Servlet Authentication Architecture

Servlet Authentication Architecture

This discussion expands on Servlet Security: The Big Picture to describe the main architectural components of Spring Security's used in Servlet authentication. If you need concrete flows that explain how these pieces fit together, look at the Authentication Mechanism specific sections.

- SecurityContextHolder - The `SecurityContextHolder` is where Spring Security stores the details of who is authenticated.
- SecurityContext - is obtained from the `SecurityContextHolder` and contains the `Authentication` of the currently authenticated user.
- Authentication - Can be the input to `AuthenticationManager` to provide the credentials a user has provided to authenticate or the current user from the `SecurityContext`.
- GrantedAuthority - An authority that is granted to the principal on the `Authentication` (i.e. roles, scopes, etc.)
- AuthenticationManager - the API that defines how Spring Security's Filters perform authentication.
- ProviderManager - the most common implementation of `AuthenticationManager`.
- AuthenticationProvider - used by `ProviderManager` to perform a specific type of authentication.
- Request Credentials with `AuthenticationEntryPoint` - used for requesting credentials from a client (i.e. redirecting to a log in page, sending a `WWW-Authenticate` response, etc.)
- AbstractAuthenticationProcessingFilter - a base `Filter` used for authentication. This also gives a good idea of the high level flow of authentication and how pieces work together.

SecurityContextHolder

At the heart of Spring Security's authentication model is the `SecurityContextHolder`. It contains the SecurityContext.

The `SecurityContextHolder` is where Spring Security stores the details of who is authenticated. Spring Security does not care how the `SecurityContextHolder` is populated. If it contains a value, it is used as the currently authenticated user.

The simplest way to indicate a user is authenticated is to set the `SecurityContextHolder` directly:

Setting `SecurityContextHolder`

Java | **Kotlin**

```
SecurityContext context = SecurityContextHolder.createEmptyContext(); 1
Authentication authentication =
    new TestingAuthenticationToken("username", "password", "ROLE_USER"); 2
context.setAuthentication(authentication);

SecurityContextHolder.setContext(context); 3
```

1. We start by creating an empty `SecurityContext`. You should create a new `SecurityContext` instance instead of using `SecurityContextHolder.getContext().setAuthentication(authentication)` to avoid race conditions across multiple threads.
2. Next, we create a new `Authentication` object. Spring Security does not care what type of `Authentication` implementation is set on the `SecurityContext`. Here, we use `TestingAuthenticationToken`, because it is very simple. A more common production scenario is `UsernamePasswordAuthenticationToken(userDetails, password, authorities)`.
3. Finally, we set the `SecurityContext` on the `SecurityContextHolder`. Spring Security uses this information for authorization.

To obtain information about the authenticated principal, access the `SecurityContextHolder`.

Access Currently Authenticated User

Java | **Kotlin**

```java
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
String username = authentication.getName();
Object principal = authentication.getPrincipal();
Collection<? extends GrantedAuthority> authorities = authentication.getAuthorities();
```

By default, `SecurityContextHolder` uses a `ThreadLocal` to store these details, which means that the `SecurityContext` is always available to methods in the same thread, even if the `SecurityContext` is not explicitly passed around as an argument to those methods. Using a `ThreadLocal` in this way is quite safe if you take care to clear the thread after the present principal's request is processed. Spring Security's FilterChainProxy ensures that the `SecurityContext` is always cleared.

Some applications are not entirely suitable for using a `ThreadLocal`, because of the specific way they work with threads. For example, a Swing client might want all threads in a Java Virtual Machine to use the same security context. You can configure `SecurityContextHolder` with a strategy on startup to specify how you would like the context to be stored. For a standalone application, you would use the `SecurityContextHolder.MODE_GLOBAL` strategy. Other applications might want to have threads spawned by the secure thread also assume the same security identity. You can achieve this by using `SecurityContextHolder.MODE_INHERITABLETHREADLOCAL`. You can change the mode from the default `SecurityContextHolder.MODE_THREADLOCAL` in two ways. The first is to set a system property. The second is to call a static method on `SecurityContextHolder`. Most applications need not change from the default. However, if you do, take a look at the JavaDoc for `SecurityContextHolder` to learn more.

SecurityContext

The `SecurityContext` is obtained from the SecurityContextHolder. The `SecurityContext` contains an Authentication object.

