§ 28. Security

If Spring Security is on the classpath then web applications will be secure by default with ‘basic’ authentication on all HTTP endpoints. To add method-level security to a web application you can also add `@EnableGlobalMethodSecurity` with your desired settings. Additional information can be found in the Spring Security Reference.

The default `AuthenticationManager` has a single user (‘user’ username and random password, printed at INFO level when the application starts up)

```
Using default security password: 78fa095d-3f4c-48b1-ad50-e24c31d5cf35
```

If you fine tune your logging configuration, ensure that the `org.springframework.boot.autoconfigure.security` category is set to log `INFO` messages, otherwise the default password will not be printed.

You can change the password by providing a `security.user.password`. This and other useful properties are externalized via `SecurityProperties` (properties prefix "security").

The default security configuration is implemented in `SecurityAutoConfiguration` and in the classes imported from there (`SpringBootWebSecurityConfiguration` for web security and `AuthenticationManagerConfiguration` for authentication configuration which is also relevant in non-web applications). To switch off the default web security configuration completely you can add a bean with `@EnableWebSecurity` (this does not disable the authentication manager configuration). To customize it you normally use external properties and beans of type `WebSecurityConfigurerAdapter` (e.g. to add form-based login). To also switch off the authentication manager configuration you can add a bean of type `AuthenticationManager`, or else configure the global `AuthenticationManager` by autowiring an `AuthenticationManagerBuilder` into a method in one of your `@Configuration` classes. There are several secure applications in the Spring Boot samples to get you started with common use cases.

The basic features you get out of the box in a web application are:

- An `AuthenticationManager` bean with in-memory store and a single user (see `SecurityProperties.User` for the properties of the user).
- Ignored (insecure) paths for common static resource locations (`/css/**`, `/js/**`, `/images/**` and `**/favicon.ico`).
- HTTP Basic security for all other endpoints.
- Security events published to Spring's `ApplicationEventPublisher` (successful and unsuccessful authentication and access denied).
- Common low-level features (HSTS, XSS, CSRF, caching) provided by Spring Security are on by default.

All of the above can be switched on and off or modified using external properties (`security.*`). To override the access rules without changing any other auto-configured features add a `@Bean` of type `WebSecurityConfigurerAdapter` with `@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)`.

28.1 OAuth2

If you have `spring-security-oauth2` on your classpath you can take advantage of some auto-configuration to make it easy to set up Authorization or Resource Server.

28.1.1 Authorization Server

To create an Authorization Server and grant access tokens you need to use `@EnableAuthorizationServer` and provide `security.oauth2.client.client-id` and `security.oauth2.client.client-secret]` properties. The client will be registered for you in an in-memory repository.

Having done that you will be able to use the client credentials to create an access token, for example:

```
$ curl client:secret@localhost:8080/oauth/token -d grant_type=password -d usernam
```

The basic auth credentials for the `/token` endpoint are the `client-id` and `client-secret`. The user credentials are the normal Spring Security user details (which default in Spring Boot to "user" and a random password).

To switch off the auto-configuration and configure the Authorization Server features yourself just add a `@Bean` of type `AuthorizationServerConfigurer`.

28.1.2 Resource Server

To use the access token you need a Resource Server (which can be the same as the Authorization Server). Creating a Resource Server is easy, just add `@EnableResourceServer` and provide some configuration to allow the server to decode access tokens. If your application is also an Authorization Server it already knows how to decode tokens, so there is nothing else to do. If your app is a standalone service then you need to give it some more configuration, one of the following options:

- `security.oauth2.resource.user-info-uri` to use the `/me` resource (e.g. `uaa.run.pivotal.io/userinfo` on PWS)
- `security.oauth2.resource.token-info-uri` to use the token decoding endpoint (e.g. `uaa.run.pivotal.io/check_token` on PWS).

If you specify both the `user-info-uri` and the `token-info-uri` then you can set a flag to say that one is preferred over the other (`prefer-token-info=true` is the default).

Alternatively (instead of `user-info-uri` or `token-info-uri`) if the tokens are JWTs you can configure a `security.oauth2.resource.jwt.key-value` to decode them locally (where the key is a verification key). The verification key value is either a symmetric secret or PEM-encoded RSA public key. If you don't have the key and it's public you can provide a URI where it can be downloaded (as a JSON object with a "value" field) with `security.oauth2.resource.jwt.key-uri`. E.g. on PWS:

```
$ curl https://uaa.run.pivotal.io/token_key
{"alg":"SHA256withRSA","value":"-----BEGIN PUBLIC KEY-----\nMIIBI...\n-----END PUI
```

If you use the `security.oauth2.resource.jwt.key-uri` the authorization server needs to be running when your application starts up. It will log a warning if it can't find the key, and tell you what to do to fix it.

28.2 Token Type in User Info

Google, and certain other 3rd party identity providers, are more strict about the token type name that is sent in the headers to the user info endpoint. The default is "Bearer" which suits most providers and matches the spec, but if you need to change it you can set `security.oauth2.resource.token-type`.

