha" class="java.util.Date" scope="session" />
  ```

  ...busca un atributo a nivel de sesión de nombre "fecha". Si lo encuentra lo deja disponible en la página, y si no lo encuentra asigna el atributo a una instancia de la clase indicada utilizando el constructor por defecto. Los valores disponibles para scope son: page (por defecto), request, session y application.

- <jsp:setProperty />

  Asigna propiedades a un JavaBean disponible como atributo del modelo. Se suele utilizar para modificar propiedades de un bean disponible con <jsp:useBean />. Por ejemplo, suponiendo la existencia de una clase com.model.VentaForm:

  ```
  <jsp:useBean id="venta" class="com.model.VentaForm"/>
  <jsp:setProperty name="venta" property="id" value="1" />
  ```

  Además, esta etiqueta también permite recuperar automáticamente un valor desde un parámetro de solicitud que posea el mismo nombre:

  ```
  <jsp:setProperty name="venta" property="fecha" />
  ```

  O un nombre diferente:

  ```
  <jsp:setProperty name="venta" property="fecha" param="date" />
  ```

  Esta etiqueta es especialmente útil para poblar un bean con los datos procedentes de un formulario. Por ejemplo, supongamos que un formulario postea datos para poblar una instancia de una clase com.model.VentaForm. Podemos instanciar un objeto de esta clase y poblar sus propiedades desde los parámetros de solicitud de la siguiente forma:

  ```
  <jsp:useBean id="venta" class="com.model.VentaForm"/>
  <jsp:setProperty name="venta" property="*" />
  ```

9.4.3. El Lenguaje de Expresiones JSTL.

Además de las librerías de etiquetas, JSTL define un llamado Lenguaje de Expresiones (EL). Las expresiones EL fueron pensadas inicialmente para acceder a los datos disponibles en los diversos contextos de datos de la aplicación, incluyendo los parámetros de solicitud, cookies, cabeceras y atributos del modelo de datos.

Las aplicaciones basadas en contenedores Servlets defines 4 contextos de atributos asociados a instancias de:

PageContext	Cada página JSP define una variable de nombre pageContext. Este objeto proporciona métodos para recuperar información asociada a la página actual.
HttpServletRequest	Cada solicitud procesada por el contenedor de Servelt provoca una instancia de esta interfaz (asociada a la variable request). Este objeto proporciona todas la información disponible en el mensaje de solicitud, incluidas las cookies de solicitud.
HttpSession	Para cada solicitud se crea una instancia de esta interfaz (asociada a la variable session) para mantener información de sesión.
ServletContext	Durante la ejecución de la aplicación se mantiene un objeto de esta interfaz (asociado a la variable application) que contiene información sobre la propia aplicación.

Cada instancia de cada una de estas interfaces dispone de métodos setAttribute() y getAttribute() para guardar y recuperar atributos de modelo.

Con EL podemos recuperar en un JSP atributos de cada uno de estos contextos:

${pageScope.*atributo*}	Recupera el valor del atributo indicado en el contexto de la página actual
${requestScope.*atributo*}	Recupera el valor del atributo indicado en el contexto creado por la instancia del objeto HttpServeltRequest actual.
${sessionScope.*atributo*}	Recupera el valor del atributo indicado en el contexto de la sesión actual.

${applicationScope.*atributo*}	Recupera el valor del atributo indicado en el contexto de la aplicación actual.
${*atributo*}	Busca un atributo en los contextos siguiendo el orden: pageContext, request, session y application.

Donde pageScope, requestScope, sessionScope y applicationScope son mapas que contienen los atributos del contexto correspondiente.

Y además podemos recuperar parámetros de solicitud, cabeceras y cookies:

${param.*name*}	Recupera el valor del parámetro de solicitud de nombre indicado.
${paramValues.*name*}	Recupera un array de todos los valores asociados al parámetro de solicitud de nombre indicado.
${header.*name*}	Recupera el valor de la cabecera de nombre indicado.
${headerValues.*name*}	Recupera un array de todos los valores de la cabecera de nombre indicado.
${cookie.*name*}	Recupera el valor de la cookie de nombre especificado.

Si un atributo no se encuentra en el contexto especificado, la expresión EL retorna el valor null.

Si el atributo recuperado es un objeto, EL presenta tres sintaxis para recuperar sus propiedades:

${*objeto.nombrePropiedad*}
${*objeto*["nombre de propiedad"]}
${objeto[*expresiónQueDevuelveUnNombreDePropiedad*]}

La sintaxis con corchetes también se usa para acceder a celdas de un array o una lista, o al valor de una mapa por su clave:

${unArray[2]}	
${unaLista[indice + 1]}	donde indice es un atributo que contiene un número entero
${unMapa[key] }	donde key es un atributo que contiene una clave existente en el mapa

Además de los operadores de propiedad y elemento array y los operadores aritméticos, relacionales, y lógicos, hay un operador especial para comprobar si un objeto está "vacío" o no puede ser usado en una expresión EL (empty). La siguiente tabla lista todos los operadores que podemos usar en expresiones EL:

Operador	Descripción
.	Accede a una propiedad
[]	Accede a un elemento de un array/lista
()	Agrupa una subexpresión
+	Suma
-	Resta o negación de un número
/ ó div	División
% ó mod	Módulo (resto)
== ó eq	Comprueba Igualdad
!= ó ne	Comprueba desigualdad
< ó lt	Comprueba menor que
> ó gt	Comprueba mayor que
<= ó le	Comprueba menor o igual que
>= ó gt	Comprueba mayor o igual que
&& ó and	Comprueba AND lógico
|| ó or	Comprueba OR lógico
! ó not	Complemento binario booleano
empty	Comprueba un valor vacío (null, string vacío, o una colección vacía)

Lo que no encontraremos en EL son sentencias como asignaciones, if/else, o while. Para este tipo de funcionalidades en JSP se usan las etiquetas de la librería Core de JSTL; y EL no está pensado para utilizarse como un lenguaje de programación de propósito general, sino como un lenguaje de acceso a datos.

9.4.4. Librería Core JSTL.

La librería Core JSTL incluye el grupo de etiquetas que suplen las instrucciones de programación de Java. La tabla siguiente describe someramente estas etiquetas:

c:if	Implementa una estructura selectiva. Si se cumple la condición asignada en el atributo test se evalúa el cuerpo, sino no se avalúa.

choose	Implementa una estructura selectiva similar a la estructura switch de Java.
c:catch	Captura las excepciones producidas en el cuerpo. El valor de la excepción es asignado en el atributo opcional var, el cual define un atributo del ámbito de página.
forEach	Implementa un bucle para iterar sobre un rango o una colección de elementos.
forTokens	Implementa un bucle para iterar sobre un texto con caracteres delimitadores.
set	Crea o modifica una variable en el ámbito especificado (por defecto a nivel de página) con un valor especificado.
remove	Elimina una variable especificada en el nivel indicado.
import	Recuperar el contenido de un recurso.
out	Expone el resultado de evaluar una expresión sobre la página.
redirect	Redirige la respuesta de la página a una Url especificada.
url	Normaliza una Url.

Acciones de selección.

La etiiqueta <c:if> es análoga a la instrucción if(){} de Java, pero si parte else. Por ejemplo, el siguiente marcado permite comprobar si existe un parámetro de solicitud:

```
<c:if test="${! empty param.user}">
  Bienvenido ${param.user}"
</c:if>
```

La etiqueta <c:choose> permite agrupar una o más acciones <c:when>, cada una de las cuales permite especificar una condición booleana diferente. Las acción <c:choose> verifica cada condición en orden y sólo se ejecutará el cuerpo de la primera acción <c:when> cuya condición se evalúe a true. El cuerpo de <c:choose> también puede contener una acción <c:otherwise>, cuyo cuerpo sólo se procesará si ninguna de las condiciones de los <c:when> es true.

Por ejemplo, el siguiente código evalúa el valor de un parámetro de solicitud para determinar a qué página redirigir.

```
<c:choose>
  <c:when test='${param.dir eq "a"}'>
    <jsp:forward page="paginaA.jsp" />
  </c:when>
  <c:when test='${param.dir eq "b"}'>
    <jsp:forward page="paginaB.jsp" />
  </c:when>
  <c:otherwise>
    <jsp:forward page="otraPagina.jsp" />
  </c:otherwise>
</c:choose>
```

Según este comportamiento, podemos usar esta etiqueta para implementar un if/else:

```
<c:choose>
  <c:when test='${param.dir eq "a"}'>
    <jsp:forward page="paginaA.jsp" />
  </c:when>
  <c:otherwise>
    <jsp:forward page="otraPagina.jsp" />
  </c:otherwise>
</c:choose>
```

Captura de excepciones.

Para capturar excepciones se utiliza la etiqueta <c:catch />. Posee la siguiente sintaxis:

```
<c:catch var="error">
  <%-- código que pueda lanzar una excepción --%>
</c:catch>
```

Donde el atributo var indica el nombre de una variable de tipo Throwable, que será creada en el ámbito de página y que contendrá la excepción generada en el cuerpo de la etiqueta.

Después de la etiqueta <c:catch /> podemos evaluar el valor de la variable error mediante una expresión EL. Como ejemplo supongamos una página que recupera un parámetro de solicitud llamado contador, el cual debe convertir a un entero. Si no existe el parámetro o no es convertible se mostrará un mensaje:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <c:catch var="ex">
      <% int contador = Integer.parseInt(request.getParameter("contador")); %>
      El valor de contador es <%= contador %>
    </c:catch>
    <c:if test="${!empty ex}">
      Falta el contador o no es convertible
    </c:if>
  </body>
</html>
```

Visualizar valores de expresiones.

Para visualizar los valores de una expresión se utiliza la acción <c:out />, como en el siguiente ejemplo:

```
Hay <c:out value="${producto.stockActual}" escapeXml="true" default="0" /> productos en el stock.
```

Donde el atributo value especifica la expresión a visualizar, el atributo escapeXml indica si hay que aplicar códigos de escape a los caracteres <, >, & y . (punto), y el atributo default indica un valor por defecto si la expresión no puede evaluarse.

Crear, modificar y eliminar variables de ámbito.

Para crear una variable en uno de los contextos (request, page, session o application) se usa la acción <c:set />:

```
<c:set var="idCliente" value="${param.numeroCliente}" scope="session" />
```

Donde el atributo var indica el nombre de la variable a crear o modificar, el atributo value especifica una expresión que evalúa al valor que se quiere asignar, y el atributo scope indica el contexto en el que se define la variable.

También se puede usar <c:set /> para asignar el valor de la propiedad de un objeto, de forma parecida a como lo hace la acción <jsp:setProperty />:

```
<jsp:useBean id="persona1" class="bean.Persona" />
<%-- Asignamos la propiedad "nombre" --%>
<c:set target="${persona1}" property="nombre" value="Juan Pérez" />
```

En este caso, el atributo target establece el objeto que queremos modificar, y el atributo property establece una propiedad del objeto especificado en target.

Si el objeto especificado en target es de tipo java.util.Map, el atributo property especificará una clave para el mapa, y el atributo value el valor asociado a dicha clave.

```
<%-- Creo un objeto de tipo HashMap y le añado un elemento con la clave "uno" y el valor 1 --%>
<jsp:useBean id="mapa1" class="java.util.HashMap" />
<c:set target="${mapa1}" property="uno" value="1" />
```

También podemos asignar el cuerpo de la etiqueta <c:set> a una variable. El siguiente código crea una variable en el ámbito de página llamado contenidoCelda con el contenido del cuerpo de la etiqueta <c:set />.

```
<c:set var="contenidoCelda">
  <td>
    <c:out value="${miCelda}"/>
  </td>
</c:set>
```

Para eliminar una variable de un ámbito se utiliza la acción <c:remove />:

```
<c:remove var="unaVariable">
```

Bucle para colecciones.

Se utiliza la etiqueta c:forEach para recorrer cualquier tipo de colección de datos o un rango de valores.

