Java EE Security API Specification

Version 1.0, August 1, 2017

Table of Contents

License

Preface

This document is the Java EE Security API Specification, version 1.0.

Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119, "Key words for use in RFCs to Indicate Requirement Levels" [RFC2119].

Other documents referenced by this specification are identified by name on first use, and thereafter by a short abbreviation. The "Bibliography" section at the end of this document provides full references.

Acknowledgments

The author would like to thank the JSR-375 Expert Group and Contributors, and, in particular, Arjan Tijms, whose contributions were critical to the success of the JSR.

Specification Lead

Will Hopkins (Oracle)

Expert Group

Adam Bien	David Blevins (Tomitribe)
Rudy De Busscher	Ivar Grimstad
Les Hazlewood (Stormpath, Inc.)	Will Hopkins (Oracle)
Werner Keil	Matt Konda (Jemurai)
Alexander Kosowski (Oracle)	Darran Lofthouse (Red Hat)
Jean-Louis Monteiro (Tomitribe)	Ajay Reddy (IBM)
Pedro Igor Silva (Red Hat)	Arjan Tijms

Contributors

Guillermo González de Agüero	John Hogan
Elder Moraes	Fatih Mutluay
Reza Rahman	Ashley Richardson

Chapter 1. Concepts and General Requirements

This chapter overview information and terminology related to this specification, and also includes a general requirements not specified elsewhere in this document.

1.1. Terminology And Acronyms

A common understanding of security-related terms is helpful for discussion or specification of security APIs. To that end, we incorporate by reference the excellent Apache Shiro Terminology [SHIROTERM], and define some additional terms used in this document.

Authentication Mechanism

The mechanism by which authentication is performed. This mechanism interacts with the caller to obtain credentials and invokes an identity store to match the given credentials with a known user (identity). If a match is found, the Authentication Mechanism uses the found identity to populate attributes (principals) to build an authenticated Subject. If a match is not found, the Authentication Mechanism reports a failed authentication, the caller is not logged in, and is unable to be given authorization.

Caller, Caller Principal

A caller is a user that is making a request to an application, or invoking an application API. A Caller Principal is a Principal object representing that user. This specification uses the term caller in preference to the term user in most contexts.

HAM

Abbreviation for *HttpAuthenticationMechanism*, an interface defined by this specification.

Identity Store

An Identity Store is a component that can access application-specific security data such as users, groups, roles, and permissions. It can be thought of as a security-specific DAO (Data Access Object). Synonyms: security provider, repository, store, login module (JAAS), identity manager, service provider, relying party, authenticator, user service. Identity Stores usually have a 1-to-1 correlation with a data source such as a relational database, LDAP directory, file system, or other similar resource. As such, implementations of the *IdentityStore* interface use data source-specific APIs to discover authorization data (roles, permissions, etc), such as JDBC, File IO, Hibernate or JPA, or any other Data Access API.

JACC

JSR-115, "Java Authorization Contract for Containers", version 1.5 [JACC].

JASPIC

JSR-196, "Java Authentication SPI for Containers", version 1.1 [JASPIC].

SAM

Abbreviation for *ServerAuthModule*, an interface defined by JASPIC.

1.2. General Requirements

The following general requirements are defined by this specification.

1.2.1. Group-To-Role Mapping

Various Java EE specifications define how roles are declared for an application, and how access to application resources can be restricted to users that have a specific role. The specifications are largely silent on the question of how users are assigned to roles, however. Most application servers have proprietary mechanisms for determining the roles a user has.

Application servers MUST provide a default mapping from group names to roles. That is, a caller who is a member of group "foo" is considered to have role "foo". This default mapping MAY be overridden by explicit proprietary configuration, but, when not overridden, provides sensible and predictable behavior for portable applications.

An application server MAY provide a default mapping from caller principal names to roles. That is, a caller with the name "bar" is considered to have role "bar". This default mapping MAY be overridden by proprietary configuration.

1.2.2. Caller Principal Types

This specification defines a principal type called *CallerPrincipal* to represent the identity of an application caller. Historically, application servers have used different principal types to represent an application’s callers, and various Java EE specifications (e.g., JASPIC), provide abstractions to accommodate, "the container’s representation of the caller principal".

This specification RECOMMENDS that Java EE application servers that rely on container-specific caller principal types derive those types by extending *CallerPrincipal*, so that portable applications can rely on a consistent representation of the caller principal.

However, we also distinguish here between a "container caller principal" and an "application caller principal", and explicitly allow for each to be represented by a different *Principal* type.

The container caller principal is a *Principal* that the container uses to represent a caller’s identity. An implementation of this specification MAY choose any *Principal* type for this purpose. The type chosen may carry additional information, or provide unique behaviors.

An application caller principal is a *Principal* that an application, or an implementation of, e.g., an *HttpAuthenticationMechanism*, uses to represent a caller’s identity. An application MAY choose any *Principal* type for that purpose. The type chosen may carry additional information, or provide unique behaviors.

Because both containers and applications can have legitimate requirements for specific *Principal* types to represent a caller, and those types may differ, it MUST be possible for the container to establish both the container's and the application's caller principal as the caller's identity; for example, by including both in a Subject representing the caller.

When both a container caller principal and an application caller principal are present, the value obtained by calling *getName()* on both principals MUST be the same.

When no specific application caller principal is supplied during authentication, the caller's identity should be represented by a single principal, the container's caller principal.

1.2.3. Expression Language Support

This specification defines a number of annotations:

```
DatabaseIdentityStoreDefinition
LdapIdentityStoreDefinition

BasicAuthenticationMechanismDefinition
CustomFormAuthenticationMechanismDefinition
FormAuthenticationMechanismDefinition

LoginToContinue
RememberMe
```

Attributes on these annotations can be provided either as actual values, or as Expression Language 3.0 expressions. In cases where the return type of an attribute is not String, an "EL alternative" attribute is provided, with "Expression" appended to the name. If an "EL alternative" attribute has a non-empty value, it takes precedence over the attribute it's an alternative to, and must contain a valid EL expression that evaluates to the same type as the attribute it's an alternative to.

For more information, see the package javadoc for the javax.security.enterprise package.

Expression Language 3.0 is specified by JSR-341, "Expression Language", version 3.0 [EL30]

Chapter 2. Authentication Mechanism

This chapter describes the *HttpAuthenticationMechanism* interface and contract. *HttpAuthenticationMechanism* is used to authenticate callers of web applications, and is specified only for use in the servlet container. It is explicitly not defined for use with other containers (EJB, JMS, JCA, etc.).

2.1. Introduction

A web application consists of resources that can be accessed by any number of callers, who are initially unknown to the application. Callers make themselves known to the application through the process of authentication.

During authentication, the caller presents proof of identity — a token or credential of some kind — which the application (or container) then validates. If the proof is valid, the application (or container) establishes the caller's identity, then proceeds to the authorization step, in which it determines whether the caller has permission to access the requested resources.

In some cases (for example, username/password authentication) the interaction between the caller and the application is simple. In other cases, a lengthier dialog is required — an application may send a random nonce to the caller, which must then use that nonce in the construction of an authentication token, or there may be interactions with a third party that vouches for the caller's identity, or the authenticity of the provided credential.

The Java EE Platform already specifies mechanisms for authenticating users of web applications. JSR-340, "Java Servlet Specification", version 3.1 [SERVLET31] specifies a declarative mechanism for configuring an application to provide BASIC, DIGEST, FORM, or CERT authentication, with authentication performed automatically by the container based on the application's configuration, which, in the case of FORM authentication, can include custom form pages.

