Get started with the Java EE 8 Security API, Part 1: Java enterprise security for cloud and microservices platforms

Overview of the new HttpAuthenticationMechanism, IdentityStore, and SecurityContext interfaces

Alex Theedom

February 10, 2018

As one of the three core specifications introduced with Java EE 8, the new Java EE Security API is an essential addition to your Java EE toolkit, and thankfully not terribly difficult to learn. Find out how the Java EE Security API supports enterprise security in cloud and microservices platforms, while introducing modern capabilities such as context and dependency injection.

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.

Experienced Java™ developers know that Java does not suffer from a scarcity of Java security mechanisms. Options include the Java Authorization for Container Contracts specification (JACC), Java Authentication Service Provider Interface for Containers (JASPIC), and a plethora of third-party container-specific security APIs and configuration management solutions.

The trouble has not been lack of options but the absence of an enterprise standard. Without a standard, there has been little motivating vendors to consistently implement core features such as authentication, to upgrade proprietary solutions for newer technologies such as context and dependency injection (CDI) and Expression Language (EL), or to stay current with security developments for cloud and microservices architectures.

This series introduces the new Java EE Security API, starting with an overview of the API and its three primary interfaces: `HttpAuthenticationMechanism`, `IdentityStore`, and `SecurityContext`.

Get the code

A new standard for Java EE security

The movement to develop a Java EE security specification was galvanized by community feedback in the 2014 Java EE 8 survey. Simplifying and standardizing Java enterprise security was a priority for many survey respondents. Once formed, the JSR 375 expert group identified the following issues:

- The various EJB and servlet containers comprising Java EE defined similar security-related APIs, but with subtly different syntax. For example, a servlet's call to check a user's role was `HttpServletRequest.isUserInRole(String role)`, while an EJB would call `EJBContext.isCallerInRole(String roleName)`.
- Existing security mechanisms like JACC were tricky to implement, and JASPIC could be difficult to use correctly.
- Existing mechanisms did not take advantage of modern Java EE programming features such as context and dependency injection (CDI).
- There was no portable way to control how authentication happened on the backend across containers.
- There was no standard support for managing identity stores or the configuration of roles and permissions.
- There was no standard support for deploying custom authentication rules.

These were the principal issues JSR 375 aimed to resolve. Simultaneously, the specification sought to enable developers to manage and control security themselves, by defining portable APIs for authentication, identity stores, roles and permissions, and authorizations across containers.

The beauty of the Java EE Security API is that it provides an alternative way to configure identity stores and authentication mechanisms, but does not replace existing security mechanisms. The Java EE Security API empowers developers to enable security in Java EE web applications in a consistent and portable manner—with or without vendor-specific or proprietary solutions.

What's in the Java EE Security API?

Version 1.0 of the Java EE Security API includes a subset of the original draft proposal and focuses on technology that is relevant for cloud-native applications. Those features are:

- An API for authentication
- An identity store API
- A security context API

These features are brought together with new, standardized terminology for all Java EE security implementations. The remaining features, slated for inclusion in the next version of the Java EE Security specification, are:

- A password aliasing API
- A role/permission assignment API
- An API for authorization interceptors

Secure web authentication

The Java EE platform already specifies two mechanisms for authenticating users of web applications: Servlet 4.0 (JSR 369) provides a declarative mechanism that is suitable for general application configuration. For more robust authentication needs, JASPIC defines a service provider interface called `ServerAuthModule`, which supports the development of authentication modules to handle any credential type. Additionally, the Servlet Container Profile specifies how JASPIC should be integrated with the servlet container.

Both of these mechanisms are meaningful and effective, but each has its restrictions for web application developers.

The servlet container mechanism is constrained to supporting only a small range of credential types defined by Servlet 4.0, and it fails to support complex interactions with callers. It also fails to provide a way for applications to establish that callers are authenticated against a desired identity store.

Conversely JASPIC is very powerful and malleable, but it's also rather complicated to use. Coding an `AuthModule` and aligning it to the web container for authentication use can be tricky. On top of this, there is no declarative configuration for JASPIC, and no clear way to override a programmatically registered `AuthModule`.

The Java EE Security API resolves some of these issues with a new interface, `HttpAuthenticationMechanism`. Essentially a simplified, servlet-container variant of the JASPIC `ServerAuthModule` interface, the new interface leverages existing mechanisms while alleviating their restrictions.

An `HttpAuthenticationMechanism` instance is a CDI bean that the container is responsible for making available for injection. Additional implementations of the `HttpAuthenticationMechanism` interface may be provided by the application or the servlet container. Note that `HttpAuthenticationMechanism` is only specified for the servlet container.