El siguiente código permite iterar sobre un rango de valores entre 1 y 10:

```
<c:forEach begin="1" end="10" var="i">
  <p>Iteración ${i}</p>
</c:forEach>
```

En el atributo var se declara un nombre de variable que estará disponible dentro del cuerpo de la etiqueta, y que en cada iteración tomará valores desde el valor inicial establecido en el atributo begin hasta el valor final establecido en el atributo end.

Pero la gran potencia de esta etiqueta es su capacidad para iterar sobre cualquier tipo de colección. El siguiente fragmento itera sobre la colección de cabeceras de la solicitud:

```
<c:forEach var="cabecera" items="${header}">
  <p>${cabecera.key} = ${cabecera.value}</p>
</c:forEach>
```

La expresión EL para el valor header obtiene el mapa de cabeceras. Cada elemento de un mapa es un objeto de tipo java.util.Map.Entry, el cual posee las propiedades key y value.

Para ilustrar todas las posibilidades que ofrece esta acción, extenderemos el ejemplo de iteración para procesar sólo un conjunto de cabeceras por cada página solicitada, añadiendo enlaces "Anterior" y "Siguiente" en la misma página.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <c:set var="lineas" value="4" />
    <ul>
      <c:forEach var="cabecera" items="${header}"
          begin="${param.inicial}" end="${param.inicial + lineas - 1}" step="1">
        <li>${cabecera.key} = ${cabecera.value}</li>
      </c:forEach>
    </ul>
    <a href="?inicial=${param.inicial-lineas}">Anterior</a>
    <a href="?inicial=${param.inicial+lineas}">Siguiente</a>
  </body>
</html>
```

La acción <c:set> define la variable lineas para indicar cuántas cabeceas se visualizarán por página. El <c:forEach> en este ejemplo toma los mismos valores para los atributos items y var como antes, pero hemos añadido dos nuevos atributos:

- E atributo begin toma el índice (base 0) del primer elemento de la colección a procesar. Aquí se selecciona al valor de un parámetro de solicitud llamado inicial. Para la primera solicitud, este parámetro no está disponible, por eso la expresión se evaluará a 0; en otras palabras, la primera fila.
- El atributo end especifica el índice del último elemento de la colección a procesar. Aquí lo hemos seleccionado al valor del parámetro inicial más lineas menos uno. Para la primera solicitud, cuando no existe el parámetro de la solicitud, este resultado es 3, por eso la acción iterará la primera vez sobre los índices del 0 al 3.
- El atributo step especifica el paso de iteración. Un valor mayor que 1 hará que el bucle se salte elementos de la colección.

Luego añadimos los enlaces "Anterior" y "Siguiente", donde realmente se define el parámetro de solicitud inicial, incrementándolo y decrementándolo apropiadamente.

Si probamos esta página se verá que realmente página las cabeceras. Pero si en la primera página pulsamos el enlace "Anterior" se producirá un error. Esto es porque el atributo begin no admite valores negativos. Para evitar esto podemos condicionar la aparición de este enlace:

```
<c:if test="${param.inicial > 0}">
  <a href="?inicial=${param.inicial-lineas}">Anterior</a>
```

```
</c:if>
```

Si avanzamos más allá de la última página de cabeceras se mostrará una página vacía pero no se producirá error. Aun así condicionaremos también la aparición de este enlace teniendo en cuenta el tamaño de la colección:

```
<c:if test="${param.inicial + lineas < header.size()}">
  <a href="?inicial=${param.inicial+lineas}">Siguiente</a>
</c:if>
```

Iteración sobre textos.

Al igual que al clase StringTokenizer, la etiqueta <c:forTokens /> permite iterar sobre trozos de un string delimitados por algún carácter especificado. En siguiente ejemplo muestra el uso de esta etiqueta:

```
<c:forTokens items="uno-dos-tres-cuatro" delims="-" var="token">
  ${token}
</c:forTokens>
```

Al igual que la etiqueta c:forEach, c:forTokens también posee atributos begin, end y step.

Procesar URLs (Reescritura URL).

La acción <c:url /> permite normalizar una URL relativa del sitio web convirtiéndola en una URL absoluta respecto a la carpeta raíz, si empiezan con una barra inclinada. Esto es interesante para no tener problemas al realizar redirecciones externas.

Además, esta acción ofrece una funcionalidad más interesante aún. La url normalizada aplica reescritura URL para asegurar el mantenimiento de las sesiones, aunque están desactivadas las cookies en el navegador cliente.

Como ejemplo de aplicación modificaremos los enlaces "Anterior" y "Siguiente" del ejemplo previo:

```
<c:if test="${param.inicial > 0}">
  <c:url var="urlAnterior" value="">
    <c:param name="inicial" value="${param.inicial-lineas}" />
  </c:url>
  <a href="${urlAnterior}">Anterior</a>
</c:if>
<c:if test="${param.inicial + lineas < header.size()}">
  <c:url var="urlSiguiente" value="">
    <c:param name="inicial" value="${param.inicial+lineas}" />
  </c:url>
  <a href="${urlSiguiente}">Siguiente</a>
</c:if>
```

La acción <c:url> soporta un atributo var, usado para especificar la variable que contendrá la URL codificada, y un atributo value para contener la URL a codificar. Si no se especifica el atributo var, la URL codificada es renderizada en la salida.

Se pueden especificar parámetros string para solicitar la URL usando acciones <c:param>. Los caracteres especiales en los parámetros especificados por elementos anidados son codificados (si es necesario) y luego añadidos a la URL como parámetros string de la consulta. El resultado final se pasa a través del proceso de reescritura de URL, añadiendo un ID de sesión si está desactivado el seguimiento de sesión usando cookies.

Redirecciones externas.

La etiqueta <c:redirect> permite redirigir externamente a otro recurso en la misma aplicación Web, en otra aplicación Web o en un servidor diferente, aplicando reescritura URL sobre la url. Por ejemplo, podemos evaluar una variable de contexto para redirigir a una u otra página:

```
<if test="${condicion}">
  <c:redirect url="/pagina1.jsp" />
</c:if>
<c:redirect url="/pagina2.jsp" />
```

La acción c:redirect es equivalente a response.sendRedirect(encodeRedirectURL(url)).

Importar y ejecutar recursos.

La etiqueta <c:import> es una acción más flexible que la acción estándar <jsp:include>. Podemos usarla para incluir contenido desde recursos dentro de la misma aplicación Web, desde otras aplicaciones Web en el mismo contenedor, o desde otros servidores, usando protocolos como HTTP y FTP.

Por ejemplo, supongamos definido el siguiente servlet:

```
package servlets;
```

```
import java.io.*;
import java.text.DateFormat;
import java.util.Date;
import javax.servlet.ServletException;
import javax.servlet.http.*;
@WebServlet(name = "FechaActual", urlPatterns = {"/fechaactual"})
public class FechaActual extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/plain;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {
      out.println(DateFormat.getDateInstance().format(new Date()));
    }
  }
}
```

El servlet FechaActual simplemente responde escribiendo la fecha actual en el canal de salida. Este servlet tiene asociado como patrón URL el recurso "fechaactual".

Ahora, en un JSP, podemos utilizar la etiqueta <c:import /> sobre este servlet de dos maneras:

```
<c:import url="fechaactual" />                    <%-- se escribe directamente la fecha --%>
<br />
<c:import url=" fechaactual " var="fecha" /> <%-- se guarda la fecha en una variable --%>
${fecha}                                          <%-- y se muestra --%>
```

Como resultado de la ejecución de este JSP se genera una página HTML que muestra en dos líneas distintas la fecha actual.

Si queremos mostrar el contenido de un archivo de texto dentro de una página JSP podemos utilizar también <c:import />.

```
Contenido del archivo de texto: <br />
<c:import url="un_archivo.txt" />
```

La etiqueta <c:import /> también admite el paso de parámetro con la etiqueta anidada <c:param />. Por ejemplo:

```
<c:import url="fechaactual" />
  <c:param name="nombre del parámetro" value="valor del parámetro" />
</c:import>
```

Los parámetros son concatenados a la URL del recurso importado como parámetros de solicitud GET, y pueden ser recuperados desde un servlet con el método HttpRequest.getParameter().

9.4.5. Librería Formatting.

La librería fmt incluye un conjunto de acciones para simplificar la internacionalización, principalmente cuando páginas compartidas se usan para varios idiomas. La tabla siguiente describe someramente sus etiquetas:

bundle	Carga un grupo de recursos de idiomas para ser utilizado por las demás etiquetas que contenga en su cuerpo. El atributo basename debe contener el nombre común del grupo de recursos de idioma.
formatDate	Da formato a un valor de fecha de acuerdo a la cultura establecida. El formato de fecha puede ser establecido mediante los atributos opcionales dateStyle, pattern, y timeStyle.
formatNumber	Da formato a un número de acuerdo a la cultura establecida. El formato de número puede ser establecido mediante los atributos opcionales currencyCode, groupingUsed, currencySymbol, maxFractionDigits, maxIntegerDigits, minFractionDigits, minIntegerDigits, y pattern.
message	Recupera el valor asociado a una clave del archivo de recursos de idioma para el idioma establecido.
parseDate	Similar a formatDate para extraer el valor de fecha de un string.
parseNumber	Similar a formatNumber para extraer el valor numérico de un string.
requestEncoding	Establece la codificación de caracteres por defecto.
setBundle	Estable un grupo de archivos de recursos de idioma para ser utilizado por etiquetas posteriores.
setLocale	Estable el idioma y/o región por defecto.
setTimeZone	Establece la zona horaria.
timeZone	Determina la zona horaria dentro del cuerpo de la etiqueta.

Formateo de fechas y números sensible a la localidad.

Primero veamos cómo formatear apropiadamente fechas y números. Este ejemplo formatea la fecha actual y un número basándose en las reglas de la localidad por defecto:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <h1>Formateando con la localidad por defecto</h1>
    <jsp:useBean id="now" class="java.util.Date" />
    Fecha: <fmt:formatDate value="${now}" dateStyle="full" />
    Numero: <fmt:formatNumber value="${now.time}" />
  </body>
</html>
```

La primera línea es la directiva taglib para la librería JSTL que contiene las acciones de formateo e internacionalización. El prefijo por defecto, usado aquí, es fmt. Para obtener un valor a formatear, se crea un objeto java.util.Date que representa la fecha y hora actuales, y lo graba como una variable llamada now.

La acción JSTL <fmt:formatDate> formatea el valor de la variable now asignado al valor del atributo value usando el tipo de expresión EL. El atributo dateStyle le dice a la acción cómo debe formatear la fecha. Podemos usar cualquiera de los valores default, short, medium, long, o full para el atributo dateStyle. El tipo de formateo que representa cada uno de estos tipos depende exactamente de la localidad. Para la localidad English, full resulta en un string como "Thursday, August 29, 2002". En este ejemplo sólo hemos formateado la parte de la fecha, pero también podemos incluir la parte de la hora (o formatear sólo la hora), y definir reglas de formateo personalizadas tanto para la fecha como para la hora en lugar de los estilos dependientes de la localidad:

```
<fmt:formatDate value="${now}" type="both" pattern="EEEE, dd MMMM yyyy, HH:mm" />
```

El atributo pattern toma un patrón de formateo personalizado del mismo tipo que la clase java.text.SimpleDateFormat. El patrón usado aquí resulta en "Thursday, 29 August 2002, 17:29" con la localidad English.

