Get started with the Java EE 8 Security API, Part 4: Interrogating caller data with SecurityContext

Authenticate and authorize user access across servlet and EJB containers

Alex Theedom

April 12, 2018

This final article in the Java EE Security API series introduces the SecurityContext API, which is used to interrogate caller data consistently across servlet and EJB containers. Find out how SecurityContext extends HttpAuthenticationMechanism's declarative capabilities, then put it to work testing caller data in a servlet container example.

About this series
The new and long-awaited Java EE Security API (JSR 375) ushers Java enterprise security into the cloud and microservices computing era. This series shows you how the new security mechanisms simplify and standardize security handling across Java EE container implementations, then gets you started using them in your cloud-enabled projects.

The previous article in this series introduced `IdentityStore`, an abstraction used to setup and configure secure access to user credential data in Java™ web applications. While developers can combine `IdentityStore` with `HttpAuthenticationMechanism` for powerful, built-in authentication and authorization, `HttpAuthenticationMechanism`'s declarative security model is insufficient for some security needs. This is where the `SecurityContext` API comes in.

In this article, you'll learn how to use `SecurityContext` to extend HttpAuthenticationMechanism programmatically, thus enabling your web applications to deny or grant access to application resources. Note that examples in this article are based on a servlet container.

Get the code

Installing Soteria

We'll use the Java EE 8 Security API reference implementation, Soteria, to explore the `SecurityContext` interface. You can get Soteria in one of two ways.

1. Explicitly specify Soteria in your POM

Use the following Maven coordinates to specify Soteria in your POM:

Listing 1. Maven coordinates for the Soteria project

```
<dependency>
  <groupId>org.glassfish.soteria</groupId>
  <artifactId>javax.security.enterprise</artifactId>
  <version>1.0</version>
</dependency>
```

2. Use built-in Java EE 8 coordinates

Java EE 8-compliant servers will have their own implementation of the new Java EE 8 Security API, or they'll rely on Sotoria's implementation. In either you only need the Java EE 8 coordinates:

Listing 2. Java EE 8 Maven coordinates

```
<dependency>
 <groupId>javax</groupId>
 <artifactId>javaee-api</artifactId>
 <version>8.0</version>
 <scope>provided</scope>
</dependency>
```

The `SecurityContext` interface is located in the `javax.security.enterprise` package.

What SecurityContext does

The `SecurityContext` API was created to provide a consistent approach to application security across servlet and EJB containers. The *security context* provides access to security-related information associated with the currently authenticated user, which can programmatically trigger the start of a web-based authentication process.

Servlet and EJB containers implement security context objects similarly, but with variation. For example, to obtain a user's identity within the servlet container you would use an `HttpServletRequest` instance and call the `getUserPrincipal()` method to return a UserPrincipal object. In the EJB container, you would call a method of the same name on an `EJBContext` instance. Likewise, if you wanted to test whether a user belonged to a certain role, you would call the `isUserRole()` method on the `HttpServletRequest` instance in the servlet container. In an EJB container you would call the `isCallerInRole()` method on the `EJBContext` instance.

The new `SecurityContext` resolves these and other inconsistencies by providing a single mechanism for programmatically obtaining authentication and authorization information across servlet and EJB containers. The new Java EE 8 Security API specification stipulates that `SecurityContext` must be available in servlet and EJB containers compatible with Java EE 8. Some server vendors may also make `SecurityContext` available in other containers.

How SecurityContext works

The SecurityContext interface provides an entry point for programmatic security and is an injectable type. It consists of the following five methods, none of which has a default implementation.

Caller data methods

- **The `getCallerPrincipal()` method** obtains the container-specific principal representing the name of the currently authenticated user. It returns `null` if the current caller is not authenticated. The principal type returned might be different from the type originally established by `HttpAuthenticationMechanism`. The important difference between this `getCallerPrincipal()` method and the method of the same name on the `EJBContext` interface is that it returns an instance of `Principal` with a null name for a user that is unauthenticated.
- **The `getPrincipalsByType()` method** returns all `Principals` of the specified type from the authenticated caller's `Subject`; if the type is not found or the current user is not authenticated then an empty `Set` is returned. You might use this method for a scenario where the container's caller principal was of a different type from the application's caller principal, or for an application needing information only available from the application's caller principal.
- **The `isCallerInRole()` method** determines if the caller is included in the role passed in as a `String`. It returns `true` if the user has the role; otherwise it returns `false`. The result returned by calling this method will be the same as if a container-specific call had been made. Calling `HttpServletRequest.isUserInRole()` or `EJBContext.isCallerInRole()` will return true if `SecurityContext.isUserInRole()` returns true.