Support for Servlet 4.0 authentication

A Java EE container must provide `HttpAuthenticationMechanism` implementations for three authentication mechanisms, which are defined in the Servlet 4.0 specification. The three implementations are:

- Basic HTTP authentication (section 13.6.1)
- Form-based authentication (section 13.6.3)
- Custom-form authentication (section 13.6.3.1)

Each implementation is triggered by the presence of its associated annotation:

- `@BasicAuthenticationMechanismDefinition`
- `@FormAuthenticationMechanismDefinition`

- `@CustomFormAuthenticationMechanismDefinition`

Upon encountering one of these annotations, the container will instantiate an instance of the associated mechanisms and make it immediately available.

In the new spec, it is no longer necessary to specify the authentication mechanism in the `web.xml` file between `<login-config>` elements, as was required by Servlet 4.0. In fact, the deployment process might fail —or at least ignore the `web.xml` configurations— if they are present when an HttpAuthenticationMechanism-based annotation is also present.

Let's take a look at examples of how each mechanism might be used.

Basic HTTP authentication

The `@BasicAuthenticationMechanismDefinition` annotation provokes basic HTTP authentication as defined by Servlet 4.0. Listing 1 shows an example. The only configuration parameter is optional, and allows a realm to be specified.

Listing 1. Basic HTTP Authentication

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

What is a realm?
A server resource can be partitioned into separate protected spaces. In this case, each will have its own authentication schema and authorization database, containing users and groups controlled by the same policy. This database of users and groups is known as a *realm*.

Form-based authentication

The `@FormAuthenticationMechanismDefinition` annotation is used for form-based authentication. It has one required parameter, `loginToContinue`, which is used to configure a web application's login page, error page, and redirect or forwarding characteristics. In Listing 2 you can see that the login page is defined with a URI and the `useForwardToLoginExpression` is configured using an Expression Language (EL) expression. There is no need to pass any parameters to the `@LoginToContinue` annotation because reasonable defaults are provided by the implementation.

Listing 2. Form-based authentication

```
@FormAuthenticationMechanismDefinition(
   loginToContinue = @LoginToContinue(
       loginPage="/login-servlet",
       errorPage="/error",
       useForwardToLoginExpression="${appConfig.forward}"
   )
)
@ApplicationScoped
public class ApplicationConfig { ... }
```

Custom-form authentication

The `@CustomFormAuthenticationMechanismDefinition` annotation triggers built-in custom-form authentication. Listing 3 shows an example.

Listing 3. Custom-form authentication

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

Custom-form authentication is intended to better align with JavaServer Pages (JSF) and related Java EE technologies. The `login.do` page is rendered and the username and password are entered and processed by the backing bean for the login page.

The IdentityStore API

An *identity store* is a database that stores user identity data such as user name, group membership, and information used to verify credentials. The Java EE Security API provides an identity-store abstraction called `IdentityStore`. Akin to the `JAAS LoginModule` interface, `IdentityStore` is used to interact with identity stores in order to authenticate users and retrieve group memberships.

As the specification is written, it is intended that `IdentityStore` is used by `HttpAuthenticationMechanism` implementations, but that isn't a requirement. `IdentityStore` can stand separate and be used by any other authentication mechanism. Nevertheless, using `IdentityStore` and `HttpAuthenticationMechanism` together enables an application to control the identity stores it uses for authentication in a portable and standard way, and is recommended for most use-case scenarios.

The `IdentityStore` API includes an `IdentityStoreHandler` interface, which the `HttpAuthenticationMechanism` must delegate to in order to validate a user credential. The `IdentityStoreHandler` then calls on the `IdentityStore` instance. `Identity` store implementations are not used directly but rather are interacted with via the dedicated handler.

The `IdentityStoreHandler` can authenticate against multiple `IdentityStores` and return an aggregate result in the form of a `CredentialValidationResult` instance. This object may do as little as relay whether the credentials are valid or not, or it could be a rich object containing any of the following information:

- `CallerPrincipal`
- A set of groups to which the principal belongs
- The caller's name or LDAP-distinguished name
- The caller's unique identifier from the identity store

Identity stores are queried in order, determined by the priority of each `IdentityStore` implementation. The list of stores is parsed twice: first for authentication and then for authorization.

As a developer, you may implement your own lightweight identity store by implementing the `IdentityStore` interface, or you may use one of the built-in `IdentityStores` for LDAP and RDBMS. These are initialized by passing configuration details to the appropriate annotation—either `@LdapIdentityStoreDefinition` or `@DataBaseIdentityStoreDefinition`.

