LABORATORIO SPRING BOOT REST

Estoy seguro de que estás buscando un **tutorial** completo sobre **Spring Rest** que cubra los temas más importantes relacionados con **Spring Boot** . ¡Estás en el lugar correcto!

Desea crear una aplicación web o una **API REST** utilizando Spring Boot (y otras tecnologías populares como Thymeleaf), pero no sabe por dónde empezar ... Permítame ayudarle a hacer las cosas. Este tutorial explica cómo crear un Api de reposo **exponiendo datos como Json** .

No te preocupes, ¡Spring no es tan difícil!

En menos de 5 minutos , creará su primera aplicación web con Spring Boot.

Spring Initializr

Para comenzar rápidamente, genere un proyecto **web de muestra** con **Spring Initializr** . Algunas de las secciones a continuación revisan parte del código generado por Spring Initializr.

Módulo Maven Pom

Primero, necesitamos crear un Proyecto Maven con el siguiente **pom.xml** :

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema- instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.maint</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>
    <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.0.1.RELEASE</version>
      <relativePath/>
    </parent>
    <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
     <project.reporting.outputEncoding>UTF8</project.reporting.outputEncoding>
     <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
          <groupId>org.projectlombok</groupId>
          <artifactId>lombok</artifactId>
```

```
        </dependency>
        <dependency>
          <groupId>com.fasterxml.jackson.core</groupId>
          <artifactId>jackson-core</artifactId>
        </dependency>
        <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-test</artifactId>
          <scope>test</scope>
        </dependency>
      </dependencies>
    <build>
      <plugins>
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
      </plugins>
    </build>
  </project>
```

Comprendamos qué se está configurando aquí:

- **Parent** : **spring-boot-starter-parent** es un conveniente Maven Parent POM configurado con todo lo necesario para ejecutar una aplicación *Spring Boot* . Al definirlo como padre, la mayoría de las dependencias como **lombok** o **jackson** ya están administradas (no es necesario especificar una versión),
- **Dependencias** :
    - **lombok** : proporciona anotaciones para generar la mayor parte del código de la placa de la caldera (como constructores, métodos de código igual y hash, etc.),
    - **jackson-core** : **Jackson** es un popular marco de serialización de Java Json,
    - **spring-boot-starter-web** es una dependencia de Spring que importa todo lo necesario para construir una aplicación web usando Spring Boot.

Al agregar Jackson como una dependencia, Spring Boot configura automáticamente los puntos finales del resto con **el** serializador **Jackson** .

Asegúrese de habilitar el proceso de **anotación** dentro del IDE. Se puede hacer en **Compilar> Compilador> Procesadores de anotación** en Intellij (Enable Annotation Processing y Obtain processors from project classpath class Obtain processors from project classpath).

Ahora, necesitamos crear una aplicación principal que arranque el Spring Boot.

Spring-Boot Bootstrap

Si ya estás tan lejos, tienes un proyecto maven funcional en tu IDE favorito (como **Intellij**). Es hora de crear la aplicación principal:

```
package com.maint;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DemoApplication {
	public static void main(String[] args) {
		SpringApplication.run(DemoApplication.class, args);
	}
}
```

DemoApplication la clase **DemoApplication** en el paquete **com.maint.demo** . Cuando se ejecutará la aplicación Java, la ejecución se delegará directamente en la clase **SpringApplication** .

SpringApplication inicia y **SpringApplication** una aplicación Spring desde un método principal de Java. De forma predeterminada, realizará los siguientes pasos para iniciar su aplicación:

- Cree una instancia *ApplicationContext* apropiada (según su classpath),
- Registre un *CommandLinePropertySource* para exponer los argumentos de línea de comando como propiedades de Spring,
- Actualice el contexto de la aplicación, cargando todos los beans simples,
- Y activa cualquier beans *CommandLineRunner* .

¿Qué es un **ApplicationContext** ?

Interfaz central para proporcionar la configuración de una aplicación. Esto es de solo lectura mientras la aplicación se está ejecutando, pero puede volver a cargarse si la implementación lo admite.

¿Qué es **CommandLinePropertySource** ?

Clase base abstracta para implementaciones de {@link PropertySource} respaldadas por argumentos de línea de comando. El tipo parametrizado **T** representa la fuente subyacente de las opciones de línea de comando.

¿Qué es un **CommandLineRunner** ?

Interfaz utilizada para indicar que un bean debe **ejecutarse** cuando está contenido dentro de una **SpringApplication** .

Si echamos un vistazo a lo que hace la aplicación **@SpringBootApplication** :

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
```

```
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class), @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication { ... }
```

Es una anotación que habilita diversas funciones, como **la configuración automática de inicio de arranque** y el **escaneo de componentes** .

Spring detectará automáticamente cualquier clase comentada de Spring (como **@Component** o **@Service**) bajo **com.maint.demo** y las **com.maint.demo** al inicio de la aplicación.

Así que sí, **hay mucha magia** detrás de la escena, ¡pero no te preocupes! Tienes tiempo de sobra para dominar las partes internas de Spring una vez que comprendes los conceptos básicos.

Person Bean

Creemos ahora un bean **Person** dentro del mismo paquete. Este frijol representa una Persona:

```
package com.maint;
import lombok.Data;
@Data public class Person {
        String firstname, lastname;
}
```

¡Gracias a la anotación **@Value** de lombok, el bean es bastante simple! Lombok se encarga de generar el constructor, equivale a / hashcode methods, getters y define todos los campos como **private final** .

Vamos a enviar este grano a través del cable serializándolo en Json.

Controlador Spring MVC

Ahora expongamos el Bean **Person** través de un controlador Spring MVC:

```
package com.maint;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/person")
class PersonController { @GetMapping("/hello")
        public Person hello() {
```

```
                final Person person = new Person();
                person.setFirstname("John");
                person.setLastname("Smith");
                return person;
        }
}
```

Este punto final expone la ruta **/person/hello** , que devuelve una instancia de **Person** con *Juan* como primer nombre y *Smith* como apellido.

Editemos application.properties para configurar la propiedad **server.port** que controla el puerto en el que se ejecutará el servidor Spring Boot:

```
server.port=8081
```

Spring Boot integra un servidor de aplicaciones **Apache Tomcat** de forma predeterminada.

Iniciando la aplicación de demostración

¡Es hora de ejecutar la aplicación! Abra la clase **DemoApplication** y haga clic derecho sobre ella. Luego seleccione, **Run DemoApplication.main()** . Debería iniciar la aplicación web:

.....

2018-01-16 14: 49: 59.373 INFO 35582 - [main] swsmmaRequestMappingHandlerMapping: asignado "{[/ persona / hello], methods = [GET]}" a com.maint.demo.Person com.maint.demo público. PersonController.hello ()

.....

initialization started 2018-04-03 16:33:15.490 INFO 188101 --- [nio-8081-exec-1] osweb.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization completed in 2.181 s

El resultado anterior muestra la salida de línea de comandos de la aplicación. Indica que la aplicación se está ejecutando en el puerto **8081** . La aplicación ha comenzado en **2.181 seconds** en mi computadora.

2018-01-16 14: 49: 59.373 INFO 35582 - [main] swsmmaRequestMappingHandlerMapping: asignado "{[/ persona / hello], methods = [GET]}" a com.maint.demo.Person com.maint.demo público. PersonController.hello ()

La línea anterior significa que **PersonController** se detectó y asignó correctamente.

Ahora ejecutemos una línea de comando **curl** para verificar si el punto final funciona correctamente: **curl http://localhost:8081/person/hello** .

El resultado debería ser:

```
{"firstname":"John","lastname":"Smith"}
```

¡Felicitaciones, acabas de construir **tu primer Rest Api con Spring Boot** !