Para escanear un string y obtener la fecha que contiene se usa la acción parseDate:

```
<fmt:parseDate value="${stringConFecha}" pattern="MM dd, YYYY" var="parsedDate" />
```

Esta acción extrae la fecha contenida en una variable stringConFecha según el patrón especificado y la asigna en una variable denominada parsedDate.
La acción <fmt:formatNumber> soporta atributos similares para especificar cómo formatear un número usando estilos dependientes de la localidad para números normales, valores de moneda, o un porcentaje, así como usar patrones personalizados de diferentes tipos. Por ejemplo:

```
<fmt:formatNumber value="1000.001" pattern="#,#00.0#"/>
```

Provoca que el número se muestre como 1,000.00 para la localidad English.
De forma análoga a la acción parseDate, si queremos escanear un número a partir de un string usaremos la acción parseNumber del siguiente modo:

```
<fmt:parseNumber value="${stringConNumero}" type="currency" var="parsedNumber"/>
```

Usar JSTL para seleccionar la localidad.
¿Cómo se determina la localidad para formatear las fechas y los números? Por defecto, JSTL se basa en una cabecera de solicitud llamada Accept-Language, que contendrá la configuración de localización preferida por el cliente.
Para sobrescribir la configuración de la localidad por defecto, podemos usar la acción <fmt:setLocale>, que selecciona la localidad por defeco dentro de un ámbito JSP específico. Aquí tenemos un ejemplo que selecciona la localidad por defecto del ámbito de la página, basándose en una cookie que sigue la pista de la localidad seleccionada por el usuario en una visita anterior:

```
<h1>Formateando con una localidad puesta por setLocale</h1>
<c:if test="${!empty cookie.preferredLocale}">
  <fmt:setLocale value="${cookie.preferredLocale.value}" />
</c:if>
<jsp:useBean id="now" class="java.util.Date" />
Fecha: <fmt:formatDate value="${now}" dateStyle="full" />
Número: <fmt:formatNumber value="${now.time}" />
```

Aquí, primero hemos usado la acción JSTL <c:if> para comprobar si con la solicitud se ha recibido una cookie llamada preferredLocale. Si es así, la acción <fmt:setLocale> sobrescribe la localidad para la página actual seleccionando la variable de configuración de la localidad en el ámbito de la página. Si lo queremos, podemos seleccionar la localidad para otro ámbito usando el atributo scope. Finalmente, se formatean la fecha y el número de acuerdo con las reglas de la localidad especificada por la cookie, o la localidad por defecto, si la cookie no está presente.
Generar texto localizado.
Aunque formatear fechas y número es importante cuando se localiza una aplicación, el contenido de texto es, por supuesto, incluso más importante. JSTL está basado en Java, por eso trata con el soporte genérico de i18n de la plataforma Java. Cuando viene con texto, este soporte está basado en lo que se llama un paquete de recursos. En su forma más simple, un paquete de recursos está representado por un fichero de texto (ubicado dentro de la carpeta del classpath) que contiene claves y un valor de texto para cada clave. Este ejemplo muestra un fichero con dos claves (hello y goodbye) y sus valores:

Archivo «labels.properties»
hello=Hola goodbye=Adiós

La acción JSTL que añade texto desde un paquete de recursos a una página es <fmt:message>. El fichero para utilizar puede especificarse de varias formas. Una es anidar las acciones <fmt:message> dentro del cuerpo de una acción <fmt:bundle>:

```
<fmt:bundle basename="labels">
  <fmt:message key="hello" /> y <fmt:message key="goodbye" />
</fmt:bundle>
```

En este caso, la acción <fmt:bundle> localiza el paquete de la localidad que es la correspondencia más cercana entre la selección de configuración de la localidad (o las localidades en el cabecera Accept-Language, si no hay localidad por defecto) y los paquetes de recursos disponibles para el nombre base especificado. Las acciones anidadas obtienen el texto desde el paquete por la clave asociada.

Al igual que con las acciones de formateo, podemos establecer un paquete por defecto, para toda la aplicación, con un parámetro de contexto o con la acción <fmt:setBundle> o la clase Config para un ámbito JSP específico:

```
<fmt:setBundle basename="labels"/>
  ..................
<fmt:message key="hello" /> y <fmt:message key="goodbye" />
```

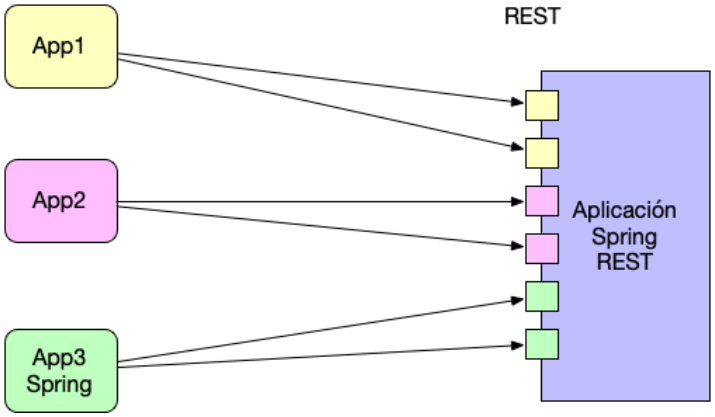
Después de que se haya definido un paquete por defecto, podemos usar acciones <fmt:message> independientes dentro del ámbito donde se estableció el valor por defecto.

10. Spring WebFlux.

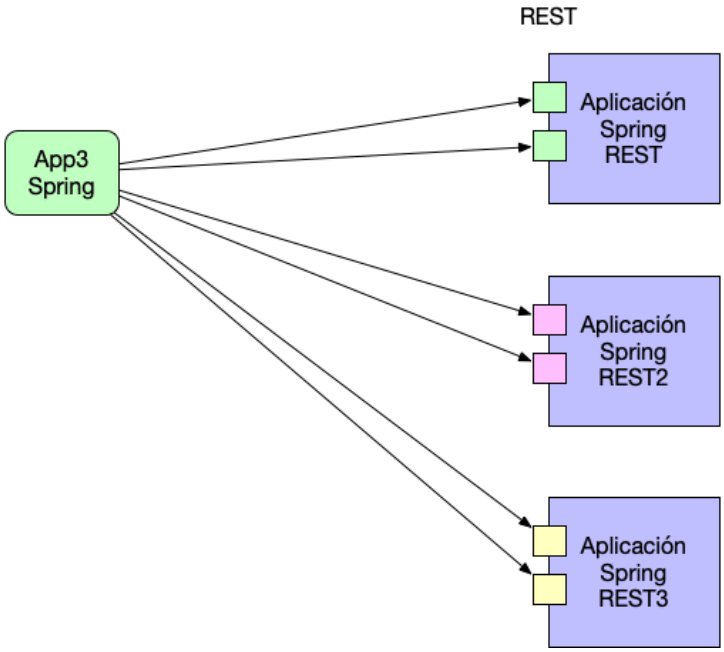
Spring WebFlux es el framework de Spring que permite manejar flujos de datos reactivos como respuesta desde un controlador.

10.1. Flujos reactivos y REST.

Hoy en día nos encontramos que una parte importante de los clientes de servicios REST pueden ser clientes que necesitan consultar una información puntual de nuestra aplicación. Es decir no necesitan acceder a todo el abanico de funcionalidad que publicamos sino que necesitan un pequeño subconjunto.

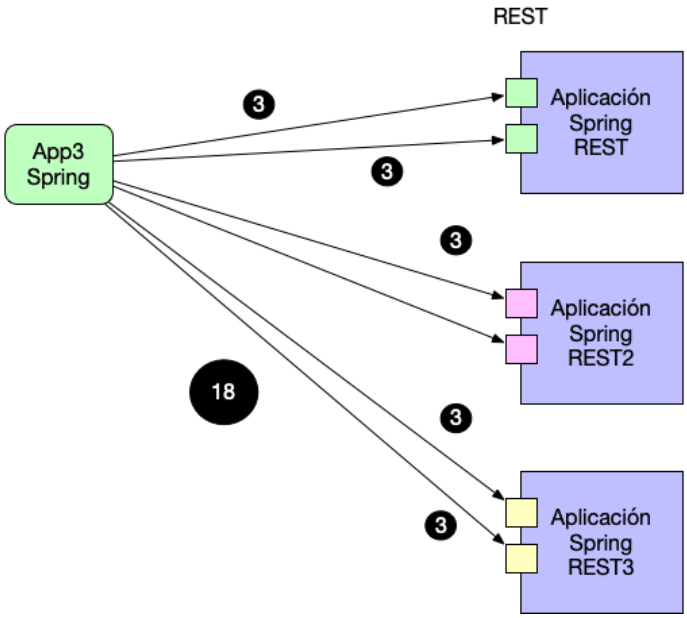


Con este tipo de arquitecturas es muy habitual tener la necesidad de que nuestra aplicación cliente de Spring necesite acceder a servicios REST de múltiples aplicaciones que en este caso hacen el rol de aplicaciones servidoras.



Es aquí donde hay un salto en cuanto a cómo podemos tratar este tipo de arquitectura. ¿Podemos usar Spring Boot de forma clásica? Está claro que sí y que podemos publicar servicios REST y podemos consumirlos mediante solicitudes HTTP. Ahora bien, existe un problema si cada petición tarda 3 segundos en hacerse a nivel de REST y tenemos que realizar las 6 peticiones que aparecen en el diagrama . Habrá que sumar los tiempos de cada una ya

que se trata de lo que habitualmente se conoce como peticiones bloqueantes, es decir, yo realizo una petición y espero a su resultado para poder realizar la siguiente:



Gracias a Spring WebFlux esto es algo que podemos resolver ya que proporciona un Framework no bloqueante. Podremos realizar peticiones de forma simultánea y en cuanto tengamos el resultado de todas dar por finalizada la tarea. Es decir, si cada petición tarda 3 segundos y hacemos todas a la vez, en 3 segundos tendremos todos los datos ya en nuestra aplicación y no en 18.

Para dar soporte a Spring WebFlux tendremos que añadir a nuestro proyecto de Spring Boot la siguiente dependencia:

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

10.2. Servicios REST Reactivos.

Vamos a construir un ejemplo que nos ayude a entender cómo funciona WebFlux. Crearemos una aplicación web con Spring Boot para devolver una lista de datos.

Para esto añadiremos un controlador REST.

```
@RestController
public class DatosController {
  @GetMapping("/datos")
  public List<String> findAll() {
    List<String> lista= List.of("dato1", "dato2", "dato3");
    try {
      Thread.sleep(3000);
    } catch (InterruptedException e) {
    }
    return lista;
  }
}
```

La solicitud "/datos" retornará una lista de tres datos pasados 3 segundos. Si usamos un navegador web para realizar esta consulta notaremos que los datos se reciben con este retraso de tiempo.

Para ser más precisos, realizaremos una consulta desde otra aplicación de Spring Boot usando el API HttpClient. Para esto crearemos un servicio que se encargue de realizar las consultas al REST:

```
@Service
public class DatosServiceClient {
  private final String URIBASE = "http://localhost:8080/";
  private HttpClient httpClient = HttpClient.newBuilder().build();
```

```
    private ObjectMapper jsonMapper = new ObjectMapper();
    private TypeReference<List<String>> typeListDatos = new TypeReference<List<String>>() { };

    private List<String> parseToList(String json) {
      try {
        return jsonMapper.readValue(json, typeListDatos);
      } catch (Exception e) {
        throw new RuntimeException(e);
      }
    }

    public List<String> queryAll() {
      var request = HttpRequest
        .newBuilder(URI.create(URIBASE+"datos"))
        .GET()
        .build();
      return httpClient.sendAsync(request, BodyHandlers.ofString())
        .thenApply(result -> result.body())
        .thenApply(result -> parseToList(result))
        .join();
    }
  }
```

Ahora invocaremos este servicio 3 vences y mediremos el tiempo de la respuesta global:

```
var init = System.currentTimeMillis();
var lista = new ArrayList<String>();
lista.addAll(servicio.queryAll());
lista.addAll(servicio.queryAll());
lista.addAll(servicio.queryAll());
var end = System.currentTimeMillis();
System.out.printf("Ha tardado %f segundos.\n", (end - init) / 1000.0);
```

El tiempo que tarda debería ser al menos de 9 segundos.

Es ahora cuando usaremos Spring WebFlux. El framework nos permitirá construir un servicio REST no bloqueante de tal forma que podamos hacer varias peticiones simultaneas y las procese en paralelo. Vamos a ver el código del nuevo Controlador REST.

```
@RestController
public class DatosController {
  @GetMapping("/datos")
  public Flux<String> findAll() {
    Flux<String> lista= Flux.just("dato1", "dato2", "dato3")
                            .delaySequence(Duration.ofSeconds(3));
    return lista;
  }
}
```

Como se puede observar, se hace uso de un objeto de tipo reactor.core.publisher.Flux. Este tipo representa una lista asíncrona de objetos. Esta clase ofrece varios métodos estáticos para crear flujos, algunos de ellos se listan a continación:

.empty()	Devuelve un flujo vacío.
.just(…)	Crea un flujo a partir de un array.
.range(start, count)	Crea un flujo de Integer entre un rango.
.from(publisher)	Crea un flujo con los datos que genera un Publisher.
.error(throwable)	Crea un flujo que provoca una excepción.
.concat(…)	Combina varios flujos en uno solo.
.fromArray()	Crea un flujo a partir de un array.
.fromIterable()	Crea un flujo a partir de un Iterable.
.fromStream()	Crea un flujo a partir de un Stream.

