

- 29%

- 24%

- 11%

- 24%

- 47%

- 49%

- 25%

- 29%

- 29%

Knowledge Base ▾ Resources ▾ Deals Join Us ▾ About ▾ Login Register

g+ f t in

Sea



API Development Simplified

Postman Tools Support Every Stage of the API Lifecycle. Try it for Free Today!

Postman

ANDROID ▾ JAVA ▾ JVM LANGUAGES ▾ SOFTWARE DEVELOPMENT AGILE CAREER COMMUNICATIONS DEVOPS META JCG ▾

Home » Java » Enterprise Java » Spring Method Security with PreAuthorize

ABOUT ANDREW HUGHES



Spring Method Security with PreAuthorize

Posted by: Andrew Hughes in Enterprise Java September 19th, 2019 0 277 Views

Friends don't let friends write user auth. Tired of managing your own users? Try Okta's API and Java SDKs today. Authenticate, manage, and secure users in any application within minutes.

This tutorial will explore two ways to configure authentication and authorization in Spring Boot using Spring Security. One method is to create a

`WebSecurityConfigurerAdapter`

and use the fluent API to override the default settings on the

`HttpSecurity`

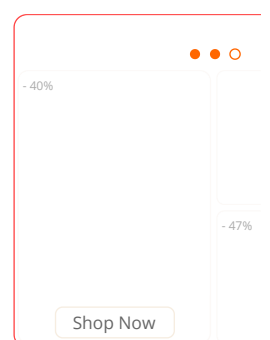
object. Another is to use the

`@PreAuthorize`

annotation on controller methods, known as method-level security or expression-based security. The latter will be the main focus of this tutorial. However, I will present some

`HttpSecurity`

code and ideas by way of contrast.



BOLETIN INFORMATIVO

¡Los iniciados ya disfrutan de ac
semanales y libros blancos de ci
Únase a ellos ahora para obtene
a las últimas noticias en el munc
como información sobre Android
otras tecnologías relacionadas.

Enter your e-mail...

never before. Try Grammarly now

Grammarly

Download

The first authentication method is

HttpSecurity

, which is global and is by default applied to all requests. Finer-grained control is possible, however, using pattern matching for endpoints, and the fluent API exposed by the

HttpSecurity

is quite powerful. This is where configuration options such as OAuth 2.0, Form Login, and HTTP Basic are exposed. It is a great place to set global authentication policies.

Method-level security is implemented by placing the

@PreAuthorize

annotation on controller methods (actually one of a set of annotations available, but the most commonly used). This annotation contains a Spring Expression Language (SpEL) snippet that is assessed to determine if the request should be authenticated. If access is not granted, the method is not executed and an HTTP Unauthorized is returned. In practice, using the

@PreAuthorize

annotation on a controller method is very similar to using

HttpSecurity

pattern matchers on a specific endpoint. There are some differences, however.

Differentiate Between Spring Security's @PreAuthorize and HttpSecurity

The first difference is subtle, but worth mentioning.

HttpSecurity

method rejects the request earlier, in a web request filter, before controller mapping has occurred. In contrast, the

@PreAuthorize

assessment happens later, directly before the execution of the controller method. This means that configuration in

HttpSecurity

is applied **before**

@PreAuthorize

.


Second,

HttpSecurity

is tied to URL endpoints while

@PreAuthorize

is tied to controller methods **and is actually located within the code adjacent to the controller definitions**. Having all of your security in one place and defined by web endpoints has a certain neatness to it, especially in smaller projects, or for more global settings; however, as projects get larger, it may make more sense to keep the authorization policies near the code being protected, which is what the annotation-based method allows.



únicos n
500 a
ubicado:
principa
relacion
Estar co
busca d
animam

nosotros. Entonces, si tiene un b
único e interesante, debe consul
programa de socios **JCG** . iTam
escritor invitado para Java Co
perfeccionar tus habilidades de e

- 24%

En de en aliexpress.co... Monc

€8,94

€6,71

- 34%

Cubo de fregona pleg... En de

€23,75

€15,44

- 25%

Cepillo de limpieza de... 2019

€10,43

€7,82

Demand nothing less than Everything.

The Data-to-Everything Platform

Le

spl

tur

built-in authentication objects (such as

```
authentication
```

and

```
principal
```

), dependency-injected method parameters, and query parameters. In this tutorial you will mostly look at two expressions:

```
hasAuthority()
```

and

```
hasRole()
```

. The Spring docs are again a great place to dig deeper.

Before we dive into the project, I want to also mention that Spring also provides a

```
@PostAuthorize
```

annotation. Not surprisingly, this is a method-level authorization annotation that is assessed **after** the method executes. Why would we want to do that? It allows the method to perform its own authorization checks based on whatever controller logic it likes before the annotation is assessed. The downside is that because the controller method is executed before the annotation is assessed, this could result in inefficiency, depending on the implementation.

Dependencies

The dependencies for this tutorial are pretty simple. You need: 1) Java 8+ installed, and 2) an Okta developer account.

If you do not have Java installed, go to AdoptOpenJDK. On *nix systems, you can also use SDKMAN.

If you do not already have a free Okta developer account, go to our website and sign up.

Start a Sample Project Using Spring Initializr

To get the project started, you can use the Spring Initializr. However, while it's worth taking a gander at the page, you don't even have to bother going there to create the project. You can use the REST API and

```
curl
```

.

Open a terminal and

```
cd
```

to wherever you want the project file .zip to end up. Run the command below, which will download the zipped Spring Boot project.

```
curl https://start.spring.io/starter.zip \
  -d dependencies=web,security \
  -d type=gradle-project \
  -d bootVersion=2.1.5.RELEASE \
  -d groupId=com.okta.preauthorize \
  -d artifactId=application \
  -o PreAuthorizeProject.zip
unzip PreAuthorizeProject.zip
```

There isn't much to the project to begin with except the

```
build.gradle
```

file and the

```
DemoApplication.java
```

class file. However, the whole project structure is there already set up for you.