Configuring a built-in IdentityStore

The simplest identity store is the *database store*. It is configured via the `@DataBaseIdentityStoreDefinition` annotation, as shown in Listing 4. The two built-in datastore annotations are based on the `@DataStoreDefinition` annotation already available in Java EE 7.

Listing 4 shows how to configure a database identity store. These configuration options are fairly self explanatory and should be familiar if you've ever configured a database definition.

Listing 4. Configuring a database identity store

```
@DatabaseIdentityStoreDefinition(
   dataSourceLookup = "${'java:global/permissions_db'}",
   callerQuery = "#{'select password from caller where name = ?'}",
   groupsQuery = "select group_name from caller_groups where caller_name = ?",
   hashAlgorithm = PasswordHash.class,
   priority = 10
)
@ApplicationScoped
@Named
public class ApplicationConfig { ... }
```

Note in Listing 4 that the priority is set to 10. This is used in case multiple identity stores are found and determines the iteration order relative to other stores. Lower numbers have higher priority.

The LDAP configuration is just as simple, as shown in Listing 5. If you have experience with LDAP configuration semantics you will find the options here familiar.

Listing 5. Configuring an LDAP identity store

```
@LdapIdentityStoreDefinition(
   url = "ldap://localhost:33389/",
   callerBaseDn = "ou=caller,dc=jsr375,dc=net",
   groupSearchBase = "ou=group,dc=jsr375,dc=net"
)
@DeclareRoles({ "admin", "user", "demo" })
@WebServlet("/admin")
public class AdminServlet extends HttpServlet { ... }
```

Customizing IdentityStore

Designing your own lightweight identity store is quite simple. You are required to implement the `IdentityStore` interface and at least the `validate()` method. There are four methods on the interface, all of which have default method implementations. The `validate()` method is the minimum required for a working identity store. It accepts an instance of `Credential` and returns an instance of `CredentialValidationResults`.

In Listing 6, the `validate()` method receives an instance of `UsernamePasswordCredential` containing login credentials to validate. It then returns an instance of `CredentialValidationResults`. If the simple configuration logic results in a successful authentication, this object is configured with the username and a set of groups to which the user belongs. If authentication fails, then the `CredentialValidationResults` instance contains only the status flag `INVALID`.

ao: updated the listing title. Is that right?

Listing 6. A custom, lightweight identity store

```
@ApplicationScoped
public class LiteWeightIdentityStore implements IdentityStore {
   public CredentialValidationResult validate(UsernamePasswordCredential userCredential) {
       if (userCredential.compareTo("admin", "pwd1")) {
           return new CredentialValidationResult("admin",
         new HashSet<>(asList("admin", "user", "demo")));
       }
       return INVALID_RESULT;
   }
}
```

Note that the implementation is annotated `@ApplicationScope`. This is required because the `IdentityStoreHandler` holds references to all `IdentityStore` bean instances managed by the CDI container. The `@ApplicationScope` annotation ensures that the instance is a CDI-managed bean, which is available to the entire application.

To use your lightweight identity store, you inject the `IdentityStoreHandler` into a custom `HttpAuthenticationMechanism`, as shown in Listing 7.

Listing 7. Injecting LiteWeightIdentityStore into a custom HttpAuthenticationMechanism

```
@ApplicationScoped
public class LiteAuthenticationMechanism implements HttpAuthenticationMechanism {
   @Inject
   private IdentityStoreHandler idStoreHandler;
   @Override
   public AuthenticationStatus validateRequest(HttpServletRequest req,
               HttpServletResponse res,
               HttpMessageContext context) {
       CredentialValidationResult result = idStoreHandler.validate(
                new UsernamePasswordCredential(
                        req.getParameter("name"), req.getParameter("password")));
       if (result.getStatus() == VALID) {
           return context.notifyContainerAboutLogin(result);
       } else {
           return context.responseUnauthorized();
       }
   }
}
```

The SecurityContext API

`IdentityStore` and `HttpAuthenticationMechanism` combine powerfully for user authentication and authorization, but the declarative model is not enough by itself. *Programmatic security* enables a

web application to perform the checks required to grant or deny access to application resources, and the `SecurityContext` API provides this functionality.

Currently, Java EE containers implement security context objects inconsistently. For example, the servlet container provides an `HttpServletRequest` instance on which the `getUserPrincipal()` method is called to obtain the `UserPrincipal` representing the user's identity. The EJB container then provides the differently named `EJBContext` instance, on which the same-named method is called. Likewise, if you want to test whether the user belongs to a certain role, you must call the method `isUserRole()` on the `HttpServletRequest` instance, and then call the `isCallerInRole()` on the EJBContext instance.

ao: Made this a sidebar.

