28 de marzo de 2019

Pruebe sus aplicaciones Spring Boot con JUnit 5

Joy Foster

En esta publicación, explicará cómo crear una aplicación Spring Boot simple y probarla con Junit 5. Una aplicación sin probar es la proverbial Caja de Pandora. ¿De qué sirve su aplicación si no sabe que funcionará bajo ninguna condición? Agregar un conjunto de pruebas genera confianza en que su aplicación puede manejar cualquier cosa que se le presente. Al crear sus pruebas, es importante utilizar un conjunto de herramientas moderno y completo. El uso de un marco moderno garantiza que pueda mantenerse al día con los cambios dentro de su idioma y bibliotecas. Un conjunto completo de herramientas garantiza que pueda probar adecuadamente todas las áreas de su aplicación sin la carga de escribir sus propias utilidades de prueba. JUnit 5 maneja bien ambos requisitos.

¡La aplicación utilizada para esta publicación será una API REST básica con puntos finales para calcular algunas cosas sobre el cumpleaños de una persona! Hay tres puntos finales POST que podrá usar para determinar el día de la semana, el signo astrológico o el signo del zodiaco chino para un cumpleaños pasado. Esta API REST estará asegurada con OAuth 2.0 y Okta. Una vez que hayamos construido la API, revisaremos la unidad probando el código con JUnit 5 y revisaremos la cobertura de nuestras pruebas JUnit.

La principal ventaja de usar Spring Framework es la capacidad de inyectar sus dependencias, lo que hace que sea mucho más fácil intercambiar implementaciones para varios propósitos, pero no menos importante para pruebas unitarias. ¡Spring Boot lo hace aún más fácil al permitirle hacer gran parte de la inyección de dependencia con anotaciones en lugar de tener que molestarse con un `applicationContext.xml` archivo complicado !

> NOTA: Para esta publicación, usaré Eclipse, ya que es mi IDE preferido. Si también usa Eclipse, necesitará instalar una versión de Oxygen o superior para poder incluir el soporte de prueba JUnit 5 (Júpiter).

Cree una aplicación Spring Boot para probar con JUnit 5

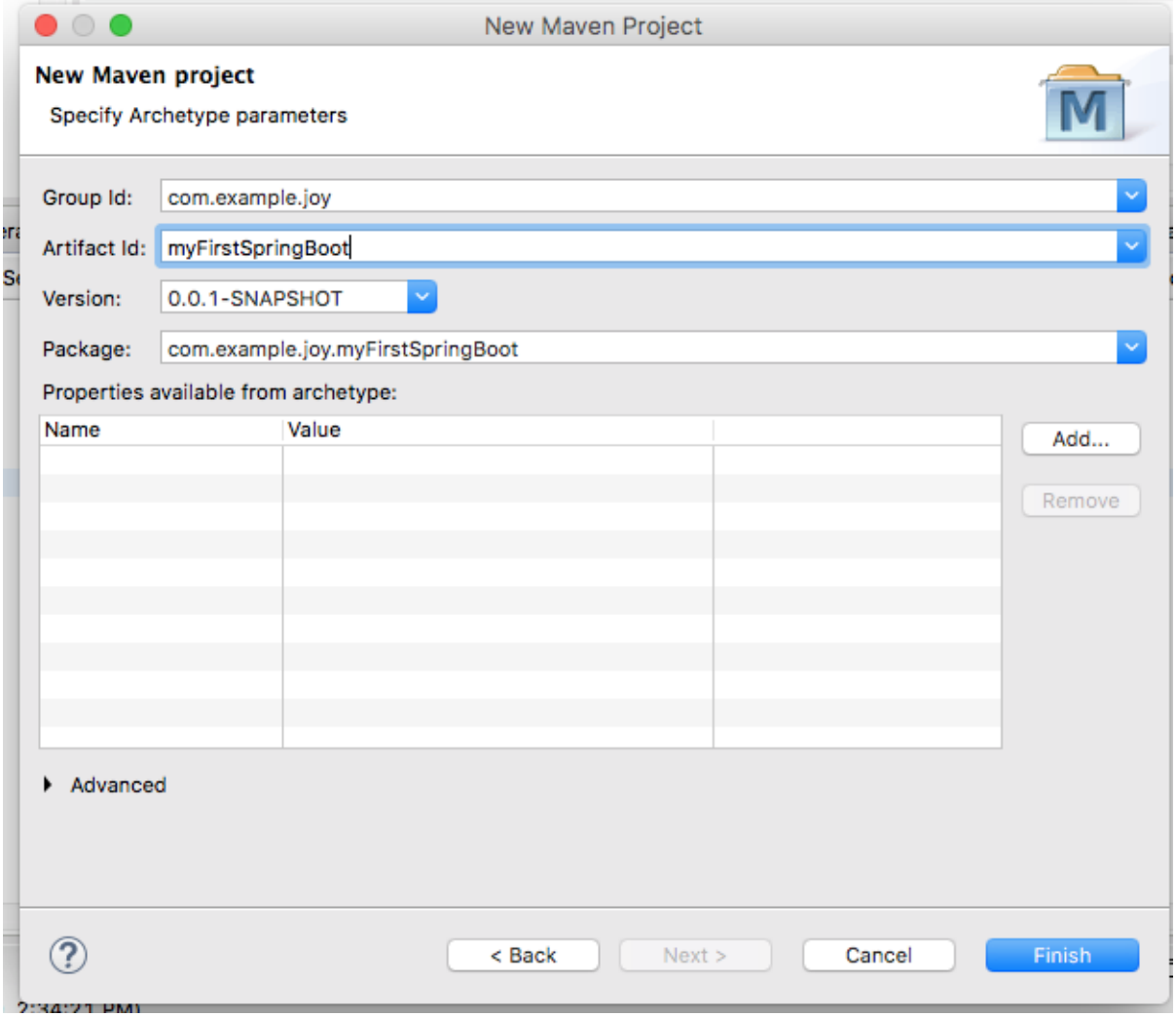
Para este tutorial, la estructura del proyecto es la que se muestra a continuación. Solo hablaré de los nombres de los archivos, pero puede encontrar su ruta utilizando la estructura a continuación, mirando la fuente completa o prestando atención al paquete.

Para comenzar, creará un proyecto Spring Boot desde cero.

NOTA: Los siguientes pasos son para Eclipse. Si usa un IDE diferente, es probable que haya pasos equivalentes. Opcionalmente, puede crear su propia estructura de directorio de proyectos y escribir el `pom.xml` archivo final en cualquier editor de texto que desee.

Cree un nuevo proyecto Maven desde **Archivo** > **Nuevo** menú. Seleccione la ubicación de su nuevo proyecto y haga clic en siguiente dos veces y luego complete la identificación del grupo, la identificación del artefacto y la versión de su aplicación. Para este ejemplo, utilicé las siguientes opciones:

- Identificación del grupo: `com.example.joy`
- Id. De artefacto: `myFirstSpringBoot`
- Versión: `0.0.1-SNAPSHOT`



> SUGERENCIA: si Maven es nuevo para usted y no está claro cómo elegir su ID de grupo, ID de artefacto o versión, revise las convenciones de nomenclatura de Maven .