Además, con el método delaySequence(), obligamos a esperar 3 segundos antes de devolver la lista. Con esto abrimos las capacidades no bloqueantes de Spring.
Ahora, el servicio cliente debe hacer uso de la clase org.springframework.web.reactive.function.client.WebClient para solicitar flujos reactivos.

```
@Service
public class DatosServiceClient {
  private final String URIBASE = "http://localhost:8080/";

  public Flux<String> queryAllFlux() {
    var client =  WebClient.create(URIBASE+"datos");
    return client.get().retrieve().bodyToFlux(String.class);
  }
}
```

La solicitud se realiza a través de un objeto WebClient, el cual permite programación reactiva no bloqueante. Al igual que antes, realizaremos tres consultas y unificaremos los flujos en uno solo:

```
var init = System.currentTimeMillis();
var lista = Flux.merge( servicio.queryAllFlux(), servicio.queryAllFlux(), servicio.queryAllFlux())
            .toStream().toList()
var end = System.currentTimeMillis();
System.out.printf("Ha tardado %f segundos.\n", (end - init) / 1000.0);
```

El tiempo que tarda debería ser de poco más de 3 segundos. Es decir, aunque hagamos varias solicitudes, se tarda lo que tarde la más perezosa de ellas.

10.3. Objetos Flux y Mono.

Al API WebFlux ofrece dos tipos de datos para retornar flujos reactivos: Flux y Mono.
El tipo Mono se utiliza para retornar un solo elemento mientras que Flux se utiliza para retornar un conjunto de elementos.
Pero aún así, podemos usar el tipo Mono para retornar una lista de datos. El ejemplo del apartado anterior quedaría así usando Mono en vez de Flux.

```
@RestController
public class DatosController {
  @GetMapping("/datos")
  public Mono<List<String>> findAll() {
    Mono<List<String>> lista = Mono.just(List.of("dato1", "dato2", "dato3"))
                               .delayElement(Duration.ofSeconds(3));
    return lista;
  }
}

@Service
public class DatosServiceClient {
  public Mono<List<String>> queryAllMono() {
    var client =  WebClient.create(URIBASE+"datos");
    return client.get().retrieve().bodyToMono(new ParameterizedTypeReference<List<String>>() { });
 }
}
```

Y podemos comprobar que varias consultas al método quereyAllMono() tardarán no más de 3 segundos:

```
var init = System.currentTimeMillis();
var lista = Flux.merge(servicio.queryAllMono(), servicio.queryAllMono(), servicio.queryAllMono())
    .toStream()
    .flatMap(ventas-> ventas.stream())
    .toList();
var end = System.currentTimeMillis();
System.out.printf("Ha tardado %f segundos.\n", (end - init) / 1000.0);
```

Entonces, ¿cuál es realmente la diferencia entre Flux y Mono? Veámoslo con las siguientes instrucciones:

```
@GetMapping("/datosflux")
```

```
public Flux<String> findAllFlux() {
  return Flux.fromIterable(List.of("dato1", "dato2", "dato3")).delayElements(Duration.ofSeconds(3));
}
@GetMapping("/datosmono")
public Mono<List<Venta>> findAllMono() {
  return Mono.just(List.of("dato1", "dato2", "dato3").delayElement(Duration.ofSeconds(3));
}
```

Al crear un Flux se retarda la emisión de cada dato cada 3 segundos, y al crear un Mono se retarda la emisión de la lista entera 3 segundos. Por tanto, un Flux permite controlar la emisión de cada uno de sus elementos mientras que un Mono emite su contenido como un único elemento. El cliente que reciba los datos podrá controlar la emisión del Flux para capturar cada dato individualmente:

```
var client =  WebClient.create(URIBASE+"ventasflux");
client.get().retrieve()
    .bodyToFlux(Venta.class)
    .subscribe(System.out::println);
```

10.4. Repositorios reactivos.

JPA no da soporte para repositorios reactivos, pero Mongo sí lo hace.

10.4.1. Repositorios reactivos para Mongo.

Para crear repositorios reactivos con Mongo necesitamos la siguiente dependencia:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>
```

Y ahora simplemente debemos crear una clase de Documento y un repositorio:

```
@Data // Lombok
@Document
public class Cuenta {
  @Id
  private String id;
  private String usuario;
  private Double cantidad;
}
```

Para crear el repositorio seguiremos el modelo de programación de repositorios de Spring Boot, creando una interfaz que en este caso debe heredar de alguna de las siguientes interfaces:

- ReactiveCrudRepository, es similar a CrudRepository.
- ReactiveMongoRepository, hereda de ReactiveCrudRepository y agrega nuevos métodos.

Por tanto, podemos crear el siguiente repositorio al cual añadimos dos métodos personalizados:

```
@Repository
public interface CuentaCrudRepository extends ReactiveCrudRepository<Cuenta, String> {
  Flux<Cuenta> findAllByCantidad(double cantidad);
  Mono<Cuenta> findFirstByUsuario(Mono<String> usuario);
}
```

Podemos pasar diferentes tipos de argumentos como simples (String), envueltos (Optional, Stream) o reactivos (Mono, Flux). Y dispondremos de métodos que retornan objeto Flux y Mono:

```
Mono<Cuenta> cuenta = repositorioi.save(new Cuenta(null, "Juan", 100.0));
Mono<Cuenta> mismaCuenta = repositorio.findById(cuenta.getId());
```

11. Seguridad en Spring.

Spring proporciona el módulo Spring Security para dar soporte a todos los aspectos de seguridad de las aplicaciones, incluyendo un marco de registro de usuarios, mecanismos de autenticación y autorización, así como compatibilidad con OAuth.

11.1. ¿Qué es Spring Security?

Spring Security es una librería que forma parte del Framework Spring, que trata de agrupar todas las funcionalidades de control de acceso de usuarios sobre proyectos Spring.

El control de acceso permite limitar las opciones que pueden ejecutar un determinado conjunto de usuarios o roles sobre la aplicación. De esta forma, Spring Security controla las invocaciones a la lógica de negocios o limita el acceso de peticiones HTTP a determinadas URLs.
Para ello, el programador debe realizar una configuración sobre la aplicación indicando a Spring Security cómo debe comportarse la capa de seguridad. Y ésta es una de las grandes ventajas de Spring Security, ya que permite realizar toda una serie de parametrizaciones y ajustes para un gran abanico de posibilidades, permitiendo que el módulo se adapte bien a casi cualquier escenario de aplicaciones realizadas con Spring Boot.

11.1.1. ¿Cómo incluir Spring Security en una aplicación web?

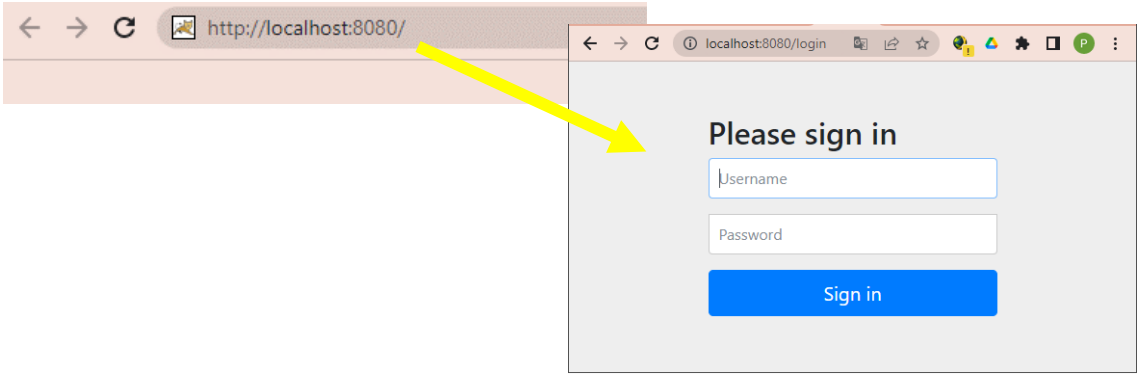
Añadir Spring Security en cualquier proyecto de Spring Boot es muy sencillo. Al menos debemos añadir la dependencia siguiente:

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

Con esto, de manera predeterminada Spring Boot protegerá todo el acceso a la aplicación, impidiendo que ningún usuario no identificado pueda invocar a cualquier controlador. Este mecanismo tan restrictivo suele ser suficiente para pequeñas aplicaciones donde sólo se necesita restringir el acceso de forma general, pero queda algo reducido en aplicaciones donde hay secciones accesibles y otras protegidas, dependiendo de si el usuario está identificado o de si éste tiene un rol determinado.
Como el resto de los aspectos configurables de Spring Boot, en el fichero de configuración de la aplicación, «application.properties» o «application.yml», hay varias propiedades que pueden ajustarse para controlar el comportamiento base de Spring Security.

```
spring.ldap.\* = ...                  # propiedades correspondientes a integración con LDAP
spring.security.oauth2.\* = ...       # parámetros para OAuth2 y JWT
spring.session.\* = ...               # configuraciones para la sesión HTTP, con persistencia SQL opcional
spring.security.user.name = user      # usuario por defecto
spring.security.user.password =       # contraseña por defecto
spring.security.user.roles =          # roles por defecto
```

De esta manera, sólo por incluir la dependencia, en el fichero de propiedades de la aplicación puede indicarse un usuario y clave por defecto que podrá usarse para tener acceso a los controladores, y como consecuencia a las diferentes vistas HTML de la aplicación. Cuando el usuario intente acceder a cualquier URL de la aplicación, Spring Boot y el Security Filter de HTTP redirigirá al usuario al formulario de identificación, donde le solicitará que inserte el nombre y contraseña para acceder.



En ese formulario se debe indicar los valores fijados en spring.security.user.name y spring.security.user.password. Si el usuario ha introducido los valores correctos, se considera al usuario autenticado, y sin tener en cuenta el rol, dejará continuar al usuario con una navegación normal.
Con la configuración por defecto no se incluye el auto registro de usuarios. En el caso de necesitar que los usuarios puedan registrarse automáticamente, por ejemplo con un correo, se deberá programar el mecanismo por el cual se confía en la información proporcionada por el usuario procediendo a la creación de éste en la tabla de datos o fuente de usuarios confiables usada por la aplicación.

11.2. Autentificación y autorización.

11.2.1. ¿Cómo proteger secciones de la aplicación?

Si la protección de forma global a toda la aplicación no es suficiente, podremos configurar secciones para que sean accesibles mientras que otras estén protegidas para un determinado número de usuarios. A este proceso se le denomina "autorización". Para determinar los usuarios sobre los que se concederá permiso de acceso se aplica un mecanismo denominado "autentificación" (o "autenticación").

Para configurar la autentificación y autorización debemos incluir una clase de configuración que declares beans que retornen un UserDetailsService para autenticación y un SecurityFilterChain para autorización.

A continuación se muestra un ejemplo:

```
@Configuration
@EnableWebSecurity
public class CustomSecurityConfig {
  // Un encriptador para contraseñas
  @Bean
  public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
  }

  // Un manejador de autentificación de usuarios
  @Bean
  @Autowired
  public UserDetailsService userDetailsService(PasswordEncoder passwordEncoder) {
    // Se crean dos usuarios de ejemplo
    UserDetails user1 = User.withUsername("abc")
      .password("abc")
      .authorities("USER")
      .passwordEncoder(pw -> passwordEncoder.encode(pw))
      .build();
    UserDetails user2 = User.withUsername("def")
      .password("def")
      .authorities("OTHER")
      .passwordEncoder(pw -> passwordEncoder.encode(pw))
      .build();