In addition, [JASPIC] specifies a general-purpose mechanism for securing messages sent between Java EE clients and servers. JASPIC defines an SPI called *ServerAuthModule*, which enables development of authentication modules to handle any credential type, or engage in interaction of arbitrary complexity with clients and third parties. [JASPIC] also defines the Servlet Container Profile, which specifies how JASPIC mechanisms, including *ServerAuthModules*, are integrated with the servlet container.

While both existing mechanisms are important and useful, each has limitations from the point of view of an application developer. The servlet container's *login-config* mechanism is limited to the *auth-method* types defined by [SERVLET31] — it doesn't support other credential types, or complex interactions with callers. It also relies on unspecified container mechanisms to associate identity stores with applications. There is no way for an application to ensure that callers are authenticated against the desired identity store, or, indeed, against *any* identity store.

JASPIC, by way of contrast, is extremely flexible and powerful, but is also complex. Writing an *AuthModule*, and arranging for the web container to use it for authentication, is a non-trivial exercise.

Additionally, there is no declarative configuration syntax for JASPIC, and there is no well-defined mechanism for a container to override an application’s programmatically-registered *AuthModule*. A container can choose to register its own *AuthModule*, or to remove one registered by an application, but JASPIC will always use the most-recently-registered module — the outcome is dependent on the order in which the application and the container attempt to register their respective modules.

The *HttpAuthenticationMechanism* interface is designed to capitalize on the strengths of existing authentication mechanisms, while mitigating the corresponding limitations. It is essentially a simplified, servlet-container-specific version of the JASPIC *ServerAuthModule* interface, retaining that interface’s flexibility and power, but reducing the cost of implementation. An *HttpAuthenticationMechanism* is a CDI bean, and is therefore made available to the container automatically by CDI. The container is responsible for placing the *HttpAuthenticationMechanism* in service.

An application MAY supply its own *HttpAuthenticationMechanism*, if desired. The servlet container MUST provide several default *HttpAuthenticationMechanism* implementations, which an application can select and configure via standard annotations. The container MAY also provide additional mechanisms beyond those required by this specification. The rules governing how the container selects an *HttpAuthenticationMechanism*, and how it is placed in service, are described in the "Installation and Configuration" section of this chapter. The required default mechanisms, and corresponding annotations, are described in the "Annotations and Built-In HttpAuthenticationMechanism Beans" section.

2.2. Interface and Theory of Operation

The *HttpAuthenticationMechanism* interface defines three methods that align closely with the methods defined by the JASPIC *ServerAuth* interface. The primary distinction is syntactic; unlike JASPIC, *HttpAuthenticationMechanism* is specified for the servlet container only, and can therefore reference servlet types in its method signatures. Only the *validateRequest()* method must be implemented; default behaviors are specified for the other two methods.

```
AuthenticationStatus validateRequest(HttpServletRequest request,
                                     HttpServletResponse response,
                                     HttpMessageContext httpMessageContext
                                    ) throws AuthenticationException;

AuthenticationStatus secureResponse(HttpServletRequest request,
                                    HttpServletResponse response,
                                    HttpMessageContext httpMessageContext
                                   ) throws AuthenticationException;

void cleanSubject(HttpServletRequest request,
                  HttpServletResponse response,
                  HttpMessageContext httpMessageContext);
```

Each method performs the same function as the corresponding *ServerAuth* method. At runtime, the methods will be invoked by a container-supplied *ServerAuthModule* that serves as a wrapper, or container, for the *HttpAuthenticationMechanism*. The container-supplied *ServerAuthModule* translates the method parameters passed to it, invokes the *HttpAuthenticationMechanism* method, and returns the resulting status to its caller. The behavior of the *HttpAuthenticationMechanism* methods should therefore be functionally equivalent to the behavior specified by the JASPIC Servlet Container Profile for the equivalent *ServerAuthModule* methods.

Summarized, this means:

- *validateRequest()* will be invoked before the *doFilter()* method of any servlet filter or the *service()* method of any servlet in the application for requests to constrained as well as to unconstrained resources, and, in addition, in response to application code calling the *authenticate()* method on the *HttpServletRequest*.
- *secureResponse()* will be invoked after the *doFilter()* method of any servlet filter or the *service()* method of any servlet in the application for requests to constrained as well as to unconstrained resources, but only if any of these two methods have indeed been invoked.
- *cleanSubject()* will be invoked in response to the application calling the *logout()* method on the *HttpServletRequest*.

The *validateRequest()* method is provided to allow a caller to authenticate. An implementation of this method can inspect the HTTP request to extract a credential or other information, or it can write to the HTTP response to, for example, redirect a caller to an OAuth provider, or return an error response. After a credential has been obtained and validated, the result of the validation can be communicated to the container using the *HttpMessageContext* parameter, which is described in more detail below.

The *secureResponse()* method is provided to allow post processing on the response generated by a servlet and/or servlet filter, such as encrypting it.

The *cleanSubject()* is provided to allow for cleanup after a caller is logged out. For example, an authentication mechanism that stores state within a cookie can remove that cookie here.

The *HttpMessageContext* interface defines methods that an *HttpAuthenticationMechanism* can invoke to communicate with the JASPIC *ServerAuthModule* (bridge module) that invokes it. The container MUST provide an implementation of the interface that supports the necessary container integrations.

The *HttpMessageContextWrapper* class implements a wrapper can be used, in a manner similar to *HttpServletRequestWrapper*, to provide custom behavior.

See javadoc for a detailed description of *HttpMessageContext* and *HttpMessageContextWrapper*. See below for more on the JASPIC bridge module.

2.3. Installation and Configuration

An *HttpAuthenticationMechanism* must be a CDI bean, and is therefore visible to the container through

CDI if it is packaged in a bean archive, which generally includes Java EE modules and application archives, as well as other archives and classes that are not part of an application, but are required by the Java EE specification to be visible to applications. See the CDI specification for details on bean archives and bean discovery. An *HttpAuthenticationMechanism* is assumed to be normal scoped.

It MUST be possible for the definition of an *HttpAuthenticationMechanism* to exist within the application archive. Alternatively such definition MAY also exists outside the application archive, for example in a jar added to the classpath of an application server.

An application packages its own *HttpAuthenticationMechanism* by including in a bean archive that is part of the application. Alternatively, it may select and configure one of the container's built-in mechanisms using the corresponding annotation, as described in the "Annotations and Built-In HttpAuthenticationMechanism Beans" section below.

The container decides which *HttpAuthenticationMechanism* to place in service using the following rules:

- The container MAY override an application's chosen *HttpAuthenticationMechanism* with one selected by the container, but SHOULD do so only if explicitly configured to.
- If the container does not override the application, it MUST place in service any *HttpAuthenticationMechanism* that is provided, either directly or via annotation, by the application.
- If the application makes more than one *HttpAuthenticationMechanism* available, either directly or via annotation or both, the results are undefined by this specification.
- If the application does not supply an *HttpAuthenticationMechanism*, or select one of the built-in mechanisms, the container MAY choose an *HttpAuthenticationMechanism* to place in service, but is NOT REQUIRED to do so.
- If the application does not make an *HttpAuthenticationMechanism* available, and the container does not choose one to place in service, then *HttpAuthenticationMechanism* is not used.

