# Table of Contents

Copyright ......................................................................................................................... 2
Eclipse Foundation Specification License ................................................................. 2
Disclaimers .................................................................................................................... 2
Preface ............................................................................................................................. 4
Notational Conventions ............................................................................................... 4
Acknowledgments .......................................................................................................... 4
Specification Lead under the JCP .............................................................................. 4
Expert Group under the JCP ...................................................................................... 4
Contributors under the JCP ....................................................................................... 5
Project Lead at the Eclipse Foundation ..................................................................... 5
Committer at the Eclipse Foundation ......................................................................... 5
Committers at the Eclipse Foundation ........................................................................ 5
1. Concepts and General Requirements .................................................................... 6
   1.1. Terminology And Acronyms ........................................................................... 6
   1.2. General Requirements .................................................................................. 7
       1.2.1. Group-To-Role Mapping ...................................................................... 7
       1.2.2. Caller Principal Types .......................................................................... 7
       1.2.3. Jakarta Expression Language Support .................................................. 8
2. Authentication Mechanism ....................................................................................... 9
   2.1. Introduction ..................................................................................................... 9
   2.2. Interface and Theory of Operation ................................................................ 10
   2.3. Installation and Configuration ...................................................................... 12
2.4. Annotations and Built-In HttpAuthenticationMechanism Beans .................... 13
   2.4.1. BASIC Annotation .................................................................................. 14
   2.4.2. FORM Annotation .................................................................................. 14
   2.4.3. Custom FORM Annotation ..................................................................... 14
   2.4.4. OpenID Connect Annotation ................................................................... 15
       2.4.4.1. Metadata configuration ................................................................. 21
       2.4.4.2. Authentication dialog ................................................................. 22
       2.4.4.3. Caller name and groups ............................................................... 25
   2.4.5. LoginToContinue Annotation ................................................................ 27
   2.4.6. RememberMe Annotation ....................................................................... 30
   2.4.7. AutoApplySession Annotation ................................................................ 32
   2.4.8. Implementation Notes .............................................................................. 33
   2.4.9. Custom FORM Notes .............................................................................. 34
   2.4.10. SecurityContext.authenticate() Notes ............................................... 35
   2.4.11. AutoApplySession Notes ...................................................................... 36
Copyright

Copyright (c) 2019, 2021 Eclipse Foundation.

Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- link or URL to the original Eclipse Foundation document.
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright (c) [$date-of-document] Eclipse Foundation, Inc. https://www.eclipse.org/legal/efsl.php"

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright (c) 2018, 2021 Eclipse Foundation. This software or document includes material copied from or derived from Jakarta™ Security https://jakarta.ee/specifications/security/2.0/.

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.
Preface

This document is the Jakarta Security 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 authors would like to thank the original JCP JSR-375 Expert Group and Contributors, and, in particular, Arjan Tijms, whose contributions were critical to the success of the JCP JSR.

We would also like to thank Alex Kosowski for his work putting together the original JSR-375 proposal, submitting it to the JCP, recruiting a diverse and highly-qualified expert group, and leading the initial development and specification efforts. Without Alex, there would be no JSR-375.

Lastly, we would like to acknowledge Ron Monzillo, and the various JCP expert groups he has led or worked with over the years, for putting in place the foundational security APIs and SPIs upon which JSR-375 was built.

Specification Lead under the JCP

Will Hopkins (October 2016 — September 2017)
Alexander Kosowski (April 2014 — September 2016)

Expert Group under the JCP

<table>
<thead>
<tr>
<th>Adam Bien</th>
<th>David Blevins (Tomitribe)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rudy De Busscher</td>
<td>Ivar Grimstad</td>
</tr>
<tr>
<td>Les Hazlewood (Stormpath, Inc.)</td>
<td>Will Hopkins (Oracle)</td>
</tr>
<tr>
<td>Werner Keil</td>
<td>Matt Konda (Jemurai)</td>
</tr>
<tr>
<td>Alexander Kosowski (Oracle)</td>
<td>Darran Lofthouse (Red Hat)</td>
</tr>
<tr>
<td>Jean-Louis Monteiro (Tomitribe)</td>
<td>Ajay Reddy (IBM)</td>
</tr>
<tr>
<td>Pedro Igor Silva (Red Hat)</td>
<td>Arjan Tijms</td>
</tr>
</tbody>
</table>
**Contributors under the JCP**