The

```
build.gradle
```

file also has the two Spring dependencies you need for this example:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.boot:spring-boot-starter-web'
}
```

```
package com.okta.preauthorize.application;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class WebController {

    @RequestMapping("/")
    @ResponseBody
    public String home() {
        return "Welcome home!";
    }

    @RequestMapping("/restricted")
    @ResponseBody
    public String restricted() {
        return "You found the secret lair!";
    }
}
```

This controller defines two endpoints:

/

and

/restricted

. You will be adding method-level security to the

/restricted

endpoint in a bit. Right now, however, no security has been configured.

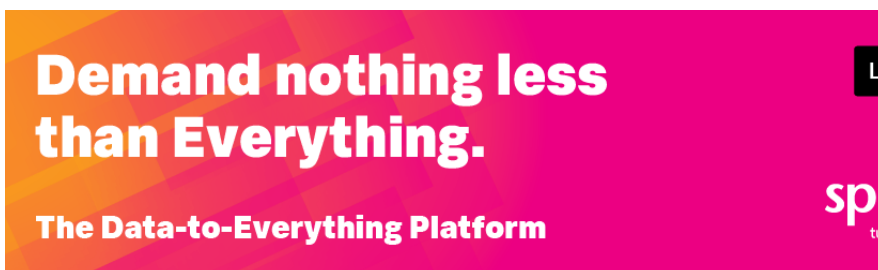
Go ahead and run the application. From the root project directory, run:

```
./gradlew bootRun
```

Once Spring Boot has finished launching, navigate to

http://localhost:8080

Notarás que aparece un formulario de inicio de sesión. Whoa! ¿De donde vino eso?!



El formulario es generado automáticamente por Spring Boot. Echa un vistazo a la clase de primavera

WebSecurityConfigurerAdapter

y el método

configure(HttpSecurity http)

. Aquí es donde se configuran los ajustes de autenticación predeterminados.

```
/**
 * Override this method to configure the {@link HttpSecurity}. Typically subclasses
 * should not invoke this method by calling super as it may override their * configuration. The default configurati
 */


```

* <pre>
* http.authorizeRequests().anyRequest().authenticated().and().formLogin().and().httpBasic();
* </pre>
* @param http the {@link HttpSecurity} to modify
* @throws Exception if an error occurs
*/
protected void configure(HttpSecurity http) throws Exception {
 ...
 http
 .authorizeRequests()
 .anyRequest().authenticated()
 .and()
 .formLogin().and()
 .httpBasic();
}
```


```

Autenticación versus autorización

Antes de continuar, quiero asegurarme rápidamente de que dos términos estén claros: autenticación y autorización. La autenticación responde a la pregunta: ¿quién está haciendo la solicitud? La autorización responde a la pregunta: ¿qué se les permite hacer?

La autenticación ocurre primero, generalmente, a menos que haya permisos específicos establecidos para usuarios anónimos (esto es una autenticación implícita en algunos aspectos). La autorización se basa en el valor del usuario autenticado. La entidad autenticada puede ser un usuario humano o un servicio automatizado, o un servicio que actúa en nombre de un usuario humano.

Download Chrome Browse

Ad Install Offline, Device-based C Policies & More. Deploy Chrome l

Google

Learn more

Dos esquemas de autorización comunes se basan en grupos y roles. Estos dos términos a menudo se combinan y se usan indistintamente en lugares menos acreditados de la web, pero hay una diferencia oficial. Los grupos definen conjuntos de usuarios y asignan permisos a esos conjuntos de usuarios. Los usuarios pueden ser miembros de múltiples grupos. Los roles definen conjuntos de permisos (acciones o recursos permitidos) que pueden asignarse a los usuarios. En la práctica, los grupos tienden a ser una forma más estática y menos flexible de acceso al controlador, mientras que los roles a menudo son finos y pueden ser dinámicos incluso dentro de una sesión, asignando roles para tareas específicas y revocándolos cuando ya no son necesarios. Cualquiera que haya usado Amazon AWS lo ha visto en acción, a menudo con un efecto desconcertante.

Habilite la seguridad de nivel de método para Spring @PreAuthorize

Lo que desea hacer ahora es configurar Spring Boot para permitir solicitudes en el punto final de inicio mientras restringe las solicitudes al

```
/restricted
```

punto final.

Inicialmente, podría pensar que podría agregar

```
@PreAuthorize("permitAll()")
```

al punto final de inicio y esto permitiría todas las solicitudes. Sin embargo, si lo prueba, descubrirá que no hace nada. Esto se debe a que la

```
HttpBuilder
```

implementación predeterminada todavía está activa, y porque se evalúa durante la cadena de filtro de solicitud web, tiene prioridad. **También debe habilitar explícitamente las anotaciones de seguridad a nivel de método, de lo contrario, se ignorarán.**

Agregue la siguiente

```
SecurityConfig
```

clase, que logrará los dos objetivos anteriores.

```
src/main/java/com/okta/preauthorize/application/SecurityConfig.java
```

```
package com.okta.preauthorize.application;

import org.springframework.context.annotation.Configuration;
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.configuration.WebSecurityConfigurerAdapter;
```

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    protected void configure(final HttpSecurity http) throws Exception {}
}
```

La

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
```

anotación es lo que permite la

El

```
configure(final HttpSecurity http)
```

método anula la

```
HttpBuilder
```

configuración predeterminada . Debido a que está vacío, deja la aplicación sin autorización o autenticación.

Ejecute la aplicación nuevamente usando