Additional methods

- **The `hasAccessToWebResource()` method** determines whether or not the caller has access to the given web resource for the given HTTP method in the current application. This is configured in the application's security constraints, in accordance with Servlet 4.0's specification on security constraints.
- **The `authenticate()` method** programmatically triggers the container to start or continue an HTTP-based authentication conversation with the caller as if the client has made the call to access the resource. This method is dependent on a valid servlet context because it requires an `HttpServletRequest` and `HttpServletResponse` instance. This method only works in the servlet container.

Now that you have an overview of the methods and how they function, we'll take a look at some code examples. All examples that follow are for `SecurityContext` methods in a Servlet 4.0 web application.

Example 1: Testing caller data in a servlet

SecurityContext's getCallerPrincipal(), getPrincipalsByType(), and isCallerInRole() methods

Listing 3 combines `SecurityContext`'s three methods used to test caller data into one servlet, in order to demonstrate their use. In the example below, the `SecurityContext` is available as a CDI bean and therefore can be injected into any context-aware instance.

Listing 3. Caller data methods in a servlet container example

```
@WebServlet("/securityContextServlet")
@ServletSecurity(@HttpConstraint(rolesAllowed = "admin"))
public class SecurityContextServlet extends HttpServlet {

   @Inject
   private SecurityContext securityContext;
   @Override
   public void doGet(HttpServletRequest request, HttpServletResponse response)
                                                           throws IOException {

       // Example 1: Is the caller is one of the three roles: admin, user and demo
       PrintWriter pw = response.getWriter();

       boolean role = securityContext.isCallerInRole("admin");
       pw.write("User has role 'admin': " + role + "\n");

       role = securityContext.isCallerInRole("user");
       pw.write("User has role 'user': " + role + "\n");

       role = securityContext.isCallerInRole("demo");
       pw.write("User has role 'demo': " + role + "\n");


       // Example 2: What is the caller principal name
       String contextName = null;
       if (securityContext.getCallerPrincipal() != null) {
           contextName = securityContext.getCallerPrincipal().getName();
       }
       response.getWriter().write("context username: " + contextName + "\n");

       // Example 3: Retrieve all CustomPrincipal
       Set<CustomPrincipal> customPrincipals = securityContext
                                          .getPrincipalsByType(CustomPrincipal.class);
       for (CustomPrincipal customPrincipal : customPrincipals) {
           response.getWriter().write((customPrincipal.getName()));
       }

   }

}
```

In the first example, the security context is used to test the logical roles in which the currently authenticated user participates. The roles being tested are admin, user, and demo.

In the second example, you see how to use the `getCallerPrincipal()` method to retrieve the platform-specific caller principal representing the name of the authenticated caller. This method returns `null` if the current user is not authenticated, so the appropriate `null` check must be done.

The final example shows how to use the `getPrincipalsByType()` method to retrieve a set of principals by type.

Next, in Listing 4, you see a custom principal implementing the `Principal` interface. The call to `getPrincipalsByType()` method will retrieve a set of this type of principal.

Listing 4. Custom principal

```
public class CustomPrincipal implements Principal {

   private final String name;

   public CustomPrincipal(String name) {
       this.name = name;
   }

   @Override
   public String getName() {
       return name;
   }
}
```

Example 2: Testing caller access to a web resource

SecurityContext’s hasAccessToWebResources() method

Listing 5 shows how to use `hasAccessToWebResource()` to test a caller’s access to a given web resource for a specified HTTP method. In this case I’ve injected the `SecurityContext` instance into the servlet and called `hasAccessToWebResource()`. We want to test if the caller has `GET` access to the resource locate at the URI `/secretServlet` (show in Listing 6), so we pass the shown arguments to the method. If the caller has the admin role the method will return `true`; otherwise it will return `false`.

Listing 5. SecurityContext’s hasAccessToWebResource()

```
@WebServlet("/hasAccessServlet")
public class HasAccessServlet extends HttpServlet {

   @Inject
   private SecurityContext securityContext;

   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse res)
                                         throws ServletException, IOException {

       boolean hasAccess = securityContext.hasAccessToWebResource("/secretServlet", "GET");

   }
}
```

Listing 6. Resource to test for access

```
@WebServlet("/secretServlet")
@ServletSecurity(@HttpConstraint(rolesAllowed = "admin"))
public class SecretServlet extends HttpServlet { }
```

Example 3: Authenticating caller access

SecurityContext’s authenticate() method

The final example shows how to use the `authenticate()` method to validate user-entered credentials. First, the user enters a username and password into the JSF shown in Listing 7. Once submitted, the `LoginBean` processes and authenticates the credentials, as shown in Listing 8.

