IBM

developerWorks®

Get started with the Java EE 8 Security API, Part 2: Web authentication with HttpAuthenticationMechanism

Classic and custom Servlet authentication with Java EE 8's new annotation-driven HTTP authentication mechanism

Alex Theedom

March 09, 2018

HttpAuthenticationMechanism's annotation-driven approach is a welcome departure from the tedium of manually configuring authentication for Java web applications. Learn how to setup and configure both classic Servlet 4.0-style authentication and custom solutions using HttpAuthenticationMechanism and the new Java EE 8 Security API.

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 first article in this series presented an overview of the Java EE Security API (JSR 375), including a high-level introduction to the new `HttpAuthenticationMechanism`, `IdentityStore`, and `SecurityContext` interfaces. In this article, the first of three deep dives, you'll learn how to use `HttpAuthenticationMechanism` to setup and configure user authentication in an example Java web application.

The `HttpAuthenticationMechanism` interface is the heart of Java™ EE's new HTTP authentication mechanism. It comes with three built-in CDI (contexts and dependency injection)-enabled implementations, which are instantiated automatically and made available for use by the CDI container. These built-in implementations support three classic authentication methods specified by Servlet 4.0: basic HTTP authentication, form-based authentication, and custom form-based authentication.

In addition to the built-in authentication methods, you have the option to use `HttpAuthenticationMechanism` to develop custom authentication. You might choose this option if you needed to support specific protocols and authentication tokens. Some servlet containers may also offer custom `HttpAuthenticationMechanism` implementations.

In this article you'll get a hands-on introduction to using the `HttpAuthenticationMechanism` interface and its three built-in implementations. I'll also show you how to write your own custom `HttpAuthenticationMechanism` authentication mechanism.

Get the code

Installing Soteria

We'll use the Java EE 8 Security API reference implementation, Soteria, to explore both built-in and custom authentication mechanisms accessible via `HttpAuthenticationMechanism`. 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>
```

Built-in authentication mechanisms

The built-in HTTP authentication mechanisms support authentication styles specified for Servlet 4.0 (section 13.6). In the next sections I'll show you how to use annotations to initiate the three authentication mechanisms, as well as how to setup and implement each one in a Java web application.

@BasicAuthenticationMechanismDefinition

The `@BasicAuthenticationMechanismDefinition` annotation triggers HTTP basic authentication as defined by Servlet 4.0 (section 13.6.1). It has one optional parameter, `realmName`, which specifies the name of the realm that will be sent via the `WWW-Authenticate` header. Listing 3 shows how to trigger HTTP basic authentication for the realm name `user-realm`.

Listing 3. HTTP basic authentication mechanism

```
@BasicAuthenticationMechanismDefinition(realmName="user-realm")
@WebServlet("/user")
@DeclareRoles({ "admin", "user", "demo" })
@ServletSecurity(@HttpConstraint(rolesAllowed = "user"))
public class UserServlet extends HttpServlet { … }
```

@FormAuthenticationMechanismDefinition

The `@FormAuthenticationMechanismDefinition` annotation provokes form-based authentication as defined by the Servlet 4.0 specification (section 13.6.3). It has one configuration option that must be set. The `loginToContinue` option accepts a configured `@LoginToContinue` annotation, which allows the application to provide "login to continue" functionality. You have the option to use reasonable defaults or specify one of four characteristics for this feature.

In Listing 4, the login page is specified to the URI `/login-servlet`. If authentication fails, flow is passed to `/login-servlet-fail`.

Listing 4. Form-based authentication mechanism

