Spring Boot Security — Core Concepts Explained

Owner: NII NARTEY
Reviewer: THOMAS DARKO

Introduction

Dealing with security risks becomes crucial for the application's long-term sustainability and development for every Java developer out there at some time. While Java is not a very easy language to learn, Spring Boot may be claimed to make it a lot easier. In this circumstance, Spring Security simplifies our job and provides alternatives for setting our application.

Terms and Concepts

Spring Security does not secure networks or computers, but applications : it is involved in the dialogue between the application and the user (or between two applications). This dialogue is managed by the Spring *Dispatcher Servlet*, which redirects http requests to the controller classes of the application. Concisely, the role of Spring Security is to insert operations in this interaction, thanks to a bunch of *Servlet Filters*. This group of filters is the **Filter Chain** of Spring Security.

Filters

A filter is a fundamental notion in Spring Boot Security since it intercepts each request before it reaches the servlets.

We may create filters in any combination, and they will prevent the request from progressing to the next filter unless all of the requirements are fulfilled.

Familiarising ourselves with these notions will tremendously assist us in navigating the authentication and authorization setup.

The *filter chain* deals with 2 fundamental concepts :

- **authentication** : the user must be identified by a **username**/**password** combination.
- **authorizations** : users are not equal regarding the operations they are allowed to do.

  As an example, a user who is not an administrator should not be allowed to modify the account of another user.

These fundamentals are explained further below:

Authentication — *Who are you? Can you prove it?*

Authentication is the process of ascertaining whether or not someone or something is who or what they claim to be. There are several methods of authentication based on the credentials a user must supply when attempting to log in to an application. The first form is **knowledge-based authentication** (*KBA*), which is based on the individual's knowledge. We may place username-password authentication in this category. Following that is **possession-based authentication** (*PBA*), which is based on something the user possesses. In this case, phone/text messaging, key cards and badges, and so on can be utilised. The third type of authentication is **multi-factor authentication** (*MFA*), which is a mixture of the first two and requires the user to give two or more verification factors in order to obtain access to the application.

Authorization — *Can you do this?*

Many people mistakenly use the phrases authentication and authorization interchangeably, although they are not the same thing. Simply defined, authorization is the process of determining which precise **rights/privileges/resources** a person *has*, whereas authentication is the act of determining who someone *is*. As a result, authorization is the act of **granting someone permission** to do or have something. Furthermore, authorization is frequently viewed as both the preparatory setting up of permissions and the actual checking of permission values when a user is granted access.

Principal

A principal is a person who is authenticated through the authentication procedure. Consider it a **presently logged-in user** of the currently logged-in account. One person can have many IDs (for example, Google), but there is generally only one logged in user. You may get it from the security context, which is connected to the current thread and hence to the current request and its session. As a result, the Spring Security principle can only be accessed as an Object and must be cast to the appropriate UserDetails instance, as I will describe in more detail in the future blogs.

Granted Authority

These are the user's fine-grained permissions that define what the user may perform. Each conferred power can be thought of as an individual privilege. *READ_AUTHORITY* and

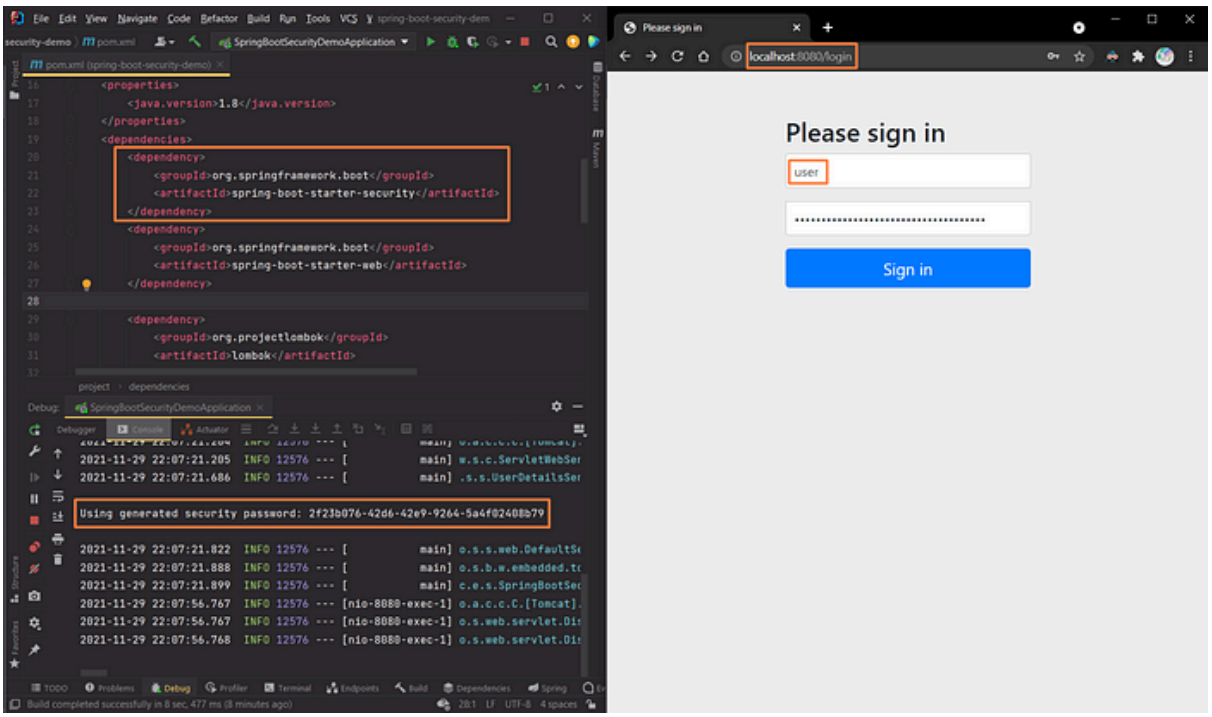
WRITE_AUTHORITY are two examples. The name is arbitrary in this context, and we may alternatively refer to the idea of authority by using privilege.