Punto final de repaso

Vamos más allá agregando un nuevo punto final a nuestro **PersonController** existente:

```
@PostMapping("/hello")
public String postHello(
        @RequestBody
        final Person person) {
                return "Hello " + person.getFirstname() + " " + person.getLastname() + "!";
                }
```

El servidor debería responder:

```
 $ curl -XPOST -H 'Content-type: application/json' -d '{"firstname": "John","lastname":"Smith"}'
http://localhost:8081/person/hello Hello John Smith!
```

¡Bonito! Pudimos enviar un objeto en formato Json y Spring lo convirtió de nuevo al Objeto Java correspondiente utilizando Jackson.

Asegurar los puntos finales

Ahora, aseguremos nuestros puntos finales para evitar el acceso no autorizado. Vamos a habilitar la **Autenticación básica** . ¿Como funciona?

El cliente debe enviar un **encabezado HTTP de** **Authorization** dentro de la solicitud como se indica a continuación:

```
 Authorization: Basic QWxhZGRpbjpPcGVuU2VzYW1l
```

El valor del encabezado comienza con la palabra clave **Basic** seguida del **username:password** codificada en Base64.

Primero, necesitamos agregar la siguiente dependencia de Maven al **pom.xml** :

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

Primero, agreguemos la siguiente configuración a nuestra **application.yml** :

```
 spring: security: user: name: admin password: passw0rd roles: USER
```

Entonces, debemos crear una clase anotada de Security **@Configuration** :

```
package com.maint;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
```

```
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
	@Override
	protected void configure(final HttpSecurity http) throws Exception {
		http .csrf() .disable() .httpBasic() .and() .authorizeRequests() .anyRequest()
		.authenticated();
	}
}
```

Esta configuración le dice a **Spring Boot 2** que habilite la autenticación básica. Tenga en cuenta que el prefijo **{noop}** le dice a Spring Security que ignore la codificación de contraseña en este caso.

Para obtener más información, consulte **Spring Security 5 Password Encoder** para obtener más información. Esto es nuevo para **Spring 5** .

Ahora es el momento de reiniciar la aplicación web:

```
Spring Boot :: (v2.0.0.RELEASE) 2018-04-03 16:32:59.059 INFO 188101 --- [ main]
com.maint.DemoApplication : Starting DemoApplication on desktop with PID 188101
(/home/ubuntu/git/demo/target/classes started by ubuntu in /home/ubuntu/git/demo) 2018-04-03
16:32:59.062 INFO 188101 --- [ main] com.maint.DemoApplication : No active profile set, falling back
to default profiles: default 2018-04-03 16:32:59.111 INFO 188101 --- [ main]
ConfigServletWebServerApplicationContext : Refreshing
org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContex
t@35ef1869: startup date [Tue Apr 03 16:32:59 CEST 2018]; root of context hierarchy 2018-04-03
16:33:00.185 INFO 188101 --- [ main] osbwembedded.tomcat.TomcatWebServer : Tomcat initialized
with port(s): 8081 (http) 2018-04-03 16:33:00.210 INFO 188101 --- [ main]
o.apache.catalina.core.StandardService : Starting service [Tomcat] 2018-04-03 16:33:00.210 INFO
188101 --- [ main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache
Tomcat/8.5.28 2018-04-03 16:33:00.220 INFO 188101 --- [ost-startStop-1]
oacatalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows
optimal performance in production environments was not found on the java.library.path:
[/usr/java/packages/lib/amd64:/usr/lib64:/lib64:/lib:/usr/lib] 2018-04-03 16:33:00.302 INFO 188101 --
- [ost-startStop-1] oaccC[Tomcat].[localhost].[/] : Initializing Spring embedded
WebApplicationContext 2018-04-03 16:33:00.302 INFO 188101 --- [ost-startStop-1]
osweb.context.ContextLoader : Root WebApplicationContext: initialization completed in 1195 ms
2018-04-03 16:33:00.437 INFO 188101 --- [ost-startStop-1] osbwservlet.FilterRegistrationBean :
Mapping filter: 'characterEncodingFilter' to: [/*] 2018-04-03 16:33:00.438 INFO 188101 --- [ost-
startStop-1] osbwservlet.FilterRegistrationBean : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2018-04-03 16:33:00.438 INFO 188101 --- [ost-startStop-1] osbwservlet.FilterRegistrationBean :
Mapping filter: 'httpPutFormContentFilter' to: [/*] 2018-04-03 16:33:00.438 INFO 188101 --- [ost-
startStop-1] osbwservlet.FilterRegistrationBean : Mapping filter: 'requestContextFilter' to: [/*] 2018-
04-03 16:33:00.438 INFO 188101 --- [ost-startStop-1] .s.DelegatingFilterProxyRegistrationBean :
Mapping filter: 'springSecurityFilterChain' to: [/*] 2018-04-03 16:33:00.439 INFO 188101 --- [ost-
startStop-1] osbwservlet.ServletRegistrationBean : Servlet dispatcherServlet mapped to [/] 2018-04-
03 16:33:00.712 INFO 188101 --- [ main] swsmmaRequestMappingHandlerAdapter : Looking for
@ControllerAdvice:
org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContex
t@35ef1869: startup date [Tue Apr 03 16:32:59 CEST 2018]; root of context hierarchy 2018-04-03
16:33:00.790 INFO 188101 --- [ main] swsmmaRequestMappingHandlerMapping : Mapped
```

"{[/person/hello],methods=[GET]}" onto public com.maint.Person com.maint.PersonController.hello() 2018-04-03 16:33:00.791 INFO 188101 --- [main] swsmmaRequestMappingHandlerMapping : Mapped "{[/person/hello],methods=[POST]}" onto public java.lang.String com.maint.PersonController.postHello(com.maint.Person) 2018-04-03 16:33:00.795 INFO 188101 --- [main] swsmmaRequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.springframework.web.servlet.ModelAndView org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController.errorHtml(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse) 2018-04-03 16:33:00.796 INFO 188101 --- [main] swsmmaRequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController.error(javax.servlet.http.HttpServletRequest) 2018-04-03 16:33:00.829 INFO 188101 --- [main] oswshandler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler] 2018-04-03 16:33:00.829 INFO 188101 --- [main] oswshandler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler] 2018-04-03 16:33:00.867 INFO 188101 --- [main] oswshandler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler] 2018-04-03 16:33:01.349 INFO 188101 --- [main] ossweb.DefaultSecurityFilterChain : Creating filter chain: org.springframework.security.web.util.matcher.AnyRequestMatcher@1, [org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter@7f02251, org.springframework.security.web.context.SecurityContextPersistenceFilter@73877e19, org.springframework.security.web.header.HeaderWriterFilter@30404dba, org.springframework.security.web.csrf.CsrfFilter@53093491, org.springframework.security.web.authentication.logout.LogoutFilter@75b3673, org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter@7dd00705, org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter@2b0b4d53, org.springframework.security.web.authentication.www.BasicAuthenticationFilter@6d4a65c6, org.springframework.security.web.savedrequest.RequestCacheAwareFilter@5bfc257, org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@4443ef6f, org.springframework.security.web.authentication.AnonymousAuthenticationFilter@dffa30b, org.springframework.security.web.session.SessionManagementFilter@4c0884e8, org.springframework.security.web.access.ExceptionTranslationFilter@23ee75c5, org.springframework.security.web.access.intercept.FilterSecurityInterceptor@267517e4] 2018-04-03 16:33:01.436 INFO 188101 --- [main] osjeaAnnotationMBeanExporter : Registering beans for JMX exposure on startup 2018-04-03 16:33:01.475 INFO 188101 --- [main] osbwembedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8081 (http) with context path '' 2018-04-03 16:33:01.479 INFO 188101 --- [main] com.maint.DemoApplication : Started DemoApplication in 2.774 seconds (JVM running for 3.16) 2018-04-03 16:33:15.467 INFO 188101 --- [nio-8081-exec-1] oaccC[Tomcat].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet' 2018-04-03 16:33:15.467 INFO 188101 --- [nio-8081-exec-1] osweb.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization started 2018-04-03 16:33:15.490 INFO 188101 --- [nio-8081-exec-1] osweb.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization completed in 23 ms

Como puede ver, han aparecido nuevas líneas de registro relacionadas con **org.springframework.security.web** , lo que significa que **Spring Security** se ha habilitado. Para aquellos que quieran comprender cómo funciona la configuración automática, consulte **SpringBootWebSecurityConfiguration javadoc** .