<table>
<thead>
<tr>
<th>Name</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guillermo González de Agüero</td>
<td>John Hogan</td>
</tr>
<tr>
<td>Elder Moraes</td>
<td>Fatih Mutluay</td>
</tr>
<tr>
<td>Reza Rahman</td>
<td>Ashley Richardson</td>
</tr>
</tbody>
</table>

**Project Lead at the Eclipse Foundation**

Arjan Tijms

**Committers at the Eclipse Foundation**

<table>
<thead>
<tr>
<th>Ajay Reddy (IBM)</th>
<th>Arjan Tijms</th>
<th>Bill Shannon (Oracle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Darran Lofthouse (Red Hat)</td>
<td>David Blevins (Tomitribe)</td>
<td>Dmitry Kornilov (Oracle)</td>
</tr>
<tr>
<td>Ed Bratt (Oracle)</td>
<td>Guillermo González de Agüero</td>
<td>Ivar Grimstad (Eclipse Foundation)</td>
</tr>
<tr>
<td>James Mulvey (IBM)</td>
<td>Padmanabha Bhat (Oracle)</td>
<td>Rudy De Busscher (Payara)</td>
</tr>
<tr>
<td>Steven Liu (Oracle)</td>
<td>Teddy Torres (IBM)</td>
<td>Tomas Langer (Oracle)</td>
</tr>
<tr>
<td>VINAY VISHAL (Oracle)</td>
<td>Werner Keil</td>
<td>Wilson Tian (Oracle)</td>
</tr>
</tbody>
</table>
Chapter 1. Concepts and General Requirements

This chapter provides overview information and terminology related to this specification, and also includes 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 (JAAAS), 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.

SAM

Abbreviation for ServerAuthModule, an interface defined by Jakarta Authentication.
1.2. General Requirements

The following general requirements are defined by this specification.

1.2.1. Group-To-Role Mapping

Various Jakarta 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 Jakarta EE specifications (e.g., Jakarta Authentication), provide abstractions to accommodate, "the container's representation of the caller principal".

This specification RECOMMENDS that Jakarta 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. Jakarta Expression Language Support

This specification defines a number of annotations:

- `DatabaseIdentityStoreDefinition`
- `LdapIdentityStoreDefinition`
- `BasicAuthenticationMechanismDefinition`
- `CustomFormAuthenticationMechanismDefinition`
- `FormAuthenticationMechanismDefinition`
- `OpenIdAuthenticationMechanismDefinition`
- `LoginToContinue`
- `RememberMe`

Attributes on these annotations can be provided either as actual values, or as Jakarta Expression Language 5.0 expressions. In cases where the return type of an attribute is not String, an "Expression Language alternative" attribute is provided, with "Expression" appended to the name. If an "Expression Language alternative" attribute has a non-empty value, it takes precedence over the attribute it's an alternative to, and must contain a valid Expression Language expression that evaluates to the same type as the attribute it's an alternative to.

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

Jakarta Expression Language, version 5.0 [EL50] is a Jakarta EE specification.
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 (enterprise beans, messaging, connectors, 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 Jakarta EE Platform already specifies mechanisms for authenticating users of web applications. The Jakarta Servlet Specification, version 4.0 [SERVLET60] 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, The Jakarta Authentication Specification, version 3.0 [AUTHENTICATION30] specifies a general-purpose mechanism for securing messages sent between Jakarta EE clients and servers. Jakarta Authentication 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. [AUTHENTICATION30] also defines the Servlet Container Profile, which specifies how Jakarta Authentication 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 [SERVLET60]—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.
Jakarta Authentication, 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 Jakarta Authentication, 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 Jakarta Authentication 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 Jakarta Authentication 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 Jakarta Authentication ServerAuth interface. The primary distinction is syntactic; unlike Jakarta Authentication, 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.
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 Jakarta Authentication 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 Jakarta Authentication `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 that 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 Jakarta Authentication 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 Jakarta EE modules and application archives, as well as other archives and classes that are not part of an application, but are required by the Jakarta 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 Jakarta Authentication when placing an `HttpAuthenticationMechanism` in
service. The container MUST supply a "bridge" ServerAuthModule that integrates HttpAuthenticationMechanism with Jakarta Authentication. 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 Jakarta Authentication 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 Jakarta Authentication, 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 Jakarta 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 [SERVLET60]. 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 [SERVLET60]. 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 [SERVLET60], 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.

