DESARROLLO DE APLICACIONES JEE CON SPRING FRAMEWORK

Dirección Nacional de Educación Continua

13-11-2021 Módulo 2

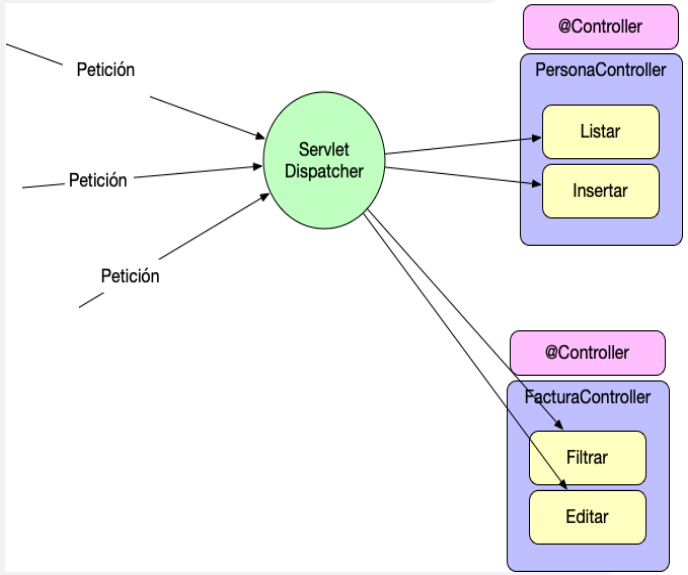
INTRODUCCIÓN

- La presente 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.

RESUMEN

- En el Módulo anterior, hemos revisado lo siguiente:
- ¿Qué es un IDE o entorno de desarrollo?, y sus capacidades.
- Se explica Spring Framework y su arquitectura
- Maven y sus aplicaciones
- Spring MVC
- Spring Boot
- Ciclo de vida de la Compilación

FRAMEWORK SPRING MVC



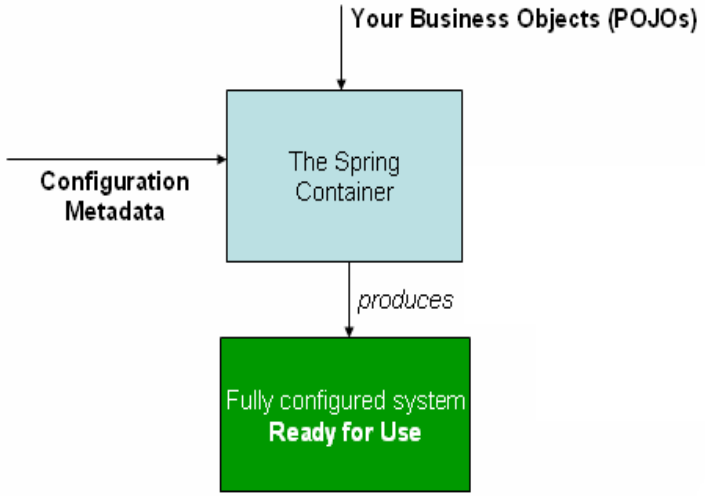
- Spring MVC es quizás el framework Web más utilizado en el mundo Java y nos permite crear aplicaciones sobre modelo MVC que generen páginas HTML sencillas en las cuales nosotros podamos cargar los contenidos que necesitemos de forma sencilla pudiendo integrarse con otras tecnologías como jQuery, React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

CARACTERÍSTICAS FRAMEWORK SPRING BOOT

- 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 ANOTACIONES

- Una anotación en Java es aquella característica que le permite incrustar información suplementaria en un archivo fuente
- 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?

- 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.