Ahora, *ejecutemos de* nuevo el comando *curl* :

```
curl -XPOST -H 'Content-type: application/json' -d '{"firstname": "John","lastname":"Smith"}' http://localhost:8081/person/hello
```

{"timestamp":1516112340374,"status":401,"error":"Unauthorized","message":"Full authentication is required to access this resource","path":"/person/hello"}

El servidor responde indicando que **el punto final requiere una autenticación** . Intentemos de nuevo especificando el nombre de usuario y la contraseña:

curl -XPOST -H 'Content-type: application/json' -d '{"firstname": "John","lastname":"Smith"}' http://admin:passw0rd@localhost:8081/person/hello Hello John Smith!

¡Estupendo! El punto final ahora está protegido por una Autenticación Básica general. Por supuesto, cuando construyas una aplicación web del mundo real, probablemente quieras asegurar tu aplicación web con un acceso específico por usuario. Cubriremos este punto en un artículo futuro.

Business Logic movido a un servicio

La cuestión es que es bastante feo tener la lógica de la aplicación escrita dentro del controlador. Claro, esta aplicación de demostración es lo suficientemente simple y realmente no importa aquí. Pero, al construir una aplicación web completamente desarrollada utilizando Spring, **tiene servicios de alto nivel que hacen el trabajo por usted** .

Vamos a crear un **PersonService** simple que hace el trabajo real en un **com.maint.demo.service** llamado **com.maint.demo.service** :

```
package com.maint;

import com.maint.demo.Person;

public interface PersonService {
	Person johnSmith();
	String hello(Person person);
}
```

La implementación es la siguiente:

```
package com.maint.demo.service;

import com.maint.demo.Person;
import org.springframework.stereotype.Service;

@Service
final class DemoPersonService implements PersonService {

	@Override
	public Person johnSmith() {
		final Person person = new Person();
		person.setFirstname("John");
		person.setLastname("Smith");
```

```
		return person;
	}

	@Override public String hello(final Person person) {
		return "Hello " + person.getFirstname() + " " + person.getLastname() + "!";
	}
}
```

En el código de arriba:

- **DemoPersonService** implementa **DemoPersonService** ,
- **DemoPersonService** está anotado Anotación **@Service** de Spring: le dice a Spring que este es un servicio que necesita ser instanciado. Un servicio es una instancia de larga ejecución que vive mientras se ejecuta la aplicación web,
- El servicio está **package protected** : la clase no es accesible fuera del paquete,
- Solo la interfaz **PersonService** es pública.

Ahora, modifique **PersonController** para delegar el trabajo al servicio definido anteriormente:

```
package com.maint.demo;

import com.maint.demo.service.PersonService;
import lombok.AllArgsConstructor;
import lombok.NonNull;
import lombok.experimental.FieldDefaults;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import static lombok.AccessLevel.PACKAGE;
import static lombok.AccessLevel.PRIVATE;

@RestController
@RequestMapping("/person")
@AllArgsConstructor(access = PACKAGE)
@FieldDefaults(level = PRIVATE, makeFinal = true)
class PersonController {
	@NonNull
	PersonService persons;
	@GetMapping("/johnsmith")
	public Person hello() {
		return persons.johnSmith();
	}

	@PostMapping("/hello")
	public String postHello(@RequestBody final Person person) {
		return persons.hello(person);
	}
}
```

Algunas cosas han cambiado:
PersonController está anotado con **@AllArgsConstructor** y **@FieldDefaults** Anotaciones de lombok: le dice a lombok que cree un constructor para los parámetros requeridos y que marque todos los campos como **private final** (lo que hace que todos sean obligatorios). El controlador es inmutable,

- **@NonNull PersonService persons;** : el servicio se define como un campo de **PersonController** , y no debe ser nulo (código nullcheck escrito por lombok).

El código generado se ve así:

```
@RestController
@RequestMapping({"/person"})
class PersonController {
	@NonNull
	private final PersonService persons;
	@GetMapping({"/johnsmith"})
	public Person hello() {
		return this.persons.johnSmith();
	}
	@PostMapping({"/hello"})
	public String postHello(@RequestBody Person person) {
		return this.persons.hello(person);
	}
	@ConstructorProperties({"persons"})
	PersonController(@NonNull PersonService persons) {
		if (persons == null) {
			throw new NullPointerException("persons");
		} else {
			this.persons = persons;
		}
	}
}
```

¡Ves lo poderoso que es **Lombok** ! Su lógica de aplicación se encuentra dentro de la implementación de **PersonService** . Primavera automáticamente

- **DemoPersonService** instancia de **DemoPersonService** ,
- **PersonController** una instancia del **PersonController** proporcionando una instancia de **PersonService** a su constructor.

Una vez que comprenda todos esos conceptos simples, diseñar incluso aplicaciones web realmente grandes no difiere mucho del modelo anterior. Se trata de servicios que se exponen a través de Rest o Web Endpoints. Y esos servicios en sí mismos pueden delegar en subservicios.

Prueba de unidad RestAssured

Ahora es el momento de escribir una prueba unitaria para verificar nuestro punto final de descanso. Las pruebas son un punto crítico para evitar regresiones cuando el código evoluciona.

Primero, agregue la dependencia de prueba **RestAssured** :

```
<dependency>
	<groupId>io.rest-assured</groupId>
	<artifactId>rest-assured</artifactId>
	<version>3.0.7</version>
	<scope>test</scope>
</dependency>
```

Luego, vamos a escribir un JUnit que realiza una llamada Rest usando **RestAssured** :

```
package com.maint;

import io.restassured.RestAssured;
import org.hamcrest.Matchers;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import static io.restassured.RestAssured.get;
import static io.restassured.RestAssured.preemptive;
import static org.springframework.boot.test.context.SpringBootTest.WebEnvironment.RANDOM_PORT;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = { DemoApplication.class }, webEnvironment = RANDOM_PORT)
public class PersonControllerTest {
	@Value("${local.server.port}")
	private int port;
	@Before
	public void setUp() {
		RestAssured.authentication = preemptive().basic("admin", "passw0rd");
	}
	@Test
	public void shouldSayHello() {
		get("http://localhost:" + port + "/person/johnsmith") .then() .assertThat() .statusCode(200)
	.body("firstname", Matchers.equalTo("John")) .and() .body("lastname", Matchers.equalTo("Smith"));
	}
}
```

El Junit anterior hace varias cosas:

- Se ejecuta con **SpringRunner** : la prueba unitaria debe integrarse en una aplicación Spring,
- **@SpringBootTest** : especifica la aplicación de arranque que se utilizará y configura un contexto web en un puerto disponible al azar,

- **@Value("${local.server.port}")** : conecta automáticamente el puerto de la aplicación web aleatoria, para que podamos reutilizarlo cuando **realicemos la** llamada Rest a través de **RestAssured** ,
- **RestAssured.authentication = preemptive().basic("admin", "passw0rd");** : configura *RestAssured* para usar la **Autenticación básica** ya que nuestros puntos finales se han asegurado previamente.