- **OpenID Connect** - Authenticates according to the Authorization Code flow and Refresh tokens as defined by the OpenID Connect specification. See [OPENID10FLOW] and [OPENID10REFRESH]. This bean is activated and configured via the @OpenIdAuthenticationMechanismDefinition 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.

```java
@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 does not 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.

```java
@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. OpenID Connect Annotation

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

```java
@Target({TYPE, METHOD})  
@Retention(RUNTIME)  
public @interface OpenIdAuthenticationMechanismDefinition {

/**
 * Required, unless providerMetadata is specified.
 * The provider URI to read / discover the metadata of the openid provider.
 *
 * @see http://openid.net/specs/openid-connect-discovery-1_0.html
 *
 * @return provider URI to read from which to read metadata
 */
String providerURI() default "";

/**
 * To override the openid connect provider's metadata property discovered
 * via providerUri.
 *
 * @return OpenIdProviderMetadata instance.
 */
OpenIdProviderMetadata providerMetadata() default @OpenIdProviderMetadata;

/**
 * Required. The client identifier issued when the application was
 * registered.
 *
 * @return the client identifier
 */
String clientId() default "";

/**
 * Required. The client secret.
 */
```
Note that it is strongly recommended to set this using an Expression so that the value is not hardcoded within the code.

```java
String clientSecret() default ";
```

Optional. The claims definition defines the custom claims mapping of caller name and groups.

```java
ClaimsDefinition claimsDefinition() default @ClaimsDefinition;
```

Optional. The Logout definition defines the logout and Relaying Party session management configuration.

```java
LogoutDefinition logout() default @LogoutDefinition;
```

The redirect URI (callback URI) to which the response will be sent by the OpenId Connect Provider. This URI must exactly match one of the Redirection URI values for the Client pre-registered at the OpenID Provider.

```java
String redirectURI() default "${baseURL}/Callback";
```

Optional. Automatically redirects the caller (the end-user) from the redirect URI defined by the <code>redirectURI</code> attribute to the resource the end-user originally requested in a "login to continue" scenario.

```java
String redirectURI() default 2.4. Annotations and Built-In HttpAuthenticationMechanism Beans
```
boolean redirectToOriginalResource() default false;

/**
 * Optional. Allows the <code>redirectToOriginalResource</code> to be specified as
 * Jakarta Expression Language expression.
 * If set, overrides the value defined by the <code>redirectToOriginalResource</code>
 * value.
 * @return
 */
String redirectToOriginalResourceExpression() default "";

/**
 * Optional. The scope value defines the access privileges. The basic (and
 * required) scope for OpenID Connect is the openid scope.
 * @return
 */
String[] scope() default {OPENID_SCOPE, EMAIL_SCOPE, PROFILE_SCOPE};

/**
 * Optional. Allows The scope value to be specified as Jakarta Expression Language
 * expression.
 * If Set, overrides any values set by scope.
 * @return
 */
String scopeExpression() default "";

/**
 * Optional. Response Type value defines the processing flow to be used. By
 * default, the value is code (Authorization Code Flow).
 * @return
 */
String responseType() default CODE;

/**
 * Optional. Informs the Authorization Server of the mechanism to be used
 * for returning parameters from the Authorization Endpoint.
 * @return
 */
String responseMode() default "";
* prompts the user for reauthentication and consent. If no value is
* specified and the user has not previously authorized access, then the
* user is shown a consent screen.
*
* @return
*/

PromptType[] prompt() default {};

/**
 * Optional. Allows the prompt value to be specified as Jakarta Expression Language
 * expression.
 * If Set, overrides the value defined by the prompt value.
 *
 */

String promptExpression() default "";

/**
 * Optional. The display value specifying how the authorization server
 * displays the authentication and consent user interface pages.
 *
 */

DisplayType display() default DisplayType.PAGE;

/**
 * Optional. Allows the display value to be specified as Jakarta Expression Language
 * expression.
 * If set, overrides the value defined by display.
 *
 */

String displayExpression() default "";

/**
 * Optional. Enables string value used to mitigate replay attacks.
 *
 */

boolean useNonce() default true;