    // Se devuelve un manejador de registro en memoria
    return new InMemoryUserDetailsManager(user1, user2);
  }

  // Un manejador de autorización de usuarios
  @Bean
  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
      .authorizeHttpRequests()
      .requestMatchers("/").permitAll()
      .anyRequest().authenticated()
      .and()
      .formLogin().permitAll()
      ;
    return http.build();
  }
}
```

Para encriptar las contraseñas primero se define un bean que retorna un PasswordEncoder:

```
return PasswordEncoderFactories.createDelegatingPasswordEncoder();
```

El método createDelegatingPasswordEncoder() proporciona un encriptador que soporta los algoritmos de encriptación habituales.

El bean UserDetailsService configura un almacén de usuarios en memoria con dos usuario. Spring Security utiliza objetos de la clase org.springframework.security.core.userdetails.User para almacenar los datos de un usuario registrado.
El bean SecurityFilterChain define qué rutas deben protegerse y cuáles no. Para el ejemplo previo, las rutas "/" y "/home" están configuradas para no requerir ninguna autenticación. Todas las demás rutas deben ser autenticadas. El método .fromLogin() habilita la página de registro de usuario por defecto, y se permite su acceso a todos.

11.2.2. Proveedores de autentificación.

Existen dos clases predefinidas que implementan UserDetailsService:

- InMemoryUserDetailsManager, implementa una base de datos en memoria de usuarios y roles.
- JdbcUserDetailsManager, se conecta a una base de datos para manejar usuarios y grupos.

Se utiliza la clase InMemoryUserDetailsManager para realizar pruebas durante la fase de desarrollo. Se puede instanciar un InMemoryUserDetailsManager vacío, o podemos pasar en su constructor un conjunto de objeto UserDetails con la información de un usuario.
La clase JdbcUserDetailsManager permite trabajar con una base de datos física. Para eso debemos proporcionarle un DataSource:

```
@Bean
@Autowired
public UserDetailsService userDetailsService(PasswordEncoder passwordEncoder, DataSource dataSource) {
  UserDetails user1 = User.withUsername("abc") …

  var manager = new JdbcUserDetailsManager(dataSource);
  manager.createUser(user1);
  manager.createUser(user2);
  return manager;
}
```

Pero la base de datos debe disponer al menos de las siguientes tablas USERS y AUTHORITIES:

```
CREATE TABLE USERS (
  USERNAME VARCHAR(50) PRIMARY KEY,
  PASSWORD VARCHAR(150) NOT NULL,
  ENABLED TINYINT NOT NULL
);

CREATE TABLE AUTHORITIES (
  USERNAME VARCHAR(50) NOT NULL REFERENCES USERS(USERNAME),
  AUTHORITY VARCHAR(30) NOT NULL
);
```

También se puede crear un manejador personalizado implementando directamente la interfaz UserDetailsService:

```
public class CustomDetailsService implements UserDetailsService {
  @Autowired
  private CredentialsRepository repo;       // Un repositorio que gestiona credenciales de usuarios

  @Override
  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    var user = repo.findByUsername(username);
    return User.builder().username(user.get)…..build();
  }
}
```

11.2.3. Seguridad CSRF.

Los ataques de falsificación de solicitudes entre sitios (CSRF) son habituales. CSRF es un ataque que obliga a un usuario final a ejecutar acciones no deseadas en una aplicación web en la que está autenticado actualmente.
Spring soporta un mecanismo que previene este tipo de ataques con las páginas Thymeleaf.
Para ello debemos agregar las siguiente dependencia:

```
<dependency>
  <groupId>org.thymeleaf</groupId>
  <artifactId>thymeleaf-spring6</artifactId>
```

```
</dependency>
```

Al aplicar Spring Security, se habilita por defecto la protección CSRF. Podemos deshabilitar esta característica útil, con la siguiente opción:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  http
    .csrf().disable()
    .....
    ;
  return http.build();
}
```

Básicamente el mecanismo CSRF que aplica Spring Security consiste en que cada vez que se solicita una vista, genera un token de solicitud que guarda en la memoria de la sesión actual. Cuando la vista postea un formulario debe retornar el mismo valor de token. Si el token falta o no coincide con el almacenado en la memoria de la sesión la aplicación fallará.

La dependencia thymeleaf-spring6 agrega automáticamente este token en los formularios como campos ocultos como el siguiente:

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

Si implementamos un formulario de autentificación personalizado tendremos que agregar este campo a mano, tal como veremos a continuación.

11.2.4. Inicio (login) y fin (logout) de sesión de registro de usuario.

Una vez que hemos configurado el mecanismo de autentificación de usuario, veremos cómo personalizar el inicio y fin de registro de un usuario.

Hemos visto que la opción .formLogin() establece un formulario por defecto que solicita las credenciales del usuario que debe autenticarse. Pero disponemos de otras opciones.

Autentificación básica.

Con la opción .httpBasic() en vez de .formLogin() se delega en el propio navegador la solicitud de las credenciales, de forma que se mostrará una ventana emergente típica del sistema operativo subyacente.

Autentificación de formulario personalizado.

Podemos personalizar el formulario de login usando una página de Thymeleaf. La siguiente página «customlogin.html» es un ejemplo sencillo:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="utf-8">
  <title>REGISTRO DE USUARIOS</title>
</head>
<body>
  <form action="/login" method="post">
    <div>
      <label>Usuario</label>
      <input type="text" name="username" />
    </div>
    <div>
      <label>Constraseña</label>
      <input type="password" name="password" />
    </div>
    <div>
      <input type="submit" value="Registrar" />
    </div>
  </form>
</body>
</html>
```

En negritas se han resaltado opciones que deben estar presentes: la acción del formulario debe ser "/login" para encadenar el mecanismo de autentificación, y debe haber dos campos de edición con los atributos name a valor "username" y "password".
Ahora configuraremos el SecurityFilterChain para que utilice nuestra página:

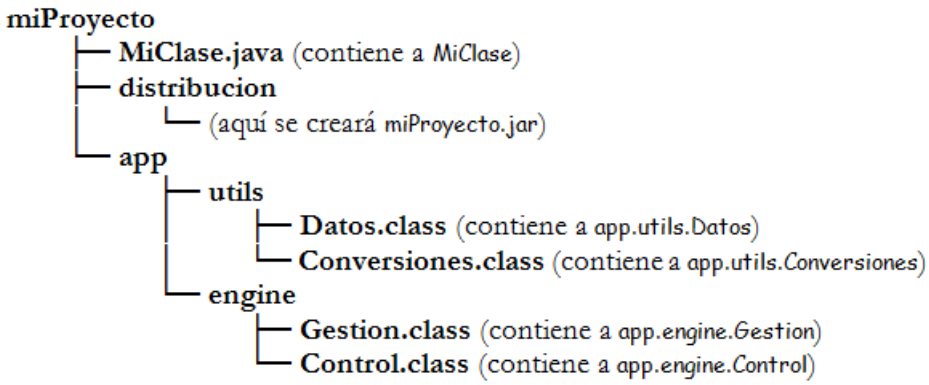
```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  http
    .csrf().disable()
    ...
    .and()
    .formLogin().permitAll()    // configura autentificación basada en formulario
    .loginPage("/customlogin.html")
    .and()
    .logout()       // permite cancelar el usuario autentificado con la uri "/logout"
    .logoutSuccessUrl("/") // redirige a la URI tras el logout
    ;
  return http.build();
}
```

12. Maven, modularidad y librerías.

Una vez que hemos construido y probado una aplicación, es posible que deseemos empaquetarla, de modo que sea más fácil de distribuir e instalar. Uno de los mecanismos que proporciona Java para estos fines son los archivo JAR. JAR significa *Java Archive*, y se utiliza para comprimir los datos (similar a los archivos en formato ZIP) y para almacenarlos. Los archivos JAR pueden ser utilizados como librerías reutilizables para otros proyectos.

12.1. Ficheros JAR.

Supongamos que una aplicación utiliza diferentes clases que se encuentran en varios paquetes. Aquí está el árbol de directorios.



Podemos crear un único archivo JAR que contenga todos los archivos class que están en app, y también que el directorio app mantenga su estructura. Cada uno de estos JAR pueden ser trasladados de lugar a lugar, y de máquina a máquina, y todas las clases dentro del JAR pueden ser accedidas a través del mecanismo *"class path"*, y usadas por java y javac. Todo esto ocurre sin necesidad de estar descomprimiendo el JAR.
El JDK proporciona la herramienta «jar» para comprimir los ficheros de un proyecto. Entornos de desarrollo como IntelliJ y VS Code disponen de opciones para empaquetar un proyecto compilado en un fichero JAR. Desde la línea de comandos podemos empaquetar el proyecto compilado de la siguiente forma:

```
(miProyecto) > jar -cf distribucion/MiProyect.jar app
```

Podemos ver lo que contiene el archivo JAR con el siguiente comando:

```
(miProyecto) > jar -tf distribucion/MiProyecto.jar
```

Mostrará algo como lo siguiente:

```
META-INF/
META-INF/MANIFEST.MF
app/
app/.DS_Store
```

```
app/utils/
app/utils/Datos.class
app/utils/Conversiones.class
app/engine/
app/engine/Gestion.class
app/engine/Control.class
```

12.1.1. El fichero de manifiesto.

El manifiesto es un fichero especial, con el nombre "MANIFEST.MF" dentro de la subcarpeta "META-INF ", que puede contener información sobre los ficheros empaquetados dentro del JAR.

El fichero de manifiesto que se crea por defecto es bastante simple y su contenido puede ser como el siguiente:

```
Manifest-Version: 1.0
Main-Class: main.Programa
```

Se trata de un típico fichero de configuración donde, en cada línea, se especifica una cabecera seguida de su valor. Con la cabecera Manifest-Version se especifica una versión para el fichero, con la cabecera Main-Class se especifica cuál es la clase principal del proyecto, aquella que deseamos ejecutar.

Podemos modificar este fichero para añadir valores adicionales. A continuación se resumen las cabeceras que podemos utilizar:

Cabecera	Significado
Main-Class	Establece la clase principal que será el punto de entrada de la aplicación.
Manifest-Version	La versión de librería.
Class-Path	Indica ubicaciones de búsqueda de clases.
Name	Nombre de un paquete para su sellado.
Sealed	Especifica el sellado de paquetes.

Descarga de extensiones.

Con la cabecera Class-Path podemos establecer rutas de búsqueda de clases, incluyendo en esas rutas los ficheros JAR, como si fuesen carpetas virtuales.

Una cabecera Class-Path, por ejemplo, se podría parecer a esto:

```
Class-Path: . servlet.jar infotest.jar utiles/beans.jar
```

Se indican rutas, separadas por espacios, de carpetas o de ficheros jar. Estas rutas pueden ser relativas a la ubicación del JAR actual o pueden ser rutas absolutas de disco. En este ejemplo, las clases ubicadas en la carpeta actual (.), y las clases de los ficheros servlet.jar, infotest.jar, y utiles/beans.jar servirán como extensiones para los propósitos de nuestra aplicación.

Sellado de paquetes.

Podemos ocultar ciertos paquetes almacenados en ficheros JAR de forma que no sean accesibles. El sellado de un paquete dentro de un fichero JAR significa que todas las clases definidas en ese paquete deben encontrarse dentro del mismo fichero JAR.

Se puede sellar un paquete añadiendo la cabecera Sealed. Por ejemplo, si queremos sellar el paquete com.utils:

```
Name: com/utils/
Sealed: true

Name: com/infos/
Sealed: true
```

Como se ve, el nombre de paquete se indica como una ruta terminada con /, esto distinguirá al paquete del nombre de una clase. Una línea en blanco le dirá a la cabecera Sealed qué paquetes declarados previamente deben quedar afectados.

Si queremos sellar todo el fichero JAR basta con poner la cabecera Sealed: true sin nombre de paquete.

Versionado de paquetes.

Podemos anotar un paquete con varias cabeceras que contendrán información de versionado. Las cabeceras de versionado deberían aparecer directamente debajo de la cabecera Name del paquete. En el siguiente ejemplo se muestran las cabeceras de versionado:

```
Name: java/util/
Specification-Title: "Java Utility Classes"
```

```
Specification-Version: "1.2"
Specification-Vendor: "Sun Microsystems Inc.".
Implementation-Title: "java.util"
Implementation-Version: "build57"
Implementation-Vendor: "Sun Microsystems. Inc."
```

12.1.2. Archivos JAR multiversión.

Desde Java 9 se puede crear un único archivo JAR que contenga algunas clases para una o varias versiones de Java. Por ejemplo, en un archivo JAR con las clases A, B, C y D compatibles con Java 8, el desarrollador puede ahora incluir versiones optimizadas de las clases A y B para la versión 9.

Esto se consigue con una estructura específica de directorios para cada versión de Java dentro de la carpeta META-INF. Veámoslo con un ejemplo de estructura de un fichero JAR:

```
appejemplo.jar
├── A.class
├── B.class
├── C.class
├── D.class
├── /META-INF
│       ├── MANIFEST.MF
│       └── /versions
│                   ├── 9
│                   │    ├── A.class
│                   │    └── B.class
│                   └── 10
│                        ├── A.class
│                        └── B.class
└── module-info.class
```

Este fichero puede incluir clases A, B, C y D compiladas para la versión 8, y versiones optimizadas de A y B para las versiones 9 y 10.

12.1.3. Empaquetado en proyectos Maven.

En los proyectos generados con Maven, podemos usar el fichero «pom.xml» para establecer las opciones de empaquetado a un fichero JAR.

Supongamos un proyecto creado con un arquetipo de Maven, denominado "proyecto1". Editaremos su fichero pom.xm para establecer el modo de empaquetado:

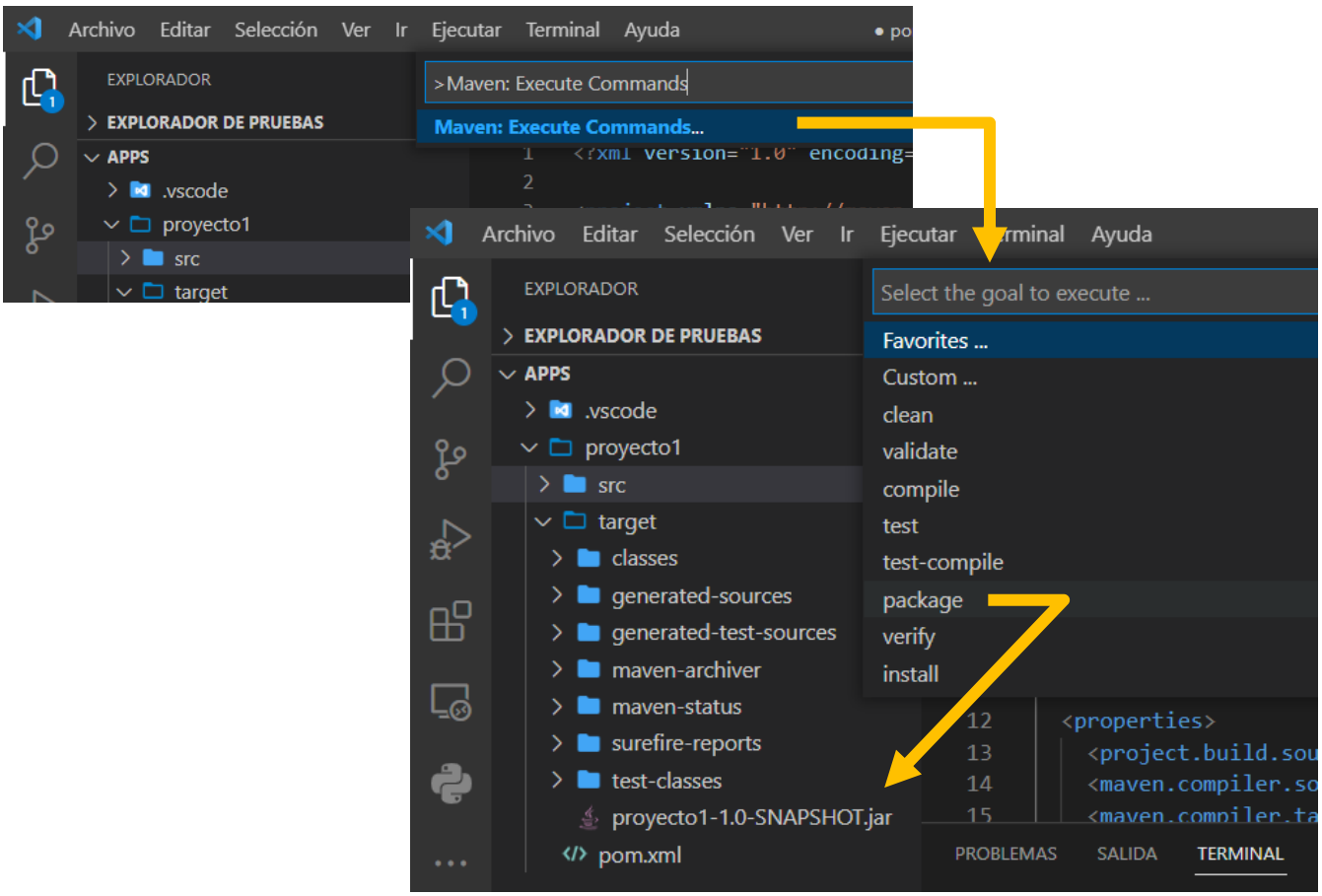
```
<?xml version="1.0" encoding="UTF-8"?>

<project …….>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.cloudftic.example</groupId>
  <artifactId>proyecto1</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>proyecto1</name>
  <packaging>jar</packaging>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-jar-plugin</artifactId>
        </plugin>
        ……
      </plugins>
    </pluginManagement>
  </build>
```

El nodo <packaging /> permite establecer alguno de los siguientes valores:

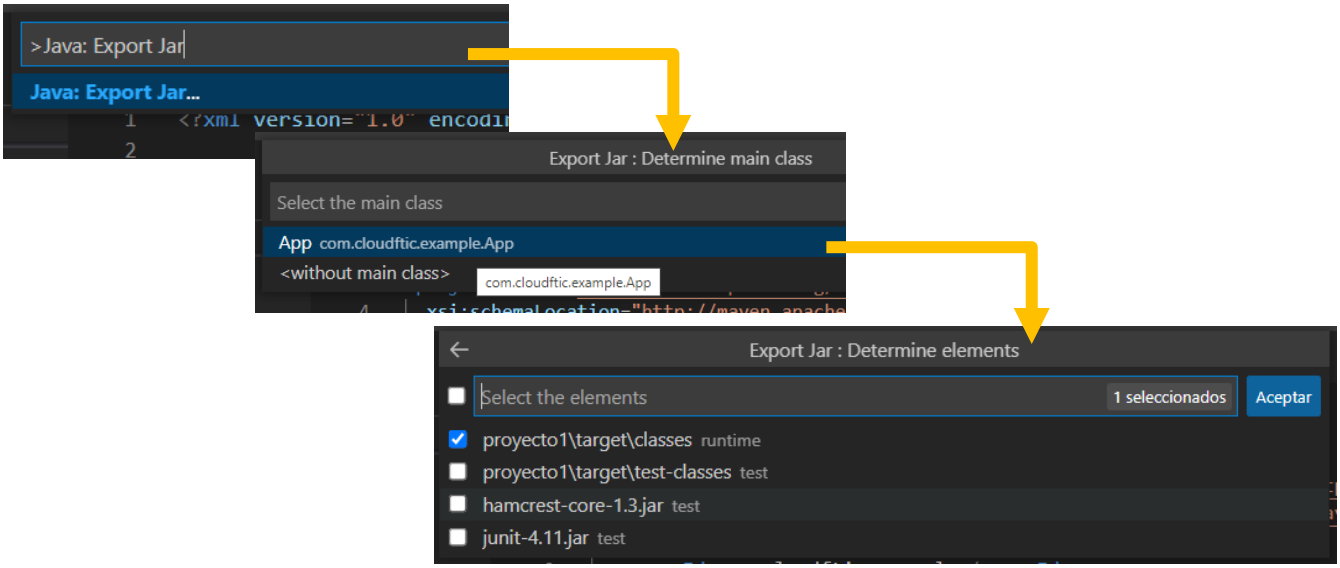
- jar: Es el valor por defecto, y provoca el empaquetado de la aplicación en un fichero con extensión ".jar".

• war: Provoca el empaquetado de la aplicación en un fichero con extensión ".war". Este formato se corresponde con aplicaciones web basadas en servlets y jsps.
• ear: Provoca el empaquetado de la aplicación en un fichero con extensión ".ear". Este formato se corresponde con aplicaciones de componentes JavaEE. Un fichero EAR puede contener a su vez fichero JAR y WAR.
• pom: Provoca el empaquetado de la aplicación en un fichero con extensión ".pom". Este formato ayuda a crear agregadores y proyectos principales. Un proyecto agregador o multimódulo ensambla submódulos provenientes de diferentes fuentes. Un proyecto principal permite definir una relación de herencia entre ficheros POM.
• ejb: Los EJB o Enterprise Java Beans son aplicaciones del lado del servidor distribuidas y escalables.
• rar: Un RAR o Adaptador de Recursos, es un archivo de almacenamiento que sirve como formato válido para la implementación de adaptadores de recursos en un servidor de aplicaciones. Básicamente, es un controlador de nivel de sistema que conecta una aplicación Java a un sistema de información empresarial (EIS)..

El comando «mvn package» permite generar el empaquetado. En VS Code podemos hacer esto usando la paleta de comandos:



Pulsa las teclas [Ctrl[[Máy][P], o pincha en el menú «Ver | Paleta de comandos», para abrir la paleta de comandos. Busca la opción «Maven: Execute Commands» y selecciona el comando, en este caso «package».
Maven compilará el proyecto y en la subcarpeta «target» creará el fichero «proyecto1-1.0-SNAPSHOT.jar». Por defecto, el nombre del fichero JAR se compone del identificador de artefacto y de la versión especificados en el fichero pom.xml.
En VS Code también podemos realizar una mayor personalización del empaquetado usando el comando «Java: Export Jar». Tras pulsar la tecla ENTER debemos seleccionar la clase principal. Tras pulsar la tecla ENTER debemos seleccionar los elementos que queremos incluir dentro del JAR. Tras pulsar el botón [Aceptar] se creará el fichero JAR.



12.1.4. Opciones avanzadas para empaquetado en Maven.

En el fichero pom.xml podemos especificar ciertas opciones que modifican el empaquetado. Esto se realiza con el plugin marven-jar-plugin, que debemos incluir dentro del nodo <plugins>, tal como se muestra a continuación:

```
<?xml version="1.0" encoding="UTF-8"?>

<project .....>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.cloudftic.example</groupId>
  <artifactId>proyecto1</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>proyecto1</name>
  <packaging>jar</packaging>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.3.0</version>
        <configuration>
          <archive>
            .........
            <manifest>
              .........
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Cambiar el nombre el fichero JAR.

Podemos personalizar el nombre final del proyecto (y por tanto del fichero JAR generado) usando el nodo <finalName>:

```
<build>
  <finalName>app1-1.0</finalName>
  ....
</build>
```

Con esta configuración se creará el fichero «app1-1.0.jar».

Configuración del fichero de manifiesto.

Dentro del nodo <archive><manifest></manifest><archive> podemos personalizar varias opciones del fichero de manifiesto:

- La clase principal, usando el subnodo <mainClass />.
- Las rutas del *class path*, usando los subnodos <addClasspath /> y <manifestEntries />.

Por ejemplo, la siguiente configuración:

```
<archive>
  <manifest>
    <mainClass>com.cloudftic.example.App</mainClass>
    <addClasspath>true</addClasspath>
  </manifest>
  <manifestEntries>
    <Class-Path>lib/ utils.jar</Class-Path>
  </manifestEntries>
</archive>
```

Genera el siguiente fichero de manifiesto:

```
Manifest-Version: 1.0
Created-By: Maven JAR Plugin 3.3.0
Build-Jdk-Spec: 17
Main-Class: com.cloudftic.example.App
Class-Path: lib/ utils.jar
```

12.1.5. Uso de librerías JAR.

Los fichero ".jar" permiten empaquetar proyecto de Java que pueden contener clases u otros tipos de datos que podemos reutilizar en otros proyecto. Un fichero JAR puede ser usado como librería o bien como un ejecutable.