Listing 7. Login form

```
<form jsf:id="form">
 <p>
   <strong>Username </strong>
   <input jsf:id="username" type="text"    jsf:value="#{loginBean.username}" />
 </p>
 <p>
   <strong>Password </strong>
   <input jsf:id="password" type="password" jsf:value="#{loginBean.password}" />
 </p>
 <p>
   <input type="submit" value="Login" jsf:action="#{loginBean.login}" />
 </p>
</form>
```

The `username` and `password` entered are set on the `LoginBean` (Listing 8) in order to generate a `Credential` instance. This credential is then used to create an `AuthenticationParameters` instance. This instance is passed to the `authenticate()` method together with the `HttpServletRequest` and `HttpServletResponse` instances, which are retrieved from the `FacesContext`. The `AuthenticationParameters` instance then returns a value of the `AuthenticationStatus` enum.

The `AuthenticationStatus` enum indicates the status of the authentication process and can be one of the following values:

- **`NOT_DONE`**: The authentication mechanism was called but it decided not to authenticate. Typically, this status would be returned in preemptive security.
- **`SEND_CONTINUE`**: The authentication mechanism was called and a multi-step authentication conversation with the caller has been initiated.
- **`SUCCESS`**: The authentication mechanism was called and the caller was successfully authenticated. The caller principal is available.
- **`SEND_FAILURE`**: The authentication mechanism was called but the caller was not successfully authenticated and therefore the caller principal is not available.

Note that in Java EE 8 the `FacesContext` has been made injectable in JSF 2.3.

Listing 8. Login bean processes and authenticates caller credentials

```
@Named
@RequestScoped
@FacesConfig(version = JSF_2_3)
public class LoginBean {

   @Inject
   private SecurityContext securityContext;

   @Inject
   private FacesContext facesContext;

   private String username, password;

   public void login() {

       Credential credential = new UsernamePasswordCredential(username, new Password(password));
```

```
        AuthenticationStatus status = securityContext.authenticate(
            getRequestFrom(facesContext),
            getResponseFrom(facesContext),
            withParams().credential(credential));

        if (status.equals(SEND_CONTINUE)) {
            facesContext.responseComplete();
        } else if (status.equals(SEND_FAILURE)) {
            addError(facesContext, "Authentication failed");
        }

    }

    private static HttpServletResponse getResponseFrom(FacesContext context) {
        return (HttpServletResponse) context
            .getExternalContext()
            .getResponse();
    }

    private static HttpServletRequest getRequestFrom(FacesContext context) {
        return (HttpServletRequest) context
            .getExternalContext()
            .getRequest();
    }


    // Getter and setters omitted
}
```

Conclusion

With this series you've seen how the new Java EE 8 Security API integrates some of the most popular and dependable Java EE technologies into common enterprise authentication and authorization routines. Among other features, the Java developer community asked for a simplified security model that was consistent over servlet and EJB containers, and that is exactly what `SecurityContext` delivers.

While my examples are based on a servlet container, `SecurityContext` makes it simple to interrogate caller principals consistently across servlet and EJB containers. Developers who have had to combine XML and annotation-based configuration for recent Java EE apps will rejoice at the move to a pure annotations framework. The new Security API also supports XML declarations, which makes migrating older projects to Java EE 8 relatively simple and stress free, without any immediate need to change security configurations.

I hope you've enjoyed this series and are able to put your new knowledge into practice. Be sure to test your understanding in the final quiz below.

Test your knowledge

1. Which of the following methods belong to the `SecurityContext` interface?
    a. `getCallerPrincipal()`
    b. `isUserRole()`
    c. `getPrincipalsByType()`
    d. `isCallerInRole()`
    e. `isCallerPrincipal()`

2. What does the `hasAccessToWebResource()` method test?
    a. If the specified user has access to the given resource
    b. If the servlet has rights to access to resource
    c. If the caller has access to the specified resource
    d. If the caller has access to the remote web resource
3. What does the `getPrincipalsByType()` method return?
    a. A set of `Principal`s of the given type from the callers `Subject`
    b. The `Principal` of the given type from the context
    c. A list of `Principal`s of the given type
    d. `Null` if the caller is not authorized
    e. An empty set if the caller is not authorized
4. Which of these are behaviors of the `getCallerPrincipal()` method?
    a. Returns the name of the authenticated caller
    b. Returns `null` if the current caller is not authenticated
    c. Returns a set of the caller’s `Principal`s
    d. Returns the platform-specific `Principal` for the authenticated caller

Check your answers.

Related topics

- Java EE Security API specification
- GitHub for the Java EE Security API specification
- Soteria reference implementation
- Java EE Security API implementation
- Presentation of the pre-final version of the new Java Security API at Devoxx 2017
- Alex's book: Java EE 8: Only What's New