/**
 * Optional. Allows the nonce activation to be specified as Jakarta Expression
 * Language expression.
 * If set, overrides the value defined by the useNonce value.
 *
 */

String useNonceExpression() default ";

/**
 * Optional. If enabled the state, nonce values and original requested resource data are stored in an HTTP session
 * otherwise in cookies.
 */

boolean useSession() default true;

/**
 * Optional. Allows the configuration of the session through a Jakarta Expression Language expression.
 * If set, overwrites the value of useSession value.
 */

String useSessionExpression() default ";

/**
 * An array of extra options that will be sent to the OAuth provider.
 * These must be in the form of {@code "key=value"} i.e. 
 * extraParameters={"key1=value", "key2=value2"} </code>
 */

String[] extraParameters() default {};

/**
 * Allows the extra parameters to be defined as a Jakarta Expression Language expression.
 * If set, overrides the extraParameters value.
 */

String extraParametersExpression() default ";

/**
 * Optional. Sets the connect timeout(in milliseconds) for Remote JWKS retrieval. Value must not be negative and if value is zero then infinite timeout.
 */

int jwksConnectTimeout() default 500;
/* Optional. Allows the connect timeout (in milliseconds) for Remote JWKS to be defined as
 * Jakarta Expression Language expression.
 * If set, overwrites the jwksConnectTimeout value.
 *
 * @return
 */

String jwksConnectTimeoutExpression() default "";

/**
 * Optional. Sets the read timeout (in milliseconds) for Remote JWKS retrieval. Value must not be negative and if value is zero then infinite timeout.
 *
 * @return
 */

int jwksReadTimeout() default 500;

/**
 * Optional. Allows the read timeout (in milliseconds) for Remote JWKS retrieval to be defined as Jakarta Expression Language expression.
 * If set, overwrites the jwksReadTimeout value.
 *
 * @return
 */

String jwksReadTimeoutExpression() default "";

/**
 * Optional. Enables or disables the automatically performed refresh of Access and Refresh Token.
 *
 * @return {code true}, if Access and Refresh Token shall be refreshed automatically when they are expired.
 */

boolean tokenAutoRefresh() default false;

/**
 * Optional. Allows the automatically performed refresh of Access and Refresh Token to be defined as Jakarta Expression Language expression.
 * If set, overwrites the value of tokenAutoRefresh.
 */

String tokenAutoRefreshExpression() default "";

/**
 * Optional. Sets the minimum validity time in milliseconds the Access Token must be valid before it is considered expired. Value must not be negative.
 */
Attributes of this annotation support Jakarta Expression Language evaluation as specified in Jakarta Expression Language Support.

The expected behavior is defined in the following sections.

2.4.4.1. Metadata configuration

The OpenID Connect authentication mechanism needs metadata about the OpenID Connect Provider to function properly. The OpenID Connect specification defines that this data can be read from the well known openid configuration endpoint which resides at [OpenID Connect Provider base URL].well-known/openid-configuration.

The following metadata values are required (since they are defined as required by the OpenID Specification):

- Authorization endpoint
- Token endpoint
- JWKS URI
- Issuer of the tokens
- Supported Subject types
- Supported Response types
- Supported Id Token Signing Algorithms

The OpenIdAuthenticationMechanismDefinition.providerURI defines the base URL of the OpenID Connect Provider where the /.well-known/openid-configuration is appended to (or used as-is when it is the well known configuration URL itself). Reading the well known openid configuration endpoint can be done eagerly when the application is deployed or lazily at the time a secured URL is accessed for the first time. The values retrieved from the well known openid configuration endpoint can be overwritten if
needed by using the `OpenIdAuthenticationMechanismDefinition.providerMetadata` structure.

### 2.4.4.2. Authentication dialog

The authentication dialog that the authentication mechanism starts and coordinates follows the flow as defined by the OpenID Connect specification. For completeness this is depicted below:

![Authentication Dialog Diagram]

When the authentication mechanism determines authentication is required, for instance when the caller ("end-user" in OpenID Connect terminology) tries to access a protected resource without being authenticated, or when the caller explicitly initiates authentication, without being authenticated for the current request, an authentication request needs to be assembled and send to the authentication endpoint of the OpenID Connect provider. This request corresponds to step (1) in the OpenID Connect diagram depicted above. The location of this endpoint is configured by the `providerURI` attribute of the `OpenIdAuthenticationMechanismDefinition` annotation.