Role

This is a coarser-grained set of authorities granted to the user (for example, *ROLE_TEACHER, ROLE_STUDENT*…). A role is expressed as a String that begins with "ROLE." The semantics we attach to how we utilise the feature is the primary distinction between granted authority and role.

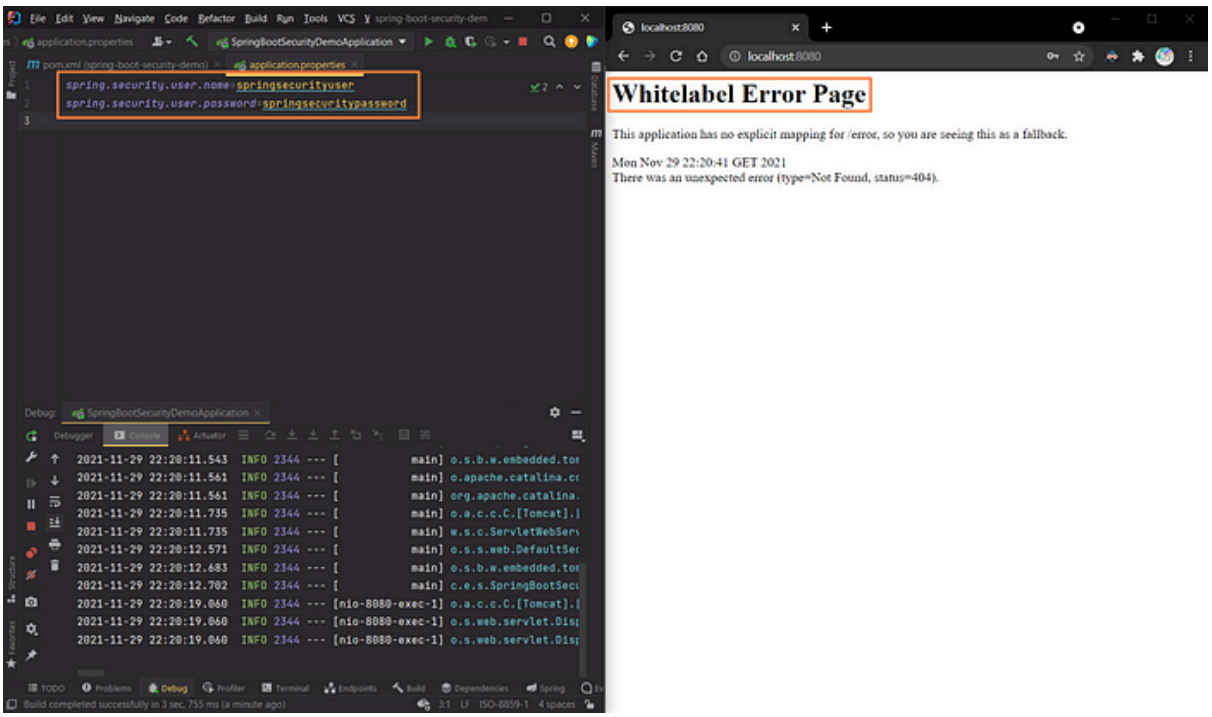
Spring Boot Starter Dependency

Spring Boot includes a *spring-boot-starter-security* package that collects all Spring Security dependencies in one place. Adding this dependency to the classpath causes the login form to be created automatically. The username is "user" by default, and the password is logged in the app, as seen in the image.