```
@FormAuthenticationMechanismDefinition(
 loginToContinue = @LoginToContinue(
     loginPage = "/login-servlet",
     errorPage = "/login-servlet-fail"
     )
)
@ApplicationScoped
public class ApplicationConfig { ... }
```

To set the manner of reaching the login page, use the `useForwardToLogin` option. To set this option to either "forward" or "redirect" you would specify `true` or `false`, with the default being `true`. Alternatively, you could set the value via an EL expression passed to the option: `useForwardToLoginExpression`.

The `@LoginToContinue` has reasonable defaults. The login page is set to `/login` and the error page is set to `/login-error`.

@CustomFormAuthenticationMechanismDefinition

The `@CustomFormAuthenticationMechanismDefinition` annotation provides options for configuring a custom login form. In Listing 5, you can see that the website's login page is identified as `login.do`. The login page is set as a value to the `loginPage` parameter to the `loginToContinue` parameter of the `@CustomFormAuthenticationMechanismDefinition` annotation. Note that `loginToContinue` is the only parameter, and it is optional.

Listing 5. Custom form configuration

```
@CustomFormAuthenticationMechanismDefinition(
   loginToContinue = @LoginToContinue(
       loginPage="/login.do"
   )
)
@WebServlet("/admin")
@DeclareRoles({ "admin", "user", "demo" })
@ServletSecurity(@HttpConstraint(rolesAllowed = "admin"))
public class AdminServlet extends HttpServlet { ... }
```

The `login.do` login page is shown in Listing 6 and is a JSF (JavaServer Pages) page supported by a login backing bean, as shown in Listing 7.

Listing 6. The login.do JSF login page

```
<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 login backing bean uses a `SecurityContext` instance to perform authentication, as shown in Listing 7. If successful, the user is given access to the resource; otherwise the flow is passed to the error page. In this case, it forwards the user to the default login URI at `/login-error`.

Listing 7. Login backing bean

```
@Named
@RequestScoped
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");
       }

   }
   // Some methods omitted for brevity
}
```

Writing a custom HttpAuthenticationMechanism

In many cases you'll find that the three built-in implementations are sufficient for your needs. In some cases, you may prefer to write your own implementation of the `HttpAuthenticationMechanism` interface. In this section I'll walk through the process of writing a custom `HttpAuthenticationMechanism` interface.

In order to ensure that it is available to your Java application, you will need to implement the `HttpAuthenticationMechanism` interface as a CDI bean with `@ApplicationScope`. The interface defines the following three methods:

- `validateRequest()` authenticates an HTTP request.
- `secureResponse()` secures the HTTP response message.
- `cleanSubject()` clears the subject of provided principals and credentials.

All methods accept the same parameter types, which are: `HttpServletRequest`, `HttpServletResponse`, and `HttpMessageContext`. These map to the corresponding methods defined on the JASPIC Server Auth Module interface, which is provided by the container. When a JASPIC method is invoked on `Server Auth`, it delegates to the corresponding method of your custom `HttpAuthenticationMechanism`.