```
./gradlew bootRun
```

, y descubrirá que ambos puntos finales están ahora completamente abiertos.

Implementar una política de seguridad global

Las aplicaciones generalmente tienen que elegir en qué política de seguridad global van a estructurar su seguridad: "predeterminado a permitido" o "predeterminado a autenticado". ¿La aplicación está abierta por defecto? O por defecto restringido? Generalmente prefiero restringir y permitir explícitamente cualquier punto final público. Este esquema tiene sentido para los tipos de aplicaciones web propietarias en las que trabajo que tienden a no ser públicas o tener una cara pública relativamente pequeña. Sin embargo, si está trabajando en algo que es en gran parte público con un respaldo discreto controlado por acceso, como un sitio web con un panel de administración, un esquema más permisivo podría tener sentido. Verás ambos aquí.

Como la aplicación ya está abierta, le mostraré primero cómo restringir un método específico. Después de eso, verá un par de formas de implementar políticas de acceso más restrictivas a nivel mundial.

Powerful and secure monit

Ad Affordable monitoring can pr
downtime and save organization:

PRTG Network Monitor

Download

En la

```
webController
```

clase, agregue la

```
@PreAuthorize
```

anotación al

```
/restricted
```

punto final, así:

```
@PreAuthorize("isAuthenticated()")
@RequestMapping("/restricted")
@ResponseBody
public String restricted() {
    return "You found the secret lair!";
}
```

Ejecute la aplicación (

```
./gradlew bootRun
```

).

Esta vez podrá navegar a la página de inicio, pero ir al

```
/restricted
```

punto final le proporciona una página de error de etiqueta blanca (ciertamente fea).

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Mon May 20 11:57:11 PDT 2019

There was an unexpected error (type=Forbidden, status=403).

Access Denied

```
webController
```

clase para que coincida con lo siguiente:

```
@Controller
@PreAuthorize("isAuthenticated()")
public class WebController {

    @PreAuthorize("permitAll()")
    @RequestMapping("/")
    @ResponseBody
    public String home() {
        return "Welcome home!";
    }

    @RequestMapping("/restricted")
    @ResponseBody
    public String restricted() {
        return "You found the secret lair!";
    }
}
```

Aquí ha utilizado la

```
@PreAuthorize
```

anotación para restringir toda la clase de controlador a usuarios autenticados y permitir explícitamente todas las solicitudes (independientemente del estado de autenticación) al punto final de inicio.

Sé que lo hemos estado llamando seguridad de "nivel de método", pero, de hecho, estas

```
@PreAuthorize
```

anotaciones también se pueden agregar a las clases de controlador para establecer un valor predeterminado para toda la clase. Aquí también es

```
@PreAuthorize("permitAll()")
```

útil porque puede anular la anotación de nivel de clase.

Encuentra tu Tienda MediaMarkt

Media Markt



Si ejecuta la aplicación (

```
./gradlew bootRun
```

) y prueba los puntos finales, encontrará que el punto final de inicio está abierto pero el

```
/restricted
```

punto final está cerrado.

Tenga en cuenta que si agrega un segundo controlador web separado, todos sus métodos seguirían abiertos de forma predeterminada y no requerirían autenticación.

Una tercera opción (mi favorita en la mayoría de las aplicaciones pequeñas y medianas) es usar

```
HttpBuilder
```

para requerir autenticación para todas las solicitudes de forma predeterminada y permitir explícitamente cualquier punto final público. Esto le permite usar

```
@PreAuthorize
```

para refinar el control de acceso para métodos específicos basados en usuarios o roles o grupos, pero deja en claro que todas las rutas, a **menos que se permitan explícitamente**, tendrán aplicada cierta seguridad básica. También significa que las vías públicas se definen en un lugar central. Nuevamente, esto funciona para cierto tipo de proyecto, pero puede no ser la mejor estructura para todos los proyectos.

Para implementar esto, cambie la

```
webController
```

clase a esto (eliminando todas las

```

    }

    @RequestMapping("/restricted")
    @ResponseBody
    public String restricted() {
        return "You found the secret lair!";
    }
}

```

Y cambie la

webSecurity

clase a esto:

```

Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    protected void configure(final HttpSecurity http) throws Exception {
        http.antMatcher("/**")
            .authorizeRequests()
            .antMatchers("/").permitAll()
            .anyRequest().authenticated()
            .and().formLogin();
    }
}

```

Lo que ha hecho es usar la

SecurityConfig

clase para permitir explícitamente todas las solicitudes en el punto final de inicio, mientras que requiere autenticación en todos los demás puntos finales. Esto establece un requisito mínimo de autenticación general para su aplicación. También volvió a habilitar la autenticación basada en formularios.

¡Intentalo!

Ejecutar la aplicación usando:

./gradlew bootRun

Navegue hasta el punto final de la casa, que está abierto:

http://localhost:8080

Y el punto final restringido, lo que requiere autenticación:

http://localhost:8080/restricted

Cuando aparezca el formulario de inicio de sesión de Spring, no olvide que puede usar las credenciales predeterminadas. El usuario es "usuario", y la contraseña se encuentra en la salida de la consola (busque

Using generated security password:

).

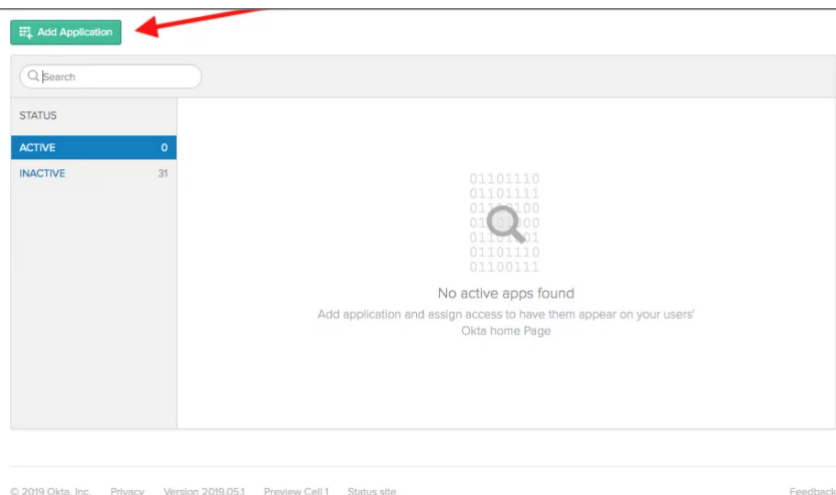
Avance al inicio de sesión de OAuth 2.0

La autenticación basada en formularios se siente bastante anticuada en estos días. Cada vez más, los usuarios esperan poder iniciar sesión utilizando sitios de terceros, y debido a las mayores amenazas de seguridad, hay menos motivación para administrar las credenciales de los usuarios en su propio servidor. Okta es una empresa de gestión de acceso e identidad de software como servicio que ofrece una gran cantidad de servicios. En esta sección, los usará para implementar rápidamente un formulario de inicio de sesión con OAuth 2.0 y OIDC (OpenID Connect).

Muy, muy brevemente: OAuth 2.0 es un protocolo de autorización estándar de la industria y OIDC es otro estándar abierto además de OAuth que agrega una capa de identidad (autenticación). Juntos, proporcionan una forma estructurada para que los programas administren la autenticación y la autorización y se comuniquen a través de redes e Internet. Sin embargo, ni OAuth ni OIDC proporcionan una implementación. Son solo especificaciones o protocolos. Ahí es donde entra Okta. Okta tiene una implementación de las especificaciones OAuth 2.0 y OIDC que permite que los programas utilicen sus servicios para proporcionar rápidamente servicios de inicio de sesión, registro e inicio de sesión único (o inicio de sesión social). En este tutorial, solo implementará una función de inicio de sesión, pero al final del tutorial, puede encontrar enlaces a otros recursos para mostrarle cómo implementar el inicio de sesión social y el registro.

Primero, regístrese para obtener una cuenta de desarrollador Okta gratuita: <https://developer.okta.com/signup/>.

Si es la primera vez que inicia sesión o se acaba de registrar, es posible que deba hacer clic en el botón **Administrador** para acceder a la consola del desarrollador.



- Haga clic en el botón verde **Agregar aplicación**
- Haga clic en Tipo de aplicación **web** y en **Siguiente**
- Dale un nombre a la aplicación. Cualquier nombre.
- Establezca **URI de redireccionamiento de inicio de sesión** en `http://localhost:8080/login/oauth2/code/okta`
- Haz clic en **Listo**.

A screenshot of the Okta Admin Console showing the configuration for a new application. The 'General Settings' tab is active, showing fields for 'Application label' (Spring PreAuthorize), 'Application type' (Web), and 'Allowed grant types' (Authorization Code, Refresh Token, Implicit (Hybrid)). The 'LOGIN' section shows 'Login redirect URIs' (http://localhost:8080/login/oauth2/code/okta), 'Logout redirect URIs', 'Login initiated by' (App Only), and 'Initiate login URI'. Below this is the 'Client Credentials' tab, which shows 'Client ID' (Ooakz4teswoV7sDZl0h7) and 'Client secret' (a masked field). The 'Client ID' is described as a 'Public identifier for the client that is required for all OAuth flows.' and the 'Client secret' is described as a 'Secret used by the client to exchange an authorization code for a token. This must be kept confidential! Do not include it in apps which cannot keep it secret, such as those running on a client.'

Hacer que Spring Boot funcione con OAuth 2.0 y Okta es notablemente fácil. El primer paso es agregar la dependencia Okta Spring Boot Starter. Es totalmente posible usar Okta OAuth 2.0 / OIDC sin usar nuestro iniciador; sin embargo, el iniciador simplifica la configuración. También se encarga de extraer el reclamo de grupos del JSON Web Token y convertirlo en una autoridad de Spring Security (que se analizará en un momento). Echa un vistazo a la página [Okta Spring Boot Starter GitHub](#) para obtener más información.

Actualice la sección de dependencias de su

```
build.gradle
```

archivo:

```
dependencies {
    implementation 'com.okta.spring:okta-spring-boot-starter:1.2.1' // <-- ADDED
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'org.springframework.security:spring-security-test'
}
```

En el

```
src/main/resources
```

directorio hay un

```
application.properties
```

archivo. Renombrarlo a

```
application.yml
```

. Agregue el siguiente contenido:

```
okta:
  oauth2:
    issuer: https://{yourOktaDomain}/oauth2/default
    client-id: {yourClientId}
    client-secret: {yourClientSecret}
```

No se olvide de actualizar los **ID de cliente**, **cliente-secreta** y **emisores** valores para que coincida con los valores de su cuenta de desarrollador y Okta PeDIP aplicación. Su emisor de Okta debería verse algo así

```
https://dev-123456.okta.com/oauth2/default
```

.

Finalmente, actualice el

```
SecurityConfiguration.java
```

archivo:

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    protected void configure(final HttpSecurity http) throws Exception {
        http.antMatcher("/**")
            .authorizeRequests()
            .antMatchers("/").permitAll()
            .anyRequest().authenticated()
            .and().oauth2Login(); // <-- THIS WAS CHANGED
    }
}
```

Tenga en cuenta que todo lo que realmente cambió aquí fue

```
formLogin()
```

a

```
oauth2Login()
```

.

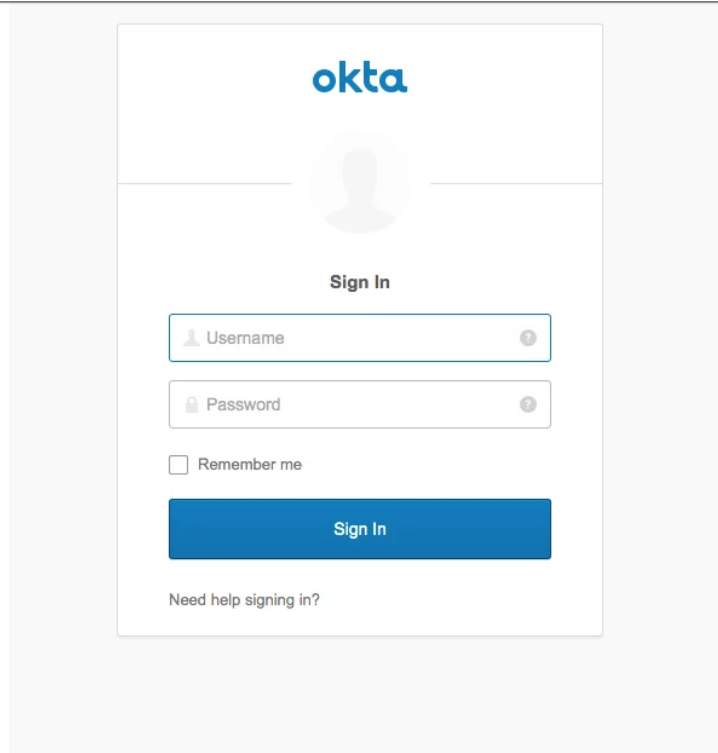
Ejecute la aplicación:

```
./gradlew bootRun
```