Cuando termine, esto producirá un `pom.xml` archivo similar al siguiente:

```
<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.(
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.3.RELEASE</version>
    </parent>
    <groupId>com.example.joy</groupId>
    <artifactId>myFirstSpringBoot</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</project>
```

A continuación, querrá actualizar el `pom.xml` con algunas configuraciones básicas y dependencias para que se vea de la siguiente manera (agregue todo después de la versión):

```
<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.(
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.3.RELEASE</version>
    </parent>
    <groupId>com.example.joy</groupId>
    <artifactId>myFirstSpringBoot</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <properties>
        <java.version>1.8</java.version>
        <spring.boot.version>2.1.3.RELEASE</spring.boot.version>
    </properties>

    <dependencies>
        <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>
           <exclusions>
                <exclusion>
                    <groupId>junit</groupId>
                    <artifactId>junit</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
             <groupId>org.junit.jupiter</groupId>
             <artifactId>junit-jupiter-engine</artifactId>
             <scope>test</scope>
         </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Tenga en cuenta que debe excluir el JUnit predeterminado de la dependencia spring-boot-starter-test. La `junit-jupiter-engine` dependencia es para JUnit 5.

Cree una API REST de Java con Spring Boot para su prueba JUnit 5

Comencemos con el archivo de la aplicación principal, que es el punto de entrada para iniciar la API de Java. Este es un archivo llamado `SpringBootRestApiApplication.java` que se ve así:

```
package com.example.joy.myFirstSpringBoot;

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

@SpringBootApplication(scanBasePackages = {"com.example.joy"})
public class SpringBootRestApiApplication {

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

La `SpringBootApplication` anotación le dice a la aplicación que debe admitir la configuración automática, el escaneo de componentes (del `com.example.joy` paquete y todo lo que contiene) y el registro de beans.

```
@SpringBootApplication(scanBasePackages = {"com.example.joy"})
```

Esta línea lanza la aplicación REST API:

```
SpringApplication.run(SpringBootRestApiApplication.class, args);
```

`BirthdayService.java` es la interfaz para el servicio de cumpleaños. Es bastante sencillo, definiendo que hay cuatro funciones auxiliares disponibles.

```
package com.example.joy.myFirstSpringBoot.services;

import java.time.LocalDate;

public interface BirthdayService {
    LocalDate getValidBirthday(String birthdayString) ;

    String getBirthDOW(LocalDate birthday);

    String getChineseZodiac(LocalDate birthday);

    String getStarSign(LocalDate birthday) ;
}
```

`BirthdayInfoController.java` maneja las tres solicitudes de publicación para obtener información de cumpleaños. Se parece a esto:

```
package com.example.joy.myFirstSpringBoot.controllers;

import java.time.LocalDate;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
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 com.example.joy.myFirstSpringBoot.services.BirthdayService;

@RestController
@RequestMapping("/birthday")
public class BirthdayInfoController {
    private final BirthdayService birthdayService;

    public BirthdayInfoController(BirthdayService birthdayService) {
        this.birthdayService = birthdayService;
    }

    @PostMapping("/dayOfWeek")
    public String getDayOfWeek(@RequestBody String birthdayString) {
        LocalDate birthday = birthdayService.getValidBirthday(birthdayString);
        String dow = birthdayService.getBirthDOW(birthday);
        return dow;
    }

    @PostMapping("/chineseZodiac")
    public String getChineseZodiac(@RequestBody String birthdayString) {
        LocalDate birthday = birthdayService.getValidBirthday(birthdayString);
        String sign = birthdayService.getChineseZodiac(birthday);
        return sign;
    }

    @PostMapping("/starSign")
    public String getStarSign(@RequestBody String birthdayString) {
        LocalDate birthday = birthdayService.getValidBirthday(birthdayString);
        String sign = birthdayService.getStarSign(birthday);
        return sign;
    }

    @ExceptionHandler(RuntimeException.class)
    public final ResponseEntity<Exception> handleAllExceptions(RuntimeException ex) {
        return new ResponseEntity<Exception>(ex, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Anotaciones de Spring MVC

Primero, notará las siguientes anotaciones cerca de la parte superior. La `@RestController` anotación le dice al sistema que este archivo es un "controlador de API de reposo", lo que simplemente significa que contiene una colección de puntos finales de API. También podría usar la `@Controller` anotación, pero significa que tendría que agregar más código repetitivo para convertir las respuestas a una respuesta HTTP OK en lugar de simplemente devolver los valores. La segunda línea le dice que todos los puntos finales tienen el prefijo "/ cumpleaños" en la ruta. Mostraré una ruta completa para un punto final más adelante.

```
@RestController
@RequestMapping("/birthday")
```

Inyección de constructor con resorte

A continuación, verá una variable de clase para `birthdayService` (de tipo `BirthdayService`). Esta variable se inicializa en el constructor de la clase. Desde Spring Framework 4.3, ya no necesita especificar `@Autowired` cuándo se usa la inyección de constructor. Esto tendrá el efecto de cargar una instancia de la `BasicBirthdayService` clase, que veremos en breve.

```
private final BirthdayService birthdayService;

public BirthdayInfoController(BirthdayService birthdayService){
    this.birthdayService = birthdayService;
}
```

Manejo de POSTs a su API

Las siguientes métodos (`getDayOfWeek` , `getChineseZodiac` y `getStarSign`) son donde se pone jugosa. Son los manejadores de los tres puntos finales diferentes. Cada uno comienza con una `@PostMapping` anotación que le indica al sistema la ruta del punto final. En este caso, la ruta sería `/birthday/dayOfWeek` (el `/birthday` prefijo proviene de la `@RequestMapping` anotación anterior).

```
@PostMapping("/dayOfWeek")
```

Cada método de punto final hace lo siguiente:

- Acepta una `birthdayString` cadena.
- Utiliza `birthdayService` para verificar si la `birthdayString` cadena se puede convertir en un `LocalDate` objeto. Si es así, devuelve el `LocalDate` objeto para usar en el código posterior. De lo contrario, arroja un error (ver Manejo de errores a continuación).
- Obtiene el valor que se devolverá (día de la semana, signo del zodiaco chino o signo astrológico) de `birthdayService` .
- Devuelve la cadena (que hará que responda con un HTTP OK debajo del capó).

Manejo de errores en un RestController

Por último, hay un método para el manejo de errores:

```
@ExceptionHandler(RuntimeException.class)
public final ResponseEntity<Exception> handleAllExceptions(RuntimeException ex) {
    return new ResponseEntity<Exception>(ex, HttpStatus.INTERNAL_SERVER_ERROR);
}
```

Aquí, la `@ExceptionHandler` anotación le dice que capture cualquier instancia de RuntimeException dentro de las funciones de punto final y que devuelva una respuesta 500.

`BasicBirthdayService.java` maneja la mayor parte de la lógica comercial real en esta aplicación. Es la clase que tiene una función para verificar si una cadena de cumpleaños es válida, así como funciones que calculan el día de la semana, el zodiaco chino y el signo astrológico de un cumpleaños.