28.3 Customizing the User Info RestTemplate

If you have a `user-info-uri`, the resource server features use an `OAuth2RestTemplate` internally to fetch user details for authentication. This is provided as a qualified `@Bean` with id `userInfoRestTemplate`, but you shouldn't need to know that to just use it. The default should be fine for most providers, but occasionally you might need to add additional interceptors, or change the request authenticator (which is how the token gets attached to outgoing requests). To add a customization just create a bean of type `UserInfoRestTemplateCustomizer` - it has a single method that will be called after the bean is created but before it is initialized. The rest template that is being customized here is *only* used internally to carry out authentication.

To set an RSA key value in YAML use the "pipe" continuation marker to split it over multiple lines ("|") and remember to indent the key value (it's a standard YAML language feature). Example:

```
security:
    oauth2:
        resource:
            jwt:
                keyValue: |
                    -----BEGIN PUBLIC KEY-----
                    MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC...
                    -----END PUBLIC KEY-----
```

28.3.1 Client

To make your webapp into an OAuth2 client you can simply add `@EnableOAuth2Client` and Spring Boot will create an `OAuth2RestTemplate` for you to `@Autowire`. It uses the `security.oauth2.client.*` as credentials (the same as you might be using in the Authorization Server), but in addition it will need to know the authorization and token URIs in the Authorization Server. For example:

application.yml.

```
security:
    oauth2:
        client:
            clientId: bd1c0a783ccdd1c9b9e4
            clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1
            accessTokenUri: https://github.com/login/oauth/access_token
```

```
            clientAuthenticationScheme: form
```

An application with this configuration will redirect to Github for authorization when you attempt to use the `OAuth2RestTemplate`. If you are already signed into Github you won't even notice that it has authenticated. These specific credentials will only work if your application is running on port 8080 (register your own client app in Github or other provider for more flexibility).

To limit the scope that the client asks for when it obtains an access token you can set `security.oauth2.client.scope` (comma separated or an array in YAML). By default the scope is empty and it is up to Authorization Server to decide what the defaults should be, usually depending on the settings in the client registration that it holds.

There is also a setting for `security.oauth2.client.client-authentication-scheme` which defaults to "header" (but you might need to set it to "form" if, like Github for instance, your OAuth2 provider doesn't like header authentication). In fact, the `security.oauth2.client.*` properties are bound to an instance of `AuthorizationCodeResourceDetails` so all its properties can be specified.

In a non-web application you can still `@Autowire` an `OAuth2RestOperations` and it is still wired into the `security.oauth2.client.*` configuration. In this case it is a "client credentials token grant" you will be asking for if you use it (and there is no need to use `@EnableOAuth2Client` or `@EnableOAuth2Sso`). To switch it off, just remove the `security.oauth2.client.client-id` from your configuration (or make it the empty string).

28.3.2 Single Sign On

An OAuth2 Client can be used to fetch user details from the provider (if such features are available) and then convert them into an `Authentication` token for Spring Security. The Resource Server above support this via the `user-info-uri` property This is the basis for a Single Sign On (SSO) protocol based on OAuth2, and Spring Boot makes it easy to participate by providing an annotation `@EnableOAuth2Sso`. The Github client above can protect all its resources and authenticate using the Github `/user/` endpoint, by adding that annotation and declaring where to find the endpoint (in addition to the `security.oauth2.client.*` configuration already listed above):

application.yml.

```
security:
    oauth2:
...
    resource:
        userInfoUri: https://api.github.com/user
        preferTokenInfo: false
```

Since all paths are secure by default, there is no “home” page that you can show to unauthenticated users and invite them to login (by visiting the `/login` path, or the path specified by `security.oauth2.sso.login-path`).

To customize the access rules or paths to protect, so you can add a “home” page for instance, `@EnableOAuth2Sso` can be added to a `WebSecurityConfigurerAdapter` and the annotation will cause it to be decorated and enhanced with the necessary pieces to get the `/login` path working. For example, here we simply allow unauthenticated access to the home page at "/" and keep the default for everything else:

```
@Configuration
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    public void init(WebSecurity web) {
        web.ignore("/");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/**").authorizeRequests().anyRequest().authenticated();
    }

}
```

28.4 Actuator Security

If the Actuator is also in use, you will find:

- The management endpoints are secure even if the application endpoints are insecure.
- Security events are transformed into `AuditEvents` and published to the `AuditService`.
- The default user will have the `ADMIN` role as well as the `USER` role.

The Actuator security features can be modified using external properties (`management.security.*`). To override the application access rules add a `@Bean` of type `WebSecurityConfigurerAdapter` and use `@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)` if you *don't* want to override the actuator access rules, or `@Order(ManagementServerProperties.ACCESS_OVERRIDE_ORDER)` if you *do* want to override the actuator access rules.

Prev | Up | Next
27. Developing web applications | Home | 29. Working with SQL databases