Ahora tiene una unidad de prueba que automáticamente activa un servidor web incorporado con su controlador y sus servicios dentro. La prueba de la unidad luego realiza una solicitud HTTP real al endpoint **/person/johnsmith** del controlador, y verifica el contenido de la respuesta.

Le sugiero que profundice en la **documentación de RestAssured** para explorar más a fondo las pruebas de **puntos finales de** descanso.

Prueba de Unidad de Retrofit

Alternativamente a **RestAssured** , puede usar **Retrofit** . Retrofit es un cliente HTTP de tipo seguro para Java. De hecho, puede usar cualquier cliente de descanso de Java que se adapte a sus necesidades.

Primero, agregue la dependencia de prueba:

```
<dependency>
	<groupId>com.squareup.retrofit2</groupId>
	<artifactId>retrofit</artifactId>
	<version>2.3.0</version>
	<scope>test</scope>
</dependency>
<dependency>
	<groupId>com.squareup.retrofit2</groupId>
	<artifactId>converter-jackson</artifactId>
	<version>2.3.0</version>
	<scope>test</scope>
</dependency>
```

Por favor, asegúrese de usar la última versión en el momento de leer esto. El código puede variar según las evoluciones futuras de la biblioteca.

Ahora escribamos la Interfaz API usando Retrofit:

```
package com.maint.demo;

import retrofit2.Call;
import retrofit2.http.GET;

public interface PersonApi {
	@GET("/person/johnsmith")
	Call<Person> johnSmith();
}
```

Entonces, necesitamos un interceptor de solicitud de Autenticación Básica:

```
package com.maint.demo;

import okhttp3.Credentials;
import okhttp3.Interceptor;
import okhttp3.Request;
import okhttp3.Response;
import java.io.IOException;

final class BasicAuthInterceptor implements Interceptor {
	private final String credentials;
	BasicAuthInterceptor(final String user, final String password) {
		this.credentials = Credentials.basic(user, password);
	}

	@Override
	public Response intercept(Chain chain) throws IOException {
		final Request request = chain.request();
		final Request authenticatedRequest = request.newBuilder() .header("Authorization",
	credentials).build();
		return chain.proceed(authenticatedRequest);
	}
}
```

Y finalmente, use esta interfaz dentro de la prueba unitaria:

```
package com.maint.demo; import com.fasterxml.jackson.databind.ObjectMapper; import okhttp3.Credentials;
import okhttp3.Interceptor; import okhttp3.OkHttpClient; import okhttp3.Request; import okhttp3.Response;
import org.junit.Before; import org.junit.Test; import org.junit.runner.RunWith; import
org.springframework.beans.factory.annotation.Value; import
org.springframework.boot.test.context.SpringBootTest; import
org.springframework.test.context.junit4.SpringRunner; import retrofit2.Retrofit; import
retrofit2.converter.jackson.JacksonConverterFactory; import java.io.IOException; import static
org.junit.Assert.assertEquals; import static
org.springframework.boot.test.context.SpringBootTest.WebEnvironment.RANDOM_PORT;
@RunWith(SpringRunner.class) @SpringBootTest(classes = { DemoApplication.class }, webEnvironment =
RANDOM_PORT) public class PersonControllerRetrofitTest { @Value("${local.server.port}") private int port;
private Retrofit retrofit; @Before public void setUp() { final OkHttpClient client = new OkHttpClient.Builder()
.addInterceptor(new BasicAuthInterceptor("admin", "passw0rd")) .build(); retrofit = new Retrofit.Builder()
.baseUrl("http://localhost:"+port) .client(client) .addConverterFactory(JacksonConverterFactory.create(new
ObjectMapper())) .build(); } @Test public void shouldSayHello() throws IOException { final PersonApi api =
retrofit.create(PersonApi.class); final Person person = api.johnSmith().execute().body(); assertEquals("John",
person.getFirstname()); assertEquals("Smith", person.getLastname()); } }
```

Depende de usted elegir el cliente Rest con el que se sienta más cómodo. Hay muchos otros clientes disponibles (Feign, Resteasy, Spring RestTemplate, UniRest y más).

Controlador de excepciones

Spring ofrece un mecanismo simple de manejo de errores centralizado mediante la anotación `@ControllerAdvice` . Aquí hay un ejemplo.

Primero, `DemoException` una `DemoException` en el paquete `com.maint.demo.exception` :

```
package com.maint.demo.exception; public class DemoException extends Exception { }
```

Ahora, vamos a crear `DemoExceptionHandler` :

```
package com.maint.demo.exception; import org.springframework.http.HttpHeaders; import
org.springframework.http.HttpStatus; import org.springframework.http.ResponseEntity; import
org.springframework.web.bind.annotation.ControllerAdvice; import
org.springframework.web.bind.annotation.ExceptionHandler; import
org.springframework.web.context.request.WebRequest; import
org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;
@ControllerAdvice class DemoExceptionHandler extends ResponseEntityExceptionHandler {
@ExceptionHandler({ DemoException.class }) protected ResponseEntity<Object> handleNotFound(
Exception ex, WebRequest request) { return handleExceptionInternal(ex, "Demo Exception
Encountered", new HttpHeaders(), HttpStatus.NOT_FOUND, request); } }
```

El controlador básicamente envía un Http 404 que no se encuentra cuando la `DemoException` es lanzada por cualquier controlador Spring MVC.

Finalmente, enriquezca nuestro `PersonController` agregando un punto final para simular esta excepción:

```
@GetMapping("/exception") public void exception() throws DemoException { throw new
DemoException(); }
```

Aquí está el resultado al ejecutar una solicitud HTTP a este punto final con `curl` :

```
ubuntu@desktop:~$ curl -v http://admin:passw0rd@localhost:8081/person/exception * Trying
127.0.0.1... * Connected to localhost (127.0.0.1) port 8081 (#0) * Server auth using Basic with user
'admin' > GET /person/exception HTTP/1.1 > Host: localhost:8081 > Authorization: Basic
YWRtaW46cGFzc3cwcmQ= > User-Agent: curl/7.47.0 > Accept: */* > < HTTP/1.1 404 < Set-Cookie:
JSESSIONID=B1AE72512170F07C4D440BF87167C014; Path=/; HttpOnly < X-Content-Type-
Options: nosniff < X-XSS-Protection: 1; mode=block < Cache-Control: no-cache, no-store, max-
age=0, must-revalidate < Pragma: no-cache < Expires: 0 < X-Frame-Options: DENY < Content-Type:
text/plain;charset=UTF-8 < Content-Length: 26 < Date: Tue, 03 Apr 2018 15:12:14 GMT < *
Connection #0 to host localhost left intact Demo Exception Encountered
```

El servidor detecta correctamente la excepción y devuelve el HTTP 404 como se indica en el `DemoExceptionHandler`. De esta forma, puede controlar cómo se comporta la aplicación para cada excepción lanzada por cualquier controlador de reposo.

Incluso `@ControllerAdvice`se pueden definir múltiples clases anotadas. Cada uno puede ser responsable de manejar las excepciones lanzadas por una parte particular de su aplicación web.

Ultimas palabras

Solo hemos arañado la superficie de lo que se puede hacer con **Spring Boot** en este tutorial. **Spring** es un potente agregado de una docena de

bibliotecas que hacen que desarrollar aplicaciones web sea tan fácil como sea posible. ¡Asegúrate de explorarlos cada vez que lo necesites, tal vez haya una biblioteca que lo haga por ti!