The container MUST use JASPIC when placing an *HttpAuthenticationMechanism* in service. The container MUST supply a "bridge" *ServerAuthModule* that integrates *HttpAuthenticationMechanism* with JASPIC. The bridge module MUST look up the correct *HttpAuthenticationMechanism* using CDI, then delegate to the *HttpAuthenticationMechanism* when the bridge module's methods are invoked. Since the method signatures and return values of the two interfaces are similar, but not the same, the bridge module MUST convert back and forth.

When an *HttpAuthenticationMechanism* is placed in service, the container MUST supply a bridge *ServerAuthModule* and the necessary supporting modules (*AuthContext*, *AuthConfig*, *AuthConfigProvider*), and arrange for the *AuthConfigProvider* to be registered with the JASPIC *AuthConfigFactory*, such that the bridge module is registered for the application context.

When an *HttpAuthenticationMechanism* is placed in service, the container MUST NOT register any *AuthConfigProvider* other than the one corresponding to the bridge *ServerAuthModule*. Given the nature of JASPIC, however, it's possible that some other entity could register a different *AuthConfigProvider* after the container has registered the bridge module's *AuthConfigProvider*. The container is NOT REQUIRED to prevent this.

2.4. Annotations and Built-In HttpAuthenticationMechanism Beans

A Java EE container MUST support built-in beans for the following *HttpAuthenticationMechanism* types, to be made available via configuration:

- BASIC - Authenticates according to the mechanism as described in 13.6.1, "HTTP Basic Authentication", in [SERVLET31]. See also RFC 7617, "The *Basic* HTTP Authentication Scheme" [RFC7617]. This bean is activated and configured via the *@BasicAuthenticationMechanismDefinition* annotation.
- FORM - Authenticates according to the mechanism as described in 13.6.3, "Form Based Authentication", in [SERVLET31]. This bean is activated and configured via the *@FormAuthenticationMechanismDefinition* annotation.
- Custom FORM - A variant on FORM, with the difference that continuing the authentication dialog as described in [SERVLET31], section 13.6.3, step 3, and further clarified in section 13.6.3.1, does not happen by posting back to j_security_check, but by invoking *SecurityContext.authenticate()* with the credentials the application collected. This bean is activated and configured via the *@CustomFormAuthenticationMechanismDefinition* annotation.

All of these beans MUST have the qualifier @Default and the scope @ApplicationScoped, as defined by the CDI specification.

All of the built-in beans MUST support authentication using *IdentityStore*, described in Chapter 3, "Identity Store", but MAY fall-back to container-specific methods if no *IdentityStore* is available.

See also the "Implementation Notes" section of this chapter.

The annotations are defined as shown in the following sections.

2.4.1. BASIC Annotation

The following annotation is used to configure the built-in BASIC authentication mechanism.

```
@Retention(RUNTIME)
@Target(TYPE)
public @interface BasicAuthenticationMechanismDefinition {

    /**
     * Name of realm that will be sent via the <code>WWW-Authenticate</code> header.
     * <p>
     * Note that this realm name <b>does not</b> couple a named identity store
     * configuration to the authentication mechanism.
     *
     * @return Name of realm
     */
    String realmName() default "";
}
```

2.4.2. FORM Annotation

The following annotation is used to configure the built-in FORM authentication mechanism.

```
@Retention(RUNTIME)
@Target(TYPE)
public @interface FormAuthenticationMechanismDefinition {

    @Nonbinding
    LoginToContinue loginToContinue();
}
```

See also the "LoginToContinue Annotation" section below.

2.4.3. Custom FORM Annotation

The following annotation is used to configure the built-in Custom FORM authentication mechanism.

```
@Retention(RUNTIME)
@Target(TYPE)
public @interface CustomFormAuthenticationMechanismDefinition {

    @Nonbinding
    LoginToContinue loginToContinue();
}
```

See also the "LoginToContinue Annotation" and "Custom FORM Notes" sections below.

2.4.4. LoginToContinue Annotation

The *LoginToContinue* annotation provides an application with the ability to declaratively add "login to continue" functionality to an authentication mechanism. "Login to continue" conceptually refers to the algorithm (flow) described by the numbered steps in [SERVLET31], Section 13.6.3, "Form Based Authentication".

The annotation is also used to configure the login page, error page, and redirect/forward behavior for the built-in form-based authentication mechanisms (implicitly suggesting, but not requiring, that those authentication mechanisms use the backing interceptor for this annotation, which is described below).

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target(TYPE)
public @interface LoginToContinue {

    @Nonbinding
    String loginPage() default "/login";

    @Nonbinding
    boolean useForwardToLogin() default true;

    @Nonbinding
    String useForwardToLoginExpression() default "";

    @Nonbinding
    String errorPage() default "/login-error";
}
```

The container MUST provide an interceptor implementation, at priority *PLATFORM_BEFORE* + 220, that backs the *LoginToContinue* annotation and intercepts calls to the configured *HttpAuthenticationMechanism*. The interceptor MUST behave as follows when intercepting calls to the *HttpAuthenticationMechanism*:

Intercepting validateRequest()

- Determine if there is any stale state in the request context, due to a previously aborted flow involving "login to continue". If so, clear the stale state.
- Determine if this request is a new caller-initiated authentication, by calling *isNewAuthentication()* on the *AuthenticationParameters* object available from *HttpMessageContext*.
  - If *isNewAuthentication()* returns true, update the request state to indicate that this is a caller-initiated authentication.
- If the request is a caller-initiated authentication, continue with flow

processCallerInitiatedAuthentication.

- Otherwise, if the request is not a caller-initiated authentication, continue with flow *processContainerInitiatedAuthentication*.

Flow processCallerInitiatedAuthentication

- Call the next *Interceptor*, and remember the resulting *AuthenticationStatus*.
- If the result was *AuthenticationStatus.SUCCESS*, and *HttpMessageContext.getCallerPrincipal()* returns a non-null principal, clear all state.
- Return the *AuthenticationStatus*.

Flow processContainerInitiatedAuthentication

- Determine how far the caller is in the "login to continue" flow by comparing the request and state against the following numbered and named steps:
  1. *OnInitialProtectedURL*: Protected resource requested and no saved request state.
  2. *OnLoginPostback*: A postback after redirecting the caller in Step 1. (Note: this is not necessarily the resource the caller was redirected to — for example, a redirect to */login* could result in a postback to *j_security_check*, or to */login2*.)
  3. *OnOriginalURLAfterAuthenticate*: A request on the original, protected URL from Step 1, with authentication data and saved request state.
- If the step, as described above, can be determined, continue with the flow having the same name as that step, otherwise return the result of calling the next *Interceptor*.

Flow OnInitialProtectedURL

- Save all request details (URI, headers, body, etc.) to the state.
- Redirect or forward to *LoginToContinue.loginPage()*, depending on the value of the *useForwardToLogin()* attribute.

Flow OnLoginPostback

- Call the next *Interceptor*, and remember the resulting *AuthenticationStatus*.
- If the result was *AuthenticationStatus.SUCCESS*:
  - If *HttpMessageContext.getCallerPrincipal()* returns *null*, return *AuthenticationStatus.SUCCESS*
  - If the current request matches the saved request state (same URI, headers, etc.), return *AuthenticationStatus.SUCCESS*
  - If the current request does not match the saved request state, save the authentication state (minimally, the caller principal and groups from the *HttpMessageContext*) and redirect to the full request URL as stored in the saved request state.
- If the result was *AuthenticationStatus.SEND_FAILURE*:
  - If *LoginToContinue.errorPage()* is non-null and non-empty, redirect to *LoginToContinue.errorPage()*.

- Return the *AuthenticationStatus*.

Flow OnOriginalURLAfterAuthenticate

- Retrieve the saved request and authentication details.
- Clear all state related to "login to continue".
- Set a wrapped request into *HttpMessageContext* that provides all the original request details (headers, body, method, etc.) from the saved request state.
- Call the *HttpMessageContext.notifyContainerAboutLogin()* method with the caller principal and groups from the saved authentication state.
- Return *AuthenticationStatus.SUCCESS*.

Intercepting secureResponse()

- The *secureResponse()* method SHOULD NOT be intercepted.

Intercepting cleanSubject()

- The *cleanSubject()* method SHOULD NOT be intercepted.

See also the SecurityContext.authenticate() Notes section below.

2.4.5. RememberMe Annotation

The *RememberMe* annotation is used to configure a *RememberMeIdentityStore*, which must be provided by the application. To use *RememberMe*, the application must provide an *HttpAuthenticationMechanism* and annotate the *HttpAuthenticationMechanism* with the *RememberMe* annotation.

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target(TYPE)
public @interface RememberMe {

    @Nonbinding
    int cookieMaxAgeSeconds() default 86400; // 1 day

    @Nonbinding
    String cookieMaxAgeSecondsExpression() default "";

    @Nonbinding
    boolean cookieSecureOnly() default true;

    @Nonbinding
    String cookieSecureOnlyExpression() default "";

    @Nonbinding
    boolean cookieHttpOnly() default true;

    @Nonbinding
    String cookieHttpOnlyExpression() default "";

    @Nonbinding
    String cookieName() default "JREMEMBERMEID";

    @Nonbinding
    boolean isRememberMe() default true;

    @Nonbinding
    String isRememberMeExpression() default "";
}
```