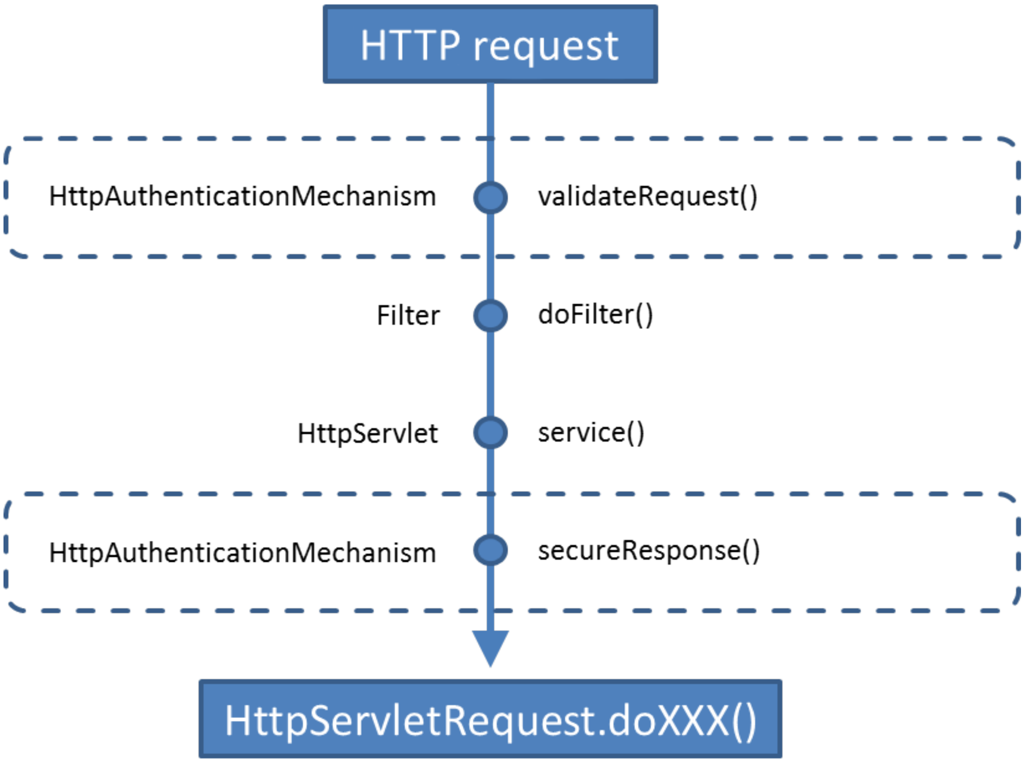
Listing 8. Custom HttpAuthenticationMechanism implementation

```
@ApplicationScoped
public class CustomAuthenticationMechanism implements HttpAuthenticationMechanism {

    @Inject
    private IdentityStoreHandler idStoreHandler;

    @Override
    public AuthenticationStatus validateRequest(HttpServletRequest req,
                  HttpServletResponse res,
               HttpMessageContext msg) {
        // use idStoreHandler to authenticate and authorize access
        return msg.responseUnauthorized(); // other responses available
    }
}
```

Method execution during an HTTP request

During an HTTP request, methods on the `HttpAuthenticationMechanism` implementation are called at fixed moments. Figure 1 shows when each method is called in relation to methods on `Filter` and `HttpServlet` instances.

Figure 1. Method call sequence



The `validateRequest()` method is invoked before the `doFilter()` or `service()` methods, and in response to calling `authenticate()` on the `HttpServletResponse` instance. The purpose of this method is to permit a caller to authenticate. To assist this action, the method has access to the caller's `HttpRequest` and `HttpResponse` instances. It can use these to extract authentication information for the request. It may also write to the HTTP response in order to redirect the caller to an OAuth provider. Following authentication, it can use the `HttpMessageContext` instance to advise the authentication state.

The `secureResponse()` method is called after `doFilter()` or `service()`. It provides post-processing functionality on a response generated by a servlet or filter. Encryption is a potential use for this method.

The `cleanSubject()` method is called following a call to the `logout()` method on an `HttpServletRequest` instance. This method also can be used to clear state related to user after a logout event.

The `HttpMessageContext` interface has methods that an `HttpAuthenticationMechanism` instance can use to communicate with the JASPIC `ServerAuthModule` that invoked it.

Custom example: Authentication with cookies

As I previously mentioned, you'll usually write a custom implementation in order to provide functionality not available from the built-in options. One example would be using cookies in your authentication flow.

At the class level, you may use the optional `@RememberMe` annotation to effectively "remember" a user authentication and apply it automatically with every request.

Listing 9. Use @RememberMe on a custom HttpAuthenticationMechanism

```
@RememberMe(
       cookieMaxAgeSeconds = 3600
)
@ApplicationScoped
public class CustomAuthenticationMechanism implements HttpAuthenticationMechanism { … }
```

This annotation has eight configuration options, all of which have sensible defaults, so you don't have to implement them manually:

- **`cookieMaxAgeSeconds`** sets the life of the "remember me" cookie.
- **`cookieMaxAgeSecondsExpression`** is the EL version of cookieMaxAgeSeconds.
- **`cookieSecureOnly`** specifies that the cookie should only be accessed via secure means (HTTPS).
- **`cookieSecureOnlyExpression`** is the EL version of cookieSecureOnly.
- **`cookieHttpOnly`** indicates that the cookie should be sent with HTTP requests only.
- **`cookieHttpOnlyExpression`** is the EL version of cookieHttpOnly.
- **`cookieName`** set the cookie's name.
- **`isRememberMe`** switches "remember me" on or off.
- **`isRememberMeExpression`** is the EL version of isRememberMe.