(es posible que deba cerrar sesión en el panel del desarrollador de Okta o usar una ventana de incógnito para ver la pantalla de inicio de sesión).

El

```
/
```



¡Inicie sesión con sus credenciales de Okta y estará autenticado!

Inspeccione los atributos de usuario de OAuth 2.0

Al desarrollar aplicaciones OAuth, he encontrado útil poder inspeccionar la información que Spring Boot tiene sobre el cliente y el usuario autenticado. Para este fin, agregue un nuevo controlador llamado

UserInfoController.java

.

src/main/java/com/okta/preauthorize/application/UserInfoController.java

```
package com.okta.preauthorize.application;

import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.oauth2.client.OAuth2AuthorizedClient;
import org.springframework.security.oauth2.client.annotation.RegisteredOAuth2AuthorizedClient;
import org.springframework.security.oauth2.core.user.OAuth2User;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.Map;

@RequestMapping("/user")
@Controller
public class UserInfoController {

    @RequestMapping("/oauthinfo")
    @ResponseBody
    public String oauthUserInfo(@RegisteredOAuth2AuthorizedClient OAuth2AuthorizedClient authorizedClient,
                               @AuthenticationPrincipal OAuth2User oAuth2User) {
        return
            "User Name: " + oAuth2User.getName() + "<br/>" +
            "User Authorities: " + oAuth2User.getAuthorities() + "<br/>" +
            "Client Name: " + authorizedClient.getClientRegistration().getClientName() + "<br/>" +
            this.prettyPrintAttributes(oAuth2User.getAttributes());
    }

    private String prettyPrintAttributes(Map<String, Object> attributes) {
        String acc = "User Attributes: <br/><div style='padding-left:20px'>";
        for (String key : attributes.keySet()){
            Object value = attributes.get(key);
            acc += "<div>" + key + ": " + value.toString() + "</div>";
        }
        return acc + "</div>";
    }
}
```

El segundo método,

```
prettyPrintAttributes()
```

es solo un poco de azúcar para formatear los atributos del usuario para que sean más legibles.

Ejecutar la aplicación:

```
./gradlew bootRun
```

Navega hacia

```
http://localhost:8080/user/oauthinfo
```

Verás algo como esto:

```
User Name: 00ab834zk7eJ18e8Y0h7
User Authorities: [ROLE_USER, SCOPE_email, SCOPE_openid, SCOPE_profile]
Client Name: Okta
User Attributes:
  at_hash: 1yq01bHDupcb8AhBNShkeQ
  sub: 00ue9mlzk7eW24e8Y0h7
  zoneinfo: America/Los_Angeles
  ver: 1
  email_verified: true
  amr: ["pwd"]
  iss: https://dev-123456.oktapreview.com/oauth2/default
  preferred_username: andrew.hughes@mail.com
  locale: en-US
  given_name: Andrew
  aud: [0oakz4teswoV7sDZI0h7]
  updated_at: 1558380884
  idp: 00oe9mlzh0xuq0T5z0h7
  auth_time: 1558454889
  name: Andrew Hughes
  exp: 2019-05-21T17:46:28Z
  family_name: Hughes
  iat: 2019-05-21T16:46:28Z
  email: andrew.hughes@mail.com
  jti: ID.CnwVJ_h1Dq5unqkwherWyf8ZFTETX_X4TP39ythQ-ZE
```

Quiero señalar algunas cosas. Primero, observe que el **Nombre de usuario** es en realidad la ID de cliente de su aplicación OIDC de Okta. Esto se debe a que, desde el punto de vista de Spring Boot OAuth, la aplicación cliente es el "usuario". Para encontrar el nombre de usuario y la información reales, debe buscar en los **Atributos del usuario**. Tenga en cuenta que el contenido real de estos atributos varía entre los proveedores de OAuth, por lo que si está apoyando a Okta, GitHub y Twitter, por ejemplo, deberá inspeccionar estos atributos para cada proveedor de OAuth para ver qué están devolviendo. .

El otro punto importante son las **Autoridades de usuario**. Las autoridades, tal como las usa Spring aquí, son un término meta para la información de autorización. Son solo cuerdas. Sus valores se extraen de la información OAuth / OIDC. Depende de la aplicación del cliente usarlos correctamente. Pueden ser roles, ámbitos, grupos, etc. En lo que respecta a OAuth / OIDC, su uso es básicamente arbitrario.

Para ver cómo funciona esto, en las siguientes secciones agregará un grupo de **administración** en Okta, asignará un usuario a ese grupo y restringirá un método al grupo de **administración** mediante la

```
@PreAuthorize
```

anotación.

ROLE_USER : Notarás que todos los usuarios autenticados son asignados

```
ROLE_USER
```

por Spring. Este es el rol predeterminado de nivel más bajo que se asigna automáticamente.