```
package com.example.joy.myFirstSpringBoot.services;

import org.springframework.stereotype.Service;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

@Service
public class BasicBirthdayService implements BirthdayService {
    private static DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

    @Override
    public LocalDate getValidBirthday(String birthdayString) {
        if (birthdayString == null) {
```

```
            throw new RuntimeException("Must include birthday");
        }
        try {
            LocalDate birthdate = LocalDate.parse(birthdayString, formatter);
            return birthdate;
        } catch (Exception e) {
            throw new RuntimeException("Must include valid birthday in yyyy-MM-dd format");
        }

    }

    @Override
    public String getBirthDOW(LocalDate birthday) {
        return birthday.getDayOfWeek().toString();
    }

    @Override
    public String getChineseZodiac(LocalDate birthday) {
        int year = birthday.getYear();
        switch (year % 12) {
            case 0:
                return "Monkey";
            case 1:
                return "Rooster";
            case 2:
                return "Dog";
            case 3:
                return "Pig";
            case 4:
                return "Rat";
            case 5:
                return "Ox";
            case 6:
                return "Tiger";
            case 7:
                return "Rabbit";
            case 8:
                return "Dragon";
            case 9:
                return "Snake";
            case 10:
                return "Horse";
            case 11:
                return "Sheep";
        }

        return "";
    }

    @Override
    public String getStarSign(LocalDate birthday) {
        int day = birthday.getDayOfMonth();
        int month = birthday.getMonthValue();

        if (month == 12 && day >= 22 || month == 1 && day < 20) {
            return "Capricorn";
        } else if (month == 1 && day >= 20 || month == 2 && day < 19) {
            return "Aquarius";
```

```
            } else if (month == 2 && day >= 19 || month == 3 && day < 21) {
                return "Pisces";
            } else if (month == 3 && day >= 21 || month == 4 && day < 20) {
                return "Aries";
            } else if (month == 4 && day >= 20 || month == 5 && day < 21) {
                return "taurus";
            } else if (month == 5 && day >= 21 || month == 6 && day < 21) {
                return "Gemini";
            } else if (month == 6 && day >= 21 || month == 7 && day < 23) {
                return "Cancer";
            } else if (month == 7 && day >= 23 || month == 8 && day < 23) {
                return "Leo";
            } else if (month == 8 && day >= 23 || month == 9 && day < 23) {
                return "Virgo";
            } else if (month == 9 && day >= 23 || month == 10 && day < 23) {
                return "Libra";
            } else if (month == 10 && day >= 23 || month == 11 && day < 22) {
                return "Scorpio";
            } else if (month == 11 && day >= 22 || month == 12 && day < 22) {
                return "Sagittarius";
            }
            return "";
        }
    }
```

La `@Service` anotación es lo que usa para inyectar esto en el `BirthdayInfoController` constructor. Dado que esta clase implementa la `BirthdayService` interfaz y está dentro de la ruta de exploración de la aplicación, Spring la encontrará, la inicializará y la inyectará en el constructor `BirthdayInfoController` .

El resto de la clase es simplemente un conjunto de funciones que especifican la lógica de negocios llamada desde `BirthdayInfoController` .

Ejecute su API REST HTTP Spring básica

En este punto, debe tener una API que funcione. En Eclipse, simplemente haga clic derecho en el `SpringBootRestApiApplication` archivo, haga clic en **Ejecutar como** > **aplicación Java** y se iniciará. Para llegar a los puntos finales, puede usar curl para ejecutar estos comandos:

Día de la semana:

Solicitud:

```
curl -X POST \
  http://localhost:8080/birthday/dayOfWeek \
  -H 'Content-Type: text/plain' \
  -H 'accept: text/plain' \
  -d 2005-03-09
```

Respuesta:

```
WEDNESDAY
```

Zodiaco chino:

Solicitud:

```
curl -X POST \
  http://localhost:8080/birthday/chineseZodiac \
  -H 'Content-Type: text/plain' \
  -H 'accept: text/plain' \
  -d 2005-03-09
```

Respuesta:

```
Rooster
```

Signo Astrológico:

Solicitud:

```
curl -X POST \
  http://localhost:8080/birthday/starSign \
  -H 'Content-Type: text/plain' \
  -H 'accept: text/plain' \
  -d 2005-03-09
```

Respuesta:

```
Pisces
```

Asegure su aplicación Java JUnit 5 con OAuth 2.0

Ahora que tenemos la API básica creada, ¡hagámosla segura! Puede hacerlo rápidamente utilizando la verificación de token OAuth 2.0 de Okta. ¿Por qué okta? Okta es un proveedor de identidad que facilita agregar autenticación y autorización a sus aplicaciones. Siempre está encendido y los amigos no dejan que sus amigos escriban autenticación.

Después de integrar Okta, la API requerirá que el usuario pase un token de acceso OAuth 2.0. Okta verificará este token para verificar su validez y autenticidad.

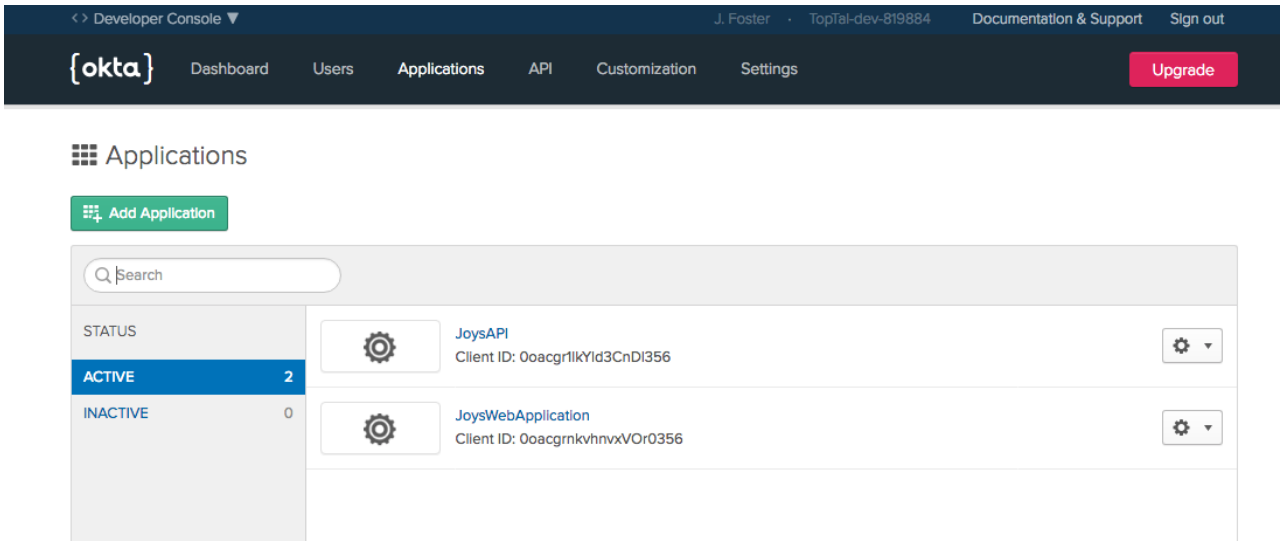
Para hacer esto, necesitará tener una "Aplicación de servicio" configurada con Okta, agregar el iniciador Okta Spring Boot al código Java y tener una forma de generar tokens para esta aplicación. ¡Empecemos!

Crear una aplicación OpenID Connect

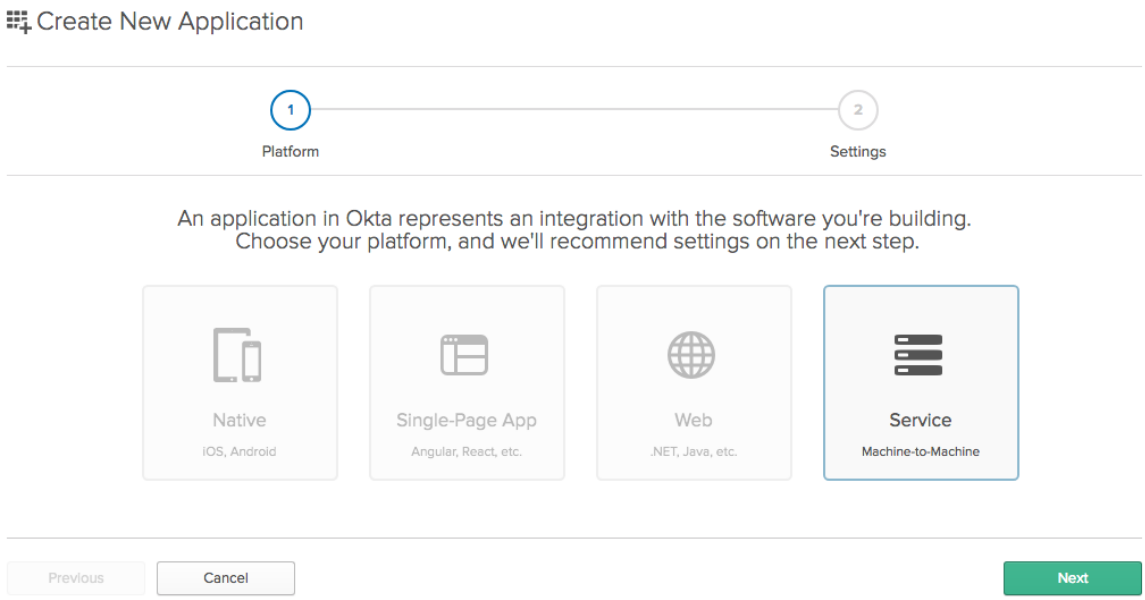
Deberá crear una aplicación OpenID Connect en Okta para obtener sus valores únicos para realizar la autenticación.

Para hacer esto, primero debe iniciar sesión en su cuenta de desarrollador Okta (o registrarse si no tiene una cuenta).

Una vez en el panel de Okta Developer, haga clic en la pestaña **Aplicaciones** en la parte superior de la pantalla y luego haga clic en el botón **Agregar aplicación** .



Verá la siguiente pantalla. Haga clic en el mosaico **Servicio** y luego haga clic en **Siguiente** .



La siguiente pantalla le pedirá un nombre para su aplicación. Seleccione algo que tenga sentido y haga clic en **Listo** .

Se creará la aplicación y se le mostrará una pantalla que muestra las credenciales de su cliente, incluida una ID de cliente y un secreto de cliente. Puede volver a esta pantalla en cualquier momento yendo a la pestaña **Aplicaciones** y luego haciendo clic en el nombre de la aplicación que acaba de crear.

Integre la autenticación segura en su código

Solo hay unos pocos pasos para agregar autenticación a su aplicación.

Cree un archivo llamado `src/main/resources/application.properties` con los siguientes contenidos:

```
okta.oauth2.issuer=https://{yourOktaDomain}/oauth2/default
okta.oauth2.clientId={clientId}
okta.oauth2.clientSecret={clientSecret}
okta.oauth2.scope=openid
```

Reemplace los elementos dentro `{...}` con sus valores. Las `{clientId}` y `{clientSecret}` los valores vendrán de la aplicación que acaba de crear. Una vez que haya configurado el contexto de la aplicación, todo lo que necesita hacer es agregar una sola dependencia a su `pom.xml` archivo y crear un archivo Java más.

Para las dependencias, agregue el iniciador Okta Spring Boot al `pom.xml` archivo en la sección de dependencias:

```
<!-- security - begin -->
<dependency>
    <groupId>com.okta.spring</groupId>
    <artifactId>okta-spring-boot-starter</artifactId>
    <version>1.1.0</version>
</dependency>
<!-- security - end -->
```

Y el último paso es actualizar el `SpringBootRestApiApplication` para incluir una subclase de configuración estática llamada `OktaOAuth2WebSecurityConfigurerAdapter` . Su `SpringBootRestApiApplication.java` archivo debe actualizarse para tener este aspecto:

```
package com.example.joy.myFirstSpringBoot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAd

@SpringBootApplication(scanBasePackages = {"com.example.joy"})
public class SpringBootRestApiApplication {