Authentication

The `Authentication` interface serves two main purposes within Spring Security:

- An input to `AuthenticationManager` to provide the credentials a user has provided to authenticate. When used in this scenario, `isAuthenticated()` returns `false`.
- Represent the currently authenticated user. You can obtain the current `Authentication` from the SecurityContext.

The `Authentication` contains:

- `principal` : Identifies the user. When authenticating with a username/password this is often an instance of `UserDetails` .
- `credentials` : Often a password. In many cases, this is cleared after the user is authenticated, to ensure that it is not leaked.
- `authorities` : The `GrantedAuthority` instances are high-level permissions the user is granted. Two examples are roles and scopes.

GrantedAuthority

`GrantedAuthority` instances are high-level permissions that the user is granted. Two examples are roles and scopes.

You can obtain `GrantedAuthority` instances from the `Authentication.getAuthorities()` method. This method provides a `Collection` of `GrantedAuthority` objects. A `GrantedAuthority` is, not surprisingly, an authority that is granted to the principal. Such authorities are usually "roles", such as `ROLE_ADMINISTRATOR` or `ROLE_HR_SUPERVISOR` . These roles are later configured for web authorization, method authorization, and domain object authorization. Other parts of Spring Security interpret these authorities and expect them to be present. When using username/password based authentication `GrantedAuthority` instances are usually loaded by the `UserDetailsService` .

Usually, the `GrantedAuthority` objects are application-wide permissions. They are not specific to a given domain object. Thus, you would not likely have a `GrantedAuthority` to represent a permission to `Employee` object number 54, because if there are thousands of such authorities you would quickly run out of memory (or, at the very least, cause the application to take a long time to authenticate a user). Of course, Spring Security is expressly designed to handle this common requirement, but you should instead use the project's domain object security capabilities for this purpose.

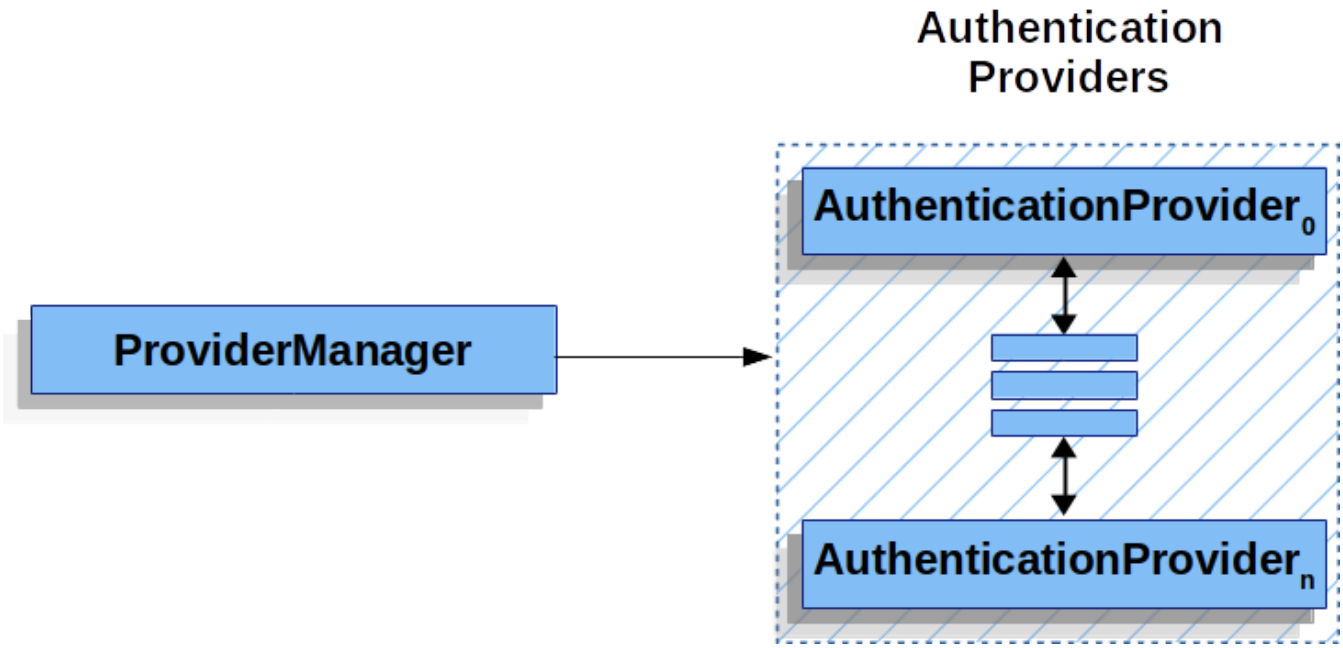
AuthenticationManager

`AuthenticationManager` is the API that defines how Spring Security's Filters perform authentication. The `Authentication` that is returned is then set on the SecurityContextHolder by the controller (that is, by Spring Security's `Filters` instances) that invoked the `AuthenticationManager` . If you are not integrating with Spring Security's `Filters` instances, you can set the `SecurityContextHolder` directly and are not required to use an `AuthenticationManager` .