ALCANCE_ : también tenga en cuenta que los ámbitos de OAuth se asignan a las autoridades de Spring y se pueden utilizar para la autorización, por ejemplo, en las anotaciones

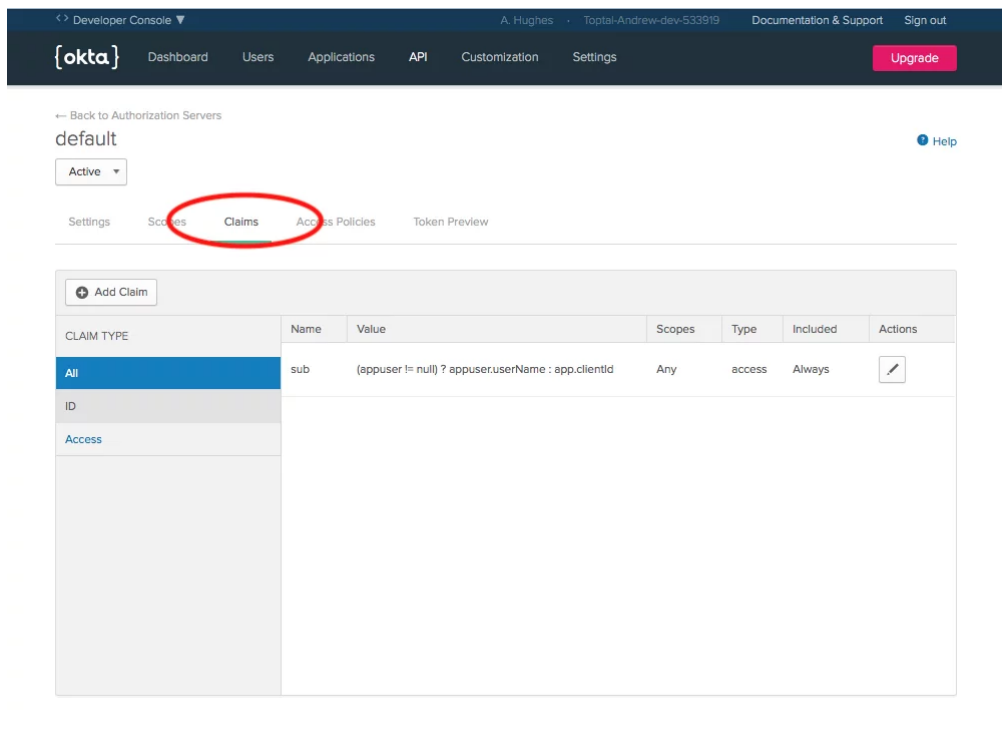
```
@PreAuthorize
```

y

```
@PostAuthorize
```

, como verá en un segundo.

Activar reclamo de grupos en Okta



Vas a crear dos asignaciones de reclamos. No está creando dos reclamos, per se, sino instruyendo a Okta para que agregue los reclamos grupales tanto al Token de acceso como al Token de ID. Debe hacer esto porque, según el flujo de OAuth, el reclamo de los grupos puede extraerse de cualquiera. En nuestro caso, con el flujo de OIDC, lo que realmente importa es el token de identificación, pero es mejor agregarlos a ambos para evitar frustraciones en el futuro. El flujo del servidor de recursos requiere que los grupos afirmen estar en el token de acceso.

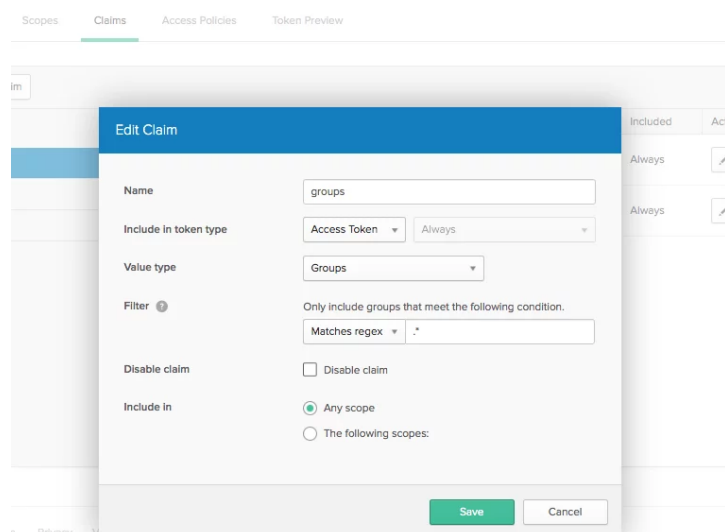
Primero, agregue una asignación de reclamo para el token de tipo **Token de acceso**.

Haz clic en **Agregar reclamo**.

Actualice los siguientes valores (los otros valores predeterminados están bien):

- **Nombre:** grupos
- **Incluir en tipo de token :** Token de acceso
- **Tipo de valor:** grupos
- **Filtro:** coincide con la expresión regular,

. *



- **Incluir en el tipo de token** : ID Token
- **Tipo de valor:** grupos
- **Filtro:** coincide con la expresión regular,

```
.*
```

¡Excelente! Entonces ahora Okta asignará todos sus grupos a un

```
groups
```

reclamo sobre el token de acceso y el token de ID.

Lo que sucede con este reclamo de grupos en el lado de Spring no es necesariamente obvio ni automático. Uno de los beneficios del iniciador Spring Boot es que extrae automáticamente el reclamo de los grupos del JWT y lo asigna a una autoridad de Spring. De lo contrario, necesitaría implementar el suyo

```
GrantedAuthoritiesExtractor
```

```
.
```

FYI: el nombre del reclamo de grupos se puede configurar usando el

```
okta.oauth2.groupsClaim
```

campo en el

```
application.yml
```

archivo. Por defecto es

```
groups
```

```
.
```

Inspeccionar los atributos del usuario con grupos

Ejecutar la aplicación:

```
./gradlew bootRun
```

```
.
```

Navega hacia

```
http://localhost:8080/user/oauthinfo
```

```
.
```

Verá algo como esto (un montón de líneas redundantes omitidas para mayor claridad):

```
User Name: 00ab834zk7eJ18e8Y0h7
User Authorities: [Everyone, ROLE_USER, SCOPE_email, SCOPE_openid, SCOPE_profile]
Client Name: Okta
User Attributes:
...
groups: ["Everyone"]
...
```

Observe un nuevo atributo de usuario de **grupos** (la reclamación de grupos asignados). El valor,

```
Everyone
```

es el grupo predeterminado asignado a, bueno, a todos. Esto se asigna a la autoridad del usuario

```
Everyone
```

```
.
```

Esa es la idea básica. Se volverá un poco más emocionante en el siguiente paso cuando agregue un grupo de administración.