What is the security context?
In a Java enterprise application, the *security context* provides access to security-related information associated with the current authenticated user. The goal of the SecurityContext API is to provide consistent access to an application's security context across all servlet and EJB containers.

The new `SecurityContext` provides a consistent mechanism across Java EE containers for obtaining authentication and authorization information. The new Java EE Security specification mandates that the `SecurityContext` be available in the servlet and EJB container as a minimum. Server vendors may also make it available in other containers.

Methods of the SecurityContext interface

The `SecurityContext` interface provides an entry point for programmatic security and is an injectable type. It has five methods, none of which have default implementations. Here's a list of the methods and their purposes:

- **Principal getCallerPrincipal();** returns the platform-specific principal representing the name of the current authenticated user or null if the current caller is not authenticated.
- **<T extends Principal> Set<T> getPrincipalsByType(Class<T> pType);** returns all principals of the given type from the authenticated caller's subject; if neither the `pType` type is found or the current user is not authenticated then an empty set is returned.
- **boolean isCallerInRole(String role);** determines whether or not the caller is included in the specified role; if the user is not authorized it returns false.
- **boolean hasAccessToWebResource(String resource, String... methods);** determines whether or not the caller has access to the given web resource via the methods provided.
- **AuthenticationStatus authenticate(HttpServletRequest req, HttpServletResponse res, AuthenticationParameters param);**: Informs the container that it should start or continue an HTTP-based authentication conversation with the caller. Being dependent on the `HttpServletRequest` and `HttpServletResponse` instances, this method only works in the servlet container.

We'll conclude with a quick look at using one of these methods to check a user's access to a web resource.

Using SecurityContext: An example

Listing 8 shows how you would use the `hasAccessToWebResource()` method to test a caller's access to a given web resource for a specified HTTP method. In this case, the `SecurityContext` instance is injected into the servlet and used in the `doGet()` method, where the caller's access to the `GET` method of the servlet located at the URI `/secretServlet` is tested.

Listing 8. Web resource access tested for caller

```
@DeclareRoles({"admin", "user", "demo"})
@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");
       if (hasAccess) {
           req.getRequestDispatcher("/secretServlet").forward(req, res);
       } else {
           req.getRequestDispatcher("/logout").forward(req, res);
       }
   }
}
```

Conclusion to Part 1

The new Java EE Security API successfully combines the power of existing authentication and authorization mechanisms with the ease of development developers expect from modern Java EE features and techniques.

While the initial drive for this API was the need for a consistent and portable way to solve security-related problems, there are improvements yet to come. In future versions, the JSR 375 expert group intends to integrate APIs for password aliasing, role and permission assignment, and intercepting authorization—all features that did not make it into the spec's v1.0.

The expert group also hopes to integrate features such as secret management and encryption, which are vital for common use cases in cloud-native and microservices applications. The 2016 Java EE Community survey additionally showed that OAuth2 and OpenID were voted the third-most important features for inclusion in Java EE 8. While time constraints ruled out these features for v1.0, there is a strong case and motivation for including these features in upcoming releases.

You've had an overview of the basic features and components of the new Java EE Security API, and I encourage you to test what you've learned with the quick quiz below. The next article will be a deep dive into the `HttpAuthenticationMechanism` interface and its three authentication mechanisms supporting Servlet 4.0.

Test your understanding

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 will trigger the built-in LDAP and RDBMS identity stores?
   a. `@LdapIdentityStore`
   b. `@DataBaseIdentityStore`
   c. `@DataBaseIdentityStoreDefinition`
   d. `@LdapIdentityStoreDefinition`
   e. `@RdbmsBaseIdentityStoreDefinition`
3. Which of the following statements are true?
   a. The `IdentityStore` can only be used by implementations of `HttpAuthenticationMechanism`.
   b. The `IdentityStore` can be used by any built-in or bespoke security solution.
   c. The `IdentityStore` should only be accessed via injected implementations of the `IdentityStoreHandler`.
   d. The `IdentityStore` cannot be used by implementations of `HttpAuthenticationMechanism`.
4. What is the goal of the `SecurityContext`?
   a. To provide consistent access to security context across servlet and EJB containers.
   b. To provide consistent access to security context to only EJB containers.
   c. To provide consistent access to security context across all containers.
   d. To provide consistent access to security context to the servlet container.
   e. To provide consistent access to security context across EJB containers.
5. Why must `HttpAuthenticationMechanism` implementations be `@ApplicationScoped`?
   a. To ensure that it is a CDI-managed bean and available to the entire application.
   b. So that the `HttpAuthenticationMechanism` can be used at all application levels.
   c. So that there is one instance of the `HttpAuthenticationMechanism` for each user.
   d. `JsonAdapter`.
   e. This is an untrue statement.

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