El código fuente completo está disponible en **Spring Boot 2 Demo** en Github.
¡Siéntete libre de compartir tus propios ejemplos de código si sientes que falta algo en este artículo!

ASEGURAR UNA API DE DESCANSO CON SPRING SECURITY

Desarrollo 8 de marzo de 2018 3 comentarios increíbles

La mayoría de los tutoriales de primavera disponibles en línea le enseñan **cómo proteger una API de descanso con Spring** con ejemplos que están lejos de ser una aplicación problemática real. Seguramente estás de acuerdo en que la **mayoría de los tutoriales carecen de casos de uso del mundo real** .

Este tutorial tiene como objetivo ayudarlo a **asegurar una aplicación en el mundo real** , no solo otro ejemplo de Hello World.

En este tutorial aprenderemos:

- Cómo asegurar una **Spring MVC** **Rest API** usando **Spring Security** ,
- Configure **Spring Security** con código Java (sin doloroso XML),
- Y delegue la autenticación a un *UserAuthenticationService* con **su propia lógica comercial** .

Pasé varias semanas modificando Spring Security para llegar a esta configuración simple. ¡Vamonos!

El código fuente completo está disponible en **Github** .

Cómo funciona

La siguiente configuración de seguridad de Spring funciona de la siguiente manera:

- El usuario inicia sesión con una solicitud POST que contiene su nombre de usuario y contraseña,
- El servidor devuelve un token de autenticación temporal / permanente,
- El usuario envía el token dentro de cada solicitud HTTP a través de un encabezado HTTP `Authorization: Bearer TOKEN` .

Cuando el usuario cierra la sesión, el token se borra en el lado del servidor. ¡Eso es!

Maven POM

Configuremos un módulo maven separado con las siguientes dependencias:

```
<?xml version="1.0" encoding="UTF-8"?> <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"> <modelVersion>4.0.0</modelVersion> <parent> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-parent</artifactId>
```

```
<version>2.0.1.RELEASE</version> </parent> <groupId>com.maint</groupId> <artifactId>securing-rest-api-spring-security</artifactId> <version>1.0-SNAPSHOT</version> <dependencies> <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-web</artifactId> </dependency> <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-security</artifactId> </dependency> <dependency> <groupId>org.projectlombok</groupId> <artifactId>lombok</artifactId> </dependency> <dependency> <groupId>javax.servlet</groupId> <artifactId>javax.servlet-api</artifactId> </dependency> <dependency> <groupId>org.apache.commons</groupId> <artifactId>commons-lang3</artifactId> <version>3.6</version> </dependency> <dependency> <groupId>com.google.guava</groupId> <artifactId>guava</artifactId> <version>23.2-jre</version> </dependency> </dependencies> <build> <plugins> <plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-compiler-plugin</artifactId> <version>3.7.0</version> <configuration> <!-- mandatory to compile project with maven 3.3.9, might be removed with latest version --> <useIncrementalCompilation>false</useIncrementalCompilation> <source>1.8</source> <target>1.8</target> <optimize>true</optimize> </configuration> </plugin> </plugins> </build> </project>
```

En este ejemplo, vamos a usar **Spring Boot** para configurar rápidamente una aplicación web utilizando Spring MVC y Spring Security.

Ahora, configuremos Spring Security.

Gestión de usuarios

En esta sección, voy a cubrir la implementación del código responsable de iniciar y cerrar usuarios.

Usuario

El `User` bean representa un solo usuario:

```
package com.maint; import com.fasterxml.jackson.annotation.JsonCreator; import com.fasterxml.jackson.annotation.JsonIgnore; import com.fasterxml.jackson.annotation.JsonProperty; import lombok.Builder; import lombok.Value; import org.springframework.security.core.GrantedAuthority; import org.springframework.security.core.userdetails.UserDetails; import java.util.ArrayList; import java.util.Collection; import static java.util.Objects.requireNonNull; @Value @Builder public class User implements UserDetails { private static final long serialVersionUID = 2396654715019746670L; String id; String username; String password; @JsonCreator User(@JsonProperty("id") final String id, @JsonProperty("username") final String username, @JsonProperty("password") final String password) { super(); this.id = requireNonNull(id); this.username = requireNonNull(username); this.password = requireNonNull(password); } @JsonIgnore @Override public Collection<GrantedAuthority> getAuthorities() { return new ArrayList<>(); } @JsonIgnore @Override public String getPassword() { return password; } @JsonIgnore @Override public boolean isAccountNonExpired() { return true; } @JsonIgnore @Override public boolean isAccountNonLocked() { return true; } @JsonIgnore @Override public boolean isCredentialsNonExpired() { return true; } @Override public boolean isEnabled() { return true; } }
```

Para que coincida con Spring Security API, la clase `User` implementa `UserDetails` . De esta forma, nuestro bean de `User`personalizado se integra perfectamente en Spring Security.

UserAuthenticationService

UserAuthenticationService es responsable de iniciar sesión y salir de los usuarios, así como de entregar los tokens de autenticación.

```
package com.maint; import java.util.Optional; public interface UserAuthenticationService { /** * Logs in with the given {@code username} and {@code password}. * * @param username * @param password * @return an {@link Optional} of a user when login succeeds */ Optional<String> login(String username, String password); /** * Finds a user by its dao-key. * * @param token user dao key * @return */ Optional<User> findByToken(String token); /** * Logs out the given input {@code user}. * * @param user the user to logout */ void logout(User user); }
```

En este tutorial, voy a utilizar una implementación muy simple que almacena a los usuarios en un HashMap local por token.

```
package com.maint; import org.springframework.stereotype.Service; import java.util.HashMap; import java.util.Map; import java.util.Optional; import java.util.UUID; @Service final class SimpleAuthenticationService implements UserAuthenticationService { Map<String, User> users = new HashMap<>(); @Override public Optional<String> login(final String username, final String password) { final String token = UUID.randomUUID().toString(); final User user = User .builder() .id(token) .username(username) .password(password) .build(); users.put(token, user); return Optional.of(token); } @Override public Optional<User> findByToken(final String token) { return Optional.ofNullable(users.get(token)); } @Override public void logout(final User user) { users.remove(user.getId()); } }
```

El servicio inicia sesión en cualquier usuario. Lo he dicho, ¡es bastante simple! Puede conectar aquí su propia lógica de autenticación en lugar de esta ficticia. Depende de usted adaptar el código a sus propias necesidades.

Spring Security Config

Estrategia de redirigir

Como estamos asegurando una API REST, en caso de falla de autenticación, el servidor no debe redirigir a ninguna página de error. El servidor simplemente devolverá un **HTTP 401 (no autorizado)** . Aquí está la NoRedirectStrategy ubicada en el paquete com.maint.security :

```
package com.maint; import org.springframework.security.web.RedirectStrategy; import javax.servlet.http.HttpServletRequest; import javax.servlet.http.HttpServletResponse; import java.io.IOException; class NoRedirectStrategy implements RedirectStrategy { @Override public void sendRedirect(final HttpServletRequest request, final HttpServletResponse response, final String url) throws IOException { // No redirect is required with pure REST } }
```

No hay nada lujoso aquí, el propósito es mantener las cosas simples.