Crear un grupo de administradores en Okta

Ahora desea agregar un grupo de **administración** en Okta. Inicie sesión en su panel de desarrollador de Okta.

Desde el menú superior, vaya a **Usuarios** y seleccione **Grupos**.

Haz clic en **Agregar grupo**.

En la ventana emergente:

punto final

En la

webController

clase, actualice el

/restricted

método de punto final. Estás agregando la siguiente anotación al método:

```
@PreAuthorize("hasAuthority('Admin')")
```

Al igual que:

```
@PreAuthorize("hasAuthority('Admin')")
@RequestMapping("/restricted")
@ResponseBody
public String restricted() {
    return "You found the secret lair!";
}
```

Esto le dice a Spring que verifique que el usuario autenticado tenga la

Admin

autoridad y, de lo contrario, rechace la solicitud.

Ejecutar la aplicación:

./gradlew bootRun

.

Navega hacia

http://localhost:8080/restricted

.

Obtendrá un error de página blanca **403 / No autorizado**.

Agregue su usuario al grupo de administración

Ahora debe agregar su usuario Okta al grupo Admin. En el menú superior, seleccione **Usuarios** y haga clic en **Grupos**. Haga clic en el grupo de **administración**. Haz clic en **Agregar miembros**. Busque su usuario en la ventana emergente y haga clic en **Agregar**.

Probar la pertenencia al grupo de administradores

¡Hecho! Veamos qué hizo eso. Una vez, otra vez, ejecutar la aplicación de arranque de primavera:

./gradlew bootRun

.

Navega hacia

http://localhost/user/oauthinfo

.

Esta vez verás el

Admin

grupo y la autoridad. (Es posible que necesite una nueva sesión del navegador para ver el cambio, o usar incógnito).

```
User Name: 00ab834zk7eJ18e8Y0h7
User Authorities: [Admin, Everyone, ROLE_USER, SCOPE_email, SCOPE_openid, SCOPE_profile]
Client Name: Okta
User Attributes:
...
groups: ["Everyone", "Admin"]
...
```

Y si navega hacia

http://localhost:8080/restricted

se le permitirá el acceso

hasAuthority()

Los roles en Spring son aut

ROLE_

prefijo (como todas las co

conjuntos de permisos, mie

uso. La implementación rea

autorización.

El punto importante a reco

hasRole()

, necesita el nombre de la autoridad en el reclamo para comenzar

ROLE_

. Por ejemplo, si agregó un

ROLE_ADMIN

grupo, y agregó su usuario, y el grupo a la aplicación OIDC, podría usarlo

hasRole('ADMIN')



Autorización basada en alcances OAuth 2.0 con Spring PreAuthorize

También puede usar la

@PreAuthorize

anotación para limitar el acceso según los ámbitos de OAuth. De la documentación de los ámbitos de OAuth 2.0 :

El alcance es un mecanismo en OAuth 2.0 para limitar el acceso de una aplicación a la cuenta de un usuario. Una aplicación puede solicitar uno o más ámbitos, esta información se presenta al usuario en la pantalla de consentimiento y el token de acceso emitido a la aplicación se limitará a los ámbitos otorgados.

Si observa el

User Authorities

retorno inspeccionado desde el

/user/oauthinfo

punto final, verá tres autoridades que comienzan con

SCOPE_

:

- ALCANCE_email
- ALCANCE_openida
- ALCANCE_perfil

Estos corresponden a los ámbitos de correo electrónico, openid y perfil. Para restringir un método para un usuario que tiene un alcance específico, se utilizaría una anotación, tales como:

@PreAuthorize("hasAuthority('SCOPE_email')")

También señalaré que puedes lograr exactamente lo mismo (más o menos exactamente) usando

HttpSecurity

en la

SecurityConfig

Puede personalizar los ámbitos que la aplicación cliente solicita del servidor de autorización Okta agregando una

```
scopes
```

propiedad al

```
application.yml
```

archivo. Por ejemplo, a continuación, configuré el

```
application.yml
```

archivo para solicitar solo el

```
openid
```

alcance, que se requiere para OAuth.

```
okta:
  oauth2:
    ...
    scopes: openid
    ...
```

Si ejecutó esta solicitud en el

```
/user/oauthinfo
```

punto final, obtendría algo como esto:

```
User Name: 00ab834zk7e318e8Y0h7
User Authorities: [Admin, Everyone, ROLE_USER, SCOPE_openid]
Client Name: Okta
...
```

Tenga en cuenta que solo se ha asignado un ámbito a las autoridades de usuario.

Intente agregar un ámbito personalizado. Cambiar

```
okta.oauth2.scopes
```

propiedad en el

```
application.yml
```

archivo para que coincida:

```
okta:
  oauth2:
    ...
    scopes: openid email profile custom
    ...
```

Antes de ejecutar la aplicación y probar esto, debe agregar el ámbito personalizado al servidor de autorización Okta (si lo ejecuta ahora obtendrá un error).

Abre tu panel de desarrollador de Okta.

Desde el menú superior, vaya a **API** y seleccione **Servidores de autorización**.

Seleccione el servidor de autorización **predeterminado**.

Haga clic en la pestaña **Ámbitos**.

Settings	Scopes	Claims	Access Policies	Token Preview
<div> <div> <div>+</div> Add Scope </div> </div>				
Name	Description	Default Scope	Metadata Publish	Actions
openid	Signals that a request is an OpenID request.	No	Yes	
profile	Allows this application to access your profile information.	No	Yes	
email	Allows this application to access your email address.	No	Yes	
address	Allows this application to access your address.	No	Yes	
phone	Allows this application to access your phone number.	No	Yes	
offline_access	Allows this application to access your data when you aren't using the application.	No	Yes	

Haga clic en el botón **Agregar alcance**.