The following values need to be passed to this endpoint **unconditionally**: 

```java
2.4. Annotations and Built-In HttpAuthenticationMechanism Beans

When the authentication mechanism determines authentication is required, for instance when the caller ("end-user" in OpenID Connect terminology) tries to access a protected resource without being authenticated, or when the caller explicitly initiates authentication, without being authenticated for the current request, an authentication request needs to be assembled and send to the authentication endpoint of the OpenID Connect provider. This request corresponds to step (1) in the OpenID Connect diagram depicted above. The location of this endpoint is configured by the `providerURI` attribute of the `OpenIdAuthenticationMechanismDefinition` annotation.

The following values need to be passed to this endpoint **unconditionally**:
```

 suffers from a failure to properly authenticate users.
• ClientId value as taken from `OpenIdAuthenticationMechanismDefinition.clientId`
• Scope value as taken from `OpenIdAuthenticationMechanismDefinition.scope`
• Response Type value as taken from `OpenIdAuthenticationMechanismDefinition.responseType`
• State value, must be generated by the authentication mechanism
• RedirectURI value as taken from `OpenIdAuthenticationMechanismDefinition.redirectURI` (evaluated)

The `redirectURI` attribute of the `OpenIdAuthenticationMechanismDefinition` annotation may contain the special 'expression' value `${baseURL}` and this must be resolved to the host and context path of the application for which the OpenID Connect authentication mechanism is installed. This requirement makes it easier to have an absolute URL as required by the OpenID Connect specification.

The following values need to be passed to this endpoint **conditionally**:

- Nonce, must be generated by the authentication mechanism if `OpenIdAuthenticationMechanismDefinition.useNonce` is set to `true`
- Response Mode value as taken from `OpenIdAuthenticationMechanismDefinition.responseMode` if defined
- Display value as taken from `OpenIdAuthenticationMechanismDefinition.display` if defined
- Prompt value as taken from `OpenIdAuthenticationMechanismDefinition.prompt` if defined
- Extra values as taken from `OpenIdAuthenticationMechanismDefinition.extraParameters` if defined

The State value, and also the Nonce value if requested, MUST be stored between requests so that these values can be validated when the OpenID Connect Provider later calls the supplied redirectURI. These values can either be stored serverside (in the HTTP Session) or clientside (as a Cookie). The value of the `OpenIdAuthenticationDefinition.useSession` attribute determines which one is used. In the case of storage through a Cookie, the Cookie must be defined as `HTTPonly` and must have the `Secure` flag set.

Before the redirect to the authentication endpoint of the OpenID Connect Provider is performed, the URL plus request parameters requested by the caller on which the authentication dialog was triggered must be stored so that it later on can be retrieved by a call to `OpenIdContext.getStoredValue(request, response, OpenIdConstant.ORIGINAL_REQUEST)`.

Additionally, if `OpenIdAuthenticationMechanismDefinition.redirectToOriginalResource` is set to 'true' and the authentication flow is container-initiated (as opposed to caller-initiated authentication) the authentication mechanism must store the full request as well. The full request here means all data that makes up the `HttpServletRequest` so that the container can restore this request later on in a similar way to how the "LoginToContinue Annotation" behaves.

Step (3) in the OpenID Connect diagram depicted above, that is, when the OpenID Connect Provider calls us back, is detected by the authentication mechanism when a request contains a `state` request parameter. When that initial condition is satisfied, the following investigation and actions must be done by the authentication mechanism:
If the request (without request parameters) does not match the redirectURI, or does not match the stored original URL (without request parameters) in case AuthenticationMechanismDefinition.redirectToOriginalResource is set to 'true', it must reply with a CredentialValidationResult.NOT_VALIDATED_RESULT value.

If there is no State value stored, it must reply with a CredentialValidationResult.NOT_VALIDATED_RESULT value.

If the State value in the request does not match the State value stored, it must reply with a CredentialValidationResult.INVALID_RESULT value.

If the request contains a parameter error, the authentication by the OpenID Connect Provider has failed and the authentication mechanism must reply with a CredentialValidationResult.INVALID_RESULT value.

If none of the above listed additional conditions apply, the request is taken to be a valid callback and the authentication between the end-user (caller) and the OpenID Connect Provider is considered to have been successful. The authentication mechanism must now move to step (4) of the OpenID Connect diagram and mark this internally by clearing the stored State value (remove it from the HTTP session or Cookie).