While the implementation of `AuthenticationManager` could be anything, the most common implementation is `ProviderManager` .

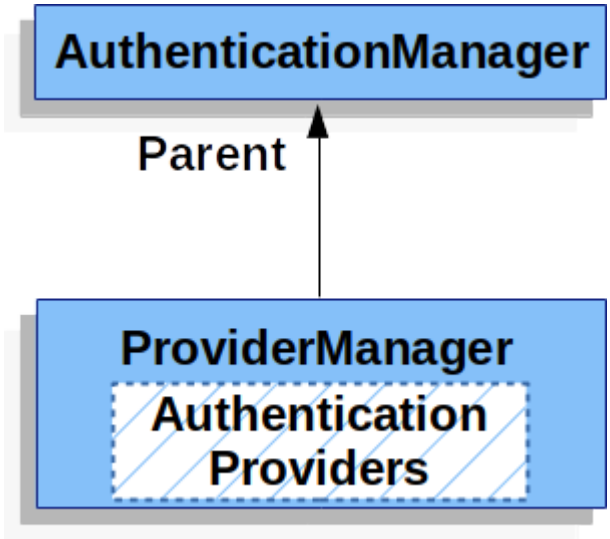
ProviderManager

`ProviderManager` is the most commonly used implementation of `AuthenticationManager`. `ProviderManager` delegates to a `List` of `AuthenticationProvider` instances. Each `AuthenticationProvider` has an opportunity to indicate that authentication should be successful, fail, or indicate it cannot make a decision and allow a downstream `AuthenticationProvider` to decide. If none of the configured `AuthenticationProvider` instances can authenticate, authentication fails with a `ProviderNotFoundException`, which is a special `AuthenticationException` that indicates that the `ProviderManager` was not configured to support the type of `Authentication` that was passed into it.