Proveedor de autenticación Token

El TokenAuthenticationProvider es responsable de encontrar al usuario mediante su token de autenticación.

```
package com.maint; import com.maint.service.UserAuthenticationService; import lombok.AllArgsConstructor; import lombok.NonNull; import lombok.experimental.FieldDefaults; import org.springframework.security.authentication.UsernamePasswordAuthenticationToken; import org.springframework.security.authentication.dao.AbstractUserDetailsAuthenticationProvider; import org.springframework.security.core.userdetails.UserDetails; import org.springframework.security.core.userdetails.UsernameNotFoundException; import org.springframework.stereotype.Component; import java.util.Optional; import static lombok.AccessLevel.PACKAGE; import static lombok.AccessLevel.PRIVATE; @Component @AllArgsConstructor(access = PACKAGE) @FieldDefaults(level = PRIVATE, makeFinal = true) final class TokenAuthenticationProvider extends AbstractUserDetailsAuthenticationProvider { @NonNull UserAuthenticationService auth; @Override protected void additionalAuthenticationChecks(final UserDetails d, final UsernamePasswordAuthenticationToken auth) { // Nothing to do } @Override protected UserDetails retrieveUser(final String username, final UsernamePasswordAuthenticationToken authentication) { final Object token = authentication.getCredentials(); return Optional .ofNullable(token) .map(String::valueOf) .flatMap(auth::findByToken) .orElseThrow(() -> new UsernameNotFoundException("Cannot find user with authentication token=" + token)); } }
```

El TokenAuthenticationProvider delega al UserAuthenticationService que hemos visto en la sección anterior.

TokenAuthenticationFilter

TokenAuthenticationFilter es responsable de extraer el token de autenticación de los encabezados de solicitud. Toma el valor del encabezado Authorization e intenta extraer el token de él.

```
package com.maint; import lombok.experimental.FieldDefaults; import org.springframework.security.authentication.BadCredentialsException; import org.springframework.security.authentication.UsernamePasswordAuthenticationToken; import org.springframework.security.core.Authentication; import org.springframework.security.web.authentication.AbstractAuthenticationProcessingFilter; import org.springframework.security.web.util.matcher.RequestMatcher; import javax.servlet.FilterChain; import javax.servlet.ServletException; import javax.servlet.http.HttpServletRequest; import javax.servlet.http.HttpServletResponse; import java.io.IOException; import static com.google.common.net.HttpHeaders.AUTHORIZATION; import static java.util.Optional.ofNullable; import static lombok.AccessLevel.PRIVATE; import static org.apache.commons.lang3.StringUtils.removeStart; @FieldDefaults(level = PRIVATE, makeFinal = true) final class TokenAuthenticationFilter extends AbstractAuthenticationProcessingFilter { private static final String BEARER = "Bearer"; TokenAuthenticationFilter(final RequestMatcher requiresAuth) { super(requiresAuth); } @Override public Authentication attemptAuthentication( final HttpServletRequest request, final HttpServletResponse response) { final String param = ofNullable(request.getHeader(AUTHORIZATION)) .orElse(request.getParameter("t")); final String token = ofNullable(param) .map(value -> removeStart(value, BEARER)) .map(String::trim) .orElseThrow(() -> new BadCredentialsException("Missing Authentication Token")); final Authentication auth = new UsernamePasswordAuthenticationToken(token, token); return getAuthenticationManager().authenticate(auth); } @Override protected void successfulAuthentication( final HttpServletRequest request, final HttpServletResponse response, final FilterChain chain, final Authentication authResult) throws IOException, ServletException { super.successfulAuthentication(request, response, chain, authResult); chain.doFilter(request, response); } }
```

Nuevamente, nada de lujoso aquí. ¡El código es bastante directo! La autenticación se delega en el AuthenticationManager . El filtro solo está habilitado para un conjunto determinado de URL. Veremos en las próximas secciones siguientes cómo se configura este filtro.

SecurityConfig

Es hora de configurar Spring Security con todos los servicios que definimos anteriormente:

```
 package com.maint; import lombok.experimental.FieldDefaults; import
org.springframework.boot.web.servlet.FilterRegistrationBean; import
org.springframework.context.annotation.Bean; import
org.springframework.context.annotation.Configuration; import
org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilde
r; import
org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity; import
org.springframework.security.config.annotation.web.builders.WebSecurity; import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity; import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.web.AuthenticationEntryPoint; import
org.springframework.security.web.authentication.AnonymousAuthenticationFilter; import
org.springframework.security.web.authentication.HttpStatusEntryPoint; import
org.springframework.security.web.authentication.SimpleUrlAuthenticationSuccessHandler; import
org.springframework.security.web.util.matcher.AntPathRequestMatcher; import
org.springframework.security.web.util.matcher.NegatedRequestMatcher; import
org.springframework.security.web.util.matcher.OrRequestMatcher; import
org.springframework.security.web.util.matcher.RequestMatcher; import static
java.util.Objects.requireNonNull; import static lombok.AccessLevel.PRIVATE; import static
org.springframework.http.HttpStatus.UNAUTHORIZED; import static
org.springframework.security.config.http.SessionCreationPolicy.STATELESS; @Configuration
@EnableWebSecurity @EnableGlobalMethodSecurity(prePostEnabled=true) @FieldDefaults(level =
PRIVATE, makeFinal = true) class SecurityConfig extends WebSecurityConfigurerAdapter {
private static final RequestMatcher PUBLIC_URLS = new OrRequestMatcher( new
AntPathRequestMatcher("/public/**") ); private static final RequestMatcher PROTECTED_URLS =
new NegatedRequestMatcher(PUBLIC_URLS); TokenAuthenticationProvider provider;
SecurityConfig(final TokenAuthenticationProvider provider) { super(); this.provider =
requireNonNull(provider); } @Override protected void configure(final AuthenticationManagerBuilder
auth) { auth.authenticationProvider(provider); } @Override public void configure(final WebSecurity
web) { web.ignoring().requestMatchers(PUBLIC_URLS); } @Override protected void configure(final
HttpSecurity http) throws Exception { http .sessionManagement()
.sessionCreationPolicy(STATELESS) .and() .exceptionHandling() // this entry point handles when you
request a protected page and you are not yet // authenticated
.defaultAuthenticationEntryPointFor(forbiddenEntryPoint(), PROTECTED_URLS) .and()
.authenticationProvider(provider) .addFilterBefore(restAuthenticationFilter(),
AnonymousAuthenticationFilter.class) .authorizeRequests() .anyRequest() .authenticated() .and()
.csrf().disable() .formLogin().disable() .httpBasic().disable() .logout().disable(); } @Bean
TokenAuthenticationFilter restAuthenticationFilter() throws Exception { final TokenAuthenticationFilter
filter = new TokenAuthenticationFilter(PROTECTED_URLS);
filter.setAuthenticationManager(authenticationManager());
filter.setAuthenticationSuccessHandler(successHandler()); return filter; } @Bean
SimpleUrlAuthenticationSuccessHandler successHandler() { final
SimpleUrlAuthenticationSuccessHandler successHandler = new
SimpleUrlAuthenticationSuccessHandler(); successHandler.setRedirectStrategy(new
NoRedirectStrategy()); return successHandler; } /** * Disable Spring boot automatic filter registration.
*/ @Bean FilterRegistrationBean<?> disableAutoRegistration(final TokenAuthenticationFilter filter) {
final FilterRegistrationBean registration = new FilterRegistrationBean<>(filter);
registration.setEnabled(false); return registration; } @Bean AuthenticationEntryPoint
forbiddenEntryPoint() { return new HttpStatusEntryPoint(UNAUTHORIZED); } }
```

Repasemos cómo se configura Spring Security aquí:

- Las URL que comienzan con `/public/**` están excluidas de la seguridad, lo que significa que cualquier URL que empiece con `/public` no estará protegida,
- `TokenAuthenticationFilter` está registrado dentro de la cadena de filtro Spring Security muy temprano. Queremos atrapar cualquier token de autenticación que pase,
- La mayoría de los otros métodos de inicio de sesión como `formLogin` o `httpBasic` se han desactivado, ya que no estamos dispuestos a usarlos aquí (queremos usar nuestro propio sistema),
- Algunos códigos de placa de caldera para deshabilitar el registro automático de filtros, relacionados con Spring Boot.