Spring Security Form Login

The password is automatically refreshed each time the program is relaunched, however we may change this setting and specify the desired username and password in the *application.properties* file. This might be beneficial for debugging. The password will no longer be logged in the software after that. We'll get the Spring Boot default Whitelabel Error Page after successfully logging in.



Overriding default settings from application.properties

All of this is handled by the filters that sit behind the servlets that serve as the framework's foundation.

Hereafter is an ordered list of the filters constituting the default *filter chain*, with a quick explanation of their action . This list aims at breaking the perceived complexity of Spring Security, but you won't need to read it at all to understand the rest of the article.

- **WebAsyncManagerIntegrationFilter** : this guy is like the cement between the *SecurityContext* and the *WebAsyncManager*, allowing it to populate the *SecurityContext* for each request.

- **SecurityContextPersistenceFilter** : sends information from the *SecurityContextRepository* to the *SecurityContextHolder*, the latter being necessary for the authentication process, which requires a valid *SecurityContext*.
- **HeaderWriterFilter** : adds headers to the current request. This Filter is particularly useful for preventing certain common attacks by adding headers like X-Frame-Options, X-XSS-Protection and X-Content-Type-Options (see the part about general protection below).
- **CsrfFilter** : adds protection against *Cross-Site Request Forgery* attacks, by involving a token, which is usually saved as a cookie in the *HttpSession*. It is common to call this filter before any request that may change the state of the application (mainly POST, PUT, DELETE and sometimes OPTIONS).
- **LogoutFilter** : manages the sign-out process by calling multiple *lougoutHandlers* in charge of clearing the security context, invalidating the user session, and redirecting to the default page…
- **UsernamePasswordAuthenticationFilter** : analyses an authentication form submission, which must provide a username and a password. This filter is activated by default on the URL */login*.
- **DefaultLoginPageGeneratingFilter** : builds a default authentication page, unless being explicitly unactivated. This filter is the reason why a login page shows up when implementing Spring Security, even before the developer builds a custom one.

- **DefaultLogoutPageGeneratingFilter** : builds a default logout page, unless being explicitly unactivated.
- **BasicAuthenticationFilter** : checks if a request includes a *basic auth header*, and attempts to sign in with the *username & password* read from this *header*.
- **RequestCacheAwareFilter** : checks in the cache if the current request is similar to an old one in order to quicken its processing.
- **SecurityContextHolderAwareRequestFilter** : offers multiple functionalities for each request, particularly concerning the authentication process : getting the user informations, checking if they are authenticated (and if not, offering the possibility to authenticate via the *Authentication Manager*), getting their roles, offering to sign out via the logout handlers, maintaining the *Security Context* across multiple threads…
- **AnonymousAuthenticationFilter** : provides an *Authentication* object to the *Security Context Holder* if there is none.
- **SessionManagementFilter** : checks whether a user is authenticated since the beginning of the request, and in that case, proceeds to session related verifications (as an example, checks whether multiple concurrent logins are currently in use).
- **ExceptionTranslationFilter** : Manage the *AccessDeniedException* and *AuthenticationException* raised by the *filterChain*. This filter is essential for the GUI to remain consistent when errors occur, for it is the bridge between Java exceptions and *http* responses.

- **FilterSecurityInterceptor** : checks whether the user's roles match the authorization requirements for the current request.

This filter list is the core of Spring Security, no more, no less. It implements the authentication and authorization concepts, and offers some additional features aiming at preventing the common attacks (see the part about general protection below). The developper just needs to configure how these filters should be used in the application : which URLs/methods to protect, which database entries to use for authentication/authorization…

Conclusion

Learning how to implement Spring Security is never finished : there is always something else to know. An exhaustive knowledge of the subject is nevertheless not required to provide decent security to your application. Here are the 3 bullet points you want to memorise, if nothing else :

➔ Spring Security is mostly a sequence of filters to use (or not) and to configure.

➔ Security provided by Spring Security is based on the authentication / authorization duo.

➔ The filters and URL protections are configured in the @EnableWebSecurity annotated class : that is the class to read in order to quickly understand how Spring Security is implemented in an application.

Sources :

- spring security architecture
- spring security reference documentation
- spring security reference expression language
- spring security reference headers

Articles & blogs :

- Marco Behler guide
- Matt Raible article