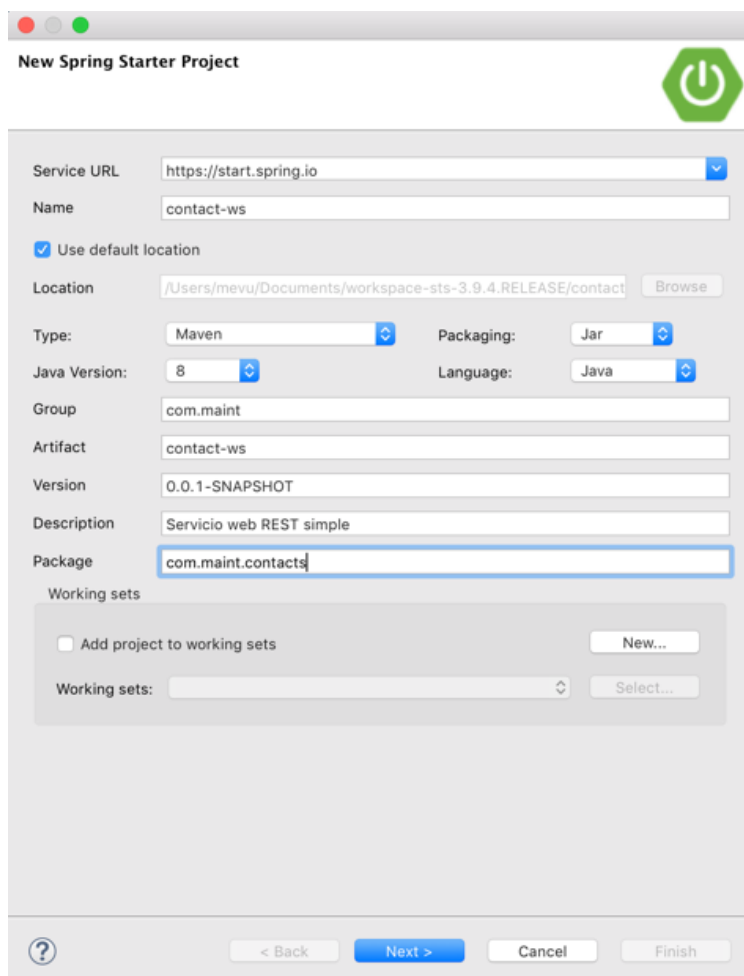


LABORATORIO SPRING BOOT REST

Este laboratorio explica cómo crear un API Rest **exponiendo datos como JSON**.

Utilizando Spring Tool Suite (STS) que es un conjunto de plugins para Eclipse, especialmente diseñados para hacernos muy productivos al desarrollar una aplicación de Spring. Usaremos esta opción que básicamente es una fachada pues STS se conecta a Spring Boot Initializr para hacer lo mismo que haríamos visitando el sitio web, sólo que importa automáticamente el proyecto en nuestro IDE.

Para crear el proyecto, en STS selecciona el menú “*File/New/Spring Starter Project...*” y en la ventana del asistente, ingresa los datos de tu proyecto, por ejemplo en nuestro caso, se verían así:



The screenshot shows the 'New Spring Starter Project' dialog box in Spring Tool Suite. The dialog has a title bar with a green power icon. The main area contains several input fields and checkboxes. The 'Service URL' field is set to 'https://start.spring.io'. The 'Name' field is 'contact-ws'. The 'Use default location' checkbox is checked. The 'Location' field shows a file path: '/Users/mevu/Documents/workspace-sts-3.9.4.RELEASE/contact'. The 'Type' is 'Maven', 'Packaging' is 'Jar', 'Java Version' is '8', and 'Language' is 'Java'. The 'Group' is 'com.maint', 'Artifact' is 'contact-ws', 'Version' is '0.0.1-SNAPSHOT', 'Description' is 'Servicio web REST simple', and 'Package' is 'com.maint.contacts'. At the bottom, there is a 'Working sets' section with an unchecked 'Add project to working sets' checkbox and a 'New...' button. Below that is a 'Working sets:' label with a dropdown menu and a 'Select...' button. At the very bottom, there are four buttons: '< Back', 'Next >', 'Cancel', and 'Finish'. The 'Next >' button is highlighted in blue.

En la siguiente ventana selecciona las librerías **Web** (clases para páginas y servicios web), **Lombok** y **JACKSON**. Presiona el botón “Finish” para completar.

Unos pocos segundos o minutos después, tendrás el proyecto listo en tu IDE. Pruébalo ejecutándolo como una aplicación Spring Boot: selecciona el proyecto, click derecho y pulsa el menú “*Run as.../Spring Boot Application*”. Si todo va bien, el proyecto debería iniciar sin errores en el log.

Agregando la capa DTO

Agregando la capa DTO: Person Bean

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

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

¡Gracias a la anotación **@Data** de lombok, el bean es bastante simple! Lombok se encarga de generar el constructor, los métodos setters, getters y define todos los campos como **private final**.

Vamos a enviar este bean a través de la capa de interface 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.

Editamos `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 el proyecto y haga clic derecho sobre él. Luego seleccione, **Run DemoApplication.main()**. Debería iniciar la aplicación web.

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"}
```

Si no dispones de curl en tu sistema, utiliza postman o un navegador para probar si funciona.

¡Felicitaciones, acabas de construir **un API Rest 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.

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 ..... 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.

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.

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 **PersonService**,
- **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**.

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.