INYECCIÓN DE DEPENDENCIAS (AUTOWIRED)

```java
@Component // en este caso podríamos usar la especia
public class InyeccionPropiedadController {

  @Autowired
  public ServicioImplementado servicioImplementado;

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

}
```

- La inyección de dependencias es quizás la característica más destacable del core de Spring Framework, que consiste 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 creara los objetos y cuando una clase necesite usarlos se le pasaran (como cuando le pasas un parámetro a un método).
- 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

CREANDO UN PROYECTO WEB SPRING BOOT

- Para la creación de un proyecto Spring boot con Spring WEB utilizaremos el IDE antes mencionado STS4, con esto podremos validar y compilar, el menú es botón derecho nuevo-> spring starter proyect

CONFIGURANDO EL PROYECTO

- 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);
    }

}
```

¿QUE ES 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.

CONFIGURACIONES INICIALES SPRING BOOT-THYMELEAF

```
<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>
```

Las librerías necesarias para springboot con Thymeleaf son:

- **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

CONFIGURACIÓN DE LOG SPRING MVC

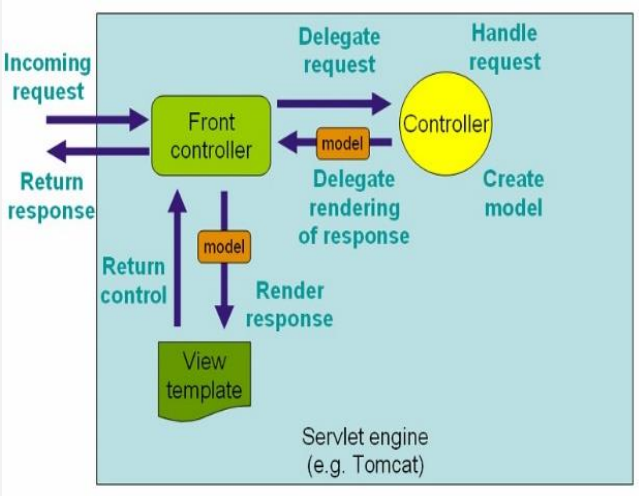
```
<!-- exclude logback , add log4j2 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>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

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

- 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.

CONFIGURACIÓN 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.

CAPA DE VISTA Y CONTROLADORES

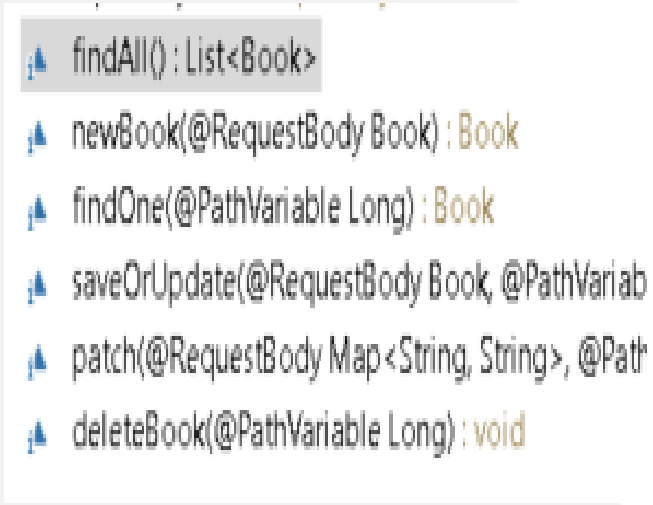


- Todas las peticiones HTTP se canalizan a través del front controller. En casi todos los frameworks MVC que siguen este patrón, el front controller no es más que un servlet cuya implementación es propia del framework. En el caso de Spring, la clase DispatcherServlet.
- El front controller averigua, 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

```java
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(() -> {
```

- Para configurar las peticiones de un proyecto con springboot utilizaremos la anotación @RestController para transformar una clase java en una API
- @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.



CONTROLADORES MULTIACCION

- 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
- Esta capacidad permite que las soluciones empresariales se puedan agrupar por negocios, capacidades, interacciones con otros sistemas, se puede aplicar patrones como DAO,Facade,ACL,etc.

RECIBIENDO DATOS EN EL CONTROLADOR

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

- 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
- Como podemos ver que recibe un parámetro de tipo PathVariable, la cual es considerada una buena práctica, mirado de 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

ENTREGANDO DATOS A LA VISTA

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

- Como vimos en el ejemplo anterior, es posible recibir datos desde un request.
- Para este caso tomaremos como ejemplo el find_by_id que devuelve un objeto JSON

DESPLEGANDO VISTA JSP CON DATOS DESDE CONTROLADOR

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


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

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

- 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 vamos a devolver datos a una plantilla ThymeLeaf llamada welcome.
- En estas líneas se puede ver el uso de un repository para acceder via JPA a la BD obtener todos los libros y devolverlos al front

CAPA 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)

ROL CAPA SERVICIOS EN 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.

