



Instituto Profesional AIEP Spa.

Desarrollo de aplicaciones JEE con Spring Framework (v2)

Manual para el Participante



Dirección Nacional de Educación Continua
Instituto Profesional AIEP Spa.
Junio 2021

Prohibida su reproducción total o parcial sin el
consentimiento explícito de AIEP.
Todos los Derechos Reservados.

INDICE

INTRODUCCIÓN	4
Gestor de Proyectos (IDE)	4
¿Qué es un gestor de proyecto?	4
¿Cuáles son sus beneficios?	5
Spring Framework.....	5
¿Por qué Spring Framework?	5
Arquitectura Spring.....	5
Maven2.....	8
Modelo Objeto Proyecto POM	8
Creación de proyectos	9
¿Qué es un arquetipo?	9
Java	10
Spring MVC	10
Spring Boot	14
Dependencias con Maven.....	16
Repositorio Local y Remoto	16
Ciclo vida Compilación	17
Compilación	18
Ejecución de Pruebas.....	19
Instalación del Artefacto.....	20
Limpieza	20
Ejecución.....	20
Empaquetamiento.	20
El Framework Spring MVC	20
Características Framework SpringBoot.....	21
El concepto y uso de las anotaciones	22
Que es un Bean Spring y como definirlo.....	22
Inyección de dependencias y la anotación autowired.....	23
Creando un Proyecto Web SpringBoot con modulo spring-WEB	24
Configurando el Proyecto.....	26
Configuraciones iniciales de spring boot	26
Configuraciones de vista Jsp Thymeleaf	28
Configuración de Log en Spring MVC	28

Configuración de DataSource	30
Capa de Vistas y Controladores	32
Configurando las peticiones	33
Controladores multilación	34
Recibiendo datos en el controlador.....	34
Entregando datos a la vista.....	34
Desplegando una vista JSP con datos entregados desde el controlador	35
Capa de Servicios.....	35
El rol de la capa de servicios en el modelo MVC	36
Creando un servicio utilizando anotaciones	37
Inyectando un servicio al controlador para su utilización.	37
Test y empaquetamiento	37
Creando Unidades de prueba con Spring	38
Empaquetando una aplicación Spring MVC en un archivo WAR	39
Accesos a datos en Spring Framework	40
Configurando un datasource en Spring	40
Utilizando Jdbc Template en Spring para el acceso a datos	41
Creando un DAO que utiliza JdbcTemplate	42
Realizando queries que reciben parámetros	44
Mapeando los resultados de una consulta a objetos	45
Invocando un DAO desde un servicio	46
Acceso a datos mediante JPA	47
La API de persistencia de Java JPA.....	48
Clase de Entidad en JPA	48
El Entity manager en JPA	49
Clases Repositorio.....	49
Recuperar, actualizar, eliminar un Objeto JPA	49
Asociaciones uno a uno, uno a muchos.....	50
Invocar un repositorio desde un servicio	51
Manejo de la transaccionalidad en los servicios	51
¿Qué es la transaccionalidad y porque es importante?	51
Configurando la transaccionalidad	52
Creando un servicio transaccional.....	53
Control de acceso mediante spring security.....	53
Incorporando el módulo spring security al proyecto.....	55
Configuración básica de Spring Security	55
Creando un formulario de Login	56
Realizando un Login y Logout de una aplicación	57
Manejo de Roles	58

Añadiendo seguridad a los elementos de la capa vista	59
Obteniendo el usuario autenticado en el controlador	61
Agregando seguridad en los controladores mediante anotaciones	62
Autenticación contra una bd	63
Autenticación utilizando JPA.	64
<i>La interoperabilidad entre los sistemas.....</i>	<i>64</i>
Protocolo de intercambio de datos	65
Tipos de protocolos de red	66
¿Qué es el Estado Representacional de Transferencia REST?	66
La notación JSON para el traspaso de información	67
Consumiendo un Servicio REST con Spring y Rest Template.	68
Principios de diseño de una API REST	69
Creando una API REST con Spring MVC	71
Segurización de una API REST mediante JWT.	73
<i>BIBLIOGRAFÍA</i>	<i>77</i>
<i>GLOSARIO</i>	<i>77</i>

INTRODUCCIÓN

El presente documento tiene como objetivo explicar las capacidades de un entorno de trabajo, sus funciones y capacidades, además explicar en detalle algunos de los módulos del framework de spring, con la intención de preparar al lector para generar soluciones empresariales simples y efectivas, conociendo la arquitectura y con esto ser un aporte al momento de diseñar una aplicación, aplicando los conocimientos de este documento.

Gestor de Proyectos (IDE)

¿Qué es un gestor de proyecto?

Un entorno de desarrollo integrado (IDE) es un sistema de software para el diseño de aplicaciones que combina herramientas comunes para desarrolladores en una sola interfaz de usuario gráfica (GUI).

Los IDE se encargan de ayudar a los desarrolladores a organizar su flujo de trabajo y solucionar problemas además analizan el código mientras se escribe, así que las fallas causadas por errores humanos se identifican en tiempo real. Gracias a que hay una sola GUI que representa todas las herramientas, los desarrolladores pueden ejecutar tareas sin tener que pasar de una aplicación a otra. El resaltado de sintaxis también es común en la mayoría de los IDE, y utiliza indicaciones visuales para distinguir la gramática en el editor de texto. Además, algunos IDE incluyen examinadores de objetos y clases, así como diagramas de jerarquía de clases para ciertos lenguajes.

¿Cuáles son sus beneficios?

Editor de código fuente: editor de texto que ayuda a escribir el código de software con funciones como el resaltado de la sintaxis con indicaciones visuales, el relleno automático específico para el lenguaje y la comprobación de errores a medida que se escribe el código.

Automatización de compilaciones locales: herramientas que automatizan tareas sencillas y repetitivas como parte de la creación de una compilación local del software para su uso por parte del desarrollador, como la compilación del código fuente de la computadora en un código binario, el empaquetado de ese código y la ejecución de pruebas automatizadas.

Depurador: programa que sirve para probar otros programas y mostrar la ubicación de un error en el código original de forma gráfica.

Spring Framework.

Spring Framework es una plataforma Java de código abierto que proporciona un soporte integral de infraestructura para desarrollar aplicaciones Java robustas de manera muy fácil y rápida. Spring Framework fue escrito inicialmente por Rod Johnson y se lanzó por primera vez bajo la licencia Apache 2.0 en junio de 2003. Este documento se ha escrito sobre la base de Spring Framework versión 4.1.6 lanzada en marzo de 2015.

¿Por qué Spring Framework?

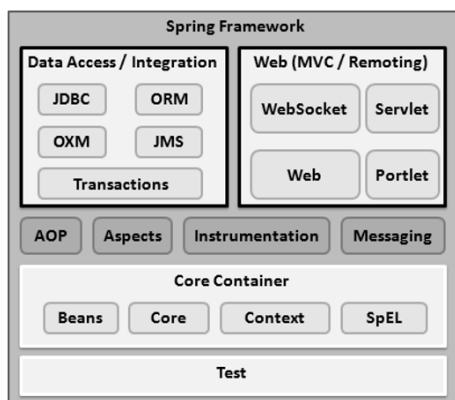
Spring es el marco de desarrollo de aplicaciones más popular para Java empresarial. Millones de desarrolladores de todo el mundo utilizan Spring Framework para crear código de alto rendimiento, fácilmente comprobable y reutilizable.

Spring es ligero en cuanto a tamaño y transparencia.

Las características principales de Spring Framework se pueden utilizar para desarrollar cualquier aplicación Java, pero existen extensiones para crear aplicaciones web sobre la plataforma Java EE. Spring Framework tiene como objetivo facilitar el uso del desarrollo J2EE y promueve las buenas prácticas de programación al permitir un modelo de programación basado en POJO.

Arquitectura Spring

Spring Framework proporciona alrededor de 20 módulos que se pueden usar en función de los requisitos de la aplicación.



Spring es un framework modular que cuenta con una arquitectura dividida en siete capas o módulos, como se muestra en la Figura, lo cual permite tomar y ocupar únicamente las partes que interesen para el proyecto y juntarlas con gran libertad.

Contenedor de núcleo

El Contenedor principal consta de los módulos Core, Beans, Context y Expression Language, cuyos detalles son los siguientes:

- El módulo Core proporciona las partes fundamentales del marco, incluidas las funciones de IoC y Dependency Injection.
- El módulo Bean proporciona BeanFactory, que es una implementación sofisticada del patrón de fábrica.
- El módulo de contexto se basa en la sólida base proporcionada por los módulos Core y Beans y es un medio para acceder a cualquier objeto definido y configurado. La interfaz ApplicationContext es el punto focal del módulo Context.
- El módulo SpEL proporciona un poderoso lenguaje de expresión para consultar y manipular un gráfico de objetos en tiempo de ejecución.

Acceso / integración de datos

La capa de Acceso / Integración de Datos consta de los módulos JDBC, ORM, OXM, JMS y Transacción cuyo detalle es el siguiente -

- El módulo JDBC proporciona una capa de abstracción JDBC que elimina la necesidad de una tediosa codificación relacionada con JDBC.
- El módulo ORM proporciona capas de integración para API populares de mapeo relacional de objetos, incluidas JPA, JDO, Hibernate e iBatis.

- El módulo OXM proporciona una capa de abstracción que admite implementaciones de mapeo Object / XML para JAXB, Castor, XMLBeans, JiBX y XStream.
- El módulo JMS de Java Messaging Service contiene características para producir y consumir mensajes.
- El módulo Transaction admite la gestión de transacciones programática y declarativa para clases que implementan interfaces especiales y para todos sus POJO.

Web

La capa Web consta de los módulos Web, Web-MVC, Web-Socket y Web-Portlet, cuyos detalles son los siguientes:

- El módulo Web proporciona funciones básicas de integración orientadas a la Web, como la funcionalidad de carga de archivos de varias partes y la inicialización del contenedor de IoC mediante oyentes de servlets y un contexto de aplicación orientado a la Web.
- El módulo Web-MVC contiene la implementación Model-View-Controller (MVC) de Spring para aplicaciones web.
- El módulo Web-Socket proporciona soporte para la comunicación bidireccional basada en WebSocket entre el cliente y el servidor en aplicaciones web.
- El módulo Web-Portlet proporciona la implementación MVC que se utilizará en un entorno de portlet y refleja la funcionalidad del módulo Web-Servlet.

Diverso

Hay algunos otros módulos importantes como AOP, Aspectos, Instrumentación, Web y módulos de prueba, cuyos detalles son los siguientes:

- El módulo AOP proporciona una implementación de programación orientada a aspectos que le permite definir interceptores de métodos y cortes de puntos para desacoplar claramente el código que implementa la funcionalidad que debe separarse.
- El módulo Aspects proporciona integración con AspectJ, que nuevamente es un marco AOP poderoso y maduro.

- El módulo de Instrumentación proporciona soporte de instrumentación de clases e implementaciones de cargadores de clases para su uso en ciertos servidores de aplicaciones.
- El módulo de mensajería proporciona soporte para STOMP como el subprotocolo WebSocket para usar en aplicaciones. También admite un modelo de programación de anotaciones para enrutar y procesar mensajes STOMP de clientes WebSocket.
- El módulo de prueba admite la prueba de componentes Spring con marcos JUnit o TestNG.

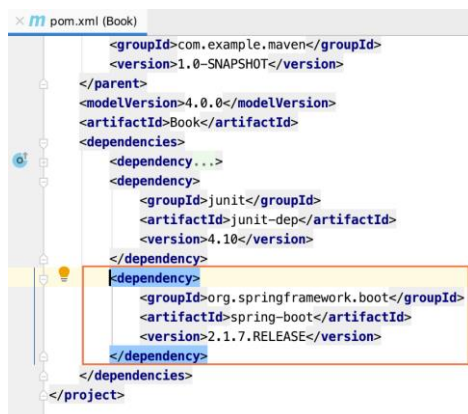
Maven2

Maven2 es una herramienta open-source, que se creó en 2001 con el objetivo de simplificar los procesos de build (compilar y generar ejecutables a partir del código fuente).

Con el fin de optimizar los tiempos de construcción asociados a los proyectos y evitar analizar qué partes de código se deben compilar, qué librerías utiliza el código, dónde incluirlas, y qué dependencias de compilación había en el proyecto.

Modelo Objeto Proyecto POM

El modelo de objetos del proyecto (POM) archivo en formato XML (lenguaje de marcado extensible). La función es similar al archivo build.xml de ant, con funciones más potentes. Este archivo se utiliza para administrar: código fuente, archivos de configuración, información y roles del desarrollador, sistema de seguimiento de problemas, información de la organización, autorización del proyecto, URL del proyecto, dependencias del proyecto



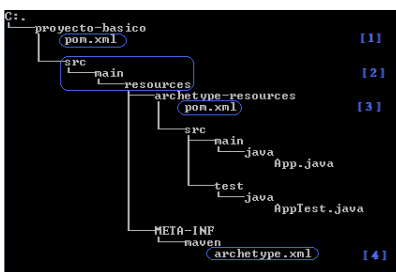
Creación de [proyectos](#)

Para la creación de proyectos utilizaremos el IDE Spring Tool Suite4 (STS), para lo cual se debe generar un Workspace.

Luego debemos elegir un arquetipo Maven para la generación de los proyectos.

¿Qué es un arquetipo?

Los arquetipos Maven, puede decirse que son plantillas, parametrizadas o configuradas para utilizar determinadas tecnologías que los desarrolladores utilizan como base para escribir y organizar el código de la aplicación.



Java

El IDE tiene la capacidad de poder importar o crear un proyecto directamente desde un arquetipo Maven, algunos datos que se solicitan en esta creación es.

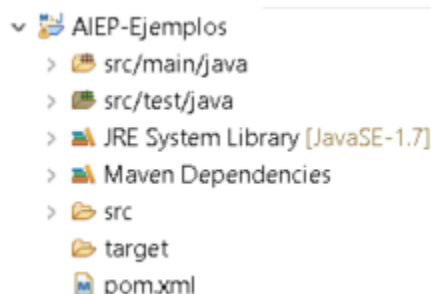
Group ID, corresponde al sistema base de package

Paquetización, debe guardar relación con lo que hace el proyecto.

ArtifactID, será el nombre del compilado (jar,war,ear) debe guardar relación con lo que hace el proyecto y si obedece a un patrón por ejemplo: AIEP-DAO-Clientes.

Versión, es la versión base de creación del compilado.

Package: es la estructura del proyecto a esta altura solo se define el base, aunque se puede editar, lo recomendado es dejarlo idéntico al group-id.



Ejemplo proyecto arquetipo java con la siguiente estructura.

Src/main/java : contendrá los packages de nuestra aplicación

Src/test/java: contendrá los packages test de nuestra aplicación

Jre System Library: indica la versión de Java con que se compilarán nuestras fuentes

Carpeta src: autogenerada

Carpeta target: carpeta de paso, uso de Maven para la compilación y generación del empaquetado.

POM.xml: archivo que permite la administración de nuestro proyecto.

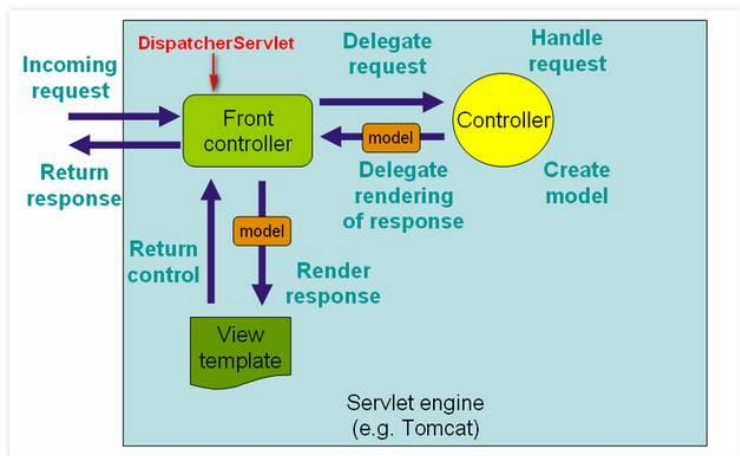
Spring MVC

En la aplicación web Spring MVC, consta de 3 componentes estándar MVC (Modelo, Vistas, Controlador):

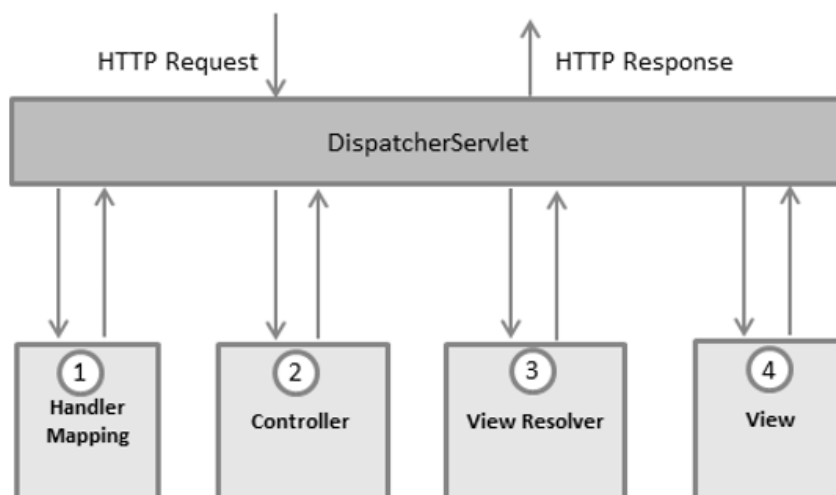
1. Modelos: objetos de dominio que son procesados por la capa de servicio (lógica empresarial) o la capa persistente (operación de la base de datos).

2. Vistas: muestra datos, normalmente es una página JSP escrita con la biblioteca de etiquetas estándar de Java (JSTL).
3. Controladores: mapeo de URL e interactúa con la capa de servicio para el procesamiento comercial y devuelve un modelo.

La siguiente figura demuestra cómo la aplicación web Spring MVC maneja una solicitud web.



El marco de Spring Web model-view-controller (MVC) está diseñado en torno a un DispatcherServlet que envía solicitudes a los controladores, con asignaciones de controladores configurables, resolución de vista, resolución de configuración regional y de tema, así como soporte para cargar archivos. El controlador predeterminado se basa en las anotaciones @Controllery @RequestMapping, lo que ofrece una amplia gama de métodos de manejo flexibles. Con la introducción de Spring 3.0, el @Controllermecanismo también le permite crear sitios web y aplicaciones RESTful, a través de la @PathVariableanotación y otras características.