For step (4) the authentication mechanism itself must call the Token endpoint to retrieve an Access Token and ID Token. This constitutes a so-called server to server call, as the end-user (caller) MUST NOT be involved here.

The call to the token endpoint must include the following parameters (as specified by the OpenID Connect specification):

- The ClientId value as taken from OpenIdAuthenticationMechanismDefinition.clientId
- The ClientSecret value as taken from OpenIdAuthenticationMechanismDefinition.clientId
- The grant_type value set to the constant authorization_code
- The RedirectURI value as taken from OpenIdAuthenticationMechanismDefinition.redirectURI
- The code received from the OpenID Connect Provider in the callback request as the code request parameter

If the call to the Token endpoint is successful, it should return a "Token Response" in JSON format.

When available in the "Token Response", the optional fields "refresh_token" and "expires_in" must be stored internally.

The authentication mechanism also MUST create a jakarta.security.enterprise.credential.Credential instance holding this Token Response and MUST validate this token using the available IdentityStoreHandler.

In order to correctly validate this token, the Jakarta Security implementation MUST install an IdentityStore capable of validating this credential. Note that in this version of the specification the type of both the Credential and IdentityStore are implementation specific. A future version of the
specification may standardise these. For this reason the implementation specific identity store is now discussed in this chapter.

The **IdentityStore** mentioned above must perform the following checks (also defined by the OpenID Connect specification):

- The **issuer** claim matches the issuer retrieved from the well known openid configuration endpoint or the **issuer** member of the `OpenIdProviderMetadata` construct.
- A **subject** claim is present and contains a value.
- The **audience** claim is present and is equal to the `OpenIdAuthenticationMechanismDefinition.clientId`
- If multiple audience values are returned by the OpenID Connect Provider, an authorized party claim (**azp**) must be present.
- If an **authorized party** claim (**azp**) is present, it must match the `OpenIdAuthenticationMechanismDefinition.clientId`
- The **expiration** claim must be present and must be 'in the future' (a clock skew might be considered or configured in an implementation specific way)
- The **issued at** claim must be present and must be 'in the past' (a clock skew might be considered or configured in an implementation specific way)
- The **not before** claim can be present and if defined, must be 'in the past' (a clock skew might be considered or configured in an implementation specific way)

For the Identity Token, the following check must be performed additionally

- When **nonce** usage is configured, verify if the **nonce** value within the Identity Token is identical to the one that was specified in the authentication request.

### 2.4.4.3. Caller name and groups

A public OpenID Connect Provider generally has no knowledge about roles or groups an end-user (caller) has in a client application (relying party), but a (private) OpenID Connect Provider operated by the same organisation may have. Therefore this specification allows groups to be provided by the client application or by the OpenID Connect Provider (or both).

Groups can be provided by the client application by means of an extra identity store with the `validationTypes` method returning `PROVIDE_GROUPS`. Groups can be provided by the OpenID Connect Provider by means of additional claims.

The claim name that is used to define the Caller Name and optionally the Caller Groups from the OpenID Connect Provider can be defined by the following attributes:

- **Caller Name**: `OpenIdAuthenticationMechanismDefinition.claimsDefinition.callerNameClaim`
- **Caller Groups**: `OpenIdAuthenticationMechanismDefinition.claimsDefinition.callerGroupsClaim`

The following logic is used to determine the value of each;
• If the specified claim exists and has a non-empty value in the Access Token, this Access Token claim value is taken.

• If not resolved yet, and the specified claim exists and has a non-empty value in the Identity Token, this Identity Token claim value is taken.

• If not resolved yet, and the specified claim exists and has a non-empty value in the User Info Token, this User Info Token claim value is taken.

An implementation may choose to not implement the call to the User Info Endpoint, in all cases or when a certain configuration value is set, since not all OpenID Connect Providers support this User Info Endpoint.

The Caller Name and optionally any Caller Groups provided by the OpenID Connect Provider must be present in the CredentialValidationResult that is returned by the implementation specific identity store that validates the Credential holding the Token Response as mentioned above.

Remembering authentication

After the end-user (caller) has been successfully authenticated, the authentication mechanism must ensure that an authenticated session is established in a way functionally equivalent to the functionality provided by the "AutoApplySession Annotation". An implementation may, but does not have to, use that annotation to fulfill this requirement.