Como puede ver, todo está unido en una **configuración de Java** que es **casi menos de 100 líneas ¡todo combinado** !

Ahora, vamos a configurar algunos Spring MVC `RestController` para poder iniciar sesión y cerrar sesión.

Controladores Spring MVC

PublicUsersController

El `PublicUsersController` permite a un usuario iniciar sesión en la aplicación:

```
 package com.maint; import com.maint.service.UserAuthenticationService; import lombok.AllArgsConstructor; import lombok.NonNull; import lombok.experimental.FieldDefaults; import org.springframework.web.bind.annotation.PostMapping; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RequestParam; import org.springframework.web.bind.annotation.RestController; import javax.servlet.http.HttpServletRequest; import static lombok.AccessLevel.PACKAGE; import static lombok.AccessLevel.PRIVATE; @RestController @RequestMapping("/public/users") @FieldDefaults(level = PRIVATE, makeFinal = true) @AllArgsConstructor(access = PACKAGE) final class PublicUsersController { @NonNull UserAuthenticationService authentication; @PostMapping("/login") String login( final HttpServletRequest request, @RequestParam("username") final String username, @RequestParam("password") final String password) { return authentication .login(username, password) .orElseThrow(() -> new RuntimeException("invalid login and/or password")); } }
```

Obviamente, el punto final de inicio de sesión debe ser de acceso público para permitir que el usuario inicie sesión. Simplemente delega el proceso de inicio de sesión en `UserAuthenticationService` .

SecuredUsersController

`SecuredUsersController` es, por definición, que permite al usuario realizar operaciones solo cuando `SecuredUsersController`sesión:

- Obtener el usuario actual Bean,
- Salir de la aplicación.

Aquí está el código:

```
package com.maint; import com.maint.service.User; import
com.maint.service.UserAuthenticationService; import lombok.AllArgsConstructor; import
lombok.NonNull; import lombok.experimental.FieldDefaults; import
org.springframework.security.core.annotation.AuthenticationPrincipal; import
org.springframework.web.bind.annotation.GetMapping; import
org.springframework.web.bind.annotation.RequestMapping; import
org.springframework.web.bind.annotation.RestController; import static
lombok.AccessLevel.PACKAGE; import static lombok.AccessLevel.PRIVATE; @RestController
@RequestMapping("/users") @FieldDefaults(level = PRIVATE, makeFinal = true)
@AllArgsConstructor(access = PACKAGE) final class SecuredUsersController { @NonNull
UserAuthenticationService authentication; @GetMapping("/current") User
getCurrent(@AuthenticationPrincipal final User user) { return user; } @GetMapping("/logout")
boolean logout(@AuthenticationPrincipal final User user) { authentication.logout(user); return true; }
}
```

Nuevamente, nada difícil aquí! Es hora de probar la aplicación. Para hacerlo, necesitamos crear una clase de arranque Spring Boot.

Aplicación Bootstrap

La clase de Application ubicada en el paquete raíz com.maint es responsable de com.maint la aplicación mediante Spring Boot:

```
package com.maint; import org.springframework.boot.SpringApplication; import
org.springframework.boot.autoconfigure.SpringBootApplication; @SpringBootApplication public
class Application { public static void main(String[] args) { SpringApplication.run(Application.class,
args); // NOSONAR } }
```

Vamos a ejecutar esta aplicación como una simple Java principal para iniciar el servidor. Finalmente, el servidor se ejecuta en el puerto 8080 de manera predeterminada. Como ya tengo un servidor ejecutándose en este puerto en mi máquina. Como resultado, configuré Spring Boot para que se ejecute en el puerto 8081 través de una application.yml :

```
server.port: 8081
```

Haga clic con el botón derecho en la clase Application y ejecútela. La aplicación debería estar ejecutándose en unos segundos:

```
. ____ _ __ _ _ /\\ / ___'_ __ _ _(_)_ __ __ _ \ \ \ \ ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \ \\/ ___)| |_)| | | | | || (_| | ) ) ) ) ' |____| .__|_| |_|_| |_\__, | / / / / ========|_|==============|___/=/_/_/_/ :: Spring Boot :: (v2.0.1.RELEASE) 2018-04-27 21:24:15.238 INFO 178584 --- [ main] com.maint.Application : Starting Application on desktop with PID 178584 (/home/ubuntu/git/spring-security-rest-api/target/classes started by ubuntu in /home/ubuntu/git/spring-security-rest-api) 2018-04-27 21:24:15.241 INFO 178584 --- [ main] com.maint.Application : No active profile set, falling back to default profiles: default 2018-04-27 21:24:15.300 INFO 178584 --- [ main] ConfigServletWebServerApplicationContext : Refreshing org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@70e8f8e: startup date [Fri Apr 27 21:24:15 CEST 2018]; root of context hierarchy 2018-04-27 21:24:16.619 INFO 178584 --- [ main] osbwembedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8081 (http) 2018-04-27 21:24:16.647 INFO 178584 --- [ main]
```

o.apache.catalina.core.StandardService : Starting service [Tomcat] 2018-04-27 21:24:16.647 INFO 178584 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.29 2018-04-27 21:24:16.658 INFO 178584 --- [ost-startStop-1] oacatalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: [/usr/java/packages/lib/amd64:/usr/lib64:/lib64:/lib:/usr/lib] 2018-04-27 21:24:16.748 INFO 178584 --- [ost-startStop-1] oaccC[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext 2018-04-27 21:24:16.748 INFO 178584 --- [ost-startStop-1] osweb.context.ContextLoader : Root WebApplicationContext: initialization completed in 1451 ms 2018-04-27 21:24:17.123 INFO 178584 --- [ost-startStop-1] osbwservlet.FilterRegistrationBean : Mapping filter: 'characterEncodingFilter' to: [/*] 2018-04-27 21:24:17.123 INFO 178584 --- [ost-startStop-1] osbwservlet.FilterRegistrationBean : Mapping filter: 'hiddenHttpMethodFilter' to: [/*] 2018-04-27 21:24:17.123 INFO 178584 --- [ost-startStop-1] osbwservlet.FilterRegistrationBean : Mapping filter: 'httpPutFormContentFilter' to: [/*] 2018-04-27 21:24:17.124 INFO 178584 --- [ost-startStop-1] osbwservlet.FilterRegistrationBean : Mapping filter: 'requestContextFilter' to: [/*] 2018-04-27 21:24:17.124 INFO 178584 --- [ost-startStop-1] .s.DelegatingFilterProxyRegistrationBean : Mapping filter: 'springSecurityFilterChain' to: [/*] 2018-04-27 21:24:17.124 INFO 178584 --- [ost-startStop-1] osboot.web.servlet.RegistrationBean : Filter tokenAuthenticationFilter was not registered (disabled) 2018-04-27 21:24:17.124 INFO 178584 --- [ost-startStop-1] osbwservlet.ServletRegistrationBean : Servlet dispatcherServlet mapped to [/] 2018-04-27 21:24:17.323 INFO 178584 --- [main] ossweb.DefaultSecurityFilterChain : Creating filter chain: OrRequestMatcher [requestMatchers=[Ant [pattern='/public/**']]], [] 2018-04-27 21:24:17.398 INFO 178584 --- [main] ossweb.DefaultSecurityFilterChain : Creating filter chain: org.springframework.security.web.util.matcher.AnyRequestMatcher@1, [org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter@325f7fa9, org.springframework.security.web.context.SecurityContextPersistenceFilter@894858, org.springframework.security.web.header.HeaderWriterFilter@60297f36, org.springframework.security.web.savedrequest.RequestCacheAwareFilter@737edcfa, org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@6f330eb9, com.maint.TokenAuthenticationFilter@3d11d526, org.springframework.security.web.authentication.AnonymousAuthenticationFilter@11ce2e22, org.springframework.security.web.session.SessionManagementFilter@2ba45490, org.springframework.security.web.access.ExceptionTranslationFilter@64711bf2, org.springframework.security.web.access.intercept.FilterSecurityInterceptor@31be6b49] 2018-04-27 21:24:17.534 INFO 178584 --- [main] oswshandler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler] 2018-04-27 21:24:17.750 INFO 178584 --- [main] swsmmaRequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@70e8f8e: startup date [Fri Apr 27 21:24:15 CEST 2018]; root of context hierarchy 2018-04-27 21:24:17.950 INFO 178584 --- [main] swsmmaRequestMappingHandlerMapping : Mapped "{[/public/users/login],methods=[POST]}" onto java.lang.String com.maint.PublicUsersController.login(javax.servlet.http.HttpServletRequest,java.lang.String,java.lang.String) 2018-04-27 21:24:17.956 INFO 178584 --- [main] swsmmaRequestMappingHandlerMapping : Mapped "{[/users/logout],methods=[GET]}" onto boolean com.maint.SecuredUsersController.logout(com.maint.User) 2018-04-27 21:24:17.956 INFO 178584 --- [main] swsmmaRequestMappingHandlerMapping : Mapped "{[/users/current],methods=[GET]}" onto com.maint.User com.maint.SecuredUsersController.getCurrent(com.maint.User) 2018-04-27 21:24:17.959 INFO 178584 --- [main] swsmmaRequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.springframework.web.servlet.ModelAndView org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController.errorHtml(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse) 2018-04-27 21:24:17.960 INFO 178584 --- [main] swsmmaRequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController.error(javax.servlet.http.HttpServletRequest) 2018-04-27 21:24:18.024 INFO 178584 --- [main] oswshandler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class

```
org.springframework.web.servlet.resource.ResourceHttpRequestHandler] 2018-04-27 21:24:18.024 INFO 178584 --- [ main] oswshandler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler] 2018-04-27 21:24:18.204 INFO 178584 --- [ main] osjeaAnnotationMBeanExporter : Registering beans for JMX exposure on startup 2018-04-27 21:24:18.241 INFO 178584 --- [ main] osbwembedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8081 (http) with context path '' 2018-04-27 21:24:18.248 INFO 178584 --- [ main] com.maint.Application : Started Application in 3.439 seconds (JVM running for 3.835)
```