Ficheros JAR como dependencias.

Para que un proyecto pueda usar el contenido del fichero JAR como librería debemos indicarle al intérprete de Java dónde localizarlos. Esto se realiza mediante un mecanismo denominado "class path".

Java proporciona varias formas de indicar dónde debe buscar referencias externas, incluyendo ficheros ".class" y ".jar". Imaginemos que hemos creado un fichero «utiles-1.0.jar» que contiene una clase «com.cloudftic.Utilidades», cuyos métodos deseamos utilizar en otro proyecto llamado «app1». La disposición de carpetas y ficheros será la siguiente:

```
(carpeta de trabajo)
   ├── utiles-1.0.jar
   ├── /app1
   │       └── /com
   │              └── /cloudftic
   │                     └── App.java
   └── /target
```

```
package com.cloudftic.example;
import com.cloudftic.Utiles;
public class App
{
    public static void main( String[] args ) {
        Utiles u = new Utiles();
    }
}
```

Para compilar la clase com.cloudftic.App debemos especificarle al compilador dónde encontrar la clase Utiles (dentro del fichero utiles.jar):

```
(carpeta de trabajo)> javac -d ./target --class-path ./utiles-1.0.jar ./app1/com/cloudftic/*.java
```

Con la opción -d especificamos la carpeta donde deben generase los fichero ".class". Con la opción --class-path se especifican rutas (o ficheros JAR) donde encontrar los paquetes de las clases referenciadas, donde varias rutas se separan con ; (en sistemas Windows) o dos : (en sistemas Unix).

Tras la ejecución de este comando, en la carpeta «target» debe crearse el fichero «com/cloudftic/App.class» con sus correspondientes subcarpetas.

Para ejecutar esta clase debemos indicarle al intérprete de Java dónde encontrar todas las clases referenciadas:

```
(carpeta de trabajo)> java -classpath ./utiles-1.0.jar;./app1 com.cloudftic.example.App
```

Si el proyecto «app1» va a ser empaquetado en un JAR, podemos especificar en el fichero de manifiesto la ubicación de las clases referenciadas:

MANIFEST.MF

```
Manifest-Version: 1.0
Main-Class: main.Programa
Class-Path: ./utiles-1.0.jar
```

De modo más general, podemos especificar rutas de búsquedas para el *"class path"* en una variable de entorno CLASSPATH del sistema operativo:

- En sistemas basados en Windows:- Como variable de entorno del S.O. (tanto en Windows como Unix)
  CLASSPATH = %CLASSPATH% ; ruta1 ; ruta2 ...
- En sistemas basados en Unix)
  CLASSPATH = $CLASSPATH : ruta1 : ruta2 ...

Ficheros JAR como ejecutables.

Si el fichero JAR contiene una clase principal (con el correspondiente método main estático) y está indicado en el fichero de manifiesto MANIFEST.MF, entonces podemos usar el fichero como un ejecutable a través del intérprete «java». A continuación se muestra un ejemplo:

```
(carpeta de trabajo) > java -jar app1.jar
```

La opción -jar le dice al intérprete que lo que viene a continuación es la ruta de un fichero JAR.

12.2. Maven y gestión de dependencias.

Maven es especialmente útil a la hora de referenciar las librerías JAR que utiliza el proyecto, y nos permite especificar cuáles de ellas queremos distribuir con el propio proyecto.

12.2.1. Configuración de dependencias con Maven.

Maven hace uso de dos repositorios a la hora de localizar librerías:

1) Un repositorio centras en la nube.
   Maven Central es una carpeta en la nube (accesible mediante conexiones de internet) que dispone de una gran cantidad de librerías, plugins y otros artefactos requeridos por nuestros proyectos.
2) Un repositorio local.
   Por defecto, se corresponde con la subcarpeta «.m2/repository» ubicada dentro de la carpeta del usuario actual del sistema. Podemos obtener la ruta de la carpeta del usuario actual con System.getProperty("user.home").

Podemos pedirle a Maven que busque y descargue librerías desde Maven Central o el repositorio local simplemente indicando su información de artefacto en la sección <dependencies />.

En el siguiente ejemplo se define la dependencia para usar Junit 4:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Cuando se compila o actualiza el proyecto, Maven busca la dependencia en el repositorio local y si no la encuentra la descarga desde Maven Central. Dentro de esta carpeta local las dependencias se organizan en subcarpetas correspondientes a los componentes del identificador de grupo, del identificar de artefacto y la versión. Por tanto, encontraremos la librería «junit-4.12.jar» dentro de «.m2/repository/junit/junit/4.12/».

Cada dependencia permite unos de los siguientes ámbitos (scope):

- **compile**: es el ámbito por defecto. Indica que la librería se usa en tiempo de compilación.
  Este ámbito indica que la librería estará disponible para el *class path* del proyecto durante la compilación, pero no durante la ejecución.
- **provided**: es similar a compile, pero la librería estará también disponible en tiempo de ejecución.
  Con este ámbito, el usuario debe garantizar que la librería estará disponible en el *class path* en tiempo de ejecución. Normalmente ocurre así si forma parte de un framework como JEE, Spring, etc.
- **runtime**: indica que la dependencia no se necesita para la compilación, pero sí para la ejecución.
  Maven suele incluir este tipo de dependencias para el intérprete y pruebas unitarias.
- **test**: indica una dependencia para pruebas unitarias.
- **system**: es similar a provided, pero la librería no estará en el repositorio local y el usuario debe proporcionar la ruta explícitamente en la etiqueta <systemPath>.

```
<dependency>
  . . .
  <scope>system</scope>
  <systemPath>${user.home}/jar/libreria.jar</systemPath>
</dependency>
```

• **import**: se utiliza para dependencias de la sección <dependencyManagement>.

Veamos ahora como incluir la librería «utiles-1.0.jar» para el proyecto «app1». Supongamos que hemos configurados el pom.xml de utiles con las siguientes opciones, y en el repositorio local ubicamos «utiles.jar» en la siguiente ubicación:

```
.m2
  └─ /repository
       └─ /com
            └─ /cloudftic
                 └─ /utiles
                      └─ /1.0
                           └─ utiles.jar
```

```
<project ….. >
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.cloudftic</groupId>
  <artifactId>utiles</artifactId>
  <version>1.0</version>

  <name>utiles</name>
```

Ahora añadiremos la dependencia en el pom.xml del proyecto «app1».

```
<dependency>
  <groupId>com.cloudftic</groupId>
  <artifactId>utiles</artifactId>
  <version>1.0</version>
</dependency>
```

Por defecto se aplica la opción <scope>compile</scope>. Si no queremos añadir el JAR al repositorio local podemos referenciar su posición en la dependencia:

```
<dependency>
  <groupId>com.cloudftic</groupId>
  <artifactId>utiles</artifactId>
  <version>1.0</version>
  <scope>system</scope>
  <systemPath>C:/Temp/apps/utiles/target/utiles-1.0.jar</systemPath>
</dependency>
```

Si especificamos <scope>system</scope> debemos añadir el nodo <systemPath /> indicando la ruta dónde encontrar la librería.

Maven nos permitirá compilar el proyecto, pero la librería «utiles-1.0.jar» debe ser añadida posteriormente al *class path* para la ejecución.

Podemos provocar que esta dependencia se añada automáticamente a una subcarpeta «lib/» junto al fichero «app1.jar» y quede establecido el *class path* en el fichero de manifiesto:

```
<build>
  <finalName>app1</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>3.3.0</version>
      <configuration>
        <archive>
          <manifest>
            <mainClass>com.cloudftic.example.App</mainClass>
            <addClasspath>true</addClasspath>
          </manifest>
          <manifestEntries>
            <Class-Path>lib/utiles-1.0.jar</Class-Path>
          </manifestEntries>
        </archive>
```

```
        </configuration>
      </plugin>
    </plugins>
    <resources>
      <resource>
        <directory>c:/Temp/apps/utiles/target/</directory>
        <includes>
          <include>*.jar</include>
        </includes>
        <targetPath>${project.build.directory}/lib</targetPath>
      </resource>
    </resources>
  </build>
```

En el nodo <manifest> se establece la clase principal, y la opción Class-Path del fichero de manifiesto. El nodo <resource> permite especificar la ubicación de ficheros de recursos (con el subnodo <directory>), patrones para incluir o excluir ficheros determinados (con los subnodos <includes> y <excludes>) y la carpeta destino (con el subnodo <targetPath>).

12.2.2. Propiedades de Maven con rutas predefinidas.

Para ayudar a establecer rutas, Maven ofrece varias propiedades predefinidas que se resumen a continuación:

Propiedad	Contiene
${project.basedir}	La carpeta raíz del proyecto (la ubicación donde está el fichero pom.xml).
${project.build.directory}	La ubicación de la carpeta "target".
${project.build.outputDirectory}	La ubicación de la carpeta "target/classes".
${project.build.testOutputDirectory}	La ubicación de la carpeta "target/test-classes".
${project.build.sourceDirectory}	La ubicación de la carpeta "src/main/java".
${project.build.testSourceDirectory}	La ubicación de la carpeta "src/test/java".
${home}	La ruta de la carpeta del usuario actual.
${settings.localRepository}	La ruta del repositorio local. Por defecto igual a ${home}/.m2/repository

Además, depondremos de las siguientes propiedades:

Propiedad	Contiene
${project.artifactId}	El identificador de artefacto.
${project.version}	La versión del proyecto.
${project.build.finalName}	El nombre completo del proyecto. Obtenido como ${project.artifactId}-${project.version}

12.2.3. Jerarquía de proyectos POM.

Cuando desarrollamos proyectos muy complejos, nos encontraremos que muchos proyectos comparten algunas o muchas librerías. Maven nos permite compartir fácilmente librerías entre proyectos mediante el concepto de Maven Parent POM. Este concepto es similar a usar herencia entre proyectos.

Lo veremos con un ejemplo. Primero crearemos un proyecto, que será el POM Parent, y que definirá las dependencias reutilizadas por otros proyectos, además de clases propias.

PASO 1. Crear el proyecto padre «appparent» con Maven.

En este proyecto configuraremos su fichero pom.xml con el tipo de empaquetado pom, y dependencias comunes (en este ejemplo Lombok):

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

  <groupId>com.cloudftic</groupId>
  <artifactId>appparent</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>

  <modules>
      <module>appchild</module>
  </modules>

  <name>appparent</name>
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.24</version>
    </dependency>
  </dependencies>
</project>
```

El nodo <modules> es opcional, y nos permite referenciar proyectos hijos. Esto es útil si queremos aplicar acciones de Maven a todos los hijos a la vez, ejecutándolas sobre el padre.

PASO 2. Crear un proyecto hijo «appchild» con Maven.

Este proyecto lo crearemos dentro de la carpeta del proyecto padre «appparent», de forma que tendremos la siguiente jerarquía de carpetas y ficheros:

```
/appparent
   ├─ /src
   ├─ /appchild
   │     ├─ /src
   │     └─ pom.xml
   └─ pom.xml
```

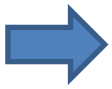
Y configuraremos su fichero pom.xml con el tipo de empaquetado jar, y una referencia al POM padre:

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

   <groupId>com.cloudftic</groupId>
   <artifactId>appchild</artifactId>
   <version>1.0</version>
   <packaging>jar</packaging>

   <parent>
     <groupId>com.cloudftic</groupId>
     <artifactId>appparent</artifactId>
     <version>1.0</version>
   </parent>
```

Ahora, en «appparent», podremos usar la anotaciones de Lombok sin tener que añadir una dependencia en su fichero pom.xm, puesto que heredará esta dependencia del POM padre.