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

    @Configuration
    static class OktaOAuth2WebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests().anyRequest().authenticated()
                .and().oauth2ResourceServer().jwt();
        }
    }
}
```

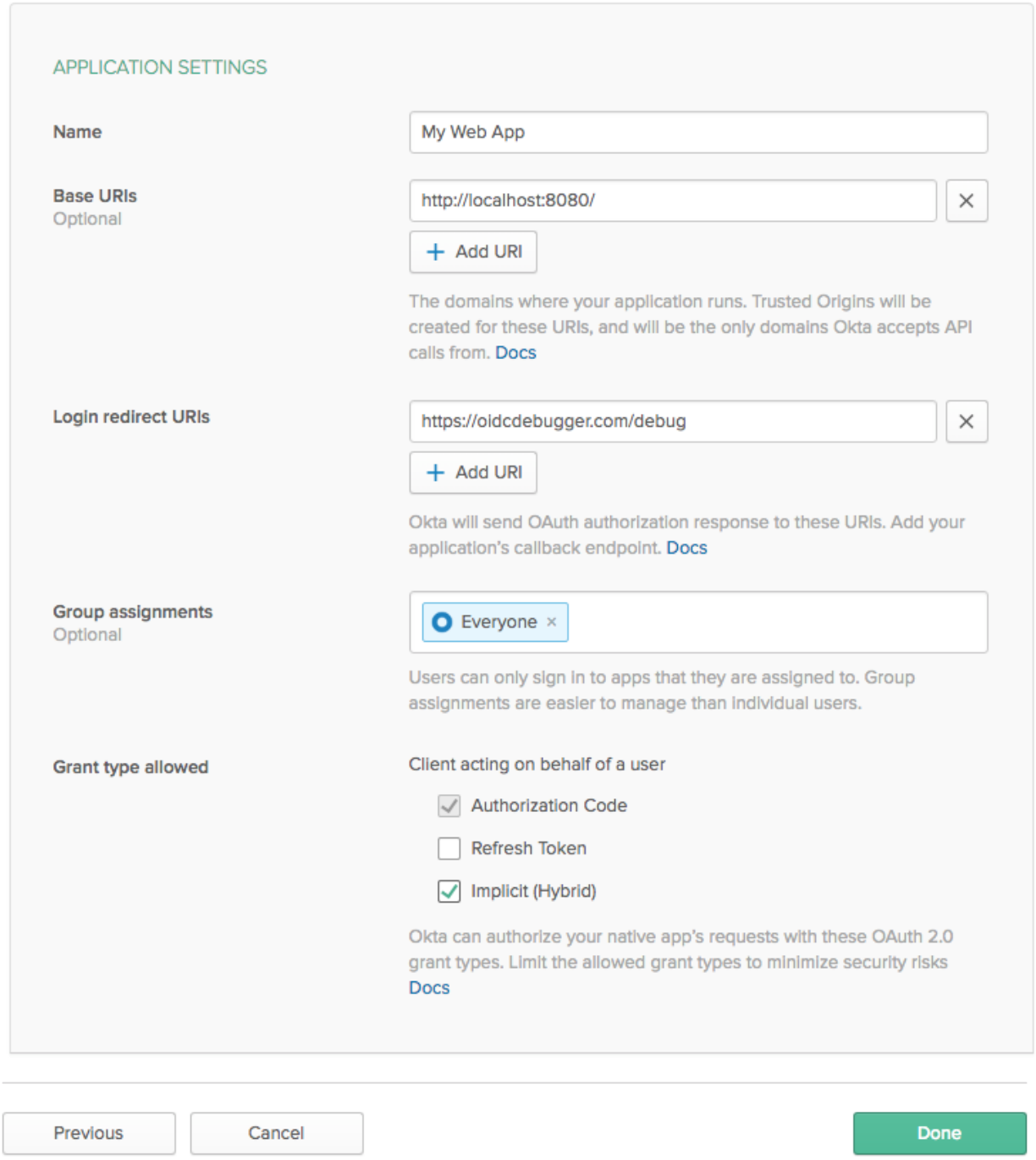
Genere un token para probar su aplicación Spring Boot con JUnit 5

Para realizar la prueba, deberá poder generar un token válido. Normalmente, la aplicación cliente sería responsable de generar los tokens que usaría para la autenticación en la API. Sin embargo, dado que no tiene una aplicación cliente, necesita una forma de generar tokens para probar la aplicación.

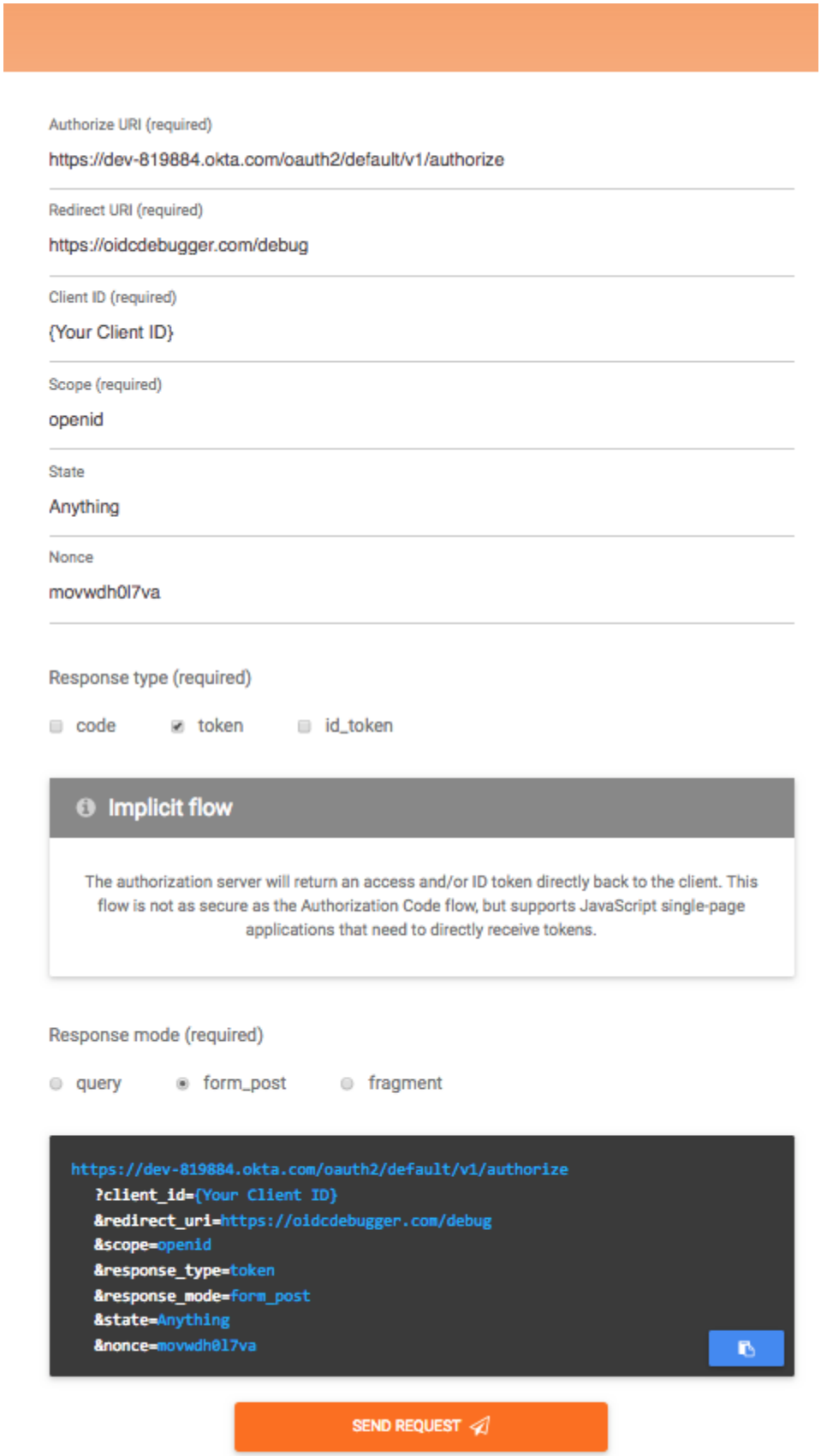
Una manera fácil de lograr un token es generar uno usando OpenID Connect Debugger . Primero, sin embargo, debe tener una configuración de aplicación **web de** cliente en Okta para usar con el flujo implícito de OpenID Connect.

Para hacer esto, regrese a la consola de desarrollador de Okta y seleccione **Aplicaciones** > **Agregar aplicación** , pero esta vez, seleccione el mosaico **web** .

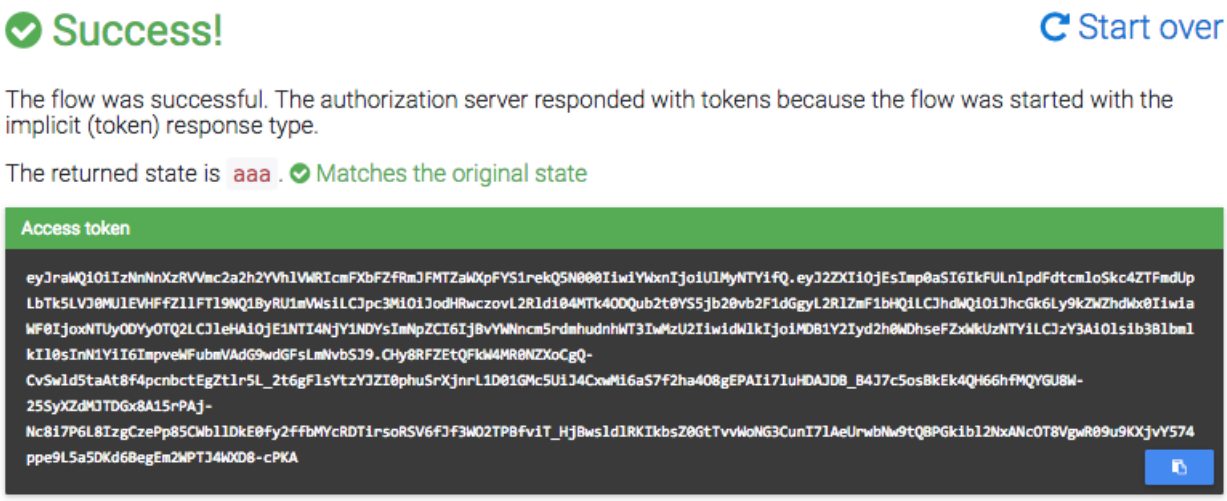
En la siguiente pantalla, deberá completar alguna información. Establezca el nombre en algo que recordará como su aplicación web. Establezca el campo **URI de redireccionamiento de inicio de sesión** en `https://oidcdebugger.com/debug` y **Tipo de concesión permitido** a **Híbrido** . Haga clic en **Listo** y copie la ID del cliente para el siguiente paso.



Ahora, navegue al sitio web del depurador OpenID Connect , complete el formulario como se muestra en la imagen a continuación (no olvide completar el ID de cliente para su aplicación **web** Okta recientemente creada). El `state` campo debe completarse pero puede contener cualquier carácter. El URI de autorización debe comenzar con la URL de su dominio (que se encuentra en el panel de Okta):



Envíe el formulario para comenzar el proceso de autenticación. Recibirás un formulario de inicio de sesión Okta si no has iniciado sesión o verás la siguiente pantalla con tu token personalizado.



NOTA: El token será válido durante una hora, por lo que es posible que tenga que repetir el proceso si está probando durante mucho tiempo.

Pruebe su aplicación Spring Boot segura con JUnit 5

Ahora debería tener una API **segura que** funcione . ¡Vamos a verlo en acción! En Eclipse, solo haga clic derecho en el `SpringBootRestApiApplication` archivo, haga clic en **Ejecutar como** > **aplicación Java** , y lo iniciará. Para llegar a los puntos finales, puede usar curl para ejecutar estos comandos, pero asegúrese de incluir el nuevo encabezado que contiene su token. Reemplace `{token goes here}` con el token real de OpenID Connect:

Día de la semana:

Solicitud:

```
curl -X POST \
  http://localhost:8080/birthday/dayOfWeek \
  -H 'Authorization: Bearer {token goes here}' \
  -H 'Content-Type: text/plain' \
  -H 'accept: text/plain' \
  -d 2005-03-09
```

Respuesta:

```
WEDNESDAY
```

Zodiaco chino:

Solicitud:

```
curl -X POST \
  http://localhost:8080/birthday/chineseZodiac \
  -H 'Authorization: Bearer {token goes here}' \
  -H 'Content-Type: text/plain' \
  -H 'accept: text/plain' \
  -d 2005-03-09
```

Respuesta:

```
Rooster
```

Signo Astrológico:

Solicitud:

```
curl -X POST \
  http://localhost:8080/birthday/starSign \
  -H 'Authorization: Bearer {token goes here}' \
  -H 'Content-Type: text/plain' \
  -H 'accept: text/plain' \
  -d 2005-03-09
```

Respuesta:

```
Pisces
```

Agregar unidad y prueba de integración a su aplicación Java con JUnit 5

¡Felicidades! ¡Ahora tiene una API segura que le brinda información útil sobre cualquier fecha de nacimiento que pueda imaginar! ¿Lo que queda? Bueno, debe agregar algunas pruebas unitarias para asegurarse de que funciona bien.

Muchas personas cometen el error de mezclar pruebas unitarias y pruebas de integración (también llamadas pruebas de extremo a extremo o E2E). Describiré la diferencia entre los dos tipos a continuación.

Antes de comenzar con las pruebas unitarias, agregue una dependencia más al `pom.xml` archivo (en la `<dependencies>` sección).

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

Pruebas unitarias

En su mayor parte, las pruebas unitarias están destinadas a probar un pequeño fragmento (o unidad) de código. Eso generalmente se limita al código dentro de una función o, a veces, se extiende a algunas funciones auxiliares llamadas desde esa función. Si una prueba unitaria está probando un código que depende de otro servicio o recurso, como una base de datos o un recurso de red, la prueba unitaria debe "burlarse" e inyectar esa dependencia para que no tenga un impacto real en ese recurso externo. También limita el enfoque solo a esa unidad que se está probando. Para simular una dependencia, puede usar una biblioteca simulada como "Mockito" o simplemente pasar una implementación diferente de la dependencia que desea reemplazar. Burlarse está fuera del alcance de este artículo y simplemente mostraré ejemplos de pruebas unitarias para `BasicBirthdayService` .

El `BasicBirthdayServiceTest.java` archivo contiene las pruebas unitarias de la `BasicBirthdayService` clase.

```
package com.example.joy.myFirstSpringBoot.services;

import static org.junit.jupiter.api.Assertions.assertEquals;
import java.time.LocalDate;
import org.junit.jupiter.api.Test;

class BasicBirthdayServiceTest {
    BasicBirthdayService birthdayService = new BasicBirthdayService();

    @Test
    void testGetBirthdayDOW() {
        String dow = birthdayService.getBirthDOW(LocalDate.of(1979, 7, 14));
        assertEquals("SATURDAY", dow);
        dow = birthdayService.getBirthDOW(LocalDate.of(2018, 1, 23));
        assertEquals("TUESDAY", dow);
        dow = birthdayService.getBirthDOW(LocalDate.of(1972, 3, 17));
        assertEquals("FRIDAY", dow);
        dow = birthdayService.getBirthDOW(LocalDate.of(1945, 12, 2));
        assertEquals("SUNDAY", dow);
        dow = birthdayService.getBirthDOW(LocalDate.of(2003, 8, 4));
        assertEquals("MONDAY", dow);
    }

    @Test
    void testGetBirthdayChineseSign() {
        String dow = birthdayService.getChineseZodiac(LocalDate.of(1979, 7, 14));
        assertEquals("Sheep", dow);
        dow = birthdayService.getChineseZodiac(LocalDate.of(2018, 1, 23));
        assertEquals("Dog", dow);
        dow = birthdayService.getChineseZodiac(LocalDate.of(1972, 3, 17));
        assertEquals("Rat", dow);
        dow = birthdayService.getChineseZodiac(LocalDate.of(1945, 12, 2));
        assertEquals("Rooster", dow);
        dow = birthdayService.getChineseZodiac(LocalDate.of(2003, 8, 4));
        assertEquals("Sheep", dow);
    }

    @Test
    void testGetBirthdayStarSign() {
        String dow = birthdayService.getStarSign(LocalDate.of(1979, 7, 14));
        assertEquals("Cancer", dow);
        dow = birthdayService.getStarSign(LocalDate.of(2018, 1, 23));
        assertEquals("Aquarius", dow);
        dow = birthdayService.getStarSign(LocalDate.of(1972, 3, 17));
        assertEquals("Pisces", dow);
        dow = birthdayService.getStarSign(LocalDate.of(1945, 12, 2));
        assertEquals("Sagittarius", dow);
        dow = birthdayService.getStarSign(LocalDate.of(2003, 8, 4));
        assertEquals("Leo", dow);
    }
}
```

Esta clase de prueba es uno de los conjuntos más básicos de pruebas unitarias que puede realizar. Crea una instancia de la `BasicBirthdayService` clase y luego prueba las

respuestas de los tres puntos finales con varias fechas de nacimiento que se pasan. Este es un gran ejemplo de una pequeña unidad que se está probando, ya que solo prueba un solo servicio y ni siquiera requiere ninguna configuración o applicationContext que se cargará para esta prueba. Debido a que solo está probando el servicio, no toca la seguridad ni la interfaz de reposo HTTP.

Puede ejecutar esta prueba desde su IDE o utilizando Maven:

```
mvn test -Dtest=BasicBirthdayServiceTest
```

Pruebas de integración con JUnit 5

Las pruebas de integración están destinadas a probar la ruta completa del código integrado (de extremo a extremo) para un caso de uso específico. Por ejemplo, una prueba de integración de la aplicación Birthday sería aquella que realiza una llamada HTTP POST al `dayOfWeek` punto final y luego prueba que los resultados son los esperados. Esta llamada finalmente afectará tanto el `BirthdayControllerInfo` código como el `BasicBirthdayService` código. También requerirá interactuar con la capa de seguridad para realizar estas llamadas. En un sistema más complejo, una prueba de integración puede afectar a una base de datos, leer o escribir desde un recurso de red o enviar un correo electrónico.

Debido al uso de dependencias / recursos reales, las pruebas de integración generalmente deben considerarse como posiblemente destructivas y frágiles (ya que los datos de respaldo podrían modificarse). Por esas razones, las pruebas de integración deben "manejarse con cuidado" y aislarse y ejecutarse independientemente de las pruebas unitarias normales. Personalmente, me gusta usar un sistema separado, particularmente para las pruebas REST API, en lugar de JUnit 5, ya que los mantiene completamente separados de las pruebas unitarias.

Si planea escribir pruebas unitarias con JUnit 5, deben nombrarse con un sufijo único como "IT". A continuación se muestra un ejemplo de las mismas pruebas con las que se ejecutó `BasicBirthdayService` , excepto que se escribió como una prueba de integración. Este ejemplo se burla de la seguridad web para esta prueba en particular, ya que el alcance no es probar OAuth 2.0, aunque se puede usar una prueba de integración para probar todo, incluida la seguridad.

El `BirthdayInfoControllerIT.java` archivo contiene las pruebas de integración de los tres puntos finales API para obtener información de cumpleaños.