A continuación se muestra la secuencia de eventos correspondientes a una solicitud HTTP entrante a DispatcherServlet :

- Después de recibir una solicitud HTTP, DispatcherServlet consulta a HandlerMapping para llamar al controlador apropiado.
- El controlador toma la solicitud y llama a los métodos de servicio apropiados según el método GET o POST utilizado. El método de servicio establecerá los datos del modelo en función de la lógica empresarial definida y devuelve el nombre de la vista al DispatcherServlet.
- El DispatcherServlet tendrá la ayuda de ViewResolver a la recolección de la vista definida para la solicitud.
- Una vez finalizada la vista, DispatcherServlet pasa los datos del modelo a la vista que finalmente se representa en el navegador.

En Spring Web MVC puede usar cualquier objeto como comando o como objeto de respaldo de formulario; no es necesario implementar una interfaz o una clase base específicas del marco. El enlace de datos de Spring es muy flexible: por ejemplo, trata las discrepancias de tipos como errores de validación que pueden ser evaluados por la aplicación, no como errores del sistema. Por lo tanto, no necesita duplicar las propiedades de sus objetos comerciales como cadenas simples sin escribir en sus objetos de formulario simplemente para manejar envíos no válidos o para convertir las cadenas correctamente. En cambio, a menudo es preferible vincular directamente a sus objetos comerciales.

El módulo web de Spring incluye muchas características únicas de soporte web:

- *Separación clara de roles.* Cada función (controlador, validador, objeto de comando, objeto de formulario, objeto de modelo DispatcherServlet, asignación de controlador, resolución de vista, etc.) puede ser cumplida por un objeto especializado.
- *Configuración potente y sencilla de clases de aplicación y marco como JavaBeans.* Esta capacidad de configuración incluye referencias sencillas en distintos contextos, como desde controladores web hasta validadores y objetos comerciales.
- *Adaptabilidad, no intrusividad y flexibilidad.* Defina cualquier firma de método de controlador que necesite, posiblemente utilizando una de las anotaciones de parámetros (como @RequestParam, @RequestHeader, @PathVariable y más) para un escenario determinado.
- *Código comercial reutilizable, sin necesidad de duplicarlo.* Utilice objetos de negocio existentes como objetos de comando o formulario en lugar de duplicarlos para ampliar una clase base de marco en particular.
- *Encuadernación y validación personalizables.* Las discrepancias de tipos como errores de validación a nivel de aplicación que mantienen el valor ofensivo, la fecha localizada y el enlace de números, etc., en lugar de objetos de formulario de solo cadena con análisis manual y conversión a objetos comerciales.
- *Mapeo de controladores y resolución de vista personalizables.* Las estrategias de resolución de vistas y mapeo de manejadores van desde una configuración simple basada en URL hasta estrategias de resolución sofisticadas y especialmente diseñadas. Spring es más flexible que los marcos web MVC que exigen una técnica en particular.
- *Transferencia de modelo flexible.* La transferencia de modelos con un nombre / valorMap permite una fácil integración con cualquier tecnología de visualización.
- *Resolución de tema y configuración regional personalizable, compatibilidad con JSP con o sin biblioteca de etiquetas Spring, compatibilidad con JSTL, compatibilidad con Velocity sin necesidad de puentes adicionales, etc.*
- *Una biblioteca de etiquetas JSP simple pero poderosa conocida como la biblioteca de etiquetas Spring que brinda soporte para funciones como el enlace de datos y los temas.* Las etiquetas personalizadas permiten la máxima flexibilidad en términos de código de marcado.
- *Una biblioteca de etiquetas de formulario JSP, introducida en Spring 2.0, que facilita mucho la escritura de formularios en páginas JSP.*

- Beans cuyo ciclo de vida tiene como ámbito la solicitud HTTP actual o HTTP Session. Esta no es una característica específica de Spring MVC en sí, sino de los WebApplicationContext contenedores que usa Spring MVC.

Spring Boot

Spring Boot es un módulo de Spring Framework. Se utiliza para crear aplicaciones independientes basadas en Spring de grado de producción con un esfuerzo mínimo.

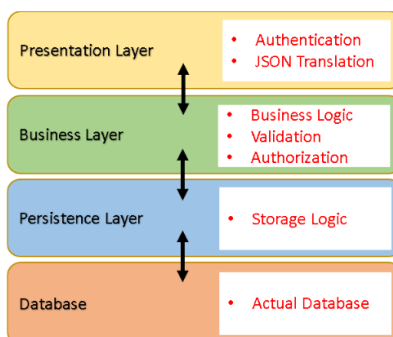
Spring de grado de producción autoconfigurable. Con él, los desarrolladores pueden comenzar rápidamente sin perder tiempo preparando y configurando su aplicación Spring. Spring Boot está construido sobre el Framework Spring y viene con muchas dependencias que se pueden conectar a la aplicación Spring. Algunos ejemplos son Spring Kafka, Spring LDAP, Spring Web Services y Spring Security.

Arquitectura.

Spring Boot sigue una arquitectura en capas en la que cada capa se comunica con la capa directamente debajo o encima (estructura jerárquica).

Antes de comprender la arquitectura Spring Boot, debemos conocer las diferentes capas y clases presentes en ella. Hay cuatro capas en Spring Boot que son las siguientes:

- Capa de presentación
- Capa empresarial
- Capa de persistencia
- Capa de base de datos

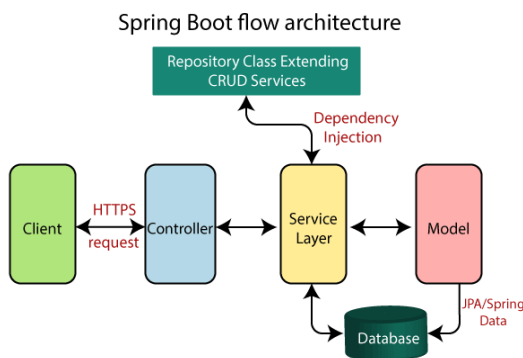


Capa de presentación: la capa de presentación maneja las solicitudes HTTP, traduce el parámetro JSON en un objeto, autentica la solicitud y la transfiere a la capa empresarial. En resumen, consta de vistas, es decir, parte de la interfaz.

Capa empresarial: la capa empresarial maneja toda la lógica empresarial. Consiste en clases de servicio y utiliza servicios proporcionados por capas de acceso a datos. También realiza autorización y validación.

Capa de persistencia: la capa de persistencia contiene toda la lógica de almacenamiento y traduce los objetos comerciales desde y hacia las filas de la base de datos.

Capa de base de datos: En la capa de base de datos, se realizan operaciones CRUD (crear, recuperar, actualizar, eliminar).



- El cliente realiza una solicitud HTTP (GET o POST).
- La solicitud se reenvía al controlador, que asigna la solicitud y la procesa. También llama a la lógica de servicio si es necesario.
- La lógica empresarial se realiza en la capa de servicio y la lógica se realiza en los datos de la base de datos que se mapea con el modelo o la clase de entidad a través de JPA.
- Se devuelve una página JSP como respuesta al cliente si no se ha producido ningún error.


```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.3.4.RELEASE</version>
</parent>
```

La imagen muestra cómo incluir las dependencias de SpringBoot, en el proyecto.

Dependencias con Maven

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

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

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

Con esta configuración, Maven buscará dicha librería y la dejará en nuestro repositorio local .M2

Repositorio Local y Remoto

Un repositorio en Maven contiene artefactos de compilación y dependencias de diversos tipos.

Hay exactamente dos tipos de repositorios: locales y remotos:

1. El repositorio local es un directorio en la computadora donde se ejecuta Maven. Almacena en caché las descargas remotas y contiene artefactos de compilación temporales que aún no ha publicado.
2. Los repositorios remotos se refieren a cualquier otro tipo de repositorio, al que se accede mediante una variedad de protocolos como file://y https://. Estos repositorios pueden ser un repositorio verdaderamente remoto configurado por

un tercero para proporcionar sus artefactos para su descarga (por ejemplo, repo.maven.apache.org). Otros repositorios "remotos" pueden ser repositorios internos configurados en un archivo o servidor HTTP dentro de su empresa, usados para compartir artefactos privados entre equipos de desarrollo y para lanzamientos.

Los repositorios locales y remotos están estructurados de la misma manera para que los scripts se puedan ejecutar en ambos lados o se puedan sincronizar para su uso sin conexión. Sin embargo, el diseño de los repositorios es completamente transparente para el usuario de Maven.

Ciclo vida Compilación

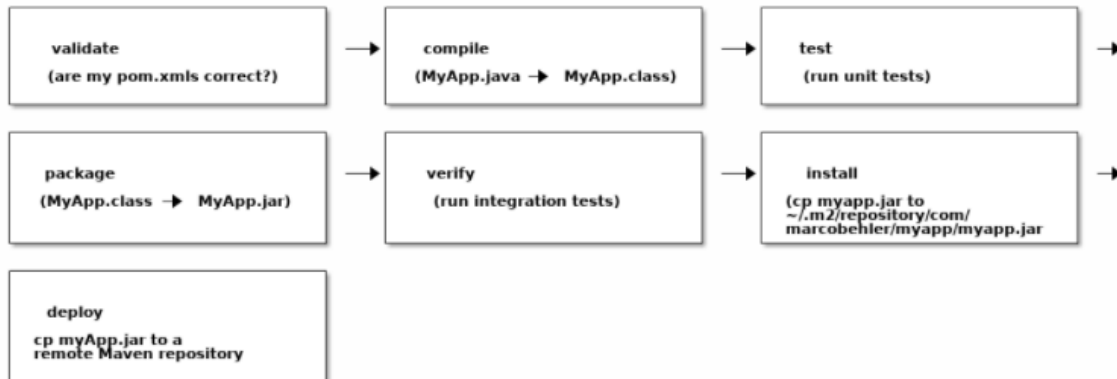
En Maven se definen tres ciclos de build del software con una serie de etapas diferenciadas. Por ejemplo, el ciclo de vida de la compilación tiene las etapas de:

- Validación (validate): Validar que el proyecto es correcto.
- Compilación (compile).
- Test (test): Probar el código fuente usando un framework de pruebas unitarias.
- Empaquetar (package): Empaquetar el código compilado y transformarlo en algún formato tipo .jar o .war.
- Pruebas de integración (integration-test): Procesar y desplegar el código en algún entorno donde se puedan ejecutar las pruebas de integración.
- Verificar que el código empaquetado es válido y cumple los criterios de calidad (verify).
- Instalar el código empaquetado en el repositorio local de Maven, para usarlo como dependencia de otros proyectos (install).
- Desplegar el código a un entorno (deploy).

Hay otros dos ciclos de vida de Maven más allá de la lista *predeterminada* anterior. Son

- **Clean:** limpia los artefactos creados por compilaciones anteriores
- **Sitio:** genera la documentación del sitio para este proyecto

En realidad, las fases se asignan a los objetivos subyacentes. Los objetivos específicos ejecutados por fase dependen del tipo de empaque del proyecto. Por ejemplo, el paquete ejecuta jar: jar si el tipo de proyecto es un JAR, y war: war si el tipo de proyecto es, lo adivinó, un WAR.

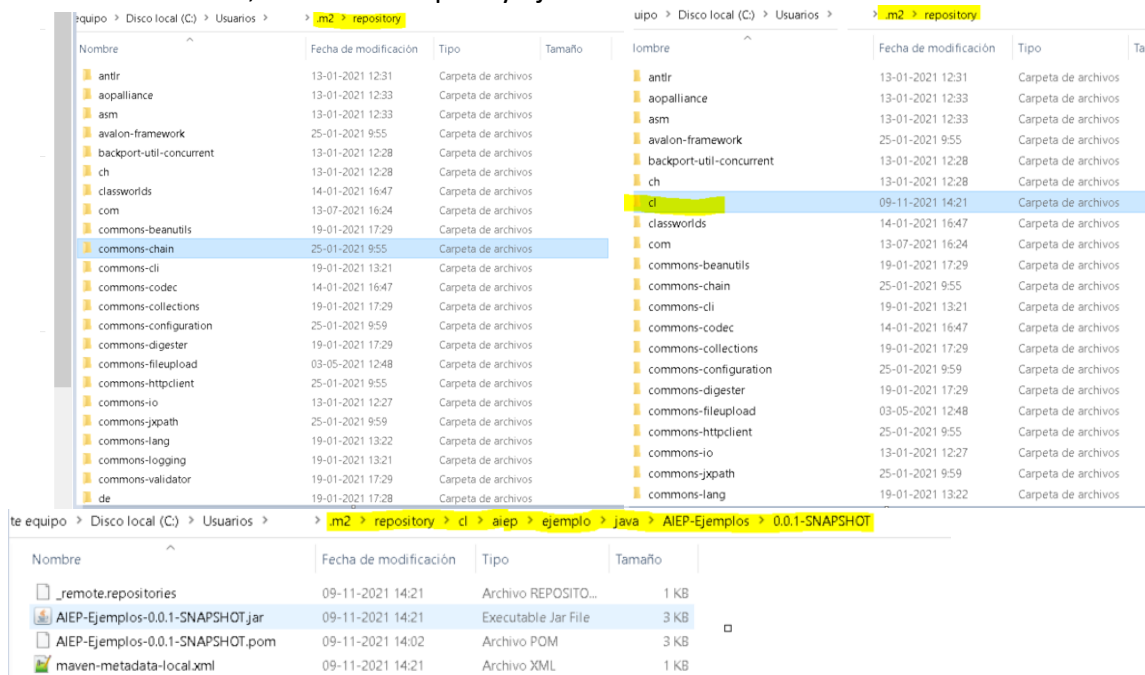


Compilación

Para la compilación utilizaremos el entorno de desarrollo, se debe ejecutar el comando “mvn clean install”

El comando **clean** realiza una limpieza de las librerías previamente instaladas en el directorio .M2

El comando **install**, vuelve a compilar y ejecuta el ciclo de vida nuevamente.



En el IDE se desplegará una consola con lo siguiente

```
[INFO] -----< cl.ejemplo.java:AIEP-Ejemplos >-----
[INFO] Building AIEP-Ejemplos 0.0.1-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ AIEP-Ejemplos ---
[INFO] Deleting C:\Users\jary\Documents\AIEP\MaterialApoyo\AIEP-Ejemplos\target
[INFO] --- maven-resources-plugin:3.0.2:resources (default-resources) @ AIEP-Ejemplos ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\Users\jary\Documents\AIEP\MaterialApoyo\AIEP-Ejemplos\src\main\resources
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ AIEP-Ejemplos ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to C:\Users\jary\Documents\AIEP\MaterialApoyo\AIEP-Ejemplos\target\classes
[INFO] --- maven-resources-plugin:3.0.2:testResources (default-testResources) @ AIEP-Ejemplos ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\Users\jary\Documents\AIEP\MaterialApoyo\AIEP-Ejemplos\src\test\resources
[INFO] --- maven-compiler-plugin:3.8.0:testCompile (default-testCompile) @ AIEP-Ejemplos ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to C:\Users\jary\Documents\AIEP\MaterialApoyo\AIEP-Ejemplos\target\test-classes
[INFO] --- maven-surefire-plugin:2.22.1:test (default-test) @ AIEP-Ejemplos ---
[INFO] Downloading from : https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-junit4/2.22.1/surefire-junit4-2.22.1.pom (3.1 KB at 3.8 KB/s)
[INFO] Downloading from : https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-junit4/2.22.1/surefire-junit4-2.22.1.pom (3.1 KB at 3.8 KB/s)
[INFO] Downloading from : https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-providers/2.22.1/surefire-providers-2.22.1.pom (2.5 KB at 11 KB/s)
[INFO] Downloading from : https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-junit4/2.22.1/surefire-junit4-2.22.1.jar (85 KB at 228 KB/s)
[INFO] Downloading from : https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-junit4/2.22.1/surefire-junit4-2.22.1.jar (85 KB at 228 KB/s)
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running cl.ejemplo.java.AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.019 s - in cl.ejemplo.java.AppTest
[INFO] Results:
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ AIEP-Ejemplos ---
[INFO] Building jar: C:\Users\jary\Documents\AIEP\MaterialApoyo\AIEP-Ejemplos\target\AIEP-Ejemplos-0.0.1-SNAPSHOT.jar
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ AIEP-Ejemplos ---
[INFO] Installing C:\Users\jary\Documents\AIEP\MaterialApoyo\AIEP-Ejemplos\target\AIEP-Ejemplos-0.0.1-SNAPSHOT.jar to C:\Users\jary\L2\repository\cl\ejemplo\java\AIEP-Ejemplos\0.0.1-SNAPSHOT\AIEP-Ejemplos-0.0.1-SNAPSHOT.jar
[INFO] Installing C:\Users\jary\Documents\AIEP\MaterialApoyo\AIEP-Ejemplos\pom.xml to C:\Users\jary\L2\repository\cl\ejemplo\java\AIEP-Ejemplos\0.0.1-SNAPSHOT\AIEP-Ejemplos-0.0.1-SNAPSHOT.pom
[INFO] BUILD SUCCESS
[INFO] Total time: 3.607 s
[INFO] Finished at: 2021-11-09T14:21:30-05:00
[INFO] -----
```

En la consola podemos ver el paso a paso de la generación del empaquetado

Ejecución de Pruebas.

En el siguiente POM podemos ver la utilización de dependencias para el uso de TestCases de nuestras clases.

```
<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.companyname.projectgroup</groupId>
  <artifactId>project</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
    </dependency>
  </dependencies>
</project>
```

Con el comando “package o install” Maven ejecutara los test cases declarados en la carpeta test previamente indicada.

Generando el paso a paso de acuerdo con el punto Ciclo de vida de este documento.

Instalación del Artefacto

El comando **install**, vuelve a compilar y ejecuta el ciclo de vida nuevamente.

Limpieza

El comando **Clean**, genera la eliminación del artefacto en el repositorio local y elimina las clases previamente compiladas en la carpeta Target

Ejecución

Después de que la compilación de Maven sea exitosa, se puede ejecutar el jar directamente desde la carpeta.M2 o desde la carpeta \target\ classes con el siguiente comando java.

```
> java -jar AIEP-Ejemplos-0.0.1-SNAPSHOT.jar
```

Empaquetamiento.