The `RememberMe` functionality is implemented as an *interceptor binding*. The container will intercept calls to `validateRequest()` and `cleanSubject()` methods. When the `validateRequest()` method is called on an implementation that includes a `RememberMe` cookie, it will attempt to authenticate the caller. If successful, the `HttpMessageConext` will be notified about a login event; otherwise the cookie will be removed. Intercepting the `cleanSubject()` method simply removes the cookie and completes the logout request.

Conclusion to Part 2

The new `HttpAuthenticationMechanism` interface is the heart of web authentication in Java EE 8. Its built-in authentication mechanisms support the three classic authentication methods specified in Servlet 4.0, and it is also very easy to extend the interface for custom implementations. In this tutorial you've learned how to use annotations to call and configure `HttpAuthenticationMechanism`'s built-in mechanisms, and how to write a custom mechanism for a special use case. I encourage you to test what you've learned with the quiz questions below.

This article is the first of three deep dives introducing major components of the new Java EE 8 Security API. The next two articles will be hands-on introductions to the `IdentityStore` and `SecurityContext` APIs.

Test your knowledge

1. What are the three default `HttpAuthenticationMechanism` implementations?
    a. `@BasicFormAuthenticationMechanismDefinition`
    b. `@FormAuthenticationMechanismDefinition`
    c. `@LoginFormAuthenticationMechanismDefinition`
    d. `@CustomFormAuthenticationMechanismDefinition`
    e. `@BasicAuthenticationMechanismDefinition`
2. Which two of the following annotations provoke form-based authentication?
    a. `@BasicAuthenticationMechanismDefinition`
    b. `@BasicFormAuthenticationMechanismDefinition`
    c. `@FormAuthenticationMechanismDefinition`
    d. `@FormBasedAuthenticationMechanismDefinition`
    e. `@CustomFormAuthenticationMechanismDefinition`
3. Which two of the following are valid configurations for basic authentication?
    a. `@BasicAuthenticationMechanismDefinition(realmName="user-realm")`
    b. `@BasicAuthenticationMechanismDefinition(userRealm="user-realm")`
    c. `@BasicAuthenticationMechanismDefinition(loginToContinue = @LoginToContinue)`
    d. `@BasicAuthenticationMechanismDefinition`
    e. `@BasicAuthenticationMechanismDefinition(realm="user-realm")`
4. Which three of the following are valid configurations for form-based authentication?
    a. `@FormAuthenticationMechanismDefinition(loginToContinue = @LoginToContinue)`
    b. `@FormAuthenticationMechanismDefinition`
    c. `@FormBasedAuthenticationMechanismDefinition`
    d. `@FormAuthenticationMechanismDefinition(loginToContinue = @LoginToContinue(useForwardToLoginExpression = "${appConfigs.forward}"))`
    e. `@FormBasedAuthenticationMechanismDefinition(loginToContinue = @LoginToContinue)`
5. During an HTTP request, in what order are methods called on the `HttpAuthenticationMechanism`, `Filter`, and `HttpServlet` implementations?
    a. `doFilter()`, `validateRequest()`, `service()`, `secureResponse()`
    b. `validateRequest()`, `doFilter()`, `secureResponse()`, `service()`
    c. `validateRequest()`, `service()`, `doFilter()`, `secureResponse()`

   d. `validateRequest()`, `doFilter()`, `service()`, `secureResponse()`
   e. `service()`, `secureResponse()`, `doFilter()`, `validateRequest()`
6. How do you set the maximum age for a `RememberMe` cookie?
   a. `@RememberMe(cookieMaxAge = (units = SECONDS, value = 3600)`
   b. `@RememberMe(maxAgeSeconds = 3600)`
   c. `@RememberMe(cookieMaxAgeSeconds = 3600)`
   d. `@RememberMe(cookieMaxAgeMilliseconds = 3600000)`
   e. `@RememberMe(cookieMaxAgeSeconds = "3600")`

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