Token Expiration

The authentication mechanism must check on each request for which there is an authenticated user if the Access Token or the Identity Token has expired.

In case a token is expired, there are 3 options:

• The token is refreshed when OpenIdAuthenticationMechanismDefinition.tokenAutoRefresh set to true

• A logout takes place when OpenIdAuthenticationMechanismDefinition.logout.accessTokenExpiry or OpenIdAuthenticationMechanismDefinition.logout.identityTokenExpiry set to true and the Access Token respectively Identity Token is expired

• The token expiration is ignored when none of the above conditions hold

In the case a refresh of the token is needed, the OpenID Connect provider refreshToken endpoint has to be called with the following parameters

• The ClientId value as taken from OpenIdAuthenticationMechanismDefinition.clientId

• The ClientSecret value as taken from OpenIdAuthenticationMechanismDefinition.clientId

• The grant_type value set to the constant refresh_token

• the refresh_token value set to the previously stored value from the refresh_token field of the Token Response
When the call is successful and a new Access Token is received, the same logic is applied as described above;

- Validate tokens
- Store in context
- Determine the caller Name and Caller groups values (which can lead to more or less permissions in the application)

When the call is not successful, or when there is no previously stored refresh_token field of the Token Response, a logout should be initiated.

### Logout

The `cleanSubject` method of the authentication mechanism has the following requirements

- Invalidate the HTTP Session, if one is available
- Call the EndSession endpoint of the OpenID Connect Provider if `OpenIdAuthenticationMechanismDefinition.logout.notifyProvider` is set to `true` and such EndSession endpoint is available. If `OpenIdAuthenticationMechanismDefinition.logout.redirectURI` is defined it should be passed along in this call.
- Redirect to `OpenIdAuthenticationMechanismDefinition.logout.redirectURI` if `OpenIdAuthenticationMechanismDefinition.logout.notifyProvider` is set to `false` and `OpenIdAuthenticationMechanismDefinition.logout.redirectURI` is defined
- Redirect to the OpenID Connect Provider Authentication endpoint for re-authentication if the previous two conditions don’t hold. Be aware that a correct `promptType` must be defined so that this option works properly. Without any prompt defined, the Openid Connect Provider can immediately redirect to the `callback` (value of `OpenIdAuthenticationMechanismDefinition.redirectURI`) of the application and the end-user (caller) is then again authenticated within the application, effectively making logging out impossible.

### 2.4.5. 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 [SERVLET60], 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).
2.4. Annotations and Built-In HttpAuthenticationMechanism Beans

```java
@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.
2.4. Annotations and Built-In HttpAuthenticationMechanism Beans

- 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.

### 2.4.6. 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.
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 theCallerPrincipal 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 callernormally by calling `proceed()` on the `InvocationContext`.

- If authentication succeeds, and the caller has requested to be remembered, as determined byevaluating 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".

### 2.4.7. AutoApplySession Annotation

The `AutoApplySession` annotation provides a way to declaratively enable Jakarta Authentication`jakarta.servlet.http.registerSession` behavior for an authentication mechanism, and automatically applyit for every request.

The `jakarta.servlet.http.registerSession` property is described in Section 3.8.4 of [AUTHENTICATION30].

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

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

The container MUST provide an interceptor implementation at priority `PLATFORM_BEFORE + 200` thatbacks 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 "jakarta.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.

### 2.4.8. Implementation Notes

Section 14.4, item 18, of [SERVLET60] 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
<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:
```java
@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(SEND_CONTINUE)) {
            facesContext.responseComplete();
        } else if (status.equals(SEND_FAILURE)) {
            addError(facesContext, "Authentication failed");
        }
    }
}
```

### 2.4.10. 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 [SERVLET60], but has been extended to support additional functionality defined by the Servlet Container Profile of [AUTHENTICATION30].

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.

### 2.4.11. AutoApplySession Notes

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

```java
```

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

```java
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 Jakarta Contexts and Dependency Injection spec, version 4.0 [CDI40].

The methods defined by the `HttpAuthenticationMechanism` closely map to the methods and semantics of a `ServerAuthModule`, as defined by the Servlet Container Profile of [AUTHENTICATION30]. (But an `HttpAuthenticationMechanism` is itself not a `ServerAuthModule`.) The servlet container MUST use Jakarta Authentication 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 [SERVLET60] (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 [SERVLET60].