• **Nombre :**

custom

• **Descripción :**

Custom test scope

Haz clic en **Crear**.

Acaba de agregar un ámbito personalizado (astutamente nombrado

custom

) a su servidor de autorización Okta predeterminado.

Una última vez, ejecutar la aplicación de arranque de primavera:

```
./gradlew bootRun
```

Navega hacia

http://localhost:8080/user/oauthinfo

User Name: 00ab834zk7eJ18e8Y0h7

User Authorities: [Admin, Everyone, ROLE_USER, SCOPE_custom, SCOPE_email, SCOPE_openid, SCOPE_profile]

Client Name: Okta

...

¡Éxito! Debería ver lo nuevo

SCOPE_custom

en las autoridades de usuario.

Spring PreAuthorize, HttpSecurity y Security en Spring Boot

Cubriste un montón de tierra! Has visto bien la seguridad a nivel de método de Spring usando

@PreAuthorize

y has visto cómo se relaciona

HttpSecurity

. Usó algunas declaraciones básicas de SpEL (Spring Expression Language) para configurar la autorización. Revisaste la diferencia entre autorización y autenticación. Configuraste Spring Boot para usar Okta como un proveedor de inicio de sesión único OAuth 2.0 / OIDC y agregaste un reclamo grupal al servidor de autenticación y a la aplicación cliente. Incluso un nuevo grupo de **administradores** y vio cómo usar la reclamación de grupos, asignada a una autoridad de Spring, para restringir el acceso. Finalmente, echó un vistazo a cómo se pueden usar los ámbitos OAuth 2.0 para definir esquemas de autorización e implementarlos en la aplicación.

- Asegure su aplicación Spring Boot con autenticación multifactor
- Cree una API segura con Spring Boot y GraphQL

Si desea profundizar, eche un vistazo al Proyecto GitHub Okta Spring Boot Starter .

Si tiene alguna pregunta sobre esta publicación, agregue un comentario a continuación. Para obtener contenido más sorprendente , siga @oktadev en Twitter, haga clic en Me gusta en Facebook o suscríbase a nuestro canal de YouTube .

"Spring Method Security with PreAuthorize" se publicó originalmente en el blog Okta Developer el 20 de junio de 2019.

Los amigos no dejan que sus amigos escriban autenticación de usuario. ¿Cansado de administrar tus propios usuarios? Pruebe la API de Okta y los SDK de Java hoy. Autentique, administre y proteja a los usuarios en cualquier aplicación en cuestión de minutos.

Etiquetado con: AUTORIZAR PREVIAMENTE SPRING SPRING SECURITY

 (0 calificación, 0 votos)

Debe ser un miembro registrado para calificar esto. 🗨 Inicie la discusión 🔄 277 reproducciones i 🐦 Tweet!

¿Quieres saber cómo desarrollar tus habilidades para convertirte en un Rockstar de Java?



Suscríbete a nuestro boletín para comenzar a rockear. Para comenzar, ¡le ofrecemos nuestros eBooks más vendidos ahora mismo! GRATIS!

1. JPA Mini libro
2. Guía de resolución de problemas de JVM
3. Tutorial JUnit para pruebas unitarias
4. Tutorial de anotaciones de Java
5. Preguntas de la entrevista Java
6. Preguntas de la entrevista de primavera
7. Diseño de la interfaz de usuario de Android

y muchos más

☐ Acepto los Términos y la Política de privacidad

[Regístrate](#)

¿TE GUSTA ESTE ARTÍCULO? LEER MÁS DE JAVA CODE GEEKS

Network Documentation

Ad tripunkt GmbH

Centralized Authorization with OAuth2 & Opaque Tokens using...

javacodegeeks.com

Conviértete en Data Scientist

Ad IE Business School

Spring Boot: Building a RESTful Web Application

javacodegeeks.com

Best Tool For Students

Ad Grammarly

Spring MVC - @RequestBody and @ResponseBody demystified

javacodegeeks.com

Deploy a Spring Boot Application into Tomcat

javacodegeeks.com

Advanced Java Tutorial

javacodegeeks.com

Deja una respuesta



Start the discussion...

Este sitio usa Akismet para reducir el spam. Aprende cómo se procesan sus datos de comentarios .

✉ Suscribirse ▼

BASE DE CONOCIMIENTOS

Cursos

Ejemplos

Minibooks

Recursos

Tutoriales

SOCIOS

Mkyong

THE CODE GEEKS NETWORK

.NET Code Geeks

Java Code Geeks

System Code Geeks

Web Code Geeks

HALL OF FAME

"Android Full Application Tutorial" series

11 Online Learning websites that you should check out

Advantages and Disadvantages of Cloud Computing – Cloud computing pros and cons

Android Google Maps Tutorial

Android JSON Parsing with Gson Tutorial

Android Location Based Services Application – GPS location

Android Quick Preferences Tutorial

Difference between Comparator and Comparable in Java

GWT 2 Spring 3 JPA 2 Hibernate 3.5 Tutorial

Java Best Practices – Vector vs ArrayList vs HashSet

ABOUT JAVA CODE GEEKS

JCGs (Java Code Geeks) is an independent online community focused on providing the ultimate Java to Java developers resource center; targeted at the technical team lead (senior developer), project manager and junior developers. JCGs serve the Java, SOA, Agile and Telecom communities with daily news, domain experts, articles, tutorials, reviews, announcements, code snippets and source projects.

DISCLAIMER

All trademarks and registered trademarks appearing on Java Code Geeks are the property of their respective owners. Java is a trademark or registered trademark of Oracle Corporation in the United States and other countries. Examples of trademarks that are not connected to Oracle Corporation and is not sponsored by Oracle Corporation are listed below.