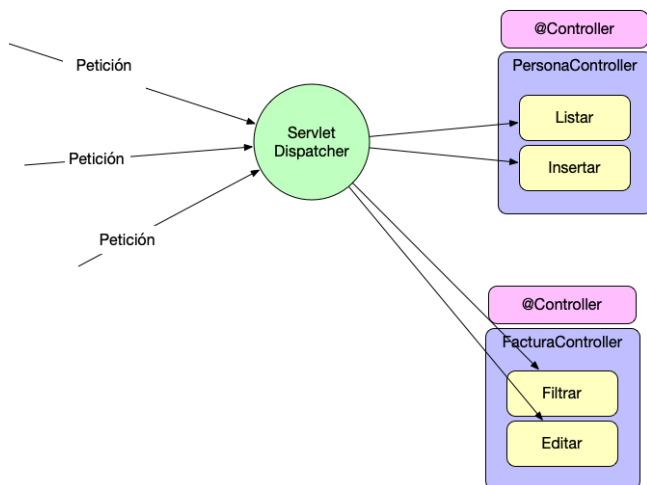
El comando **package**, genera el tipo de empaquetamiento indicado en el POM ya sea un Jar o War o Ear.

El Framework Spring MVC

Spring MVC es quizás el framework Web más utilizado en el mundo Java y nos permite crear aplicaciones sobre modelo MVC que generen páginas HTML sencillas en las cuales nosotros podamos cargar los contenidos que necesitemos de forma sencilla pudiendo integrarse con otras tecnologías como jQuery, React o Vue a la hora de generar aplicaciones modernas y flexibles.

Spring Framework proporciona un modelo integral de programación y configuración para aplicaciones empresariales modernas basadas en Java, en cualquier tipo de plataforma de implementación.

Un elemento clave de Spring es el soporte de infraestructura a nivel de aplicación: Spring se enfoca en la "plomaría" de aplicaciones empresariales para que los equipos puedan enfocarse en la lógica de negocios a nivel de aplicación, sin vínculos innecesarios con entornos de implementación específicos.



Características Framework SpringBoot

Como se indica previamente en el documento Spring Boot es parte del Framework Spring.
Principales Beneficios.

- *Reduce el tiempo de desarrollo y aumenta la productividad general del equipo de desarrollo.*
- *Le ayuda a configurar automáticamente todos los componentes para una aplicación Spring de nivel de producción.*
- *Facilita a los desarrolladores la creación y prueba de aplicaciones basadas en Java al proporcionar una configuración predeterminada para las pruebas unitarias y de integración.*
- *Evita escribir mucho código repetitivo, anotaciones y configuración XML.*
- *Viene con servidores HTTP integrados como Tomcat o Jetty para probar aplicaciones web.*
- *Agrega muchos complementos que los desarrolladores pueden usar para trabajar fácilmente con bases de datos integradas y en memoria. Spring le permite conectarse fácilmente con bases de datos y servicios de cola como Oracle, PostgreSQL, MySQL, MongoDB, Redis, Solr, Elasticsearch, Rabbit MQ, ActiveMQ y muchos más .*

- *Permite la asistencia de administrador, lo que significa que puede administrar mediante acceso remoto a la aplicación.*

El concepto y uso de las anotaciones

El lenguaje de programación Java proporcionó soporte para anotaciones desde Java 5.0 en adelante. Los principales frameworks de Java adoptaron rápidamente anotaciones, y Spring Framework comenzó a usar anotaciones de la versión 2.5. Debido a la forma en que se definen, las anotaciones proporcionan mucho contexto en su declaración.

Antes de las anotaciones, el comportamiento de Spring Framework se controlaba en gran medida a través de la configuración XML. Hoy, el uso de anotaciones nos proporciona capacidades tremendas en la forma en que configuramos los comportamientos de Spring Framework.

Que es un Bean Spring y como definirlo

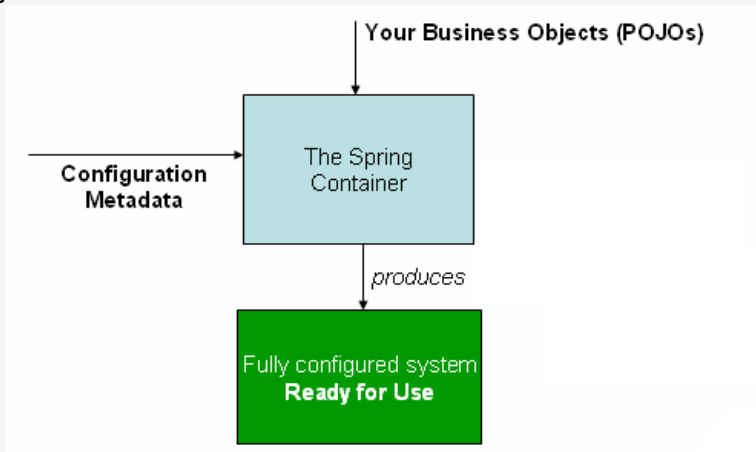
Un Java Bean, también son conocidos un Bean, es una clase simple en Java que cumple con ciertas normas con los nombres de sus propiedades y métodos. Es un componente (similar a una caja) que nos permite encapsular el contenido, con la finalidad de otorgarle una mejor estructura. Que, además, permite la reutilización de código mediante a una estructura sencilla.

Se aconseja que un bean cumpla las siguientes condiciones:

- Constructor sin argumentos: aunque ya existe por defecto (no se ve pero está), se aconseja el declararlo.
- Atributos privados.
- Getters&Setters de todos los atributos.
- El bean puede implementar Serializable.

Requieren una anotación o crear un fichero .xml donde se detalle que es un Bean. Mediante estos JavaBeans, desarrollamos nuestro modelo de objetos (o modelo de dominio) para la aplicación. Nuestros POJOS realmente son Beans siempre y que añadamos una anotación que indique que dicho POJO es un Bean.

El siguiente diagrama muestra una vista de alto nivel de cómo funciona Spring. Las clases de su aplicación se combinan con los metadatos de configuración para que, después de que ApplicationContext se cree e inicialice, tenga un sistema o aplicación completamente configurado y ejecutable.



Inyección de dependencias y la anotación autowired

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

La inyección de dependencias es una forma distinta de diseñar aplicaciones. La DI consiste en que en lugar de que sean las clases las encargadas de crear (instanciar) los objetos que van a usar (sus atributos), los objetos se inyectarán mediante los métodos setters o mediante el constructor en el momento en el que se cree la clase y cuando se quiera usar la clase en cuestión ya estará lista, en cambio sin usar DI la clase necesita crear los objetos que necesita cada vez que se use.

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

Lo que nos ofrece la inyección de dependencias es desacoplamiento y también que los objetos son instanciados en el Contenedor DI y se inyectan donde sea necesario de forma que pueden ser reutilizados.

@Autowired

La anotación `@Autowired` nos permite no tener que definir la propiedad que se quiere inyectar el bean. La anotación `@Autowired` se puede poner encima del atributo que se quiere inyectar, encima del método setter de dicho método o también encima del constructor y dependiendo de donde se ponga la inyección se haría por atributo, por setter o por constructor como es lógico, ejemplo:

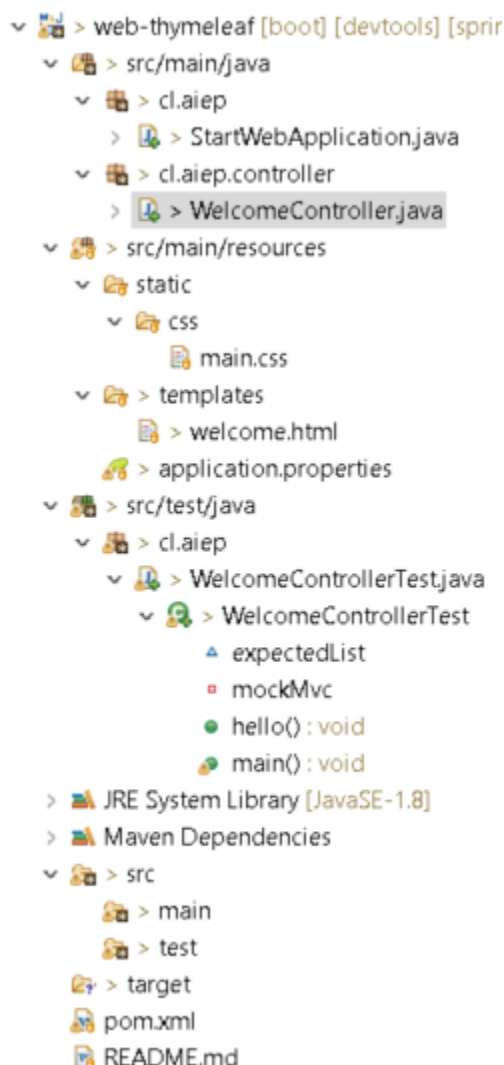
```
@Component // en este caso podríamos usar la especialización @Controller, al tratarse de un controlador
public class InyeccionPropiedadController {

    @Autowired
    public ServicioImplementado servicioImplementado;

    String miMetodo() {
        return servicioImplementado.metodo();
    }
}
```

Creando un Proyecto Web SpringBoot con modulo spring-WEB

Para la creación de un proyecto Spring boot con Spring WEB utilizaremos el IDE antes mencionado STS4, con esto podremos validar y compilar.



Las clases generadas corresponden a lo siguiente.

StartWebApplication: clase principal, es el inicio de SpringBoot

WelcomeController: Clase que contiene el o los Servicios (rest-services) para ser invocados desde un front o backend

application.properties: Archivo de parámetros principal de SpringBoot, en ella se pueden realizar todas las configuraciones de BD, Logs, Accesos, Variables de entorno.

Welcome.html: Template TymeLeaf, Html estático que permite eficiencia en el despliegue.

WelcomeControllerTest: Clase que permite probar de manera automática los servicios de la app, guarda relación con el comando “test” Maven descrito en el documento.

POM.xml: Archivo de Configuración Maven base del proyecto.

Configurando el Proyecto

Para que un proyecto spring boot funcione debe contar con las siguientes configuraciones base.

- JVM, máquina virtual de java para compilación
- IDE, Entorno de Trabajo. (opcional)
- Application.properties, archivo base de configuración.
- POM.xml, archivo Maven con las dependencias de SpringBoot

La clase SpringApplication proporciona una forma conveniente de arrancar una aplicación Spring que se inicia desde un método main(). En muchas situaciones, puede delegar al SpringApplication.run método estático, como se muestra en el siguiente ejemplo

```
@SpringBootApplication
public class MyApplication {

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

}
```

Configuraciones iniciales de spring boot

Las dependencias iniciales de Spring-Boot las cuales se configuran en el archivo POM.xml, son las siguientes.

```
<modelVersion>4.0.0</modelVersion>

<artifactId>web-thymeleaf</artifactId>
<packaging>jar</packaging>
<name>Spring Boot Web Thymeleaf Example</name>
<description>Spring Boot Web Thymeleaf Example</description>
<version>1.0</version>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.2.RELEASE</version>
</parent>

<properties>
  <java.version>1.8</java.version>
  <bootstrap.version>4.2.1</bootstrap.version>
</properties>

<dependencies>

  <!-- web mvc -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

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

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

  <!-- hot swapping, disable cache for template, enable live reload -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
  </dependency>

  <!-- Optional, for bootstrap -->
  <dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>${bootstrap.version}</version>
  </dependency>

</dependencies>
```

org.springframework.boot: Paquete principal contiene las librerías base del framework spring para spring boot

org.springframework.boot: Paquete utilizado para trabajar con el componente MVC del framework, que incluye servicios rest y sus anotaciones.

spring-boot-starter-thymeleaf: Paquete utilizado para soportar el manejo dinámico de las plantillas JSP para Thymeleaf

spring-boot-starter-test: Paquete utilizado para ejecutar las pruebas automáticas desde Maven, comando Test

Configuraciones de vista Jsp Thymeleaf

Thymeleaf es un motor de plantillas, es decir, **es una tecnología que nos va a permitir definir una plantilla** y, juntamente con un modelo de datos, obtener un nuevo documento, sobre todo en entornos web.

Ventajas de Thymeleaf:

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

Es integrable con muchos de los frameworks más utilizados, como por ejemplo Spring MVC, Play, Java EE, etc. Y está basado en el uso de nuevas etiquetas, de nuevos atributos.

Configuración de Log en Spring MVC

Log4j es una biblioteca open source desarrollada en Java por la Apache Software Foundation que permite a los desarrolladores de software escribir mensajes de registro, cuyo propósito es dejar constancia de una determinada transacción en tiempo de ejecución.

Log4j permite filtrar los mensajes en función de su importancia. La configuración de salida y granularidad de los mensajes es realizada en tiempo de ejecución mediante el uso de archivos de configuración externos. Log4J ha sido implementado en otros lenguajes como: C, C++, C#, Perl, Python, Ruby y Eiffel.

Para el caso de Spring boot debemos configurarlo en el POM.xml de la siguiente forma.

```
<!-- exclude logback , add log4j2 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

Lo primero es excluir del paquete Spring-boot-starter, las librerías de logback y log4j2, con esto evitamos que se autoconfiguren.

Segundo se debe agregar el paquete

spring-boot-starter-log4j2: librería que maneja el Log4J2

Luego debemos crear el fichero Log4j2.xml para la generación de la configuración

Quedando así el proyecto.

```

> logging-log4j2 [boot] [spring-boot master]
  > src/main/java
    > cl.aiep
      > HelloController.java
      > StartApplication.java
    > src/main/resources
      > templates
        > welcome.html
        > application.properties
        > log4j2.xml
        > log4j2.yml.bk
  > JRE System Library [JavaSE-1.8]
  > Maven Dependencies
  > target/generated-sources/annotations
  > target/generated-test-sources/test-annotati
  > src
    > main
  > target
  > pom.xml
  > README.md
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Configuration status="DEBUG">
3   <Appenders>
4     <Console name="LogToConsole" target="SYSTEM_OUT">
5       <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
6     </Console>
7     <File name="LogToFile" fileName="logs/app.log">
8       <PatternLayout>
9         <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
10      </PatternLayout>
11    </File>
12  </Appenders>
13  <Loggers>
14    <Logger name="cl.aiep" level="debug" additivity="false">
15      <AppenderRef ref="LogToFile"/>
16      <AppenderRef ref="LogToConsole"/>
17    </Logger>
18    <Logger name="org.springframework.boot" level="error" additivity="false">
19      <AppenderRef ref="LogToConsole"/>
20    </Logger>
21    <Root level="error">
22      <AppenderRef ref="LogToFile"/>
23      <AppenderRef ref="LogToConsole"/>
24    </Root>
25  </Loggers>
26 </Configuration>

```

Luego una vez dentro de las clases se debe invocar de la siguiente forma.

```

13 @Controller
14 public class HelloController {
15
16     private static final Logger logger = LogManager.getLogger(HelloController.class);
17
18     private List<Integer> num = Arrays.asList(1, 2, 3, 4, 5);
19
20     @GetMapping("/")
21     public String main(Model model) {
22
23         // pre-java 8
24         if (logger.isDebugEnabled()) {
25             logger.debug("Hello from Log4j 2 - num : {}", num);
26         }
27
28         // java 8 lambda, no need to check log level
29         logger.debug("Hello from Log4j 2 - num : {}", () -> num);
30
31         model.addAttribute("tasks", num);
32
33         return "welcome"; //view
34     }
35
36     private int getNum(){
37         return 100;
38     }
39 }

```

Configuración de DataSource

Para el caso de las conexiones a BD utilizaremos Hikari parte del paquete `spring-boot-starter-data-jpa` como pool de conexión con una Bd H2

HikariCP es un grupo de conexiones JDBC rápido, simple, confiable y listo para producción.

En la versión Spring Boot 2.0, la tecnología de agrupación de bases de datos predeterminada se ha cambiado de Tomcat Pool a HikariCP. Esto se debe a que HikariCP proporciona un rendimiento excelente. Ahora, desde el lanzamiento de Spring Boot 2.0, `spring-boot-starter-jdbc` y `spring-boot-starter-data-jpa` resuelven las dependencias de HikariCP de forma predeterminada, y la propiedad `spring.datasource.type` toma `HikariDataSource` como valor predeterminado. Spring Boot primero elige HikariCP y luego el grupo Tomcat, y luego elige Commons DBCP2 según la disponibilidad.

Crearemos una aplicación de demostración en la que realizaremos operaciones de creación y lectura en la base de datos. Configuraremos propiedades de HikariCP como `connectionTimeout`, `minimumIdle`, `maximumPoolSize`, `idleTimeout`, `maxLifetime` y `autoCommit` en el archivo.

H2 es un sistema administrador de bases de datos relacionales programado en Java. Puede ser incorporado en aplicaciones Java o ejecutarse de modo cliente-servidor. Una de las características más importantes de H2 es que se puede integrar completamente en aplicaciones Java y acceder a la base de datos lanzando SQL directamente, sin tener que pasar por una conexión a través de sockets.

Está disponible como software de código libre bajo la Licencia Pública de Mozilla o la Eclipse Public License.

Para configurar el proyecto se deben considerar 3 pasos.

- POM.xml se deben incluir las librerías. `spring-boot-starter-data-jpa` y `com.h2database`
- Se debe configurar el `Application.properties` para configurar el Pool Hikari.
`spring.datasource.hikari.connectionTimeout=20000`
`spring.datasource.hikari.maximumPoolSize=5`
`spring.datasource.hikari.poolName=HikariPoolZZZ`
- Se deben configurar las entidades (Beans) y un Repositorio con Inyección de dependencias.

Entidad

```
package cl.aiep;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;

    public Book() {
    }

    public Book(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Book{" +
            "id=" + id +
            ", name='" + name + '\'' +
            '}';
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }
}
```

Se pueden ver algunas anotaciones como @Entity, la cual se utiliza para identificar una tabla en BD

@Id, es la primary key de una entidad, (Obligatorio)

@GeneratedValue, se utiliza para que el id sea autoincremental.

Repositorio.