The container MUST provide an interceptor implementation at priority *PLATFORM_BEFORE* + 210 that backs the *RememberMe* annotation and intercepts calls to the configured *HttpAuthenticationMechanism*. The interceptor MUST behave as follows when intercepting calls to the *HttpAuthenticationMechanism*:

Intercepting validateRequest()

- Determine whether there is a RememberMe cookie in the request
- If the cookie is present:
  - Use it to construct a *RememberMeCredential* and call the *validate()* method of the *RememberMeIdentityStore*.

    - If the validate succeeds, call _HttpMessageContext.notifyContainerAboutLogin(), passing the CallerPrincipal and CallerGroups returned by _validate().
    - If the validate fails, remove the cookie from the request.
- If no cookie is present, or if the attempt to validate a cookie failed, authenticate the caller normally by calling *proceed() on the _InvocationContext*.
- If authentication succeeds, and the caller has requested to be remembered, as determined by evaluating the *isRememberMeExpression()*, then:
    - Call the *generateLoginToken()* method of the *RememberMeIdentityStore*.
    - Set the new cookie with parameters as configured on the *RememberMe* annotation.

Intercepting secureResponse()

- The *secureResponse()* method SHOULD NOT be intercepted.

Intercepting cleanSubject()

- If there is a RememberMe cookie in the request, then:
    - Remove the cookie.
    - Call the *removeLoginToken()* method of the *RememberMeIdentityStore*.

See also the description of *RememberMeIdentityStore* in Chapter 3, "Identity Store".

2.4.6. AutoApplySession Annotation

The *AutoApplySession* annotation provides a way to declaratively enable JASPIC *javax.servlet.http.registerSession* behavior for an authentication mechanism, and automatically apply it for every request.

The *javax.servlet.http.registerSession* property is described in Section 3.8.4 of [JASPIC].

This annotation embodies the concept of a caller being authenticated over a series of multiple HTTP requests (together, a "session"). The built-in form-based authentication mechanisms use this same concept. It is therefore implicitly suggested, but not required, that the form-based authentication mechanisms use the backing interceptor for this annotation to establish and maintain their sessions.

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target(TYPE)
public @interface AutoApplySession {
}
```

The container MUST provide an interceptor implementation at priority *PLATFORM_BEFORE* + 200 that backs the *AutoApplySession* annotation and intercepts calls to the configured *HttpAuthenticationMechanism*. The interceptor MUST behave as follows when intercepting calls to the *HttpAuthenticationMechanism*:

Intercepting validateRequest()

- Get the *HttpServletRequest* from the *HttpMessageContext* that is passed as an argument to *validateRequest()*.
- Get the *Principal* from the *HttpServletRequest* (via *getUserPrincipal()*).
- If the *Principal* is null:
    - Call the next *Interceptor*, and remember the resulting *AuthenticationStatus*.
        - If the result is *AuthenticationStatus.SUCCESS*, get the *Map* object from the *MessageInfo* in the *HttpMessageContext*, and add an entry to the *Map* with key "*javax.servlet.http.registerSession*" and value "*true*".
    - Return the *AuthenticationStatus*.
- If the *Principal* is not null:
    - Create a new *CallerPrincipalCallback* instance, passing the *Principal* and client subject obtained from *HttpMessageContext* to the constructor.
    - Obtain the *CallbackHandler* from *HttpMessageContext*, and have it handle the *CallerPrincipalCallback*.
    - Return *AuthenticationStatus.SUCCESS*.

Intercepting secureResponse()

- The *secureResponse()* method SHOULD NOT be intercepted.

Intercepting cleanSubject()

- The *cleanSubject()* method SHOULD NOT be intercepted.

See also the AutoApplySession Notes section below.

2.4.7. Implementation Notes

Section 14.4, item 18, of [SERVLET31] describes requirements for supporting BASIC and FORM authentication via the web.xml *login-config* element. This specification requires that implementations of BASIC and FORM be made available as *HttpAuthenticationMechanism* CDI beans. The servlet container is NOT REQUIRED to implement separate and independent mechanisms to satisfy each requirement. Instead, the container MAY choose to provide a single mechanism, for each of BASIC and FORM, that meets the requirements of both specifications; i.e., an implementation that can be configured via *login-config*, but which is also made available as an *HttpAuthenticationMechanism* if the application uses the corresponding annotation. Equally, the container is NOT REQUIRED to provide a unified implementation, and MAY satisfy the two requirements using separate, independent implementations.

An implementation of BASIC or FORM is NOT REQUIRED to support *IdentityStore* when configured via *login-config*, regardless of whether the container has provided a single mechanism or separate mechanisms to satisfy the *login-config* and *HttpAuthenticationMechanism* requirements. Implementations MAY support *IdentityStore* for all configuration methods.

If an application provides an *HttpAuthenticationMechanism*, and also configures a *login-config* element in web.xml, the container MAY fail deployment, but is NOT REQUIRED to. If the container does not fail deployment, it MUST use only the *HttpAuthenticationMechanism* to authenticate the application's callers (i.e., it MUST ignore the *login-config* from web.xml).

2.4.8. Custom FORM Notes

The Custom FORM variant is intended to align better with modern Java EE technologies such as CDI, Expression Language, Bean Validation and specifically JSF.

Below is an example showing how the mechanism can be used with those technologies.

Consider the following JSF Facelet:

```
    <h:messages />

    <body>
        <p>
            Login to continue
        </p>

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

    </body>
```

The "Username" and "Password" inputs are bound via expression language to properties of a named CDI bean, and the bean's login() method is invoked to authenticate the user:

```
@Named
@RequestScoped
public class LoginBacking {

    @NotNull
    private String username;

    @NotNull
    private String password;

    @Inject
    private SecurityContext securityContext;

    @Inject
    private FacesContext facesContext;

    public void login() {

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

        AuthenticationStatus status = securityContext.authenticate(
            getRequest(facesContext),
            getResponse(facesContext),
            withParams()
                .credential(credential));