CREANDO UNA CAPA SERVICIO UTILIZANDO ANOTACIONES

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

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

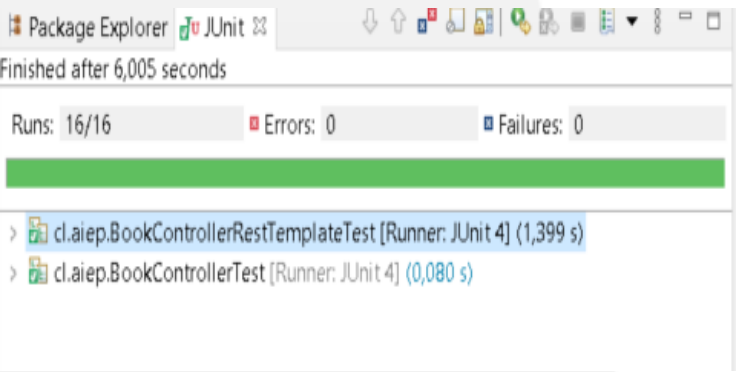
- Como vimos en el ejemplo anterior, un controller permite la generación de múltiples acciones, desde una consulta hasta interacciones con otros sistemas, legados o backends
- En este caso generaremos un Service para el manejo de lógica de negocio utilizando la anotación @Service

INYECTANDO UN SERVICIO AL CONTROLADOR

```java
@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 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.

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”

```java
package cl.aiep;

import com.fasterxml.jackson.core.JsonProcessingException;

//@WebMvcTest(BookController.class)
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
@ActiveProfiles("test")
public class BookControllerTest {

    private static final ObjectMapper om = new ObjectMapper();

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private BookRepository mockRepository;

    @Before
    public void init() {
        Book book = new Book(1L, "Book Name", "Mkyong", new BigDecimal("9.99"));
        when(mockRepository.findById(1L)).thenReturn(Optional.of(book));
    }

    @Test
    public void find_bookId_OK() throws Exception {

        mockMvc.perform(get("/books/1"))
                /*.andDo(print())*/
                .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.id", is(1)))
                .andExpect(jsonPath("$.name", is("Book Name")))
                .andExpect(jsonPath("$.author", is("Mkyong")))
                .andExpect(jsonPath("$.price", is(9.99)));

        verify(mockRepository, times(1)).findById(1L);

    }
```

CREANDO UNIDADES DE PRUEBA

- Las unidades de pruebas son clases java que utilizan las dependencias de Junit,y las anotaciones @@SpringBootTest o @RunWith
- Para este caso analizaremos los test asociados al rest controller de previamente vistos en el módulo.
- @Runwith anotación que permite indicar con que tipo de ejecutor vamos a probar nuestra clase, en este caso utilizaremos el defecto de Spring SpringRunner
- @SpringBoot test 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 WAR

```xml
<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.apac
    <modelVersion>4.0.0</modelVersion>

    <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>
```

- Para empaquetar el proyecto de ejemplo en WAR, solo se debe cambiar un parámetro en el POM.xml de jar a war, sin mayores configuraciones o descriptores.
- Luego ejecutar comando mvn clean install y se genera el proyecto WAR

PRUEBA PRÁCTICA NRO2

- El alumno deberá crear un proyecto starter spring boot
- Configurar el proyecto para thymeleaf
- Crear una nueva clase de tipo service con 2 métodos que entreguen 1 listado de números pares y otro de impares
- Crear una clase java que exponga un Servicio Rest de búsqueda con un párametro de entrada que indique par o impar
- Inyectar el service en la clase rest controller e invocar el método según parametro de entrada
- Crear una clase TEST que valide el cumplimiento de las clase previamente realizadas.
- Incluir las librerías necesarias en el POM.xml para ejecutar las pruebas
- Mostrar la consola con los principales paso a paso del ciclo de vida Maven, Clean, validate, compile, install.

AIEP
de la Universidad
Andrés Bello

Certified
B
Corporation

MUCHAS GRACIAS