```
package cl.aiep;

import org.springframework.data.repository.CrudRepository;

public interface BookRepository extends CrudRepository<Book, Long> {

    List<Book> findByName(String name);

}
```

Clase que se encarga de realizar el puente con BD, toma la entidad y genera su CRUD

Finalmente


```
@SpringBootApplication
public class StartApplication implements CommandLineRunner {

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

    @Autowired
    private BookRepository repository;

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

    @Override
    public void run(String... args) {
        log.info("StartApplication...");

        repository.save(new Book("Java"));
        repository.save(new Book("Node"));
        repository.save(new Book("Python"));

        System.out.println("\nfindAll()");
        repository.findAll().forEach(x -> System.out.println(x));

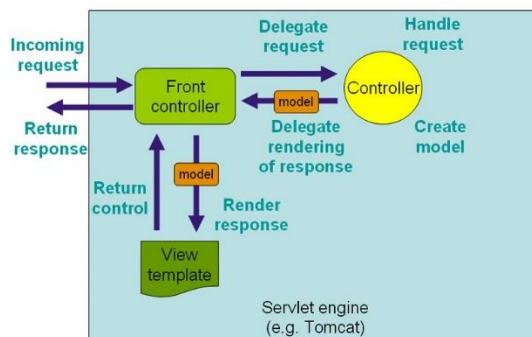
        System.out.println("\nfindById(1)");
        repository.findById(1L).ifPresent(x -> System.out.println(x));

        System.out.println("\nfindByName('Node')");
        repository.findByName("Node").forEach(x -> System.out.println(x));
    }
}
```

Clase principal que permite por inyección de dependencias acceder a la tabla antes creada.

Capa de Vistas y Controladores

Como lo revisamos en el módulo anterior, el modelo MVC de spring framework obedece al siguiente diagrama.



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

- Finalmente, el front controller (el DispatcherServlet) redirige la petición hacia la vista, que muestra los resultados de la operación realizada.

Configurando las peticiones

Para la creación de un servicio en SpringBoot necesitamos lo siguiente:

- POM.xml se deben incluir las librerías. `spring-boot-starter-web`, `spring-boot-starter-data-jpa` y `com.h2database`
- Se debe configurar el `Application.properties` para configurar el Pool Hikari.
`spring.datasource.hikari.connectionTimeout=20000`
`spring.datasource.hikari.maximumPoolSize=5`
`spring.datasource.hikari.poolName=HikariPoolZZZ`
- Se deben configurar las entidades (Beans) y un Repositorio con Inyección de dependencias.
- Se debe configurar un RestController para el manejo y administración de los Rest Services

La entidad y Repositorio se explicaron en modulo anterior

El Rest controller debería ser de la siguiente forma

```
import com.mkyong.error.BookNotFoundException;

@RestController
public class BookController {

    @Autowired
    private BookRepository repository;

    // Find
    @GetMapping("/books")
    List<Book> findAll() {
        return repository.findAll();
    }

    // Save
    @PostMapping("/books")
    //return 201 instead of 200
    @ResponseStatus(HttpStatus.CREATED)
    Book newBook(@RequestBody Book newBook) {
        return repository.save(newBook);
    }

    // Find
    @GetMapping("/books/{id}")
    Book findOne(@PathVariable Long id) {
        return repository.findById(id)
            .orElseThrow(() -> new BookNotFoundException(id));
    }

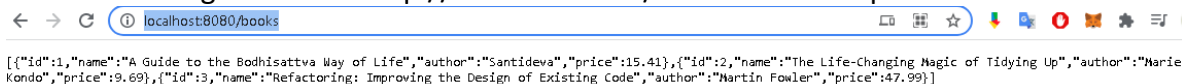
    // Save or update
    @PutMapping("/books/{id}")
    Book saveOrUpdate(@RequestBody Book newBook, @PathVariable Long id) {

        return repository.findById(id)
            .map(x -> {
                x.setName(newBook.getName());
                x.setAuthor(newBook.getAuthor());
                x.setPrice(newBook.getPrice());
                return repository.save(x);
            })
            .orElseGet(() -> {
```

Se generan las siguientes anotaciones

@RestController anotación que indica que se desplegarán Apis en dicha clase.
 @GetMapping, anotación que indica que el servicio responderá por el método GET
 @PostMapping, anotación que indica que el servicio responderá por el método Post
 @PutMapping, anotación que indica que el servicio genera un save/update sobre una entidad.

Al entrar a la siguiente URL <http://localhost:8080/books> debiera aparecer:



```
[{"id":1,"name":"A Guide to the Bodhisattva Way of Life","author":"Santideva","price":15.41}, {"id":2,"name":"The Life-Changing Magic of Tidying Up","author":"Marie Kondo","price":9.69}, {"id":3,"name":"Refactoring: Improving the Design of Existing Code","author":"Martin Fowler","price":47.99}]
```

Controladores multiacción

Como vimos en el ejemplo anterior, un controller permite la generación de múltiples acciones, desde una consulta hasta interacciones con otros sistemas, legados o backends

```
findAll(): List<Book>
newBook(@RequestBody Book): Book
findOne(@PathVariable Long): Book
saveOrUpdate(@RequestBody Book, @PathVariable
patch(@RequestBody Map<String, String>, @Pathr
deleteBook(@PathVariable Long): void
```

Recibiendo datos en el controlador

Como vimos en el ejemplo anterior, un controller permite la generación de múltiples acciones, desde una consulta hasta interacciones con otros sistemas, legados o backends
 En este caso tomaremos como ejemplo el método de búsqueda por ID

```
// Find
@GetMapping("/books/{id}")
Book findOne(@PathVariable Long id) {
    return repository.findById(id)
        .orElseThrow(() -> new BookNotFoundException(id));
}
```

En este método podemos ver que recibe un parámetro de tipo PathVariable, la cual es considerada una buena práctica desde el punto de vista seguridad ya que es mucho más difícil entender el valor del atributo si no estas indicando que atributo es, Ej.

<http://localhost:8080/user/22233445/aglp12>

<http://localhost:8080/user?nroMovil=22233445&patenteAuto=aglp12>

HATEOAS nos dice que debemos mostrar las URIs de forma que el cliente este menos acoplado al servidor y no sea necesario hacer cambios en estos casos.

Entregando datos a la vista

Como vimos en el ejemplo anterior, un controller permite la generación de múltiples acciones, desde una consulta hasta interacciones con otros sistemas, legados o backends
 Para este caso tomaremos como ejemplo el método find by id

```
// Find
@GetMapping("/books/{id}")
Book findOne(@PathVariable Long id) {
    return repository.findById(id)
        .orElseThrow(() -> new BookNotFoundException(id));
}
```

El método retorna el objeto Book en modo JSON

Desplegando una vista JSP con datos entregados desde el controlador

Como vimos en el ejemplo anterior, un controller permite la generación de múltiples acciones, desde una consulta hasta interacciones con otros sistemas, legados o backends. En este caso tomaremos como ejemplo el método de búsqueda.

```
@GetMapping("/")
public String main(Model model) {
    List<Book> tasks = repository.findAll();

    model.addAttribute("message", message);
    model.addAttribute("tasks", tasks);

    return "welcome"; //view
}
```

En este método se obtienen todos los libros desde el repositorio generado con JPA y se entregan a la plantilla Thymeleaf welcome.

```
src/main/resources
├── static
│   ├── css
│   └── templates
│       └── welcome.html
└── application.properties
```

Capa de Servicios

REST abraza los preceptos de la web, incluida su arquitectura, beneficios y todo lo demás. Esto no es ninguna sorpresa dado que su autor, Roy Fielding, estuvo involucrado probablemente en una docena de especificaciones que rigen el funcionamiento de la web.

¿Qué beneficios? La web y su protocolo principal, HTTP, proporcionan una pila de características:

- Acciones adecuadas (GET, POST, PUT, DELETE, ...)
- Almacenamiento en caché

- Redirección y reenvío
- Seguridad (cifrado y autenticación)

Todos estos son factores críticos en la construcción de servicios resilientes. Pero eso no es todo. La web está construida a partir de muchas especificaciones diminutas, por lo que ha podido evolucionar fácilmente, sin atascarse en "guerras de estándares".

Los desarrolladores pueden recurrir a kits de herramientas de terceros que implementan estas diversas especificaciones y al instante tienen la tecnología de cliente y de servidor al alcance de la mano.

Al construir sobre HTTP, las API REST brindan los medios para construir:

- APIS compatibles con versiones anteriores
- APIS evolucionables
- Servicios escalables
- Servicios asegurables

El rol de la capa de servicios en el modelo MVC

Clases de Servicio

Es recomendable definir clases de servicio en las que se implementa la lógica de negocio de la aplicación. Las clases *controller* llaman a las clases servicio, que son las que realmente realizan todo el procesamiento.

De esta forma se separan las responsabilidades. Las clases *controller* se encargan de procesar las peticiones y las respuestas HTTP y las clases de servicio son las que realmente realizan la lógica de negocio y devuelven el contenido de las respuestas. Si en algún momento hay que añadir una nueva capa de presentación en la que, por ejemplo, se trabaje con objetos JSON, no será necesario cambiar la capa de servicios, sólo añadir nuevas clases *controller*.

La separación de la lógica de negocio en las clases de servicio permite también realizar tests que trabajan sobre objetos Java, independientes de los formatos de entrada/salida manejados por los controladores.

Creando un servicio utilizando anotaciones

Como vimos en el ejemplo anterior, un controller permite la generación de múltiples acciones, desde una consulta hasta interacciones con otros sistemas, legados o backends. En este caso generaremos un Service para el manejo de lógica de negocio utilizando la anotación `@Service`.

```
@Service
public class BookService {
    @Autowired
    private BookRepository repository;

    public List<Book> findAll() {
        return repository.findAll();
    }
}
```

Inyectando un servicio al controlador para su utilización.

Como vimos en el ejemplo anterior, un controller permite la generación de múltiples acciones, desde una consulta hasta interacciones con otros sistemas, legados o backends. Una vez realizado el `@Service` se puede utilizar inyección de dependencias para invocarlo desde el controller.

```
@Autowired
BookService service;

@GetMapping("/")
public String main(Model model) {
    List<Book> tasks = service.findAll();

    model.addAttribute("message", message);
    model.addAttribute("tasks", tasks);

    return "welcome"; //view
}
```

Como se puede ver en el ejemplo, el encargado de invocar la entidad de BD es el service y no el controller.

Test y empaquetamiento

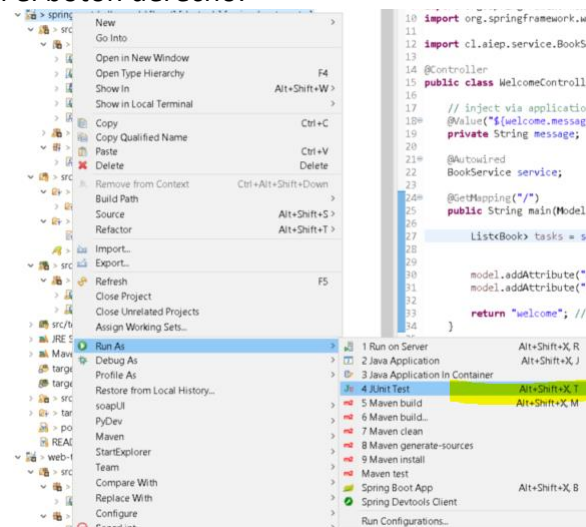
En Spring Boot 2.4 se usa JUnit 5 como librería de tests.

En la aplicación de demostración hay varios ejemplos que muestran posibles formas de realizar pruebas en una aplicación Spring Boot.

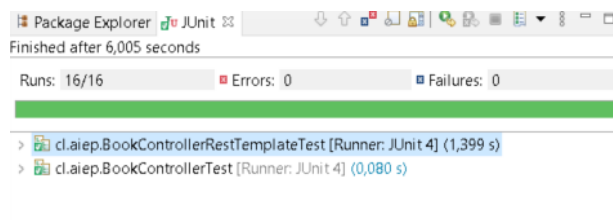
Spring Boot incluye el framework AssertJ que permite realizar expresiones de prueba con un lenguaje muy expresivo.

Los tests se pueden ejecutar usando el comando típico de Maven “test”

También se pueden lanzar desde el propio STS, pulsando en el panel del proyecto sobre el directorio de tests con el botón derecho.

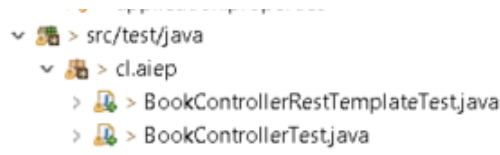


Los tests se lanzarán y aparecerá un panel en el que se mostrará si pasan correctamente (verde) o no.



Creando Unidades de prueba con Spring

Como vimos en el ejemplo anterior, una clase puede ser de pruebas Unitarias estas pueden ser ejecutadas on-demand o automáticamente por los comandos de Maven. Para este caso analizaremos los test asociados al rest controller de books.



En el caso del BookControllerTest.java

```

1 package cl.aiep;
2
3 import com.fasterxml.jackson.core.JsonProcessingException;
4
5 // @WebMvcTest(BookController.class)
6 @RunWith(SpringRunner.class)
7 @SpringBootTest
8 @AutoConfigureMockMvc
9 @ActiveProfiles("test")
10 public class BookControllerTest {
11
12     private static final ObjectMapper om = new ObjectMapper();
13
14     @Autowired
15     private MockMvc mockMvc;
16
17     @MockBean
18     private BookRepository mockRepository;
19
20     @Before
21     public void init() {
22         Book book = new Book(1L, "Book Name", "Mkyong", new BigDecimal("9.99"));
23         when(mockRepository.findById(1L)).thenReturn(Optional.of(book));
24     }
25
26     @Test
27     public void find_bookId_OK() throws Exception {
28
29         mockMvc.perform(get("/books/1"))
30             .andExpect(status().isOk())
31             .andExpect(jsonPath("$.id", is(1)))
32             .andExpect(jsonPath("$.name", is("Book Name")))
33             .andExpect(jsonPath("$.author", is("Mkyong")))
34             .andExpect(jsonPath("$.price", is(9.99)));
35
36         verify(mockRepository, times(1)).findById(1L);
37     }
38 }

```

Tenemos las siguientes anotaciones

@RunWith anotación que permite indicar con que tipo de ejecutor vamos a probar nuestra clase, en este caso utilizaremos el defecto de Spring SpringRunner

@SpringBootTest anotación que indica que esta clase es una de tipo Test.

@AutoconfigureMockMvc Anotación que se puede aplicar a una clase de prueba para habilitar y configurar la configuración automática de MockMvc

@Before anotación que permite configurar el caso de prueba antes de la ejecución

@test anotación que indica que método se va a evaluar.

Empaquetando una aplicación Spring MVC en un archivo WAR

Para empaquetar el proyecto de ejemplo en WAR, solo se debe cambiar un parámetro en el POM.xml de jar a war.


```
<?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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-rest-hello-world</artifactId>
  <packaging>war</packaging>
  <name>Spring Boot REST API Example</name>
  <version>1.0</version>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.2.RELEASE</version>
  </parent>
  <!-- Java 8 -->
  <properties>
    <java.version>1.8</java.version>
    <downloadSources>true</downloadSources>
    <downloadJavadocs>true</downloadJavadocs>
    <bootstrap.version>4.2.1</bootstrap.version>
  </properties>
</project>
```

Luego ejecutar comando mvn clean install y se genera el proyecto.

ring-boot > spring-rest-hello-world > target	
Nombre	Fecha de modificación
classes	10-11-2021 18:34
generated-sources	10-11-2021 18:34
generated-test-sources	10-11-2021 18:34
maven-archiver	10-11-2021 18:34
maven-status	10-11-2021 18:34
spring-rest-hello-world-1.0	10-11-2021 18:34
surefire-reports	10-11-2021 18:34
test-classes	10-11-2021 18:34
spring-rest-hello-world-1.0.war	10-11-2021 18:34
spring-rest-hello-world-1.0.war.original	10-11-2021 18:34

Accesos a datos en Spring Framework

Configurando un datasource en Spring

Generalmente el uso y explotación de bases de datos SQL en Java lo realizamos a través de herramientas de mapeo ORM tales como Hibernate o Apache OpenJPA que permiten desarrollar la capa de persistencia de nuestras aplicaciones de forma muy rápida con poco código y una alta abstracción sobre el modelo relacional. Eso sí, es necesario saber utilizar estas herramientas muy bien para evitar graves problemas de rendimiento.

Spring no sólo es muy fácil de integrar con estos productos tal y hemos visto en el documento, sino que además proporciona con Spring Data JPA su propia capa de abstracción que nos facilita y simplifica aún más el trabajo

Básicamente para la configuración de un datasource en Spring se debe importar la dependencia de la BD en el POM.xml y configurar en el Application.properties, en el siguiente topico mostraremos un ejemplo de esto.

Utilizando Jdbc Template en Spring para el acceso a datos

JdbcTemplate es una de las formas de acceder a la base de datos proporcionada por Spring, y es la forma de implementación más básica y de nivel inferior para acceder a la base de datos en Spring.

Para este caso utilizaremos un proyecto específico creado con el IDE STS para el manejo de JDBC directo sobre JPA.

Lo primero es configurar el proyecto para lo cual se debe considerar los siguientes ficheros.

- POM.xml
- Application.properties

Pom.xml

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

<!-- in-memory database -->
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>

<!-- MySQL JDBC driver
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
-->

<!-- PostgreSQL
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.2.5</version>
</dependency>
-->
```

Se deben incluir las dependencias de la BD a la que nos conectaremos y el `spring-boot-starter-jdbc`, para el ejemplo será H2

En el `application.properties`, se deben incluir las configuraciones de Hikari y la BD elegida. Ahora analizaremos el repositorio que utilizan `JdbcTemplate`

```
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ParameterizedPreparedStatementSetter;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.support.AbstractJdbcCreatingPreparedStatementCallback;
import org.springframework.jdbc.support.LobCreator;
import org.springframework.jdbc.support.LobHandler;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import cl.aiep.Book;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.math.BigDecimal;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Optional;

@Repository
public class JdbcBookRepository implements BookRepository {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Autowired
    LobHandler lobHandler;

    @Override
    public int count() {
        return jdbcTemplate
            .queryForObject("select count(*) from books", Integer.class);
    }
}
```

Método count(), método que se encarga de contar la cantidad de Libros, utiliza jdbcTemplate y un SQL nativo para H2.

Creando un DAO que utiliza JdbcTemplate

Como vimos en el ejemplo anterior, una vez configurada la BD en nuestro proyecto podemos generar implementar el patrón DAO.

¿Qué es el patrón DAO?

El patrón Data Access Object (DAO) propone separar por completo la lógica de negocio de la lógica para acceder a los datos, de esta forma, el DAO proporcionará los métodos necesarios para insertar, actualizar, borrar y consultar la información; por otra parte, la capa de negocio solo se preocupa por lógica de negocio y utiliza el DAO para interactuar con la fuente de datos.

El patrón DAO se compone de 3 partes.

- Una Interfaz, esta se puede invocar desde la capa backend.
- Una Clase Implements, clase que implementa los métodos de la interfaz, en ella se incluye la lógica de negocio orientada a datos.
- Repositorios de Información, son todas las clases Pojos, asociadas a las tablas del modelo de datos.