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

    }
```

2.4.9. SecurityContext.authenticate() Notes

Any *LoginToContinue*-annotated *HttpAuthenticationMechanism*, as well as the two built-in FORM authentication mechanisms, can be triggered via a call to the *SecurityContext.authenticate()* method. This method is based on the *HttpServletRequest.authenticate()* method, as defined by [SERVLET31], but has been extended to support additional functionality defined by the Servlet Container Profile of [JASPIC].

The extended behavior is facilitated by the *AuthenticationParameters* parameter passed to *SecurityContext.authenticate()*. *AuthenticationParameters* includes a *newAuthentication* field.

When *newAuthentication* is set to *true*, the container MUST discard any state that it holds for an *HttpAuthenticationMechanism*, and that is associated with the current caller. Specifically, this means that any associated state, such as described for the LoginToContinue Annotation above, MUST be cleared, and the request must proceed as if processing a new request.

When *newAuthentication* is set to *false*, the container MUST NOT discard any state that it holds for an *HttpAuthenticationMechanism*, and that is associated with the current caller. Instead, the container MUST resume the in-progress authentication dialog, based on the associated state. Specifically, the container MUST:

- Determine how far the caller is in the "login to continue" flow, based on the previously saved state (or lack thereof), and;
- Continue processing from that point as it would normally do.

2.4.10. AutoApplySession Notes

As an example, idiomatic code for setting the *javax.servlet.http.registerSession* key as per the requirements is:

```
httpMessageContext.getMessageInfo().getMap().put("javax.servlet.http.registerSession",
TRUE.toString());
```

As another example, idiomatic code for setting the *CallerPrincipalCallback* as per the requirements is:

```
httpMessageContext.getHandler().handle(new Callback[] {
    new CallerPrincipalCallback(httpMessageContext.getClientSubject(), principal) }
);
```

2.5. Relationship to other specifications

An *HttpAuthenticationMechanism* is a CDI bean, as defined by JSR-346, "Contexts and Dependency Injection for the Java EE platform", version 1.2 [CDI12].

The methods defined by the *HttpAuthenticationMechanism* closely map to the methods and semantics of a *ServerAuthModule*, as defined by the Servlet Container Profile of [JASPIC]. (But an *HttpAuthenticationMechanism* is itself not a *ServerAuthModule*.) The servlet container MUST use JASPIC mechanisms to arrange for an *HttpAuthenticationMechanism* to be placed in service.

This specification mandates that when a *ServerAuthModule* is called by the Servlet container, CDI services (such as the *BeanManager*) MUST be fully available, and all scopes that are defined to be active during the *service()* method of a servlet, or during the *doFilter()* method of a servlet filter, MUST be active. Specifically this means that the request, session, and application scopes MUST be active, and that a *ServerAuthModule* method such as *validateRequest()* MUST be able to obtain a reference to the CDI *BeanManager* programmatically (for example, by doing a JNDI lookup), and MUST be able to use that reference to obtain a valid request-scoped, session-scoped, or application-scoped bean. This specification does not mandate that a *ServerAuthModule* must itself be a CDI bean, or that a *ServerAuthModule* must be injectable.

An *HttpAuthenticationMechanism* implementation is logically equivalent to a built-in authentication mechanism as defined by [SERVLET31] (i.e., HTTP Basic Authentication, HTTP Digest Authentication, Form Based Authentication, and HTTPS Client Authentication); more specifically, it corresponds to an "additional container authentication mechanism", as described in section 13.6.5 of [SERVLET31].

The BASIC and FORM authentication mechanisms as defined by this specification are logically equivalent to the similarly named authentication mechanisms in [SERVLET31], respectively sections 13.6.1, "HTTP Basic Authentication", and 13.6.3, "Form Based Authentication".

Chapter 3. Identity Store

This chapter describes the *IdentityStore* and *IdentityStoreHandler* interfaces and contracts.

3.1. Introduction

IdentityStore provides an abstraction of an identity store, which is a database or directory (store) of identity information about a population of users that includes an application's callers. An identity store holds caller names, group membership information, and information sufficient to allow it to validate a caller's credentials. An identity store may also contain other information, such as globally unique caller identifiers (if different from caller name), or other caller attributes.

Implementations of the *IdentityStore* interface are used to interact with identity stores to authenticate users (i.e., validate their credentials), and to retrieve caller groups. *IdentityStore* is roughly analogous to the JAAS *LoginModule* interface, which is often integrated into Java EE products (albeit in vendor-specific ways). Unlike *LoginModule*, *IdentityStore* is intended specifically for Java EE, and provides only credential validation and group retrieval functions (i.e., functions that require interaction with an identity store). An *IdentityStore* does not collect caller credentials, or manipulate *Subject*s.

IdentityStore is intended primarily for use by *HttpAuthenticationMechanism* implementations, but could in theory be used by other types of authentication mechanisms (e.g., a JASPIC *ServerAuthModule*, or a container's built-in authentication mechanisms). *HttpAuthenticationMechanism* implementations are not required to use *IdentityStore* — they can authenticate users in any manner they choose — but *IdentityStore* will often be a useful and convenient mechanism.

A significant advantage of using *HttpAuthenticationMechanism* and *IdentityStore* over container-provided BASIC or FORM implementations is that it allows an application to control the identity stores it will authenticate against, in a standard, portable way.

An *IdentityStore* is expected to perform only context- and environment-independent processing (for example, verifying usernames and passwords and returning caller data). It should provide a pure *{credentials in, caller data out}* function. An *IdentityStore* should not directly interact with the caller, or attempt to examine request context or application state.

The *IdentityStoreHandler* interface defines a mechanism for invoking on *IdentityStore* to validate a user credential. An *HttpAuthenticationMechanism* (or other caller) should not interact directly with an *IdentityStore*, but instead invoke the *IdentityStoreHandler* to validate credentials. The *IdentityStoreHandler*, in turn, invokes on the *IdentityStore*. An *IdentityStoreHandler* can also orchestrate an authentication across multiple *IdentityStore* instances, returning an aggregated result.

A default *IdentityStoreHandler* implementation is supplied by the container, but applications can also supply their own implementation. The orchestration behavior of the default *IdentityStoreHandler* is described in the "Handling Multiple Identity Stores" section below.

3.2. Interface and Theory of Operation

The *IdentityStore* interface defines two methods that are used by the runtime to validate a *Credential* or obtain caller information:

- validate(Credential)
- getCallerGroups(CredentialValidationResult)

An implementation of *IdentityStore* can choose to handle either or both of these methods, depending on its capabilities and configuration. It indicates which methods it handles through the set of values returned by its *validationTypes()* method:

- *VALIDATE* to indicate that it handles *validate()*
- *PROVIDE_GROUPS* to indicate that it handles *getCallerGroups()*
- Both *VALIDATE* and *PROVIDE_GROUPS* to indicate that it handles both methods

This method of declaring capabilities was chosen so that an *IdentityStore* could be written to support both methods, but configured to support just one or the other in any particular deployment.

The full interface is shown below (without default behaviors; signatures only).

```
public interface IdentityStore {

    enum ValidationType { VALIDATE, PROVIDE_GROUPS }

    CredentialValidationResult validate(Credential credential);

    Set<String> getCallerGroups(CredentialValidationResult validationResult);

    int priority();

    Set<ValidationType> validationTypes();
}
```

3.2.1. Validating Credentials

The *validate()* method determines whether a *Credential* is valid, and, if so, returns information about the user identified by the *Credential*. It is an optional method that an *IdentityStore* may choose not to implement.