The BASIC and FORM authentication mechanisms as defined by this specification are logically equivalent to the similarly named authentication mechanisms in [SERVLET60], 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 Jakarta EE products (albeit in vendor-specific ways). Unlike LoginModule, IdentityStore is intended specifically for Jakarta 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 Subjects.

IdentityStore is intended primarily for use by HttpAuthenticationMechanism implementations, but could in theory be used by other types of authentication mechanisms (e.g., a Jakarta Authentication 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).

```java
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.

```java
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.

```java
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 \texttt{getCallerGroups()} when there is only one \textit{IdentityStore}, because the same groups are returned by the \textit{validate()} method.

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

Because \texttt{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 \texttt{getCallerGroups()} are strongly encouraged to check that the calling context has \textit{IdentityStorePermission}, as shown below, before proceeding. (The built-in identity stores are \textbf{REQUIRED} to do so, see Annotations and Built-In IdentityStore Beans.)

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

### 3.2.3. Declaring Capabilities

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

```java
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 \textit{priority()} method allows an \textit{IdentityStore} to be configured with an ordinal number indicating the order in which it should be consulted when multiple \textit{IdentityStores} 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:

```java
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.

```java
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 Jakarta 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.

```java
@Retention(RUNTIME)
@Target(TYPE)
public @interface LdapIdentityStoreDefinition {
    enum LdapSearchScope { ONE_LEVEL, SUBTREE }

    String url() default "";
    String bindDn() default "";
    String bindDnPassword() default "";
    String callerBaseDn() default "";
}
```
3.4.2. Database Annotation

The `DatabaseIdentityStoreDefinition` annotation configures an instance of the built-in database identity store.
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 [CDI40].
Chapter 4. Security Context

This chapter describes the main SecurityContext interface and contract and other context interfaces and their contract.

4.1. Introduction

The Jakarta 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 enterprise beans 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:

```java
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 EJBCurrent.getCallerPrincipal(), which, per Jakarta Enterprise Beans spec, version 3.2 [JEB32], 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:

```java
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 [SERVLET60] 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:

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

And the following call to hasAccessToWebResource():

```java
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:

```java
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 [CDI40].

Various specifications in Jakarta EE provide similar or even identical methods to those provided by 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`
- Enterprise Beans - `EJBContext#getCallerPrincipal`, `EJBContext#isCallerInRole`
- XML Web Services - `WebServiceContext#getUserPrincipal`, `WebServiceContext#isUserInRole`
- RESTful Web Services - `SecurityContext#getUserPrincipal`, `SecurityContext#isUserInRole`
- Server Faces - `ExternalContext#getUserPrincipal`, `ExternalContext#isUserInRole`
- Contexts and Dependency Injection - `@Inject Principal`
- WebSocket - `Session#getUserPrincipal`

4.6. Other context interfaces
4.6.1. OpenIdContext

An implementation must provide a CDI bean for the OpenIdContext interface with scope SessionScoped. This bean is used to programmatically access data received from an OpenID Connect Provider when the OpenID Connect authentication mechanism (via the OpenID Connect Annotation) has been used to authenticate the caller.
Bibliography

The following documents are referenced by this specification.

[CDI40]
Jakarta Contexts and Dependency Injection, version 4.0
https://jakarta.ee/specifications/cdi/4.0/

[EB32]
Jakarta Enterprise Beans, version 3.2
https://jakarta.ee/specifications/enterprise-beans/3.2/

[EL50]
Jakarta Expression Language, version 5.0
https://jakarta.ee/specifications/expression-language/5.0/

[AUTHORIZATION21]
Jakarta Authorization, version 2.1
https://jakarta.ee/specifications/authorization/2.1/

[AUTHENTICATION30]
Jakarta Authentication, version 3.0
https://jakarta.ee/specifications/authentication/3.0/

[RFC2119]
RFC 2119, "Key words for use in RFCs to Indicate Requirement Level"

[RFC7617]
RFC 7617, "The 'Basic' HTTP Authentication Scheme"

[SECAPI]
Jakarta Security, version 3.0
https://jakarta.ee/specifications/security/3.0/

[SERVLET60]
Jakarta Servlet, version 6.0
https://jakarta.ee/specifications/servlet/6.0/

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