Interfaz.

```
public interface DAOBookRepository {  
  
    int count();  
  
    int save(Book book);  
  
    int update(Book book);  
  
    int deleteById(Long id);  
  
    List<Book> findAll();  
  
    List<Book> findByNameAndPrice(String name, BigDecimal price);  
  
    Optional<Book> findById(Long id);  
  
    String findNameById(Long id);  
  
    int[] batchInsert(List<Book> books);  
    int[][] batchInsert(List<Book> books, int batchSize);  
    int[] batchUpdate(List<Book> books);  
    int[][] batchUpdate(List<Book> books, int batchSize);  
  
    void saveImage(Long bookId, File image);  
  
    List<Map<String, InputStream>> findImageByBookId(Long bookId);  
  
}
```

La interfaz es parte del patrón Facade, este patrón permite la separación de la lógica asociada a cada capa.

Implementador.

```
public class DAOBookRepositoryImpl implements DAOBookRepository {  
  
    @Autowired  
    private BookRepository bookRepository;  
  
    @Autowired  
    LobHandler lobHandler;  
  
    @Override  
    public int count() {  
        return bookRepository.count();  
    }  
  
    @Override  
    public int save(Book book) {  
        return bookRepository.save(book);  
    }  
  
    ...  
}
```

Clase que tiene lógica de negocio orientada a Orígenes de datos, no necesariamente las entidades consultadas deben pertenecer a una base de datos, puede ser un fileSystem o un Servidor FTP.

Repositorio

```
1  
2 @Repository  
3 public class BookRepository {  
4  
5     @Autowired  
6     private JdbcTemplate jdbcTemplate;  
7  
8     @Autowired  
9     LobHandler lobHandler;  
10  
11  
12     public int count() {  
13         return jdbcTemplate  
14             .queryForObject("select count(*) from books", Integer.class);  
15     }  
16  
17     public int save(Book book) {  
18         return jdbcTemplate.update(  
19             "insert into books (name, price) values(?,?)",  
20             book.getName(), book.getPrice());  
21     }  
22 }
```

Clase que contiene las query's asociadas a una entidad, en este caso las query's son generadas con JdbcTemplate librería que es parte de Spring Framework.

Realizando queries que reciben parámetros

Como vimos en el ejemplo anterior, las query's generadas con JdbcTemplate permiten consultar directamente las tablas asociadas a una Base de datos, además nos permite pasar parámetros de consulta como en el siguiente ejemplo.

```
// jdbcTemplate.queryForObject, populates a single object  
  
public Optional<Book> findById(Long id) {  
    return jdbcTemplate.queryForObject(  
        "select * from books where id = ?",  
        new Object[]{id},  
        (rs, rowNum) ->  
            Optional.of(new Book(  
                rs.getLong("id"),  
                rs.getString("name"),  
                rs.getBigDecimal("price")  
            ))  
    );  
}
```

Como se puede ver, en la query generada se agrega un carácter de tipo "?" para indicar que se debe incluir un valor asociado, este carácter permite evitar un SqlInjection, recibiendo como parámetro anexo el valor del dicho "?".

Finalmente entrega un Optional que contendrá el book asociado al ID

Sql Injection ó Inyección SQL

Es una vulnerabilidad que permite al atacante enviar o “inyectar” instrucciones SQL de forma maliciosa y malintencionada dentro del código SQL programado para la manipulación de bases de datos, de esta forma todos los datos almacenados estarían en peligro. La finalidad de este ataque es poder modificar del comportamiento de nuestras consultas a través de parámetros no deseados, pudiendo así falsificar identidades, obtener y divulgar información de la base de datos (contraseñas, correos, información relevante, entre otros), borrar la base de datos, cambiar el nombre a las tablas, anular transacciones, el atacante puede convertirse en administrador de la misma.

Esto ocurre normalmente a la mala filtración de las variables en un programa que tiene o crea SQL, generalmente cuando solicitas a un usuario entradas de cualquier tipo y no se encuentran validadas, como por ejemplo su nombre y contraseña, pero a cambio de esta información el atacante envía una sentencia **SQL** invasora que se ejecutará en la base de datos.

Ejemplo de ataque SQL Injection

```
"SELECT * FROM usuarios WHERE id_usuario = "+ variable_idUser ;
```

En este caso el atacante puede enviar como valor en variable un sql que se sume por ejemplo si el valor de **variable_idUser** es **"10 or 1=1"** responderá igualmente con datos. En cambio con una herramienta de análisis que se indique que el valor a recibir por la query es Numérico, arrojará un error en runtime al ejecutar la query.

Mapeando los resultados de una consulta a objetos

El patrón DAO indica como buena práctica el volcar la data desde el origen a DTO's de Lógica de negocio, los DTO's permiten abstraer la lógica de negocio de la información contenida en bd, en un DTO se pueden contener varias entidades de Base de Datos o fuentes de Información, o lo que se necesite desde las capas mas arriba.

En este caso utilizaremos las entidades Customer y Books y las uniremos en un DTO con el listado de ambas.

```
▼ > cl.aiep.dto
  > DTOBook.java
  > DTOCustomer.java
  > DTOCustomersBooks.java
```

DtoBooks, DTOCustomer, corresponden a una conversión desde la entidad a un objeto mas cercano al negocio.

DTOCustomerBooks, corresponde al listado de ambos.
Dicha conversión se realiza en la Clase DAO,

```
@Override
public DTOCustomerBooks getCustBooks() {
    DTOCustomerBooks dtoCustBooks= new DTOCustomerBooks();
    List<DTOCustomer> listaClientes= new ArrayList<DTOCustomer>();
    List<DTOBook> listaLibros= new ArrayList<DTOBook>();

    for(Book libro:findAll()) {
        listaLibros.add(new DTOBook(libro.getId(), libro.getName(), libro.getPrice()));
    }
    for(Customer cust:findAllCustomer()) {
        listaClientes.add(new DTOCustomer(cust.getId(), cust.getName(), cust.getAge(),cust.getCreatedDate()));
    }

    dtoCustBooks.setListaClientes(listaClientes);
    dtoCustBooks.setListaLibros(listaLibros);

    return dtoCustBooks;
}
```

Se podrían cambiar o agregar los valores que vienen desde la entidad.
Luego se pasan los listados a una clase que contiene ambos listados.

Invocando un DAO desde un servicio

Como se explica anteriormente, un `@Service` permite generar una capa de abstracción entre las capas de controlador y backend, para este caso utilizaremos la siguiente estructura Controller(servicioRest) -> Service(CapaAbstracción -> DAO(Acceso a Datos).
Controller

```
@RestController
public class BookController {

    @Autowired // Test NamedParameterJdbcTemplate
    private BookService bookService;

    // Find
    @GetMapping("/books")
    List<Book> findAll() {
        return bookService.findAll();
    }

    // Find
    @GetMapping("/all")
    DTOCustomerBooks findAllBookCustomer() {
        return bookService.findCustomerBooks();
    }
}
```

En este caso se genera un servicio tipo GET que obtiene todos los libros y clientes el endpoint asociado es <http://localhost:8080/all> además se genera la inyección del service BookService
Service.

```
@Service
public class BookService {

    @Autowired
    private DAOBookRepository bookRepository;

    public DTOCustomerBooks findCustomerBooks() {
        return bookRepository.getCustBooks();
    }
}
```

Se realiza la Inyección del DAO y se invoca al método que retorna el DTO con el listado de ambas entidades.

DAO

```
public class DAOBookRepositoryImpl implements DAOBookRepository {  
    @Autowired  
    private BookRepository bookRepository;  
    @Autowired  
    private CustomerRepository customerRepository;  
    @Autowired  
    LobHandler lobHandler;  
    @Override  
    public DTOCustomersBooks getCustBooks() {  
        DTOCustomersBooks dtoCustBooks= new DTOCustomersBooks();  
        List<DTOCustomer> listaClientes= new ArrayList<DTOCustomer>();  
        List<DTOBook> listaLibros= new ArrayList<DTOBook>();  
        for(Book libro:findAll()) {  
            listaLibros.add(new DTOBook(libro.getId(), libro.getName(), libro.getPrice()));  
        }  
        for(Customer cust:findAllCustomer()) {  
            listaClientes.add(new DTOCustomer(cust.getID(), cust.getName(), cust.getAge(),cust.getCreateDate()));  
        }  
        dtoCustBooks.setlistaClientes(listaClientes);  
        dtoCustBooks.setlistaLibros(listaLibros);  
        return dtoCustBooks;  
    }  
}
```

Como se explicó anteriormente el DAO se encarga de generar la conversión.

Acceso a datos mediante JPA

La persistencia de datos es un medio mediante el cual una aplicación puede recuperar información desde un sistema de almacenamiento no volátil y hacer que esta persista. La persistencia de datos es vital en las aplicaciones empresariales debido al acceso necesario a las bases de datos relacionales. Las aplicaciones desarrolladas para este entorno deben gestionar por su cuenta la persistencia o utilizar soluciones de terceros para manejar las actualizaciones y recuperaciones de las bases de datos con persistencia. JPA (Java™ Persistence API) proporciona un mecanismo para gestionar la persistencia y la correlación relacional de objetos y funciona desde las especificaciones EJB 3.0.

La especificación JPA define la correlación relacional de objetos internamente, en lugar de basarse en implementaciones de correlación específicas del proveedor. JPA se basa en el modelo de programación Java que se aplica a los entornos Java EE (Java Enterprise Edition) pero JPA puede funcionar en un entorno Java SE para probar las funciones de las aplicaciones.

JPA representa una simplificación del modelo de programación de persistencia. La especificación JPA define explícitamente la correlación relacional de objetos, en lugar de basarse en implementaciones de correlación específicas del proveedor. JPA crea un estándar para la importante tarea de la correlación relacional de objetos mediante la utilización de anotaciones o XML para correlacionar objetos con una o más tablas de una base de datos. Para simplificar aún más el modelo de programación de persistencia

La API de persistencia de Java JPA

Una de las principales características de JPA es que nos entrega APIS que nos evitan el escribir código SQL para acceder a la Base de Datos

- La API EntityManager puede actualizar, recuperar, eliminar o aplicar la persistencia de objetos de una base de datos.
- La API EntityManager y los metadatos de correlación relacional de objetos manejan la mayor parte de las operaciones de base de datos sin que sea necesario escribir código JDBC o SQL para mantener la persistencia.
- JPA proporciona un lenguaje de consulta, que amplía el lenguaje de consulta EJB independiente, conocido también como JPQL, el cual puede utilizar para recuperar objetos sin grabar consultas SQL específicas en la base de datos con la que está trabajando.

JPA está diseñado para funcionar dentro y fuera de un contenedor Java Enterprise Edition (Java EE). Cuando se ejecuta JPA dentro de un contenedor, las aplicaciones pueden utilizar el contenedor para gestionar el contexto de persistencia. Si no hay ningún contenedor para gestionar JPA, la aplicación debe manejar ella misma la gestión del contexto de persistencia. Las aplicaciones diseñadas para la persistencia gestionada por contenedor no requieren tanta implementación de código para manejar la persistencia, pero estas aplicaciones no se pueden utilizar fuera de un contenedor. Las aplicaciones que gestionan su propia persistencia pueden funcionar en un entorno de contenedor o en un entorno Java SE.

Clase de Entidad en JPA

La clase Entidad es una clase Java simple que representa una fila en una tabla de base de datos con su formato más sencillo. Los objetos de entidades pueden ser clases concretas o clases abstractas. Mantienen estados mediante la utilización de propiedades o campos, este tipo de clases ya la hemos visto como ejemplos en el documento.

```
@Entity
public class Book {

    * @Id
    * @GeneratedValue
    * private Long id;
    * private String name;
    * private String author;
    * private BigDecimal price;

    * // avoid this "No default constructor for entity"
    * public Book() {
    * }

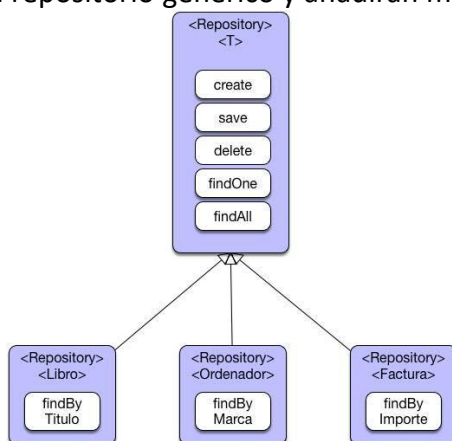
    * public Book(Long id, String name, String author, BigDecimal price) {
    *     this.id = id;
    *     this.name = name;
    *     this.author = author;
    *     this.price = price;
    * }
}
```

El Entity manager en JPA

El entity manager en JPA es un Gestor de recursos que mantiene la colección activa de objetos de entidad que está utilizando la aplicación. EntityManager maneja la interacción y metadatos de bases de datos para las correlaciones relacionales de objetos. Una instancia de EntityManager representa un contexto de persistencia. Una aplicación en un contenedor puede obtener el EntityManager mediante inyección en la aplicación o buscándolo en el espacio de nombres del componente Java.

Clases Repositorio

El concepto de Java Generic Repository es muy habitual cuando trabajamos con tecnologías de persistencia. El concepto de Repository como clase que se encarga de gestionar todas las operaciones de persistencia contra una tabla de la base de datos. La mayor parte de los lenguajes de programación soportan el uso de clases genéricas. Estas permiten avanzar en el uso del patrón Repository y generar repositorios genéricos que reducen de forma significativa el código que tenemos que construir. Todos nuestros repositorios extenderán del repositorio genérico y añadirán más operaciones.



Recuperar, actualizar, eliminar un Objeto JPA

Como se explica anteriormente, un Repository nos permite generar todas las acciones CRUD asociadas a una entidad.

Recuperar.

```
// Find
@GetMapping("/books")
List<Book> findAll() {
    return repository.findAll();
}
```

En el ejemplo se puede ver que no es necesario generar un SQL para obtener todas las entidades de desde una BD

Actualizar.

```
// update or insert
@PostMapping("/books/{id}")
Book saveOrUpdate(@RequestBody Book newBook, @PathVariable Long id) {
    return repository.findById(id)
        .map(x -> {
            x.setName(newBook.getName());
            x.setAuthor(newBook.getAuthor());
            x.setPrice(newBook.getPrice());
            return repository.save(x);
        })
        .orElseGet(() -> {
            newBook.setId(id);
            return repository.save(newBook);
        });
}
```

En el ejemplo se puede ver como se inserta o actualiza un registro.
Eliminar.

```
@DeleteMapping("/books/{id}")
void deleteBook(@PathVariable Long id) {
    repository.deleteById(id);
}
```

En este caso se elimina el registro por el Id de Libro.

Asociaciones uno a uno, uno a muchos

@OneToOne Relation

En una relación uno-a-uno, un elemento puede vincularse al único otro elemento. Significa que cada fila de una entidad se refiere a una y sólo una fila de otra entidad.

@ManyToOne

En una relación uno-a-muchos, un elemento puede vincularse a muchos otros elementos. Significa que cada fila de una entidad se refiere a una y sólo una fila de otra entidad.

Nos permite considerar el ejemplo anterior. Cliente y Libro de manera unidireccional, la relación es relación uno-a-uno. Este tipo de relación fue explicado en el Modulo de Base de datos del curso.

En el caso de JPA se representa de la siguiente forma.

@OneToMany

```
@Entity
@Table(name = "book_category")
public class BookCategory {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;

    @OneToMany(mappedBy = "bookCategory", cascade = CascadeType.ALL)
    private Set<Book> books;

    public BookCategory(){
    }
}
```

En donde un libro pertenece a una categoría

@ManyToOne

```

1
2
3 @Entity
4 public class Book{
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     private int id;
8
9     private String name;
10
11     @ManyToOne
12     @JoinColumn(name = "book_category_id")
13     private BookCategory bookCategory;
14
15     public Book() {
16
17     }
18 }

```

Acá se representa que una categoría puede estar en muchos libros

Invocar un repositorio desde un servicio

Como se explica anteriormente, un @Service permite generar una capa de abstracción entre las capas de controlador y backend, para este caso utilizaremos la siguiente estructura Controller(servicioRest) --> Service(CapaAbstracción --> Repository

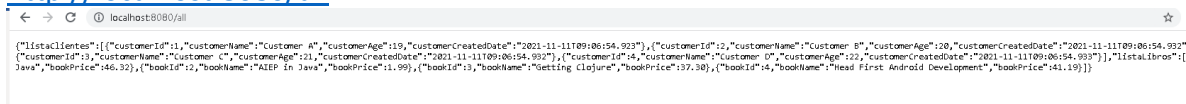
```

1
2 @Service
3 public class BookService {
4
5     @Autowired
6     private BookRepository repository;
7
8     public List<Book> findAll() {
9         return repository.findAll();
10    }
11 }

```

Se solicita desde el Service el repositorio para la búsqueda y al acceder al servicio

<http://localhost:8080/all>



```

{"listClientes":[{"customerId":1,"customerName":"Customer A","customerAge":19,"customerCreatedDate":"2021-11-11T09:06:54.933Z"},{"customerId":2,"customerName":"Customer B","customerAge":20,"customerCreatedDate":"2021-11-11T09:06:54.933Z"},{"customerId":3,"customerName":"Customer C","customerAge":21,"customerCreatedDate":"2021-11-11T09:06:54.933Z"},{"customerId":4,"customerName":"Customer D","customerAge":22,"customerCreatedDate":"2021-11-11T09:06:54.933Z"},{"customerId":5,"customerName":"Customer E","customerAge":23,"customerCreatedDate":"2021-11-11T09:06:54.933Z"}], "listLibros":[{"bookId":1,"bookName":"Java in a Nutshell","bookPrice":46.32}, {"bookId":2,"bookName":"AIEP in Java","bookPrice":11.99}, {"bookId":3,"bookName":"Getting Clojure","bookPrice":37.30}, {"bookId":4,"bookName":"Head First Android Development","bookPrice":41.19}, {"bookId":5,"bookName":"Spring in Action","bookPrice":39.99}]}

```

Entrega la información en formato JSON.

Manejo de la transaccionalidad en los servicios

¿Qué es la transaccionalidad y porque es importante?

Una Transacción es un conjunto de operaciones ejecutadas como una unidad cuyo resultado final es una acción que manipula la base de datos. Un fallo en la ejecución de

una de las operaciones provoca una reversión de la transacción, recuperando el estado inicial de la base de datos.

Spring Transaction Management es una de las características más utilizadas e importantes del marco Spring. La gestión de transacciones es una tarea trivial en cualquier aplicación empresarial. Ya hemos aprendido a utilizar la API de JDBC para la gestión de transacciones. Spring proporciona un amplio soporte para la gestión de transacciones y ayuda a los desarrolladores a centrarse más en la lógica empresarial en lugar de preocuparse por la integridad de los datos en caso de fallas en el sistema.