```
CredentialValidationResult validate(Credential credential);
```

The result of validation is returned as a *CredentialValidationResult*, which provides methods to obtain the resulting status value, and, for successful validations, the ID of the identity store that validated the credential, the caller principal, the caller's unique ID in the identity store, and the caller's group memberships, if any. Only the caller principal is required to be present for a successful validation.

The identity store ID, caller DN, and caller unique ID are provided to assist implementations of *IdentityStore* in cooperating across invocations of *validate()* and *getCallerGroups()*. They can be used to ensure that the correct caller's groups are returned from *getCallerGroups()* even in environments where caller principal name alone is insufficient to uniquely identify the correct user account.

```
public class CredentialValidationResult {

    public enum Status { NOT_VALIDATED, INVALID, VALID };

    public Status getStatus();

    public String getIdentityStoreId();

    public CallerPrincipal getCallerPrincipal();

    public String getCallerDn();

    public String getCallerUniqueId();

    public Set<String> getCallerGroups();
}
```

The defined status values are:

- *VALID*: Validation succeeded and the user is authenticated. The caller principal and groups (if any) are available ONLY with this result status.
- *INVALID*: Validation failed. The supplied *Credential* was invalid, or the corresponding user was not found in the user store.
- *NOT_VALIDATED*: Validation was not attempted, because the *IdentityStore* does not handle the supplied *Credential* type.

The *Credential* interface is a generic interface capable of representing any kind of token or user credential. An *IdentityStore* implementation can support multiple concrete *Credential* types, where a concrete *Credential* is an implementation of the *Credential* interface that represents a particular type of credential. It can do so by implementing the *validate(Credential)* method and testing the type of the *Credential* that's passed in. As a convenience, the *IdentityStore* interface provides a default implementation of *validate(Credential)* that delegates to a method that can handle the provided *Credential* type, assuming such a method is implemented by the *IdentityStore*:

```
default CredentialValidationResult validate(Credential credential) {
    try {
        return CredentialValidationResult.class.cast(
            MethodHandles.lookup()
                .bind(this, "validate",
                      methodType(CredentialValidationResult.class,
                                 credential.getClass()))
                .invoke(credential));
    } catch (NoSuchMethodException e) {
        return NOT_VALIDATED_RESULT;
    } catch (Throwable e) {
        throw new IllegalStateException(e);
    }
}
```

So, for example, *validate(Credential)* would delegate to the following method of *ExampleIdentityStore* if passed a *UsernamePasswordCredential*:

```
public class ExampleIdentityStore implements IdentityStore {

    public CredentialValidationResult validate(
        UsernamePasswordCredential usernamePasswordCredential) {
        // Implementation ...
        return new CredentialValidationResult(...);
    }
}
```

3.2.2. Retrieving Caller Information

The *getCallerGroups()* method retrieves the set of groups associated with a validated caller. It is an optional method that an *IdentityStore* may choose not to implement.

```
Set<String> getCallerGroups(CredentialValidationResult validationResult);
```

The *getCallerGroups()* method supports aggregation of identity stores, where one identity store is used to authenticate users, but one or more other stores are used to retrieve additional groups. In such a scenario, it is necessary to query identity stores without validating the caller’s credential against the stores.

If an *IdentityStore* supports both *validate()* and *getCallerGroups()*, the behavior of both methods should be consistent with respect to groups. That is, for a given user "foo", the set of groups returned when calling *validate()* to authenticate user "foo" should be the same as the set of groups returned when calling *getCallerGroups()* for *CallerPrincipal* "foo". (Assuming no errors occur during either call — this requirement is intended as a normative description of expected behavior; it does not imply that an implementation must "make it right" if errors or other uncontrollable factors cause results to vary between any two calls.)

As a result, it is never necessary to call *getCallerGroups()* when there is only one *IdentityStore*, because the same groups are returned by the *validate()* method.

Note that *getCallerGroups()* is not intended as a general purpose API for retrieving user groups. It should be called only by an *IdentityStoreHandler*, in the course of orchestrating a *validate()* call across multiple identity stores.

Because *getCallerGroups()* enables its callers to access an external store as a privileged user (i.e., as an LDAP or database user with permission to search the store and retrieve information about arbitrary user accounts), it should be protected against unauthorized access.

Implementors of *getCallerGroups()* are strongly encouraged to check that the calling context has *IdentityStorePermission*, as shown below, before proceeding. (The built-in identity stores are REQUIRED to do so, see Annotations and Built-In IdentityStore Beans.)

```
SecurityManager securityManager = System.getSecurityManager();
if (securityManager != null) {
    securityManager.checkPermission(new IdentityStorePermission("getGroups"));
}
```

3.2.3. Declaring Capabilities

The *IdentityStore* interface includes methods for an implementation to declare its capabilities and ordinal priority. An *IdentityStore* implementation may allow these "capabilities" to be configured, so that an application can determine what a store is used for.

```
enum ValidationType { VALIDATE, PROVIDE_GROUPS }

Set<ValidationType> DEFAULT_VALIDATION_TYPES = EnumSet.of(VALIDATE, PROVIDE_GROUPS);

default int priority() {
    return 100;
}

default Set<ValidationType> validationTypes() {
    return DEFAULT_VALIDATION_TYPES;
}
```

The *priority()* method allows an *IdentityStore* to be configured with an ordinal number indicating the order in which it should be consulted when multiple *IdentityStore*s are present (more precisely, when multiple enabled CDI Beans with type *IdentityStore* are available). Lower numbers represent higher priority, so an *IdentityStore* with a lower priority value is called before an *IdentityStore* with a higher priority value.

The *validationTypes()* method returns a Set of enum constants of type *ValidationType*, indicating the purposes for which an *IdentityStore* should be used:

- *VALIDATE*, to indicate that it handles *validate()*
- *PROVIDE_GROUPS* to indicate that it handles *getCallerGroups()*
- Both *VALIDATE* and *PROVIDE_GROUPS* to indicate that it handles both methods

An *IdentityStore*'s validation types determine whether the store is used for authentication only (meaning any group data it returns must be ignored), for providing groups only (meaning it’s not used for authentication, but only to obtain group data for a caller that was authenticated by a different *IdentityStore*), or for both (meaning it’s used for authentication and any group data it returns is used).

This method of declaring capabilities was chosen to enable applications to enable or disable *IdentityStore* capabilities via configuration.

3.2.4. Handling Multiple Identity Stores

Access to the *IdentityStore* is abstracted by the *IdentityStoreHandler* interface, which provides a single method:

```
public interface IdentityStoreHandler {
    CredentialValidationResult validate(Credential credential);
}
```

For the caller, the semantics of the *validate()* method are as described for the *IdentityStore* method with the same signature.

The purpose of the *IdentityStoreHandler* is to allow for multiple identity stores to logically act as a single *IdentityStore* to the *HttpAuthenticationMechanism*. A compliant implementation of this specification MUST provide a default implementation of the *IdentityStoreHandler* that is an enabled CDI bean with qualifier @Default, and scope @ApplicationScoped, as defined by the CDI specification.

The *validate()* method of the default implementation MUST do the following:

- Call the *validate(Credential credential)* method on all available *IdentityStore* beans that declared themselves capable of doing validation, in the order induced by the return value of the *getPriority()* method of each *IdentityStore*. (Lower priority values imply a lower order, causing the corresponding *validate(Credential credential)* method to be called sooner. The calling order is undefined when two *IdentityStore* implementations return the same value.)
    - If a call to *validate()* returns a result with status *INVALID*, remember it, in case no *IdentityStore*

    returns a VALID result.
  - If a call to *validate()* returns a result with status *VALID*, remember this result and stop calling *validate()*.

- If all *IdentityStore* beans have been called but no result was returned with status *VALID*, then:
  - If a result was previously returned with status *INVALID*, return that result.
  - Otherwise, return a result with status *NOT_VALIDATED*.

- If there is a *VALID* result:
  - Create an empty set of groups.
  - Add any groups returned in the CredentialValidationResult to the set of groups, if and only if the identity store that returned the *VALID* result declared the *PROVIDE_GROUPS* validation type.
  - Call the *getCallerGroups()* method on all available *IdentityStore* beans that declared *only* the *PROVIDE_GROUPS* validation type, in the order induced by the return value of the *getPriority()* method of each *IdentityStore*, passing in the *CredentialValidationResult* obtained during the previous phase. Add the groups returned by each call to the set of accumulated groups.

- Return a new *CredentialValidationResult* with status *VALID*; the *CallerPrincipal*, *CallerUniqueId*, *CallerDn*, and *IdentityStoreId* that were returned from the successful *validate()*; and the accumulated collection of groups.

The default *IdentityStoreHandler* MUST make all calls to *getCallerGroups()* in the context of a *PrivilegedAction*. Other implementations of *IdentityStoreHandler* are strongly encouraged to do so as well.

The necessary permission grants (i.e., for *IdentityStorePermission("getGroups")*) should be configured if running with a *SecurityManager*.

See javadoc for additional information.

3.2.5. State

An *IdentityStore* is logically stateless. An *IdentityStoreHandler* should not make any assumptions about the state of an *IdentityStore* before, during, or after making calls to it. In particular, an *IdentityStore* store should not be aware of the point its caller has reached in the authentication process, and, even more specifically, an *IdentityStore* should not keep track of whether a caller is authenticated or not at any given moment in time.

An *IdentityStore* instance may make use of instance variables; for example, to store configuration data like an LDAP URL, to store actual caller data for in-memory lookup, for the caching, etc.

3.2.6. RememberMeIdentityStore

The *RememberMeIdentityStore* is a specialized interface that is similar to the standard *IdentityStore* interface, but is a distinct type (no inheritance relationship) and works differently.

Applications often want to remember logged in callers for extended periods of time — days or weeks — so that callers don't have to log in every time they visit the application. A *RememberMeIdentityStore* can be used to:

- Generate a login token ("remember me token") for a caller
- Remember the caller associated with the login token
- Validate the login token when the caller returns, and re-authenticate the caller without the need to provide additional credentials.

If the caller does not have a login token, or if the login token has expired, then the normal authentication process takes place.

```
public interface RememberMeIdentityStore {

    CredentialValidationResult validate(RememberMeCredential credential);

    String generateLoginToken(CallerPrincipal callerPrincipal, Set<String> groups);

    void removeLoginToken(String token);
}
```

RememberMeIdentityStore can only be used when an application includes an *HttpAuthenticationMechanism* or configures one of the built-in ones. The application must specify the *RememberMe* annotation on the *HttpAuthenticationMechanism* to configure the *RememberMeIdentityStore*.

See the description of the *RememberMe* annotation in Chapter 2, "Authentication Mechanism".

3.3. Installation and Configuration

Installation of an *IdentityStore* depends on the CDI specification. That is, an *IdentityStore* is considered installed and available for usage when it's available to the CDI runtime as an enabled Bean. An *IdentityStore* is assumed to be normal scoped.

It MUST be possible for the definition of an *IdentityStore* to exist within the application archive. Alternatively such definition MAY also exists outside the application archive, for example in a jar added to the classpath of an application server.

As described above, in the "Declaring Capabilities" section, the *IdentityStore* interface includes two methods, *validationTypes()* and *priority()*, that enable an *IdentityStore* to declare its capabilities. Those capabilities may be intrinsic — determined by the *IdentityStore*'s implementation — or they may be determined by the *IdentityStore*'s configuration.

3.4. Annotations and Built-In IdentityStore Beans

A Java EE container MUST support built-in beans for the following *IdentityStore* types, to be configured and made available via corresponding annotations:

- LDAP — Supports caller data that is stored in an external LDAP server. This bean is activated and configured via the *@LdapIdentityStoreDefinition* annotation.
- Database — Supports caller data that is stored in an external database accessible via a DataSource bound to JNDI. This bean is activated and configured via the *@DatabaseIdentityStoreDefinition* annotation.

Each of these beans MUST have the qualifier @Default and the scope @ApplicationScoped, as defined by the CDI specification.

The built-in identity stores MUST support validating *UsernamePasswordCredential*. They MAY support other credential types, but are NOT REQUIRED to.

The built-in identity stores MUST check whether a *SecurityManager* is configured, and, if so, check whether the calling context has *IdentityStorePermission*, as described in Retrieving Caller Information above, before proceeding.

Note that implementations are explicitly NOT REQUIRED to provide an LDAP server or database. The requirement is only to provide *IdentityStore* implementations that can work with an external LDAP or database server that may be present in the operating environment.

The corresponding annotations are defined as shown in the following sections.

3.4.1. LDAP Annotation

The *LdapIdentityStoreDefinition* annotation configures an instance of the built-in LDAP identity store. See javadoc for details of the configuration attributes.

```
@Retention(RUNTIME)
@Target(TYPE)
public @interface LdapIdentityStoreDefinition {

    enum LdapSearchScope { ONE_LEVEL, SUBTREE }

    String url() default "";

    String bindDn() default "";

    String bindDnPassword() default "";
    String callerBaseDn() default "";

    String callerNameAttribute() default "uid";

    String callerSearchBase() default "";

    String callerSearchFilter() default "";

    LdapSearchScope callerSearchScope() default LdapSearchScope.SUBTREE;

    String callerSearchScopeExpression() default "";

    String groupSearchBase() default "";

    String groupSearchFilter() default "";

    LdapSearchScope groupSearchScope() default LdapSearchScope.SUBTREE;

    String groupSearchScopeExpression() default "";

    String groupNameAttribute() default "cn";

    String groupMemberAttribute() default "member";

    String groupMemberOfAttribute() default "memberOf";

    int readTimeout() default 0;

    String readTimeoutExpression() default "";

    int maxResults() default 1000;

    String maxResultsExpression() default "";

    int priority() default 80;

    String priorityExpression() default "";

    ValidationType[] useFor() default {VALIDATE, PROVIDE_GROUPS};

    String useForExpression() default "";

}
```

3.4.2. Database Annotation

The *DatabaseIdentityStoreDefinition* annotation configures an instance of the built-in database identity store.