¡Estupendo! El servidor está en funcionamiento, listo para ser utilizado. Ahora realicemos algunas solicitudes usando `curl` .

Probando la aplicación

Primero, inicie sesión en la API REST:

```
ubuntu@ubuntu-Aspire-V3-772:~$ curl -XPOST -d 'username=john&password=smith' http://localhost:8081/public/users/login b856850e-1ad4-456d-b5ca-1c2bfc355e5
```

Al enviar una solicitud de formulario codificada en la url al punto final, se devuelve como se esperaba un `UUID` aleatorio. Ahora, usemos el UUID en una solicitud posterior para recuperar al usuario actual:

```
ubuntu@ubuntu-Aspire-V3-772:~$ curl -H 'Authorization: Bearer b856850e-1ad4-456d-b5ca-1c2bfc355e5e' http://localhost:8081/users/current {"id":"b856850e-1ad4-456d-b5ca-1c2bfc355e5e","username":"john","enabled":true}
```

¡Bonito! Iniciamos sesión en el sistema y pudimos recuperar al usuario actual en formato Json. Por defecto, Spring Boot usa **Jackson Json** API para serializar beans en Json.

Vamos a cerrar la sesión del sistema:

```
ubuntu@ubuntu-Aspire-V3-772:~/git/site$ curl -H 'Authorization: Bearer b856850e-1ad4-456d-b5ca-1c2bfc355e5e' http://localhost:8081/users/logout true
```

Si intentamos volver a obtener el usuario actual con el mismo token de autenticación, deberíamos recibir un error:

```
ubuntu@ubuntu-Aspire-V3-772:~/git/site$ curl -H 'Authorization: Bearer b856850e-1ad4-456d-b5ca-1c2bfc355e5e' http://localhost:8081/users/current {"timestamp":1516184750678,"status":401,"error":"Unauthorized","message":"Authentication Failed: Bad credentials","path":"/users/current"}
```

Como se esperaba, el servidor denegó el acceso al recurso seguro porque el token de autenticación se ha revocado previamente.

Ahora se está preguntando: **¿Cómo pueden mis usuarios crear una cuenta?**

registro de usuario

El registro de nuevos usuarios también es bastante simple. Todo lo que tiene que hacer es agregar un punto final al `PublicUsersController` . Aquí hay un código de ejemplo extraído de nuestro propio servidor.

La interfaz del UserRegistrationService :

```
package com.maint; import com.maint.service.User; import java.util.Optional; public interface UserRegistrationService { /** * @return true if registration is enabled */ boolean isEnabled(); /** * In the case the username already exists, it returns * the already registered user. * * @throws IllegalArgumentException if username is empty or already exists * @throws IllegalStateException if username is a disposable mail */ User register(String username, Optional<String> password); }
```

El PublicUsersController mejorado:

```
package com.maint; import lombok.AllArgsConstructor; import lombok.NonNull; import lombok.experimental.FieldDefaults; import org.springframework.web.bind.annotation.PostMapping; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RequestParam; import org.springframework.web.bind.annotation.RestController; import javax.servlet.http.HttpServletRequest; import static com.google.common.base.Strings.emptyToNull; import static java.util.Optional.ofNullable; import static lombok.AccessLevel.PACKAGE; import static lombok.AccessLevel.PRIVATE; @RestController @RequestMapping("/public/users") @FieldDefaults(level = PRIVATE, makeFinal = true) @AllArgsConstructor(access = PACKAGE) final class PublicUsersController { @NonNull UserAuthenticationService authentication; @NonNull UserRegistrationService registration; @PostMapping("/register") String register( final HttpServletRequest request, @RequestParam("username") final String username, @RequestParam(value = "password", required = false) final String password) { registration.register(username, ofNullable(emptyToNull(password))); return authentication.login(username, password).orElseThrow(RuntimeException::new); } @PostMapping("/login") String login( final HttpServletRequest request, @RequestParam("username") final String username, @RequestParam("password") final String password) { return authentication .login(username, password) .orElseThrow(() -> new RuntimeException("invalid login and/or password")); } }
```

Y un ejemplo de UserRegistrationService que no hace nada (pero puede ser UserRegistrationService con su propia lógica):

```
package com.maint; import org.springframework.stereotype.Service; import java.util.Optional; import java.util.UUID; @Service final class NoopRegistrationService implements UserRegistrationService { @Override public boolean isEnabled() { return true; } @Override public User register(final String username, final Optional<String> password) { final String token = UUID.randomUUID().toString(); return User .builder() .id(token) .username(username) .password(password.orElse("12345")) // Unsecure .build(); } }
```

Este servicio es responsable de crear una cuenta dentro del sistema. Puede almacenar la cuenta en una base de datos relacional, o mantenerlos en la memoria, por ejemplo. La implementación depende de tus necesidades.

Ultimas palabras

Espero que esta capa de seguridad esqueleto lista para usar te permita construir una API de descanso segura usando Spring Security.Nos llevó un tiempo descubrir cómo funciona Spring Security y cómo crear esta configuración.

Pensamos que sería una buena idea compartir este tutorial para ayudarlo a evitar pasar semanas jugando con Spring Security (como lo hicimos nosotros).