Configurando la transaccionalidad

Un aspecto muy interesante, es la propagación de la transaccionalidad a través de nuestra lógica de negocio. Por ejemplo, necesitamos un método transaccional, pero contemplamos la posibilidad de que exista una transacción superior, aun así queremos que esta transacción sea atómica y se ejecute de manera anidada. Para manejar este tipo de lógicas tan concretas tenemos la configuración de la propagación.

Opciones de Propagación

REQUIRED: Da soporte a la transacción actual o crea una nueva sino existe.

SUPPORTS: Da soporte a la transacción actual, sino existe, ejecuta el método sin transacción.

MANDATORY: Da soporte a la transacción actual, sino existe, lanza una excepción.

REQUIRES_NEW: Crea una nueva transacción, si existe una, la suspende.

NOT_SUPPORTED: Ejecuta el método sin transacción, si existe, la suspende.

NEVER: Ejecuta el método sin transacción, si existe, lanza una excepción.

NESTED: Ejecuta dentro una transacción si existe una.

Creando un servicio transaccional

Como lo hemos visto en ejemplos anteriores para configurar el proyecto con transaccionalidad, es necesario modificar las clases `@Service` agregando la anotación `@Transactional` y seleccionando la forma de propagar si es necesario.

```
1 import java.util.List;
2
3 @Service
4 @Transactional
5 public class BookService {
6
7     public DTOCustomersBooks findCustomerBooks() {
8         return bookRepository.getCustBooks();
9     }
10
11
12
13
14 @Autowired
15 private DAOBookRepository bookRepository;
16
17 }
```

Control de acceso mediante spring security

Spring Security es un framework de apoyo al marco de trabajo Spring, que dota al mismo de una serie de servicios de seguridad aplicables para sistemas basados en la arquitectura basados en J2EE, enfocado particularmente sobre proyectos construidos usando SpringFramework. De esta dependencia, se minimiza la curva de aprendizaje si ya es conocido Spring.

Los procesos de seguridad están destinados principalmente, a comprobar la identidad del usuario mediante la autenticación y los permisos asociados al mismo mediante la autorización. La autorización es dependiente de la autenticación ya que se produce posteriormente a su proceso.

Por regla general muchos de estos modelos de autenticación son proporcionados por terceros o son desarrollados por estándares importantes como el IETF adicionalmente,

Spring Security proporciona su propio conjunto de características de autenticación. Específicamente, Spring Security actualmente soporta integración de autenticación con todas las siguientes tecnologías:

- HTTP BASIC authentication headers (an IEFT RFC-based standard).
- HTTP Digest authentication headers (an IEFT RFC-based standard).
- HTTP X.509 client certificate exchange (an IEFT RFC-based standard).
- LDAP (un enfoque muy común para necesidades de autenticación multiplataforma, específicamente en entornos extensos).
- Form-based authentication (necesario para interfaces de usuario simples).
- OpenID authentication.
- Computer Associates Siteminder.
- JA-SIG Central Authentication Service.
- Transparent authentication context propagation for Remote Method Invocation (RMI) and HttpInvoker.
- Automatic "remember-me" authentication.
- Anonymous authentication.
- Run-as authentication.
- Java Authentication and Authorization Service (JAAS)
- Container integration with JBoss, Jetty, Resin and Tomcat (tambien podemos usar autenticación gestionada por el contenedor)
- Java Open Source Single Sign On (JOSSO)*
- OpenNMS Network Management Platform*
- AppFuse*
- AndroMDA*
- Mule ESB*
- Direct Web Request (DWR)*
- Grails*
- Tapestry*
- JTrac*
- Jasypt*
- Roller*
- Elastic Plath*
- Atlassian Crowd*
- Nuestros propios sistemas de autenticación.

(*Indica proporcionado por un tercero)

Spring Security tiene un alto grado de aceptación entre la comunidad de desarrolladores por su gran flexibilidad sobre los modelos de autenticación. De esta manera, permite una rápida integración de las soluciones que presentan ante los requerimientos de los clientes

potenciales, sin implicar una migración de los sistemas a un determinado entorno. Además, es una plataforma abierta y en constante evolución, lo que permite ir añadiendo nuevos mecanismos de autenticación, o directamente programar los propios de manera poco compleja

En ocasiones un proceso simple de autenticación no es eficiente. Hay que controlar como se relaciona el usuario con la aplicación. Puede resultar interesante como llegan las solicitudes, si son automatizadas o por medio de una persona, asegurar que se realizan mediante el protocolo Http. Para completar estos objetivos, Spring Security proporciona "canales de seguridad" automáticos integrándose con JCAPTCHA para detección de usuarios humanos.

SpringSecurity también facilita el proceso de la autorización. Para ello, aporta tres características esenciales: Autorización en base a solicitudes web, autorización en base a llamadas, autorización en base al acceso a instancias.

Incorporando el módulo spring security al proyecto

Para incorporar el módulo de seguridad al proyecto es necesario modificar el POM.xml

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

Con esto ya podemos configurar la seguridad de los Servicios.

Configuración básica de Spring Security

Existen dos formas de configurar Spring Security: una 100% manual y otra algo más automática. La forma manual consiste en definir todos los filtros necesarios para su correcto funcionamiento.

La otra forma, que es la que mostraremos, consiste en usar el elemento http. Este elemento configurará por nosotros todos los filtros obligatorios, beans y la cadena de seguridad. Si además se le pone el atributo auto-config a true, configurará todos los filtros restantes necesarios para el funcionamiento.

El primero de los beans que se crea es el springSecurityFilterChain, que es el encargado de gestionar los diferentes filtros de la cadena de seguridad.

Obviamente, se pueden personalizar los filtros que se configuran automáticamente e incluso añadir los nuestros propios. Esto se realiza anidando un nuevo elemento en el http. Con el custom-filter, se pueden añadir nuevos filtros, así como posicionarlos donde sea necesario en la cadena de seguridad. Es posible que alguno de los nuevos filtros entren en

conflicto con algunos de los existentes. Esto ocurrirá si estamos sobrescribiendo un filtro existente, con lo cual, habrá que poner el atributo auto-config a false.

Creando un formulario de Login

Para la creación de un formulario Login primero crearemos una clase filtro de seguridad.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(
    prePostEnabled = true,
    securedEnabled = true,
    jsr250Enabled = true)

public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

    // Create 2 users for demo
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user").password("{noop}password").roles("USER")
            .and()
            .withUser("admin").password("{noop}password").roles("USER", "ADMIN");
    }
}
```

utilizaremos, lo aprendido en el módulo anterior de Thymeleaf, agregando las configuraciones asociadas quedando de la siguiente forma

```
1 <!DOCTYPE HTML>
2 <html lang="en" xmlns:th="http://www.thymeleaf.org">
3 <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
6
7     <title>Spring Boot Thymeleaf Hello World Example</title>
8
9     <link rel="stylesheet" th:href="@{webjars/bootstrap/4.2.1/css/bootstrap.min.css}" />
10    <link rel="stylesheet" th:href="@{/css/main.css}" />
11
12 </head>
13
14 <body>
15
16 <nav class="navbar navbar-expand-md navbar-dark bg-dark fixed-top">
17     <a class="navbar-brand" href="#">AIEP</a>
18     <span sec:authentication="name"></span> | <a style="color: white;" onclick="document.forms['logoutForm'].submit()">Logout</a> | </h3>
19 </nav>
20
21 <main role="main" class="container">
22     <form id="logoutForm" method="POST" th:action="@{/logout}">
23         <input type="hidden" name="{_csrf.parameterName}" value="{_csrf.token}" />
24     </form>
25
26     <div class="starter-template">
27         <h1>Spring Boot Web Thymeleaf Example</h1>
28         <h2>
29             <span th:text="Hello, ' + ${message}"></span>
30         </h2>
31     </div>
32
33
34     <ol>
35         <li th:each="task : ${tasks}" th:text="${task}"></li>
36     </ol>
37
38 </main>
39 <!-- /.container -->
40
41 <script type="text/javascript" th:src="@{webjars/bootstrap/4.2.1/js/bootstrap.min.js}"></script>
42 </body>
43 </html>
```

En este fichero Welcome.html podemos ver el código asociado al mensaje de bienvenida y el código de Logout

El fichero Login.html se ve de la siguiente forma.

```

<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
<meta name="description" content="">
<meta name="author" content="">
<title>Spring Security Example</title>
<link href="/webjars/bootstrap/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div class="container">
<h2 class="form-signin-heading">Welcome to AIEP, please login</h2>
<div th:if="${param.error}" class="alert alert-danger">
Invalid username and password.
</div>
<div th:if="${param.logout}" class="alert alert-success">
You have been logged out.
</div>
<form class="form-signin" method="POST" th:action="@{/login}">
<p>
<label for="username" class="sr-only">Username</label>
<input type="text" id="username" name="username" class="form-control"
placeholder="Username" required autofocus>
</p>
<p>
<label for="password" class="sr-only">Password</label>
<input type="password" id="password" name="password" class="form-control"
placeholder="Password" required>
</p>
<button class="btn btn-lg btn-primary btn-block" type="submit">Login</button>
</form>
</div>
</body>
</html>

```

En él podemos validar el Html asociado a el ingreso de usuario y password

Realizando un Login y Logout de una aplicación

Para el caso de un Login y Logout debemos modificar la clase filtro de Spring Security

```

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

    http
        //HTTP Basic authentication
        .httpBasic()
        .and()
        .authorizeRequests()
        .antMatchers(HttpMethod.GET, "/books/**").hasRole("USER")
        .antMatchers(HttpMethod.POST, "/books").hasRole("ADMIN")
        .antMatchers(HttpMethod.PUT, "/books/**").hasRole("ADMIN")
        .antMatchers(HttpMethod.PATCH, "/books/**").hasRole("ADMIN")
        .antMatchers(HttpMethod.DELETE, "/books/**").hasRole("ADMIN")
        .and()
        .csrf().disable()
        .formLogin().loginPage("/login")
        .permitAll()
        .and()
        .logout()
        .permitAll();
}

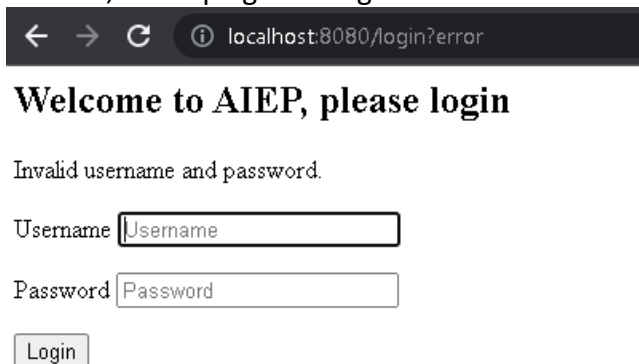
```

En la imagen podemos ver como se configura el Login y su url , además la configuración del logout

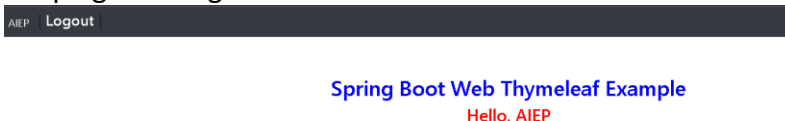
Una vez realizado el cambio, procedemos a probar el funcionamiento de la app construida con STS



Si ingresamos mal el usuario, se desplegará el siguiente error.



Caso contrario desplegara lo siguiente



Manejo de Roles

Para el manejo de roles deberemos incluir algunos atributos a la clase
SpringSecurityConfig

```

@Configuration
public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

    // Create 2 users for demo
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

        auth.inMemoryAuthentication()
            .withUser("user").password("{noop}password").roles("USER")
            .and()
            .withUser("admin").password("{noop}password").roles("USER", "ADMIN");

    }
}

```

Para este ejemplo crearemos 2 usuarios en memoria.
Y además se asignarán roles asociados a cada tipo de servicio.

```

// Secure the endpoints with HTTP Basic authentication
@Override
protected void configure(HttpSecurity http) throws Exception {

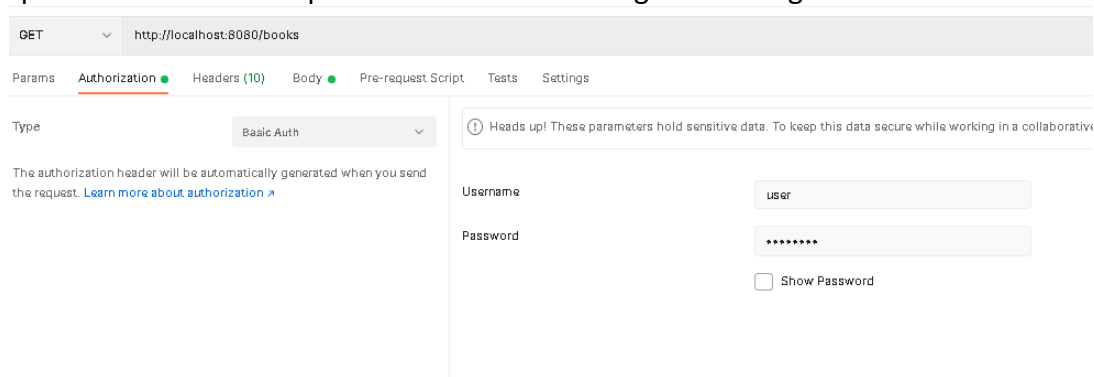
    http
        // HTTP Basic authentication
        .httpBasic()
        .and()
        .authorizeRequests()
        .antMatchers(HttpMethod.GET, "/books/**").hasRole("USER")
        .antMatchers(HttpMethod.POST, "/books/**").hasRole("ADMIN")
        .antMatchers(HttpMethod.PUT, "/books/**").hasRole("ADMIN")
        .antMatchers(HttpMethod.PATCH, "/books/**").hasRole("ADMIN")
        .antMatchers(HttpMethod.DELETE, "/books/**").hasRole("ADMIN")
        .and()
        .csrf().disable()
        .formLogin().disable();
}

```

Como se ve en la imagen, en el caso de los métodos que obtienen información cualquier usuario puede consultarlos, y los que administran solo el Admin.

Añadiendo seguridad a los elementos de la capa vista

De acuerdo con lo revisado anteriormente los métodos ya se encuentran securizados. Para probarlos utilizaremos una herramienta libre llamada PostMan, en el primer método GET puede ser consultado por un usuario como la siguiente imagen



The screenshot shows the Postman interface for a GET request to `http://localhost:8080/books`. The 'Authorization' tab is selected, showing 'Basic Auth' configuration. The 'Username' field contains 'user' and the 'Password' field contains 'password'. A warning message states: 'Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, Postman will automatically mask sensitive data in the request and response bodies.' Below the fields, there is a checkbox for 'Show Password' which is currently unchecked.

Desplegando la información necesitada.

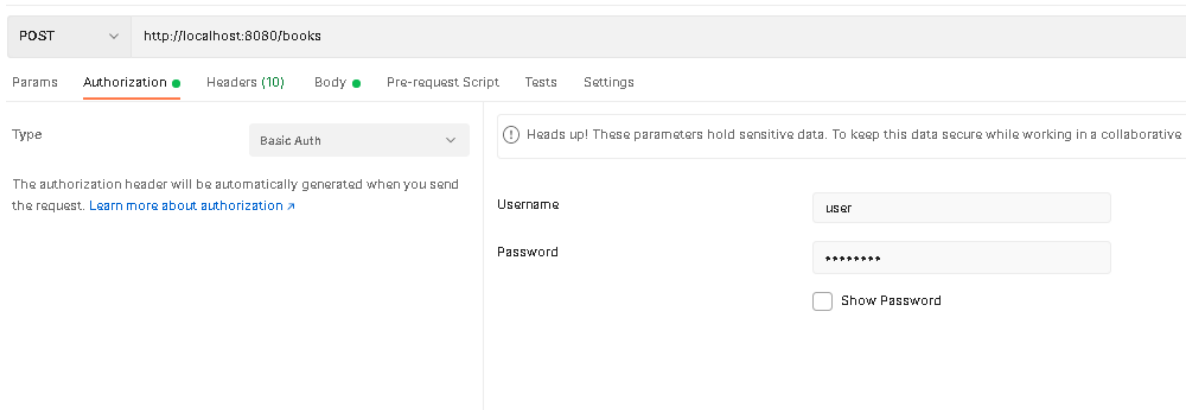
```

1  {
2    {
3      "id": 1,
4      "name": "A Guide to the Bodhisattva Way of Life",
5      "author": "Santideva",
6      "price": 15.41
7    },
8    {
9      "id": 2,
10     "name": "The Life-Changing Magic of Tidying Up",
11     "author": "Marie Kondo",
12     "price": 9.69
13   },
14   {
15     "id": 3,
16     "name": "Refactoring: Improving the Design of Existing Code",
17     "author": "Martin Fowler",
18     "price": 47.99
19   },
20   {
21     "id": 4,
22     "name": "The Pragmatic Programmer: The Journey from Rambo to

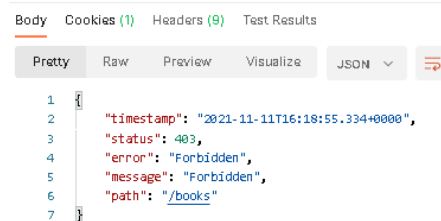
```

En los casos de administración POST(save,update,delete) solo se permite el acceso del administrador, por ejemplo.

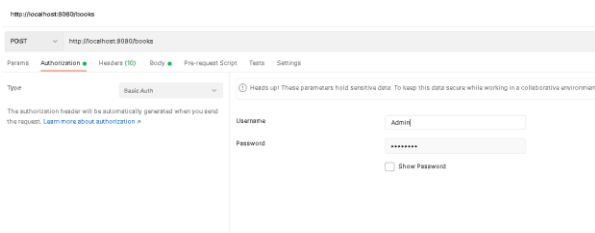
<http://localhost:8080/books>



El servicio rechaza la petición por no contar con los permisos.



En cambio, si se realiza con el usuario admin.



Se ejecuta correctamente



Obteniendo el usuario autenticado en el controlador

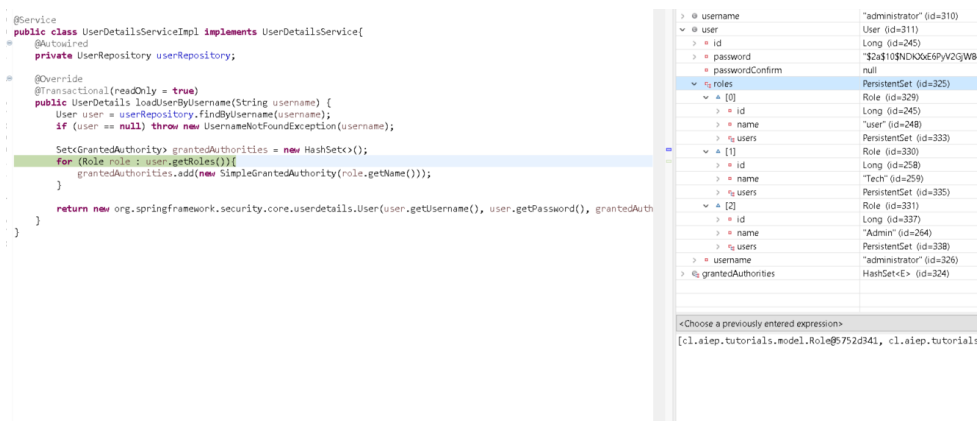
Una vez autenticado el usuario, es factible captura sus roles, para esto construiremos un service que permita obtener los roles desde BD y este sea invocado desde el Controller

```

1  }
2  }
3  }
4  }
5  }
6  }
7  }
8  }
9  }
10 }
11 }
12 }
13 }
14 }
15 }
16 }
17 }
18 }
19 }
20 }
21 }
22 }
23 }
24 }
25 }
26 }
27 }
28 }
29 }
30 }
31 }
32 }
33 }
34 }
35 }
36 }
37 }
38 }
39 }
40 }
41 }
42 }
43 }
44 }
45 }
46 }
47 }
48 }
49 }
50 }
51 }
52 }
53 }
54 }
55 }
56 }
57 }
58 }
59 }
60 }
61 }
62 }
63 }
64 }
65 }
66 }
67 }
68 }
69 }
70 }
71 }
72 }
73 }
74 }
75 }
76 }
77 }
78 }
79 }
80 }
81 }
82 }
83 }
84 }
85 }
86 }
87 }
88 }
89 }
90 }
91 }
92 }
93 }
94 }
95 }
96 }
97 }
98 }
99 }
100 }

```

En el método /login se invoca el service para obtener los roles.



Utilizando el STS se puede entrar en modo Debug que permite ir línea a línea revisando los valores asignados, en este caso se recorren los roles del usuario.

Agregando seguridad en los controladores mediante anotaciones

Para agregar seguridad en los controladores, utilizaremos la anotación `@PreAuthorize`, anotación que permite evitar los accesos no validados a los métodos del controlador. Teniendo lo siguiente.