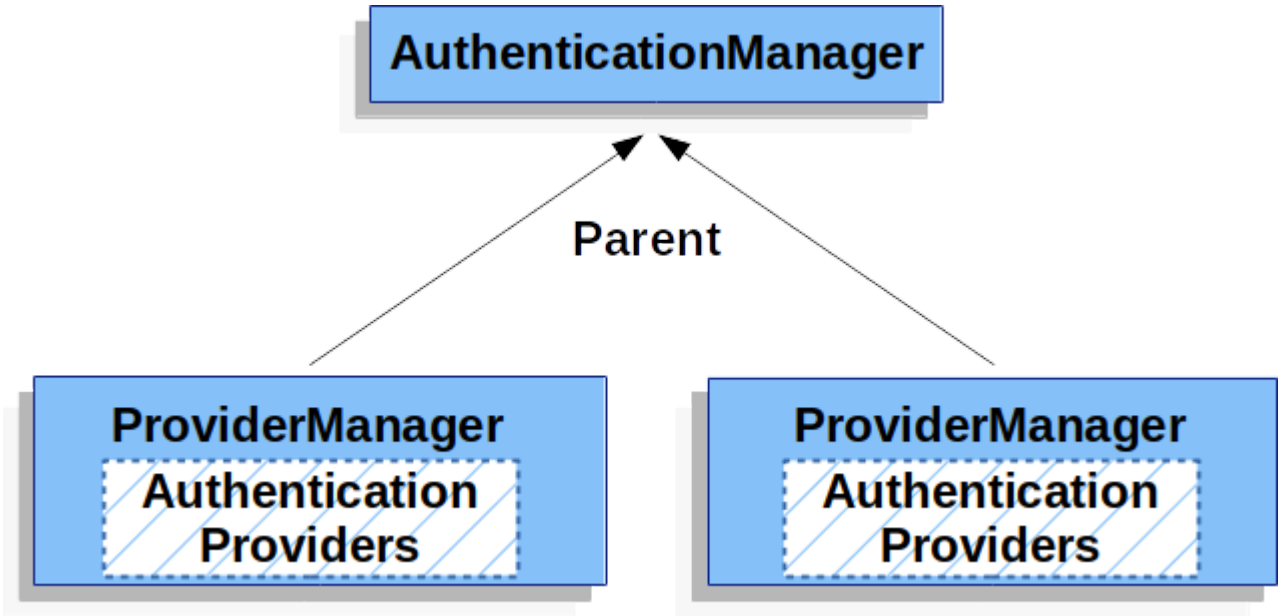


In practice each `AuthenticationProvider` knows how to perform a specific type of authentication. For example, one `AuthenticationProvider` might be able to validate a username/password, while another might be able to authenticate a SAML assertion. This lets each `AuthenticationProvider` do a very specific type of authentication while supporting multiple types of authentication and expose only a single `AuthenticationManager` bean.

`ProviderManager` also allows configuring an optional parent `AuthenticationManager`, which is consulted in the event that no `AuthenticationProvider` can perform authentication. The parent can be any type of `AuthenticationManager`, but it is often an instance of `ProviderManager`.



In fact, multiple `ProviderManager` instances might share the same parent `AuthenticationManager`. This is somewhat common in scenarios where there are multiple `SecurityFilterChain` instances that have some authentication in common (the shared parent `AuthenticationManager`), but also different authentication mechanisms (the different `ProviderManager` instances).



By default, `ProviderManager` tries to clear any sensitive credentials information from the `Authentication` object that is returned by a successful authentication request. This prevents information, such as passwords, being retained longer than necessary in the `HttpSession`.

This may cause issues when you use a cache of user objects, for example, to improve performance in a stateless application. If the `Authentication` contains a reference to an object in the cache (such as a `UserDetails` instance) and this has its credentials removed, it is no longer possible to authenticate against the cached value. You need to take this into account if you use a cache. An obvious solution is to first make a copy of the object, either in the cache implementation or in the `AuthenticationProvider` that creates the returned `Authentication` object. Alternatively, you can

disable the `eraseCredentialsAfterAuthentication` property on `ProviderManager`. See the Javadoc for the ProviderManager class.

AuthenticationProvider

You can inject multiple `AuthenticationProvider`s instances into `ProviderManager`. Each `AuthenticationProvider` performs a specific type of authentication. For example, `DaoAuthenticationProvider` supports username/password-based authentication, while `JwtAuthenticationProvider` supports authenticating a JWT token.

Request Credentials with `AuthenticationEntryPoint`

`AuthenticationEntryPoint` is used to send an HTTP response that requests credentials from a client.