```
@Retention(RUNTIME)
@Target(TYPE)
public @interface DatabaseIdentityStoreDefinition {

    String dataSourceLookup() default "java:comp/DefaultDataSource";

    String callerQuery() default "";

    String groupsQuery() default "";

    Class<? extends PasswordHash> hashAlgorithm() default Pbkdf2PasswordHash.class;

    String[] hashAlgorithmParameters() default {};

    int priority() default 70;

    String priorityExpression() default "";

    ValidationType[] useFor() default {VALIDATE, PROVIDE_GROUPS};

    String useForExpression() default "";
}
```

Password hashing/hash verification is provided by an implementation of the *PasswordHash* interface, which must be made available as a dependent-scoped bean, and is configured by type on the *hashAlgorithm()* attribute. The specified type may refer to the actual implementation class, or to any type it implements or extends, as long as the specified type implements the *PasswordHash* interface.

Parameters for the configured *PasswordHash* can be provided using the *hashAlgorithmParameters* attribute, and will be passed to the *initialize()* method of the *PasswordHash* when the identity store is initialized.

The default hash algorithm, *Pbkdf2PasswordHash*, is an interface denoting a standard, built-in *PasswordHash*. All implementations of this specification MUST provide an implementation of the *Pbkdf2PasswordHash* interface, with configuration and behavior as described by the interface’s javadoc.

See javadoc for further details on *PasswordHash* and the *DatabaseIdentityStoreDefinition* annotation.

3.5. Relationship to Other Specifications

IdentityStore and *IdentityStoreHandler* implementations are CDI beans, as defined by [CDI12].

Chapter 4. Security Context

This chapter describes the *SecurityContext* interface and contract.

4.1. Introduction

The Java EE platform defines a declarative security model for protecting application resources. The declared constraints on access are then enforced by the container. In some cases the declarative model is not sufficient; for example, when a combination of tests and constraints is needed that is more complex than the declarative model allows for. Programmatic security allows an application to perform tests and grant or deny access to resources.

This specification provides an access point for programmatic security — a security context — represented by the *SecurityContext* interface.

In this version of the specification, the *SecurityContext* MUST be available in the Servlet container and the EJB container. Application servers MAY make *SecurityContext* available in other containers, but are NOT REQUIRED to.

4.2. Retrieving and Testing for Caller Data

The *SecurityContext* interface defines two methods that allow the application to test aspects of the caller data:

```
Principal getCallerPrincipal();

<T extends Principal> Set<T> getPrincipalsByType(Class<T> pType);

boolean isCallerInRole(String role);
```

The *getCallerPrincipal()* method retrieves the *Principal* representing the caller. This is the container-specific representation of the caller principal, and the type may differ from the type of the caller principal originally established by an *HttpAuthenticationMechanism*. This method returns null for an unauthenticated caller. (Note that this behavior differs from the behavior of *EJBContext.getCallerPrincipal()*, which, per JSR-345, "Enterprise JavaBeans", version 3.2 [EJB32], returns a principal with a "product-specific unauthenticated principal name" to represent an unauthenticated caller.)

The *getPrincipalsByType()* method retrieves all principals of the given type. This method can be used to retrieve an application-specific caller principal established during authentication. This method is primarily useful in the case that the container’s caller principal is a different type than the application caller principal, and the application needs specific information behavior available only from the application principal. This method returns an empty *Set* if the caller is unauthenticated, or if the requested type is not found.

Where both a container caller principal and an application caller principal are present, the value returned by *getName()* MUST be the same for both principals.

See the Chapter 1, "Concepts", for more information on principal handling.

The *isCallerInRole()* method takes a String argument that represents the role that is to be tested for. It is undefined by this specification how the role determination is made, but the result MUST be the same as if the corresponding container-specific call had been made (i.e., *HttpServletRequest.isUserInRole()*, *EJBContext.isCallerInRole()*), and MUST be consistent with the result implied by other specifications that prescribe role-mapping behavior.

4.3. Testing for Access

The *SecurityContext* interface defines a method for programmatically testing access to a resource:

```
boolean hasAccessToWebResource(String resource, String... methods);
```

The *hasAccessToWebResource()* method determines if the caller has access to the specified web resource for the specified HTTP methods, as determined by the security constraints configured for the application. See section 13.8 of [SERVLET31] for a description of web application security constraints.

The resource parameter is an *URLPatternSpec* that identifies an application-specific web resource. See the javadoc for more detail.

This method can only be used to check access to resources in the current application — it cannot be called cross-application, or cross-container, to check access to resources in a different application.

As an example, consider the following Servlet definition:

```
@WebServlet("/protectedServlet")
@ServletSecurity(@HttpConstraint(rolesAllowed = "foo"))
public class ProtectedServlet extends HttpServlet { ... }
```

And the following call to *hasAccessToWebResource()*:

```
securityContext.hasAccessToWebResource("/protectedServlet", GET)
```

The above *hasAccessToWebResource()* call would return true if and only if the caller is in role "foo".

4.4. Triggering the Authentication Process

The *SecurityContext* interface defines a method that allows an application to programmatically trigger the authentication process:

```
AuthenticationStatus authenticate(HttpServletRequest request,
                                  HttpServletResponse response,
                                  AuthenticationParameters parameters);
```

Programmatically triggering means that the container responds as if the caller had attempted to access a constrained resource. It causes the container to invoke the authentication mechanism configured for the application. If the configured authentication mechanism is an *HttpAuthenticationMechanism*, then the AuthenticationParameters argument is meaningful and extended capabilities of *HttpAuthenticationMechanism* are available. If not, the behavior and result is as if *HttpServletRequest.authenticate()* were called.

The *authenticate()* method allows an application to signal to the container that it should start the authentication process with the caller. This method requires a *HttpServletRequest* and *HttpServletResponse* parameters to be passed in, and can therefore only be used in a valid Servlet context.

4.5. Relationship to Other Specifications

The *SecurityContext* implementation is a CDI bean, as defined by [CDI12].

Various specifications in Java EE provide similar or even identical methods to those provided be the *SecurityContext*. It is the intention of this specification to eventually supersede those methods and provide a cross-specification, platform alternative. The following gives an overview:

- Servlet - *HttpServletRequest#getUserPrincipal*, *HttpServletRequest#isUserInRole*
- EJB - *EJBContext#getCallerPrincipal*, *EJBContext#isCallerInRole*
- JAX-WS - *WebServiceContext#getUserPrincipal*, *WebServiceContext#isUserInRole*
- JAX-RS - *SecurityContext#getUserPrincipal*, *SecurityContext#isUserInRole*
- JSF - *ExternalContext#getUserPrincipal*, *ExternalContext#isUserInRole*
- CDI - *@Inject Principal*
- WebSockets - *Session#getUserPrincipal*

Bibliography

The following documents are referenced by this specification.

[CDI12]

JSR-346, "Contexts and Dependency Injection for the Java EE platform", version 1.2, Maintenance Release
https://jcp.org/aboutJava/communityprocess/mrel/jsr346/index.html

[EJB32]

JSR-345, "Enterprise JavaBeans", version 3.2
https://jcp.org/aboutJava/communityprocess/final/jsr345/index.html

[EL30]

JSR-341, "Expression Language", version 3.0
https://jcp.org/aboutJava/communityprocess/final/jsr341/index.html

[JACC]

JSR-115, "Java Authorization Contract for Containers", version 1.5, Maintenance Release 3
https://jcp.org/aboutJava/communityprocess/mrel/jsr115/index3.html

[JASPIC]

JSR-196, "Java Authentication SPI for Containers", version 1.1, Maintenance Release B
https://jcp.org/aboutJava/communityprocess/mrel/jsr196/index2.html

[RFC2119]

RFC 2119, "Key words for use in RFCs to Indicate Requirement Level"
https://tools.ietf.org/html/rfc2119

[RFC7617]

RFC 7617, "The *Basic* HTTP Authentication Scheme"
https://tools.ietf.org/html/rfc7617

[SECAPI]

JSR-375, "Java EE Security API", version 1.0
https://jcp.org/en/jsr/detail?id=375

[SERVLET31]

JSR-340, "Java Servlet Specification", version 3.1
https://jcp.org/aboutJava/communityprocess/final/jsr340/index.html

[SHIROTERM]

"Apache Shiro Terminology"
https://shiro.apache.org/terminology.html