```
@GetMapping("/")
public String defaultAfterLogin(Model model) {

    String retorno="welcome";
    model.addAttribute("message", "AIEP");
    return retorno;
}
```

En este caso el controlador no tiene ningún control sobre la autenticación del usuario. Si se accede al sitio <http://localhost:8080/> el browser desplegará lo siguiente.



Spring Boot Web Thymeleaf Example

Hello, AIEP

Generando así una fuga de seguridad ya que el usuario no está autenticado. Para solucionar lo anterior es que agregamos la anotación `@PreAuthorize`

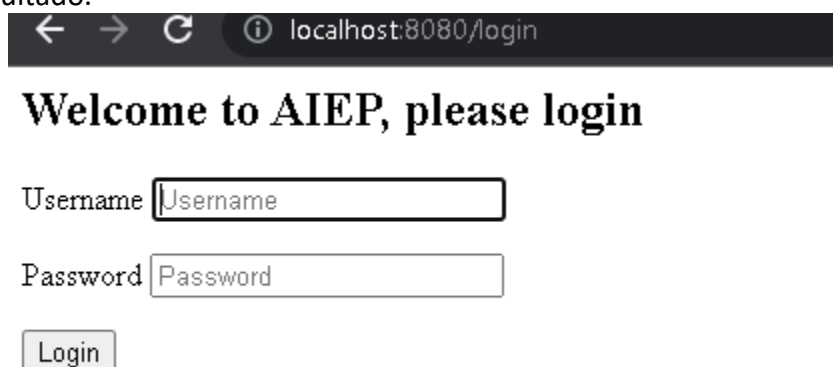
```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(
    prePostEnabled = true,
    securedEnabled = true,
    jsr250Enabled = true)

public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {
```

```
@GetMapping("/")
@PreAuthorize("isAuthenticated()")
public String defaultAfterLogin(Model model) {

    String retorno="welcome";
    model.addAttribute("message", "AIEP");
    return retorno;
}
```

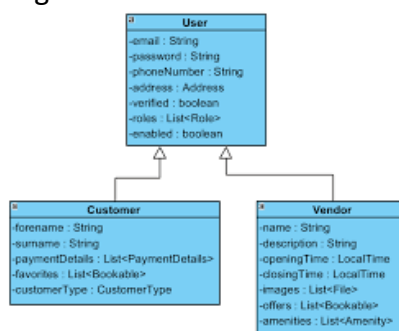
Con esto le indicamos que solo puede acceder si el usuario está autenticado, generando el siguiente resultado.



Se redirecciona automáticamente al login.

Autenticación contra una bd

La autenticación contra una BD indica que los usuarios sus password y atributos serán almacenadas en Tablas para luego ser consultadas.



Este método no es recomendable desde el punto de vista seguridad y de reutilización ya que a pesar de que es factible de realizar, en la actualidad existen herramientas para la administración de cuentas de usuario y sus contraseñas como los LDAP, ActiveDirectory, etc. Que separan los usuarios y sus cuentas de un negocio específico o modelo.

Autenticación utilizando JPA.

Complementando lo previamente indicado en el documento la autenticación contra Database se puede ejecutar con jdbcTemplate o Entidades, para el ejemplo a continuación utilizaremos Entidades.

Lo primero que debemos modificar es el archivo POM.xml, en este caso se debe incluir la BD.

```
<dependency>  
  <groupId>com.h2database</groupId>  
  <artifactId>h2</artifactId>  
</dependency>
```

Una vez realizado esto, se deben construir las entidades que soporten el modelo asociado a usuarios. En este caso son las siguientes clases

- 2 entidades, User y Role como se explica en el módulo anterior las entidades son una representación de las tablas en BD
- 2 services usuarioService y RoleService, los services son capas de abstracción para la capa de acceso a datos
- 2 Repositorios UserRepo, RoleRepo, los repositorios son clases que permiten realizar las consultas a Base de Datos
- 1 Controller que invoca al service, Clase en donde se encuentran las Urls de la aplicación y se realizan las validaciones de formulario.

UserService

```
@Service  
public class UserDetailsServiceImpl implements UserDetailsService {  
    @Autowired  
    private UserRepository userRepository;  
  
    @Override  
    @Transactional(readOnly = true)  
    public UserDetails loadUserByUsername(String username) {  
        User user = userRepository.findByUsername(username);  
        if (user == null) throw new UsernameNotFoundException(username);  
  
        Set<GrantedAuthority> grantedAuthorities = new HashSet<>();  
        for (Role role : user.getRoles()) {  
            grantedAuthorities.add(new SimpleGrantedAuthority(role.getName()));  
        }  
  
        return new org.springframework.security.core.userdetails.User(user.getUsername(), user.getPassword(), grantedAuthorities);  
    }  
}
```

En este service se accede a BD para obtener la data asociada al username ingresado en el login.

La interoperabilidad entre los sistemas

La interoperabilidad es un concepto definido en el **Vocabulario de Información y Tecnología ISO/ISO/IEC 2382** como “la capacidad de comunicar, ejecutar programas o transferir datos entre varias unidades funcionales de manera que el usuario no tenga que conocer las características únicas de estas unidades”. (ISO, 2000). Otros autores como Lueders en 2004 lo definieron como “la habilidad de los Sistemas TIC y de los procesos de

negocios que ellas soportan, de **intercambiar datos y posibilitar compartir información y conocimiento**”.

Estas definiciones nos recuerdan aquella vieja idea del ser humano de **poder compartir la información de manera universal**, más allá de la tecnología que se encargue de su almacenamiento, su procesamiento o su distribución. Persiguiendo este objetivo se inventó la imprenta en 1440 y en 1969 se creó la red ARPANET, que pronto se desarrollaría en el Internet que hoy conocemos.

Hoy en día, en plena explosión de datos digitales, **cobran fuerza las tecnologías que facilitan la interoperabilidad de las empresas** y que les permiten aprovecharse de distintos beneficios.

Beneficios de la interoperabilidad

Las organizaciones que aplican tecnologías que facilitan la comunicación y sincronía entre sistemas consiguen los siguientes beneficios:

- **Información cohesionada.** Se identifican sistemas de información que operan de manera aislada y se localiza información redundante para conseguir una comunicación fluida entre los mismos.
- **Mayor adaptabilidad.** Los sistemas que se encargan de capturar la información se conectan entre sí y transfieren los datos que han detectado de manera automática y flexible, adaptándose a los cambios más rápido.
- **Más productividad y control.** Los datos que se capturan se relacionan con producción y se gestionan para que estén disponibles y accesibles por parte de los distintos usuarios de un modo más sencillo.

Una empresa puede aprovechar estas ventajas si aplica la interoperabilidad del sistema en todas las áreas de su negocio

Protocolo de intercambio de datos

Un protocolo es un conjunto de reglas: los **protocolos de red** son estándares y políticas formales, conformados por restricciones, procedimientos y formatos que definen el intercambio de **paquetes** de información para lograr la comunicación entre dos **servidores** o más dispositivos a través de una red.

Los **protocolos de red** incluyen mecanismos para que los dispositivos se identifiquen y establezcan conexiones entre sí, así como reglas de formato que especifican cómo se forman los **paquetes** y los datos en los mensajes enviados y recibidos. Algunos protocolos admiten el reconocimiento de mensajes y la compresión de datos diseñados para una comunicación de red confiable de alto rendimiento.

Tipos de protocolos de red

Los protocolos para la **transmisión de datos** en internet más importantes son TCP (Protocolo de Control de Transmisión) e IP (**Protocolo de Internet**). De manera conjunta (TCP/IP) podemos enlazar los dispositivos que acceden a la red, algunos otros **protocolos de comunicación** asociados a internet son POP, SMTP y HTTP.

Estos los utilizamos prácticamente todos los días, aunque la mayoría de los usuarios no lo sepan ni conozcan su funcionamiento. Estos protocolos permiten la **transmisión de datos** desde nuestros dispositivos para navegar a través de los sitios, enviar correos electrónicos, escuchar música online, etc.

Existen varios tipos de protocolos de red:

- Protocolos de comunicación de red: protocolos de comunicación de **paquetes** básicos como TCP / IP y HTTP.
- Protocolos de seguridad de red: implementan la seguridad en las comunicaciones de red entre **servidores**, incluye HTTPS, SSL y SFTP.
- Protocolos de gestión de red: proporcionan mantenimiento y gobierno de red, incluyen SNMP e ICMP.

Un grupo de protocolos de red que trabajan juntos en los niveles superior e inferior comúnmente se les denomina *familia de protocolos*.

El modelo OSI (Open System Interconnection) organiza conceptualmente a las familias de protocolos de red en **capas de red** específicas. Este Sistema de Interconexión Abierto tiene por objetivo establecer un contexto en el cual basar las arquitecturas de comunicación entre diferentes sistemas.

¿Qué es el Estado Representacional de Transferencia REST?

La transferencia de estado representacional (REST) es un estilo de arquitectura de software que utiliza un subconjunto de HTTP . Se utiliza comúnmente para crear aplicaciones interactivas que utilizan servicios WEB. Un servicio web que sigue estas pautas se llama RESTful. Dicho servicio web debe proporcionar sus recursos WEB en una representación textual y permitir que se lean y modifiquen con un protocolo sin estado y un conjunto predefinido de operaciones. Este enfoque permite la interoperabilidad entre los sistemas informáticos en Internet que prestan estos servicios. REST es una alternativa a, por ejemplo, SOAP como forma de acceder a un servicio web.

Los "recursos web" se definieron por primera vez en la World Wide Web como documentos o archivos identificados por sus URL . Hoy en día, la definición es mucho más genérica y abstracta, e incluye todo lo, entidad o acción que se puede identificar, nombrar, abordar, manejar o realizar de cualquier forma en la Web. En un servicio web RESTful, las solicitudes realizadas al URI de un recurso generan una respuesta con una carga útil formateada en HTML , XML , JSON o algún otro formato. Por ejemplo, la respuesta puede confirmar que se ha cambiado el estado del recurso. La respuesta también puede incluir enlaces de hipertexto a recursos relacionados. El protocolo más común para estas solicitudes y respuestas es HTTP. Proporciona operaciones (métodos HTTP) como GET, POST, PUT y DELETE. Mediante el uso de un protocolo sin estado y operaciones estándar, los sistemas RESTful apuntan a un rendimiento rápido, confiabilidad y la capacidad de crecer reutilizando componentes que se pueden administrar y actualizar sin afectar el sistema en su totalidad, incluso mientras está en ejecución.

El objetivo de REST es aumentar el rendimiento, la escalabilidad, la simplicidad, la modificabilidad, la visibilidad, la portabilidad y la confiabilidad. Esto se logra siguiendo los principios REST, como arquitectura cliente-servidor, ausencia de estado, capacidad de almacenamiento en caché, uso de un sistema en capas, soporte para código bajo demanda y uso de una interfaz uniforme. Estos principios deben seguirse para que el sistema se clasifique como REST.

El término transferencia de estado representacional fue introducido y definido en 2000 por Roy Fielding en su tesis doctoral. La disertación de Fielding explicó los principios REST que se conocían como el "modelo de objeto HTTP" a partir de 1994, y se utilizaron en el diseño de los estándares HTTP 1.1 y Uniform Resource Identifiers (URI).

La notación JSON para el traspaso de información

JSON, cuyo nombre corresponde a las siglas JavaScript Object Notation o Notación de Objetos de JavaScript, es un formato ligero de intercambio de datos, que resulta sencillo de leer y escribir para los programadores y simple de interpretar y generar para las máquinas.

JSON es un formato de texto completamente independiente de lenguaje, pero utiliza convenciones que son ampliamente conocidos por los programadores, entre ellos:

- C

- C++
- C#
- Java
- JavaScript
- Perl
- Python
- Entre otros

Dichas propiedades hacen de JSON un formato de intercambio de datos ideal para usar con API REST o AJAX. A menudo se usa en lugar de XML, debido a su estructura ligera y compacta.

Muchos lenguajes de programación proporcionan métodos para analizar una cadena de texto con este formato en un objeto nativo y viceversa.

Consumiendo un Servicio REST con Spring y Rest Template.

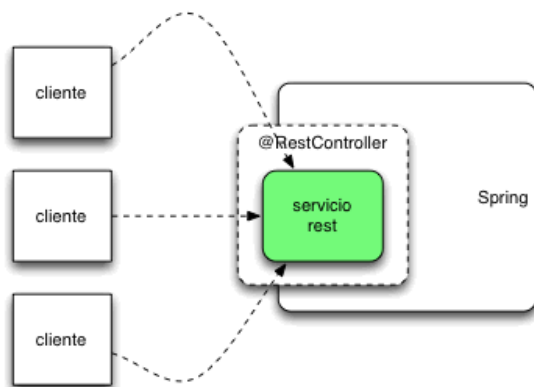
Para consumir un servicio rest construido con el IDE STS utilizaremos una herramienta libre llamada Postman.

Primero que todo se debe configurar el archivo POM.xml

```
<dependencies>

<!-- spring mvc, rest -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Luego crear una clase que use la anotación `@RestController` y automáticamente se publicara como un Spring REST Service.



Ejemplo de clase RestController

```
@RestController
public class BookController {

    @Autowired
    private BookRepository repository;

    // Find
    @GetMapping("/books")
    List<Book> findAll() {
        return repository.findAll();
    }

    // Save
    @PostMapping("/books")
    //return 201 instead of 200
    @ResponseStatus(HttpStatus.CREATED)
    Book newBook(@RequestBody Book newBook) {
        return repository.save(newBook);
    }

    // Find
    @GetMapping("/books/{id}")
    Book findOne(@PathVariable Long id) {
        return repository.findById(id)
            .orElseThrow(() -> new BookNotFoundException(id));
    }

    // Save or update
    @PutMapping("/books/{id}")
    Book saveOrUpdate(@RequestBody Book newBook, @PathVariable Long id) {

        return repository.findById(id)
            .map(x -> {
                x.setName(newBook.getName());
                x.setAuthor(newBook.getAuthor());
                x.setPrice(newBook.getPrice());
                return repository.save(x);
            })
    }
}
```

Principios de diseño de una API REST

Al desarrollar una API RESTful de Java, los diseñadores deben considerar dos elementos clave:

- Patrón de URL
- Qué método HTTP usar

El primer principio importante que enfatizamos es que siempre se debe acceder a los recursos a través de una URL que los identifique de manera única.

Para cualquiera que haya utilizado un navegador web, este es un concepto completamente nuevo. Cuando visitamos una página web o descargamos un archivo PDF basado en la web, dirigimos el navegador a la URL que identifica el recurso. El mismo concepto también se aplica cuando se utilizan los servicios web RESTful Java para acceder a los recursos del lado del servidor. Si el cliente jQuery o Angular necesita manipular recursos, debe haber una URL única que permita que el código JavaScript relevante identifique y ubique el recurso RESTful correspondiente.

Regla de diseño RESTful: la llamada GET no puede cambiar el estado del servidor. Para lidiar con los métodos HTTP, debe seguir importantes reglas de diseño RESTful. Si los diseñadores de la API de RESTful Java violan estas reglas, se extraviarán.

El primer principio es, las llamadas GET nunca pueden cambiar el estado de ningún recurso RESTfull en el servidor.

El segundo principio es, los métodos RESTfull PUT y DELETE deben cumplir con el principio de idempotencia

Aunque no es una regla estricta, los métodos PUT y DELETE mapean aproximadamente los conceptos de guardar y eliminar. Si los diseñadores desean eliminar recursos del servidor, deben usar el método HTTP DELETE. Si necesita crear un nuevo recurso o necesita actualizar un recurso existente, debe usar el método PUT.

Los métodos PUT y DELETE son relativamente simples para guardar y eliminar datos. Pero esta es otra trampa que los diseñadores de API RESTful Java encuentran a menudo. Esto conduce a la segunda regla: El método HTTP debe ser idempotente.

Si algo es idempotente, significa que se puede repetir, pero el resultado es siempre el mismo.

Por ejemplo, suponga que el cliente envía una solicitud RESTful DELETE para eliminar el registro número 271. Esta llamada se puede realizar una o 100 veces. En cualquier caso, el resultado final debe ser el mismo, es decir, el final de la vida del número 271.