```
package com.example.joy.myFirstSpringBoot.controllers;

import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostP
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostP
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

import com.example.joy.myFirstSpringBoot.services.BasicBirthdayService;

@AutoConfigureMockMvc
@ContextConfiguration(classes = {BirthdayInfoController.class, BasicBirthdayService.class})
@WebMvcTest
class BirthdayInfoControllerIT {
    private final static String TEST_USER_ID = "user-id-123";
    String bd1 = LocalDate.of(1979, 7, 14).format(DateTimeFormatter.ISO_DATE);
    String bd2 = LocalDate.of(2018, 1, 23).format(DateTimeFormatter.ISO_DATE);
    String bd3 = LocalDate.of(1972, 3, 17).format(DateTimeFormatter.ISO_DATE);
    String bd4 = LocalDate.of(1945, 12, 2).format(DateTimeFormatter.ISO_DATE);
    String bd5 = LocalDate.of(2003, 8, 4).format(DateTimeFormatter.ISO_DATE);

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testGetBirthdayDOW() throws Exception {
        testDOW(bd1, "SATURDAY");
        testDOW(bd2, "TUESDAY");
        testDOW(bd3, "FRIDAY");
        testDOW(bd4, "SUNDAY");
        testDOW(bd5, "MONDAY");
    }

    @Test
    public void testGetBirthdayChineseSign() throws Exception {
        testZodiak(bd1, "Sheep");
        testZodiak(bd2, "Dog");
        testZodiak(bd3, "Rat");
        testZodiak(bd4, "Rooster");
        testZodiak(bd5, "Sheep");
```

```
    }

    @Test
    public void testGetBirthdaytestStarSign() throws Exception {
        testStarSign(bd1, "Cancer");
        testStarSign(bd2, "Aquarius");
        testStarSign(bd3, "Pisces");
        testStarSign(bd4, "Sagittarius");
        testStarSign(bd5, "Leo");
    }

    private void testDOW(String birthday, String dow) throws Exception {
        MvcResult result = mockMvc.perform(MockMvcRequestBuilders.post("/birthday/dayOfWeek")
                .with(user(TEST_USER_ID))
                .with(csrf())
                .content(birthday)
                .contentType(MediaType.APPLICATION_JSON)
                .accept(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andReturn();

        String resultDOW = result.getResponse().getContentAsString();
        assertNotNull(resultDOW);
        assertEquals(dow, resultDOW);
    }

    private void testZodiak(String birthday, String czs) throws Exception {
        MvcResult result = mockMvc.perform(MockMvcRequestBuilders.post("/birthday/chineseZodia
                .with(user(TEST_USER_ID))
                .with(csrf())
                .content(birthday)
                .contentType(MediaType.APPLICATION_JSON)
                .accept(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andReturn();

        String resultCZ = result.getResponse().getContentAsString();
        assertNotNull(resultCZ);
        assertEquals(czs, resultCZ);
    }

    private void testStarSign(String birthday, String ss) throws Exception {
        MvcResult result = mockMvc.perform(MockMvcRequestBuilders.post("/birthday/starSign")
                .with(user(TEST_USER_ID))
                .with(csrf())
                .content(birthday)
                .contentType(MediaType.APPLICATION_JSON).accept(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andReturn();

        String resultSS = result.getResponse().getContentAsString();
        assertNotNull(resultSS);
        assertEquals(ss, resultSS);
    }
}
```

Esta clase de prueba tiene bastante; repasemos algunos elementos clave.

Hay algunas líneas de código que le indican al sistema que se burle de la seguridad, por lo que no necesita generar un token antes de ejecutar esta prueba de integración. Las siguientes líneas le dicen al sistema que finja que ya tenemos un token y un usuario válidos:

```
.with(user(TEST_USER_ID))
.with(csrf())
```

MockMvc es simplemente un sistema práctico integrado en Spring Framework para permitirnos hacer llamadas a una API REST. La `@AutoConfigureMockMvc` anotación de clase y la `@Autowired` variable de miembro MockMvc le indican al sistema que configure e inicialice automáticamente el `MockMvc` objeto (y en segundo plano, un contexto de aplicación) para esta aplicación. Cargará `SpringBootRestApiApplication` y permitirá que las pruebas le hagan llamadas HTTP.

Si lees sobre el *corte de prueba* , podrías encontrarte en una madriguera de conejo y sentir ganas de arrancarte el cabello. Sin embargo, si retrocede fuera de la madriguera del conejo, podría ver que el corte de prueba es simplemente el acto de recortar lo que está cargado dentro de su aplicación para una prueba de unidad particular o una clase de prueba de integración. Por ejemplo, si tiene 15 controladores en su aplicación web, con servicios con conexión automática para cada uno, pero su prueba solo prueba uno de ellos, ¿por qué molestarse en cargar los otros 14 y sus servicios con conexión automática? En cambio, ¡simplemente cargue el controlador que está probando y las clases de soporte necesarias para ese controlador! ¡Entonces, veamos cómo se usan los segmentos de prueba en esta prueba de integración!

```
@ContextConfiguration(classes = {BirthdayInfoController.class, BasicBirthdayService.class})
@WebMvcTest
```

La `WebMvcTest` anotación es el núcleo de segmentar una aplicación WebMvc. Le dice al sistema que está cortando y `@ContextConfiguration` le dice exactamente qué controladores y dependencias cargar. He incluido el `BirthdayInfoController` servicio porque ese es el controlador que estoy probando. Si lo dejara fuera, estas pruebas fallarían. También he incluido el `BasicBirthdayService` ya que esta es una prueba de integración y quiero que siga adelante y conecte automáticamente ese servicio como una dependencia del controlador. Si esto no fuera una prueba de integración, podría burlarme de esa dependencia en lugar de cargarla con el controlador.

¡Y eso es todo! ¡Cortar no tiene que ser demasiado complicado!

Puede ejecutar esta prueba desde su IDE o utilizando Maven:

```
mvn test -Dtest=BirthdayInfoControllerIT
```

Unidad de aislamiento y pruebas de integración

En Eclipse, si hace clic derecho en una carpeta y selecciona **ejecutar como** > **Prueba JUnit** , ejecutará todas las pruebas unitarias y pruebas de integración sin perjuicio. Sin embargo, a menudo se desea, especialmente si se ejecuta como parte de un proceso automatizado, ya sea para ejecutar las pruebas unitarias o para ejecutar ambas. De esta manera, puede haber una comprobación rápida de la cordura de las unidades, sin ejecutar las pruebas de integración a veces destructivas. Hay muchos enfoques para hacer esto, pero una forma fácil es agregar el complemento **Maven Failsafe** a su proyecto. Esto se realiza actualizando la `<build>` sección del `pom.xml` archivo de la siguiente manera:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

El complemento Failsafe diferenciará los tipos de pruebas por los nombres. Por defecto, considerará cualquier prueba que comience o termine `IT` como una prueba de integración. También considera las pruebas que terminan en `ITCase` una prueba de integración.

Una vez que `pom.xml` se configura, puede ejecutar los objetivos `test` o `verify` para probar las pruebas unitarias o las pruebas unitarias y de integración respectivamente. Desde Eclipse, esto se hace yendo al proyecto y haciendo clic derecho y seleccionando

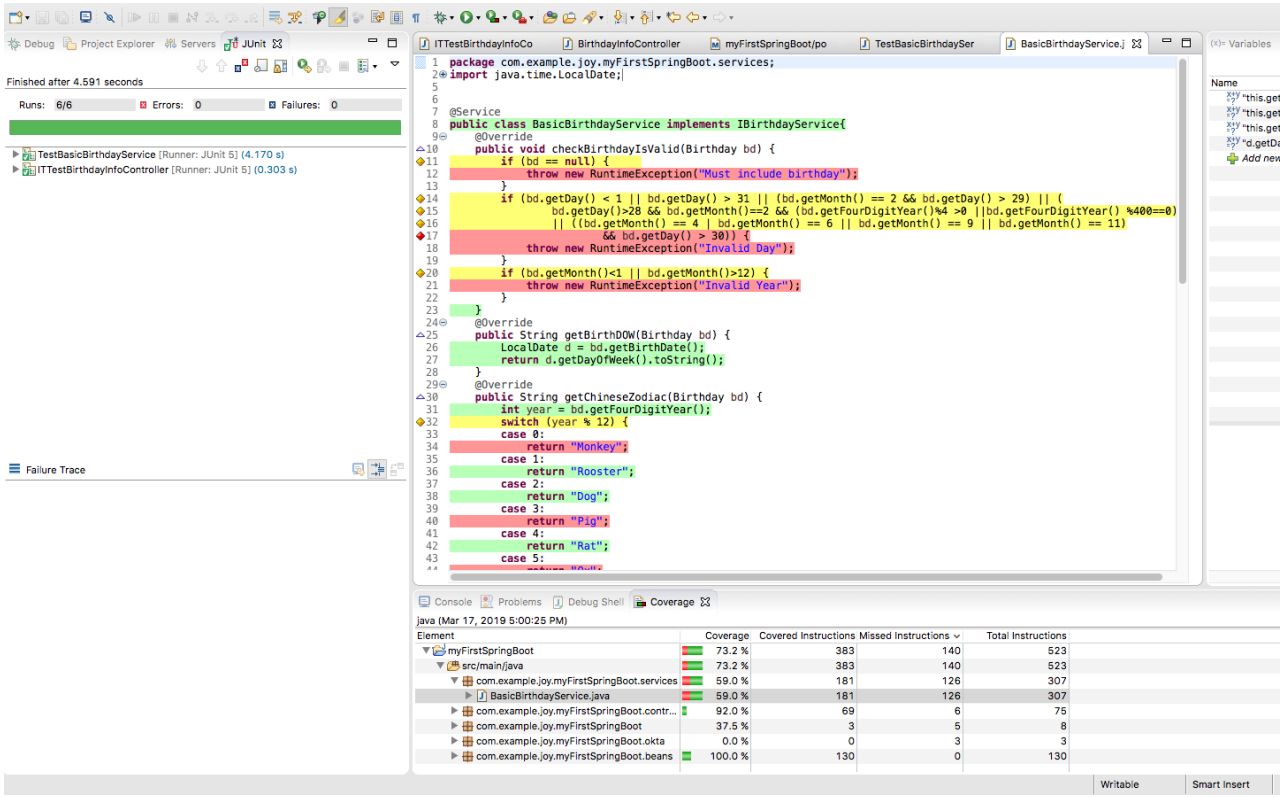
ejecutar como > **prueba Maven** para el `test` objetivo. Para el `verify` objetivo, debe hacer clic en **ejecutar como** > **Maven build ...** y luego ingresar "verificar" en el cuadro de texto de objetivos y hacer clic en ejecutar. Desde la línea de comandos, esto se puede hacer con `mvn test` y `mvn verify` .

Agregue cobertura de código a su aplicación Java con JUnit 5

La idea de "cobertura de código" es la cuestión de cuánto de su código se prueba con su unidad y / o pruebas de integración. Hay muchas herramientas que un desarrollador puede usar para hacer eso, pero como me gusta Eclipse, generalmente uso una herramienta llamada EclEmma . En versiones anteriores de Eclipse, solíamos tener que instalar este complemento por separado, pero parece que actualmente está instalado de forma predeterminada al instalar versiones de Eclipse EE. Si no se puede encontrar, siempre puede ir al Eclipse Marketplace (desde el menú de ayuda de Eclipse) e instalarlo usted mismo.

Desde Eclipse, ejecutar EclEmma es muy simple. Simplemente haga clic derecho en una sola clase de prueba o una carpeta y seleccione **cobertura como** > **Prueba JUnit** . Esto ejecutará su prueba o pruebas unitarias, pero también le proporcionará un informe de cobertura (consulte la parte inferior de la imagen a continuación). Además, resaltará cualquier código en su aplicación que esté cubierto en verde y cualquier cosa que no esté en rojo. (Cubrirá una cobertura parcial, como una declaración if que se prueba como verdadera, pero no como falsa con amarillo).

SUGERENCIA: si observa que está evaluando la cobertura de sus casos de prueba y desea que se elimine, vaya a **Preferencias** > **Java** > **Cobertura de código** y configure la opción "Solo coincidencia de entradas de ruta" `src/main/java` .



Obtenga más información sobre Java y Spring Boot, API REST seguras y OIDC

Espero que hayas llegado hasta aquí y hayas disfrutado de este tutorial sobre cómo construir y probar una API REST segura con Spring Boot y JUnit 5.

El código fuente completo está disponible en GitHub .

Para obtener más información sobre Spring Boot o OpenID Connect, consulte estos tutoriales:

- Comience con Spring Boot, OAuth 2.0 y Okta
- Cree una aplicación CRUD básica con Angular 7.0 y Spring Boot 2.1
- Comience con Spring Security 5.0 y OIDC
- Identidad, reclamos y tokens: un manual de OpenID Connect, parte 1 de 3

Para obtener más información sobre JUnit 5 y Test Slices, eche un vistazo a estas fuentes:

- Sergio Martin: lleve las pruebas unitarias al siguiente nivel con JUnit 5

- Biju Kunjummen - Spring Boot Web Test Slicing

Si ha llegado hasta aquí, puede estar interesado en ver futuras publicaciones de blog. Sigue a mi equipo, @oktadev en Twitter, o mira nuestro canal de YouTube . Para preguntas, por favor deje un comentario a continuación.

VISITA OKTA.COM

Social

GITHUB GORJEO FORO BLOG RSS YOUTUBE

Más información

INTEGRARSE CON OKTA
BLOG
CAMBIAR REGISTRO
AVISOS DE TERCEROS
KIT DE HERRAMIENTAS DE LA COMUNIDAD

Contacto y Legal

CONTACTA A NUESTRO EQUIPO
CONTACTO DE VENTAS
SOPORTE DE CONTACTO
TÉRMINOS Y CONDICIONES
POLÍTICA DE PRIVACIDAD