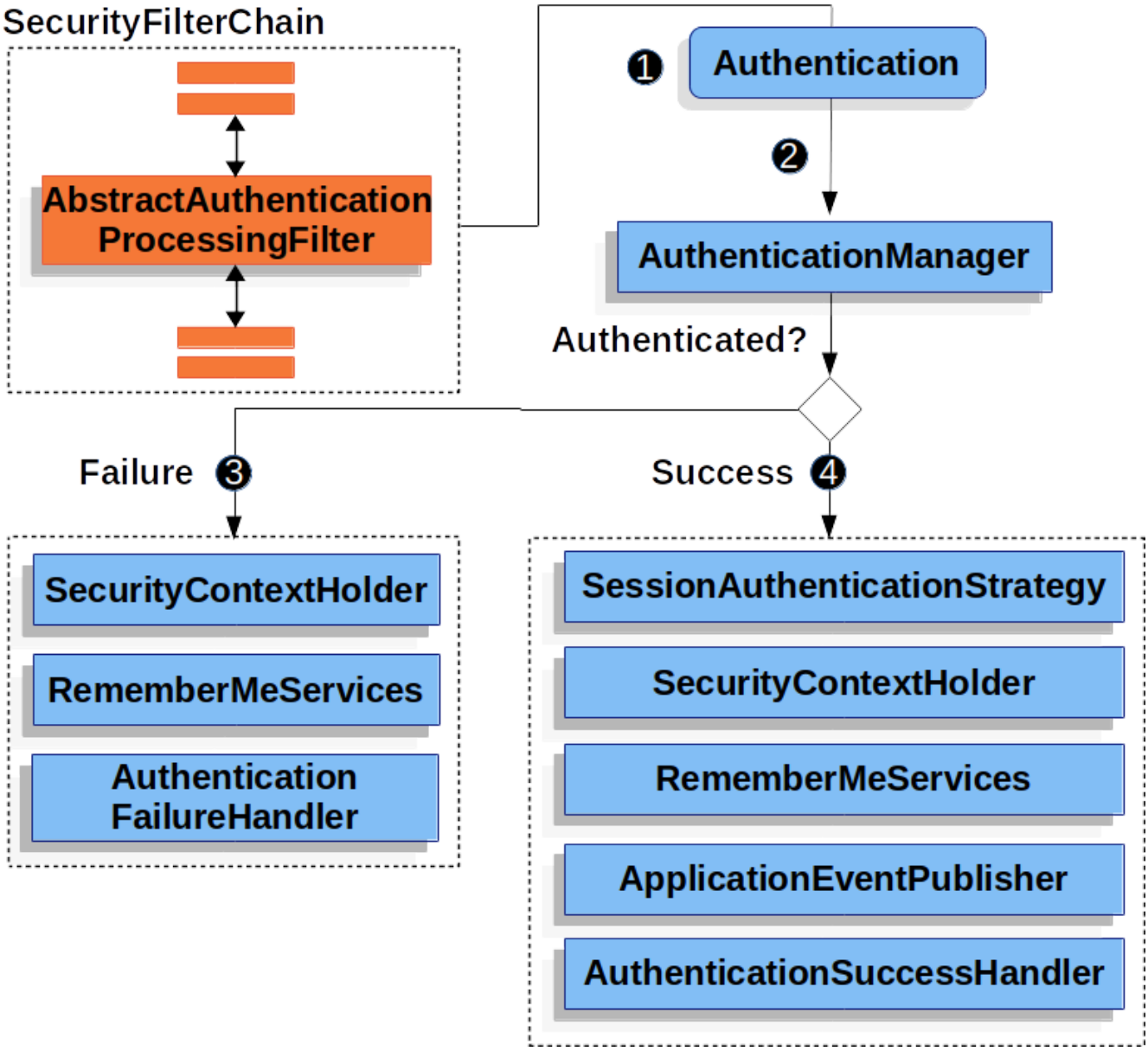
Sometimes, a client proactively includes credentials (such as a username and password) to request a resource. In these cases, Spring Security does not need to provide an HTTP response that requests credentials from the client, since they are already included.

In other cases, a client makes an unauthenticated request to a resource that they are not authorized to access. In this case, an implementation of `AuthenticationEntryPoint` is used to request credentials from the client. The `AuthenticationEntryPoint` implementation might perform a redirect to a log in page, respond with an WWW-Authenticate header, or take other action.

AbstractAuthenticationProcessingFilter

`AbstractAuthenticationProcessingFilter` is used as a base `Filter` for authenticating a user's credentials. Before the credentials can be authenticated, Spring Security typically requests the credentials by using `AuthenticationEntryPoint`.

Next, the `AbstractAuthenticationProcessingFilter` can authenticate any authentication requests that are submitted to it.



1 When the user submits their credentials, the `AbstractAuthenticationProcessingFilter` creates an `Authentication` from the `HttpServletRequest` to be authenticated. The type of `Authentication` created depends on the subclass of `AbstractAuthenticationProcessingFilter`. For example, `UsernamePasswordAuthenticationFilter` creates a `UsernamePasswordAuthenticationToken` from a *username* and *password* that are submitted in the `HttpServletRequest`.

2 Next, the `Authentication` is passed into the `AuthenticationManager` to be authenticated.

3 If authentication fails, then *Failure*.

- The SecurityContextHolder is cleared out.
- `RememberMeServices.loginFail` is invoked. If remember me is not configured, this is a no-op. See the `rememberme` package.
- `AuthenticationFailureHandler` is invoked. See the `AuthenticationFailureHandler` interface.

4 If authentication is successful, then *Success*.

- `SessionAuthenticationStrategy` is notified of a new login. See the `SessionAuthenticationStrategy` interface.
- The Authentication is set on the SecurityContextHolder. Later, if you need to save the `SecurityContext` so that it can be automatically set on future requests, `SecurityContextRepository#saveContext` must be explicitly invoked. See the `SecurityContextHolderFilter` class.
- `RememberMeServices.loginSuccess` is invoked. If remember me is not configured, this is a no-op. See the `rememberme` package.
- `ApplicationEventPublisher` publishes an `InteractiveAuthenticationSuccessEvent`.
- `AuthenticationSuccessHandler` is invoked. See the `AuthenticationSuccessHandler` interface.