El contra ejemplo es una solicitud para eliminar los 10 registros más antiguos de la base de datos, en este caso no se estaría cumpliendo la regla, lo que se recomienda en estos casos es que se utilice el método GET para obtener dichos registros y eliminarlos uno a uno o utilizar el método POST

El método PUT también debe ser idempotente. Por lo tanto todas las actualizaciones no deben ser masivas, sino que registro a registro. En algunos casos por performance de algunos motores, se recomienda eliminar para crear. Podemos llamar a este método una y otra vez, y después de cada llamada, el servidor estará en el mismo estado:

Técnicamente hablando, los parámetros de consulta al final de la URL solo deben usarse para consultas. En este ejemplo, usamos parámetros de consulta para pasar la carga útil al

servidor. Hacerlo simplifica el ejemplo, pero también rompe el propósito original del parámetro de consulta.

RESTful API: método POST

Ya sabemos que eliminar los 10 registros más antiguos de la base de datos es un uso incorrecto del método DELETE, y un simple incremento numérico es una mala aplicación del método PUT. ¿Significa esto que no podemos usar API RESTful para lograr estas cosas? por supuesto no.

En cualquier escenario distinto de las reglas anteriores, se puede utilizar el método POST. Por lo tanto, si desea eliminar los 10 registros más antiguos de la base de datos, puede utilizar el método POST.

Por supuesto, si el método POST se utiliza como un método universal para cumplir con todos los desafíos del diseño de API RESTful, aún existen riesgos. El hecho de que no haya una violación de las reglas sobre idempotencia o abuso de los métodos GET, PUT y DELETE no significa que la API RESTful se haya diseñado correctamente. El uso excesivo del método POST en sí mismo es también uno de los malentendidos del diseño RESTful.

Creando una API REST con Spring MVC

Para la creación de un api rest, utilizaremos un proyecto construido con STS con las librerías del framework para web.

```
<!-- spring mvc, rest -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Librería que contiene todo lo necesario para construir un api REST full con Spring MVC
Para levantar la vista (Front) es necesario la siguiente librería.

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

Con ella podemos generar plantillas HTML que pueden ser consumidas desde los controller.


```
1 <!DOCTYPE html>
2 <html lang="en" xmlns:th="http://www.thymeleaf.org">
3 <head>
4   <meta charset="utf-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
6   <title>Spring Boot Thymeleaf Hello World Example</title>
7   <link rel="stylesheet" th:href="@{webjars/bootstrap/4.2.1/css/bootstrap.min.css}">
8   <link rel="stylesheet" th:href="@{css/main.css}">
9
10 </head>
11
12 <body>
13
14 <nav class="navbar navbar-expand-md navbar-dark bg-dark fixed-top">
15   <a class="navbar-brand" href="#">AIEP</a>
16 </nav>
17
18 <main role="main" class="container">
19
20   <div class="starter-template">
21     <h1>Spring Boot Web Thymeleaf Example</h1>
22     <h2>
23       <span th:text="'Hello, ' + ${message}"></span>
24     </h2>
25   </div>
26
27   <div>
28     <li th:each="task : ${tasks}" th:text="${task}"></li>
29   </div>
30
31 </main>
32 <!-- /.container -->
33
34 <script type="text/javascript" th:src="@{webjars/bootstrap/4.2.1/js/bootstrap.min.js}"></script>
35 </body>
36 </html>
```

Luego debemos construir una clase java que contenga el api completa restfull con los métodos descritos anteriormente.

```
@RestController
public class BookController {
    @Autowired
    private BookRepository repository;
    @GetMapping("/books/{id}")
    Book findOne(@PathVariable Long id) {
        return repository.findById(id)
            .orElseThrow(() -> new BookNotFoundException(id));
    }
    @PutMapping("/books/{id}")
    Book saveOrUpdate(@RequestBody Book newBook, @PathVariable Long id) {
        return repository.findById(id)
            .map(x -> {
                x.setName(newBook.getName());
                x.setAuthor(newBook.getAuthor());
                x.setPrice(newBook.getPrice());
                return repository.save(x);
            })
            .orElseGet(() -> {
                newBook.setId(id);
                return repository.save(newBook);
            });
    }
    @DeleteMapping("/books/{id}")
    void deleteBook(@PathVariable Long id) {
        repository.deleteById(id);
    }
    @PostMapping("/books")
    //return 201 instead of 200
    @ResponseStatus(HttpStatus.CREATED)
    Book newBook(@RequestBody Book newBook) {
        return repository.save(newBook);
    }
    @PatchMapping("/books/{id}")
    Book patch(@RequestBody Map<String, String> update, @PathVariable Long id) {
        return repository.findById(id)
            .map(x -> {
                String author = update.get("author");
                if (!StringUtils.isEmpty(author)) {
                    x.setAuthor(author);
                    // better create a custom method to update a value = :newValue
                    return repository.save(x);
                } else {
                    throw new BookUnsupportedFieldPatchException(update.keySet());
                }
            })
            .orElseGet(() -> {
                throw new BookNotFoundException(id);
            });
    }
}
```

Segurización de una API REST mediante JWT.

¿Qué es JWT?

JSON Web Token (abreviado JWT) es un estándar abierto basado en JSON propuesto por IETF (RFC 7519) para la creación de tokens de acceso que permiten la propagación de identidad y privilegios o claims en inglés. Por ejemplo, un servidor podría generar un token indicando que el usuario tiene privilegios de administrador y proporcionarlo a un cliente. El cliente entonces podría utilizar el token para probar que está actuando como un administrador en el cliente o en otro sistema. El token está firmado por la clave del servidor, así que el cliente y el servidor son ambos capaz de verificar que el token es

legítimo. Los JSON Web Tokens están diseñados para ser compactos, poder ser enviados en las URLs -URL-safe- y ser utilizados en escenarios de Single Sign-On (SSO). Los privilegios de los JSON Web Tokens pueden ser utilizados para propagar la identidad de usuarios como parte del proceso de autenticación entre un proveedor de identidad y un proveedor de servicio, o cualquiera otro tipo de privilegios requeridos por procesos empresariales.

El estándar de JWT se basa en otros estándares basados en JSON Web Signature (RFC 7515) y JSON Web Encryption (RFC 7516)

Para la segurización de las APIS RESTfull con JWT, utilizaremos el IDE STS y para probar las APIS el programa PostMan.

Dentro del proyecto, se debe considerar la modificación en los siguientes puntos.

POM.xml

```
<!-- JSON WEB TOKEN -->
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
  <version>0.9.0</version>
</dependency>
<dependency>
  <groupId>org.json</groupId>
  <artifactId>json</artifactId>
  <version>20160810</version>
</dependency>
<!-- API, java.xml.bind module -->
<dependency>
  <groupId>jakarta.xml.bind</groupId>
  <artifactId>jakarta.xml.bind-api</artifactId>
  <version>2.3.2</version>
</dependency>
<!-- Runtime, com.sun.xml.bind module -->
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
  <version>2.3.2</version>
</dependency>

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

Se debe incluir la configuración de seguridad al proyecto.

```
@EnableWebSecurity
@Configuration
class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .addFilterAfter(new JWTAuthorizationFilter(), UsernamePasswordAuthenticationFilter.class)
            .authorizeRequests()
            .antMatchers(HttpMethod.POST, "/user").permitAll()
            .anyRequest().authenticated();
    }
}
```

Además, el filtro que permite interceptar las peticiones.

```

4 public class JWTAuthorizationFilter extends OncePerRequestFilter {
5     private final String HEADER = "Authorization";
6     private final String PREFIX = "Bearer ";
7     String SECRET = "A1EP5Secret";
8
9     @Override
10    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws ServletException,
11    {
12        try {
13            if (checkJWTToken(request, response)) {
14                Claims claims = validateToken(request);
15                if (claims.get("authorities") != null) {
16                    setSpringAuthentication(claims);
17                } else {
18                    SecurityContextHolder.clearContext();
19                }
20            } else {
21                SecurityContextHolder.clearContext();
22            }
23            chain.doFilter(request, response);
24        } catch (ExpiredJwtException | UnsupportedJwtException | MalformedJwtException e) {
25            response.setStatus(HttpServletResponse.SC_FORBIDDEN);
26            ((HttpServletResponse) response).sendError(HttpServletResponse.SC_FORBIDDEN, e.getMessage());
27            return;
28        }
29    }
30
31    private Claims validateToken(HttpServletRequest request) {
32        String jwtToken = request.getHeader(HEADER).replace(PREFIX, "");
33        return Jws.parser().setSigningKey(SECRET.getBytes()).parseClaimsJws(jwtToken).getBody();
34    }
35 }

```

Una vez configurado, procedemos a probar con postman.

Primero se debe obtener el token de autenticación de tipo Bearer.

<http://localhost:8080/user>

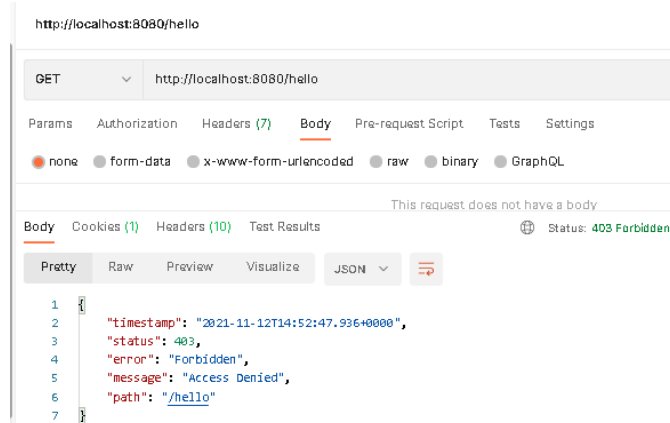
The screenshot shows the REST Client interface. At the top, the method is set to **POST** and the URL is `http://localhost:8080/user`. Below this, there are tabs for **Params**, **Authorization**, **Headers (9)**, **Body** (which is selected and underlined in red), **Pre-request Script**, **Tests**, and **Settings**. Under the **Body** tab, there are radio buttons for **none**, **form-data**, **x-www-form-urlencoded**, **raw**, **binary**, **GraphQL**, and **JSON** (which is selected). The JSON body is displayed in a text area with line numbers 1 through 4 on the left. The body content is:


```
1 {
2   "user": "azep",
3   "psword": "password"
4 }
```

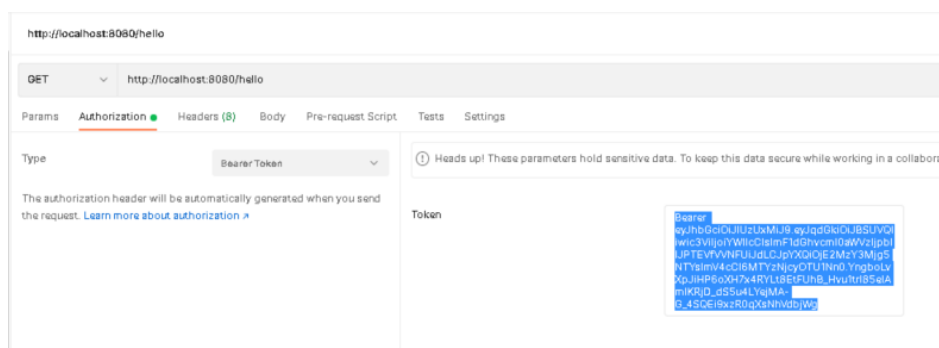
Nos responderá lo siguiente

[illegible]

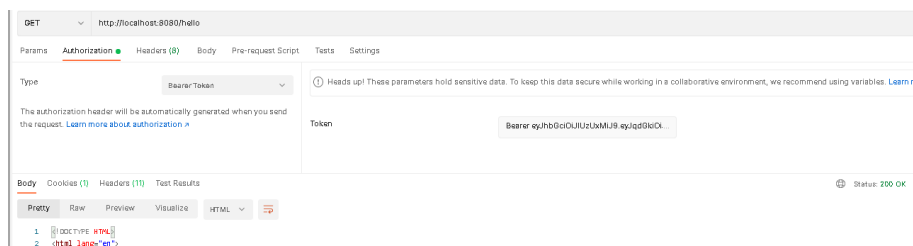
Sin este token no se podrá utilizar el resto de las APIs, por ejemplo.



Si se ingresa el bearer token.



En este caso permite acceder al sitio



BIBLIOGRAFÍA

Las urls Utilizadas para la generación de este documento fueron.

<https://spring.io/learn>

<https://spring.io/tools>

<https://www.thymeleaf.org/doc/articles/springsecurity.html>

https://www.tutorialspoint.com/spring_boot/index.htm

<https://www.baeldung.com/?s=spring>

<https://es.wikipedia.org/>

<https://jwt.io/introduction>

<https://mkyong.com/>

GLOSARIO

Término	Descripción
Single Sign-On (SSO)	Es un procedimiento de autenticación donde se habilita al usuario para que pueda acceder a múltiples sistemas, recursos o aplicaciones con una sola identificación en base a un único "usuario" y "contraseña"
Java EE	Java Enterprise Edition, Java EE en adelante, es un conjunto de estándares de tecnologías dedicadas al desarrollo de Java del lado del servidor. ... Es decir, desarrollaremos aplicaciones empresariales distribuidas, con arquitecturas multicapa, escritas en Java y que se ejecutan en un servidor de aplicaciones.

J2EE	J2EE es una plataforma para el cómputo empresarial a partir de la cual es posible el desarrollo profesional de aplicaciones empresariales distribuidas sobre una arquitectura multicapa, que son escritas con el lenguaje de programación Java y son ejecutadas desde un servidor de aplicaciones.
POJO	POJO son las iniciales de " Plain Old Java Object ", que puede interpretarse como "Un objeto Java Plano Antiguo". Un POJO es una instancia de una clase que no extiende ni implementa nada en especial. Para los programadores Java sirve para enfatizar el uso de clases simples y que no dependen de un framework en especial. Este concepto surge en oposición al modelo planteado por los estándares EJB anteriores al 3.0, en los que los Enterprise JavaBeans (EJB) debían implementar interfaces especiales.
XMLBeans	es una herramienta que nos permite manipular XML asociando cada uno de los elementos de un documento XML a objetos y tipos de datos de Java, haciendo más amigable el uso de XML. XMLBeans utiliza un esquema de XML (archivo XSD) para compilar las interfaces y las clases necesarias para poder acceder y modificar un documento XML. Una vez que se han generado las clases y las interfaces, es posible manipular archivos de XML como si fueran una colección de objetos y sin tener que hacer parsing (análisis sintáctico), ya que las funciones encargadas de hacer el análisis sintáctico o parsing son generadas por el XMLBeans a partir del archivo de esquema XSD.
JiBX.	Es un framework de libre distribución que permite trabajar con datos desde documentos XML usando nuestras propias estructuras de clases. JiBX maneja todo lo referente a la conversión de datos en XML y viceversa basándose en nuestras

	<p>instrucciones. Las principales características de esta herramienta:</p> <ul style="list-style-type: none"> • Flexibilidad: puede usar la estructura de clases que prefiera, así como indicar cómo traducirla a XML y viceversa. Es interesante realizar el mapeo de forma manual, porque de esta manera se asegura el correcto tratamiento de los datos y el formato que deseamos en la transformación. Si se utiliza un intérprete automático se corre el riesgo de una transformación inadecuada. • Rendimiento: el parseo y las técnicas de mejora de los ficheros class permiten a JiBX construir código de marshalling y unmarshalling de alto rendimiento partiendo directamente de sus clases. • Código limpio: nosotros escribimos el código y JiBX trabaja con él, y no al revés.
XStream	XStream es una librería Java que permite serializar objetos Java a XML y realizar su posterior recuperación. Se caracteriza por su eficiencia y simplicidad de uso, generando documentos XML relativamente legibles.
Web-Portlet	Son componentes pluggables que se gestionan y muestran en un portal web. Un portlet puede integrar y personalizar contenido de diferentes fuentes dentro de una página web y responde a las solicitudes de un cliente web generando contenido dinámico.
Web-Socket	WebSocket es un protocolo de red basado en TCP que establece cómo deben intercambiarse datos entre redes. Puesto que es un protocolo fiable y eficiente, es utilizado por prácticamente todos los clientes. El protocolo TCP establece conexiones entre dos puntos finales de comunicación, llamados sockets. De esta

	manera, el intercambio de datos puede producirse en las dos direcciones.
SpEL	Spring expression language (SpEL) es un lenguaje de expresiones que permite realizar operaciones sobre la información en tiempo de ejecución.
IoC	<p>Spring se basa en el principio de Inversión de Control (IoC) o Patrón Hollywood («No nos llames, nosotros le llamaremos») consiste en:</p> <ul style="list-style-type: none"> • Un Contenedor que maneja objetos por ti, este contenedor es un archivo XML. • El contenedor generalmente controla la creación de estos objetos. Por decirlo de alguna manera, el contenedor hace los “new” de las clases java para que no los realices tu. • El contenedor resuelve dependencias entre los objetos que contiene.
Velocity	<p>Es un motor de plantillas basado en Java. Con Velocity se pueden crear páginas web, SQL, clases java y cualquier otro tipo de salida de plantillas.</p> <p>Se puede utilizar como una aplicación independiente para generar código fuente e informes, o como un componente integrado en otros sistemas.</p> <p>Spring nos proporciona un motor de Velocity para generar, de forma dinámica, por ejemplo, el cuerpo de nuestros mensajes de correo electrónico.</p>

Serializable	Serializable es una clase ubicada en el paquete <code>java.io.Serializable</code> . Serializable, la cual no cuenta con ningún método, por lo que es una clase que sirve solamente para especificar que todo el estado de un objeto instanciado podrá ser escrito o enviado en la red como una trama de bytes
AssertJ	AssertJ es un proyecto de código abierto que ha surgido a partir del desaparecido Fest Assert. Es compatible con otras librerías como Guava, Joda Time y Neo4J. También, dispone de un generador automático de comprobaciones para los atributos de las clases, lo que añade más semántica a nuestras clases de prueba.
Hikari	HikariCP es un grupo de conexiones JDBC rápido, simple, confiable y listo para producción. En la versión Spring Boot 2.0, la tecnología de agrupación de bases de datos predeterminada se ha cambiado de Tomcat Pool a HikariCP. Esto se debe a que HikariCP proporciona un rendimiento excelente.
PostMan	Postman es una aplicación que nos permite realizar pruebas API. Es un cliente HTTP que nos da la posibilidad de testear 'HTTP requests' a través de una interfaz gráfica de usuario, por medio de la cual obtendremos diferentes tipos de respuesta que posteriormente deberán ser validados.