Si no queremos que los proyectos hijos queden anidados en el interior del proyecto padre, podemos aplicar la siguiente configuración:

```
./
├─ /appparent
│     ├─ /src
│     └─ pom.xml
└─ /appchild
      ├─ /src
      └─ pom.xml
```



```
<parent>
  <groupId>com.cloudftic</groupId>
  <artifactId>appparent</artifactId>
  <version>1.0</version>
  <relativePath>../appparent</relativePath>
</parent>
```

12.2.4. Administración de dependencias.

En proyecto Maven Parent POM el nodo <dependencyManagement> permite centralizar la información de dependencias con un control mayor que el nodo <dependencies>.

En el POM principal, la principal diferencia entre <dependencies> y <dependencyManagement>es la siguiente:

1. Los artefactos especificados en <dependencies> SIEMPRE se incluirán como una dependencia de los módulos secundarios.
2. Los artefactos especificados en <dependencyManagement> solo se incluirán en el módulo secundario si también se especificaron en el nodo <dependencies> del propio módulo secundario.
3. En el módulo secundario podemos omitir la versión y/o ámbito declarados en el padre.

Veremos con un ejemplo cómo aplicar este mecanismo. Debemos crear dos proyectos: el proyecto AChild necesita las dependencias alpha y betaShared, mientras que el proyecto BChild necesita las dependencias gamma y betaShared.

En un módulo padre definiremos estas tres dependencias:

Proyecto «Parent»

```
<project>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>alpha</groupId>
        <artifactId>alpha</artifactId>
        <version>1.0</version>
      </dependency>
      <dependency>
        <groupId>gamma</groupId>
```

```
        <artifactId>gamma</artifactId>
        <version>1.0</version>
        <type>war</type>
        <scope>runtime</scope>
      </dependency>
      <dependency>
        <groupId>betaShared</groupId>
        <artifactId>betaShared</artifactId>
        <version>1.0</version>
        <type>bar</type>
        <scope>runtime</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

En los proyectos hijos referenciaremos las dependencias que necesiten:

Proyecto «AChild»

```
<project>
  <dependencies>
    <dependency>
      <groupId>alpha</groupId>
      <artifactId>alpha</artifactId>
    </dependency>
    <dependency>
      <groupId>betaShared</groupId>
      <artifactId>betaShared</artifactId>
      <type>bar</type>
    </dependency>
  </dependencies>
</project>
```

Proyecto «BChild»

```
<project>
  <dependencies>
    <dependency>
      <groupId>gamma</groupId>
      <artifactId>gamma</artifactId>
      <type>war</type>
    </dependency>
    <dependency>
      <groupId>betaShared</groupId>
      <artifactId>betaShared</artifactId>
      <type>bar</type>
    </dependency>
  </dependencies>
</project>
```

En estos POM no necesitamos indicar la versión del artefacto.
Además, la definición de dependencias en <dependencyManagement> nos permitirá controlar la transitividad entre dependencias. Esto ocurre cuando un artefacto define dependencias de otros artefactos, los cuales, a su vez, definen dependencias de otros artefactos.
Por ejemplo, si el artefacto alpha define dependencias de artefactos beta y gamma, en un POM podemos incluir una dependencia para alpha excluyendo la dependencia transitiva beta:

```
<dependency>
  <groupId>alpha</groupId>
  <artifactId>alpha</artifactId>
  <exclusions>
    <exclusion>
      <groupId>gamma</groupId>
      <artifactId>gamma</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

12.3. Modularidad en Java 9.

Los módulos fueron la principal novedad de Java 9. Un módulo permite agrupar el código y los recursos externos usados por una aplicación añadiendo un fichero descriptor que restringe el acceso a sus paquetes, y describe sus dependencias (independientemente de las configuraciones de dependencias de Maven).
De la misma forma que hemos aplicado encapsulación en los datos, al tratar de métodos y clases, la modularización aplica encapsulación sobre los paquetes definidos en una aplicación, de forma que podemos decidir qué paquetes serán visibles para otras aplicaciones y cuáles no. Este nivel de encapsulación permite:

• Encapsulación fuerte de paquetes: la encapsulación se cumple durante la compilación y la ejecución, incluso frente a intentos de accesos mediante técnicas de reflexión.
• Configuración fiable. El intérprete comprueba la disponibilidad de las dependencias de una aplicación antes de ejecutarla. En versiones previas esta comprobación se realizaba en tiempo de ejecución, provocando que los programas fallasen al no encontrarse una determinada librería.
• Creación de imágenes del JDK ajustadas, que incluyen sólo los paquetes que nuestra aplicación necesite. Esto implica menores requerimientos de memoria y disco (útil para microservicios y dispositivos pequeños).
• Mayor seguridad, porque el código implicado es menor.
• Optimización mejorada: se evita la inclusión de código no accedido por la aplicación.
• Servicios desacoplados sin escaneo del classpath (las implementaciones de un interfaz se indican explícitamente).
• Carga rápida de tipos. El sistema sabe dónde está cada paquete sin tener que escanear el classpath.
• Mantiene las fronteras establecidas por la arquitectura de nuestra aplicación.

La encapsulación fuerte implica otros beneficios, como la posibilidad de realizar pruebas aisladas de un módulo, evitar la decadencia del código al introducir dependencias accidentales, y la reducción de dependencias cuando varios equipos trabajan en paralelo.

12.3.1. ¿Qué es un módulo?

Podemos definir un "módulo" Como un artefacto que puede contener código, recursos, y metadatos. Los metadatos describen las dependencias con otros módulos, y regulan el acceso a los paquetes del módulo.
Por ejemplo, un módulo en formato JAR podría contener lo siguiente:

```
appejemplo.jar
├── /com
│       └── /ejemplo
│                   └── HolaMundo.class
└── module-info.class
```

El fichero descriptor «module-info.class» se obtiene por la compilación del fichero «module-info.java» y contiene la meta-información sobre el módulo. Esto incluye:

• El nombre del módulo
• Los paquetes que se exponen
• Las dependencias con otros módulos
• Los servicios consumidos e implementados

El fichero descriptor module-info.java debe ubicarse en la raíz del fichero JAR o directorio de la aplicación. Este fichero se compila junto al resto de ficheros Java como si fuese una clase, pero como su nombre incluye un guion no es un nombre de clase válido, y así, otras herramientas no lo confundirán con un fichero Java.
El fichero module-info.java de ejemplo podría ser como sigue:

```
module ejemplo {
  requires java.util.logging;
  exports com.ejemplo;
}
```

El módulo se llama ejemplo, depende del módulo java.util.logging y expone el paquete com.ejemplo.

12.3.2. Sintaxis del descriptor.

En el descriptor, un módulo se define con las siguientes palabras clave:

module	Comienza la definición de un módulo.
exports ... to	Expone un paquete, opcionalmente a un módulo concreto.
import	Al igual que en los ficheros fuente, se utiliza para importar tipos. Lo normal es usar nombres completos de paquetes en vez de import, pero si debemos repetir mucho un tipo, podemos usarlo.
open	Permite que se puedan aplicar técnicas de reflexión sobre un módulo.
opens ... to	Permite técnicas de reflexión en un paquete concreto, para alguno o para todos los paquetes.
provides ... with	Indica un servicio y su implementación.
requires	Indica la dependencia con un módulo.

requires static	Indica la dependencia con un módulo requerido durante la compilación y opcional durante la ejecución.
requieres transitive	Permite indicar dependencias con las dependencias del módulo requerido.
uses	Indica que se usa un servicio.

El siguiente contenido muestra cómo utilizar estas cláusulas:

```
// nombre del módulo. Con open permite la reflexión en todo el módulo
open module com.ejemplo {
  // exporta un paquete para que otros módulos accedan a sus miembros públicos
  exports com.ejemplo;
  // indica una dependencia con el módulo com.utiles
  requires com.utiles;
  // indica una dependencia obligatoria durante la compilación con el módulo
  // com.config, pero opcional durante ejecución
  requires static com.config;
  // indica una dependencia al módulo com.multi y sus dependencias
  requires transitive com.multi;
  // permite técnicas de reflexión desde el módulo com.negocio
  opens com.negocio;
  // permite reflexión sobre el paquete com.naranjas pero solo desde el módulo com.frutas
  opens com.naranjas to com.frutas;
  // expone el tipo IArchivo, el cual implementa el servicio Archivo
  provides com.service.Archivo with com.consumer.IArchivo
  // usa la clase de servicio com.service.Registro
  uses com.service.Registro
}
```

El uso de modularización cambia el modo en cómo se compilan y ejecutan los proyectos de Java. Por eso vamos a ver cómo compilar "a mano" el proyecto. Para eso abriremos una consola del sistema y nos ubicaremos en la carpeta del proyecto, en este caso "c:\appejemplo":

```
(c:\appejemplo)> javac -d build src\com\ejemplo\HolaMundo.java
```

Con el comando javac se compilan los ficheros de Java. La opción -d indica la carpeta destino y a continuación especificamos los ficheros a compilar.

En la carpeta build debemos añadir el resto de la estructura de ficheros del módulo:

```
/build
├── /com
│       └── /ejemplo
│                   └── HolaMundo.class
└── module-info.class
```

Podemos ejecutar el módulo con el siguiente comando:

```
(c:\appejemplo)> java --module-path build --module com.ejemplo/com.ejemplo.HolaMundo
```

El comando java lanza el intérprete de Java para que ejecute el módulo. La opción --module-path especifica la carpeta del fichero descriptor y la opción --module especifica el nombre del módulo, una barra, y el nombre cualificado (con sus paquetes) de la clase principal (la que posee el método main).

Si empaquetamos todo el proyecto en un fichero JAR:

```
appejemplo.jar
├── /META-INF
│       └── MANIFEST.MF
├── /com
│       └── /ejemplo
│                   └── HolaMundo.class
└── module-info.class
```

Podemos ejecutarlo de la siguiente manera:

```
d:\appejemplo> java -p dist\appejemplo.jar -m com.ejemplo
```

En este caso, con la opción -p se especifica la ruta de un JAR que empaqueta un módulo, y con la opción -m el nombre el módulo.

12.3.3. Intérpretes mínimos generados con «jlink».

Desde Java 9, la herramienta jlink, y gracias a la modularización, permite generar imágenes del JDK con únicamente los módulos que necesite una aplicación. Esto es especialmente útil para los contenedores de Docker y los entornos cloud ya que permite generar imágenes de contenedores con un tamaño significativamente menor.

Una invocación básica de la herramienta de enlace jlink, es la siguiente:

```
> jlink --module-path <modulepath> --add-modules <modules> --limit-modules <modules> --output <path>
```

dónde:

--module-path es la ruta donde el enlazador descubrirá los módulos. Pueden ser archivos JAR modulares, archivos JMOD o módulos explotados.

--add-modules indica los módulos que se agregarán a la imagen en tiempo de ejecución. Estos módulos pueden, mediante dependencias transitivas, hacer que se agreguen módulos adicionales.

--limit-modules limita el universo de módulos observables.

--output es el directorio que contendrá la imagen de tiempo de ejecución resultante.

Para ejecutar un ejemplo, supongamos un módulo con el siguiente descriptor:

```
module test.example {
  requires java.logging;
}
```

Y todos los ficheros del módulo los ubicamos dentro de la carpeta c:\app\out. Podemos comprobar las dependencias con el comando jdeps:

```
(c:\app)> jdeps --module-path out --module test.example
test.example            -> java.io                  java.base
test.example            -> java.lang                java.base
test.example            -> java.util.logging        java.logging
```

El módulo java.base siempre será incluido por defecto y por ello no es necesario requerirlo en el manifiesto. Sólo tenemos que incluir la ruta de estos módulos separados por comas, todos los demás módulos que dependan de estos serán incluidos automáticamente.

Por tanto, el comando será el siguiente:

```
(c:\app)> jlink --module-path D:\java\jdk-11\jmods;out --add-modules test.example --output build
```

Los módulos java.base y java.logging se encuentran en la carpeta de instalación del JDK (en este caso, c:\java\jdk-11\jmods), y añadimos la carpeta out que contiene nuestro módulo. La imagen será creada en la carpeta de salida build.

El tamaño de esta imagen (dentro de la carpeta build) está en torno a los 40MB, en comparación con los 277MB que ocupa JDK 11.