Get started with the Java EE 8 Security API, Part 3: Securely access user credentials with IdentityStore

Authenticate and authorize users with the new IdentityStore API

Alex Theedom

April 12, 2018

Learn how to use the new IdentityStore interface to setup and configure RDBMS or LDAP identity storage in your Java web applications.

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

The first article in this series presented a high-level introduction to basic features and components of the new Java™ EE Security API (JSR 375), including the new `IdentityStore` interface. In this article you'll learn how to use `IdentityStore` to securely store and access user credential data in your Java web applications.

The new `IdentityStore` abstraction is one of three headline features in the Java EE Security API specification release. An *identity store* is a database that stores user identity data such as user name, group membership, and other information used to verify a caller's credentials. While `IdentityStore` is designed to be used with any authentication mechanism, it is especially well suited to integrating with Java EE 8's `HttpAuthenticationMechanism`, which I introduced in Part 2.

Get the code

Installing Soteria

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

1. Explicitly specify Soteria in your POM

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

Listing 1. Maven coordinates for the Soteria project

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

2. Use built-in Java EE 8 coordinates

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

Listing 2. Java EE 8 Maven coordinates

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

Interfaces, classes, and annotations related to `IdentityStore` are located in the `javax.security.enterprise.identitystore` package.

How IdentityStore works

Similar to the JAAS LoginModule interface, `IdentityStore` is an abstraction used to interact with identity stores and authenticate users and retrieve group memberships. `IdentityStore` is designed to work well with `HttpAuthenticationMechanism` but you may use any authentication mechanism you wish. You also have your choice of whether to use containers, but using a container with the `IdentityStore` mechanism is recommended for most scenarios. Combining `IdentityStore` with a container lets you control the identity stores in a portable, standard way.

IdentityStoreHandler

Instances of `IdentityStore` are managed with the `IdentityStoreHandler`, which provides mechanisms for querying all available identity stores. An instance of the handler type is made available for injection via CDI, as shown in Listing 3. This will be used wherever authentication needs to happen. (See Part 1 for an overview of CDI in the Java EE Security API.)

Listing 3. Inject the identity store handler

```
@Inject
private IdentityStoreHandler idStoreHandler;
```

The `IdentityStoreHandler` interface has one method, `validate()`, which accepts a credential instance. Implementations of this method will typically invoke the `validate()` and `getCallerGroups()` methods associated with one or more `IdentityStore` implementations, then return an aggregated result. I'll explain more about this feature later in the tutorial.

The Java EE Security API comes with a default implementation of the `IdentityStoreHandler` interface, which should suffice in most cases. You also have the option to replace the default with a custom implementation.

The default implementation of `IdentityStoreHandler` authenticates against multiple `IdentityStore`s. It iterates over a list of stores, and returns an aggregated result in the form of a `CredentialValidationResult` instance. This object can be very simple or more complex. At its simplest, it delivers a status value of `NOT_VALIDATED`, `INVALID`, or `VALID`. In many cases, you will want some combination of these additional values:

- `CallerPrincipal`, with or without the caller's groups
- The caller's name or LDAP-distinguished name
- The caller's unique identifier from the identity store

For now we will focus on defaults, but later in the article I will show you how to setup your own lightweight identity store by implementing the `IdentityStore` interface.

Built-in identity stores

The Java EE Security API comes with built-in `IdentityStore` implementations for LDAP and RDBMS. Like other features in the new Security API, these are easily invoked with annotations.

Calling a built-in RDBMS integration

External databases are accessible via a `DataSource` bound to JNDI. You'll use the `@DataBaseIdentityStoreDefinition` annotation to activate an external database. Once activated, you'll configure connection details by passing values to the annotation.

Calling a built-in LDAP integration

You'll use the `@LdapIdentityStoreDefinition` annotation to call and configure an LDAP `IdentityStore` bean. After you've called the bean, you can pass in the configuration details required to connect to an external LDAP server.

Note that these implementations are application-scoped CDI beans and are based on the `@DataStoreDefinition` annotation already available in Java EE 7.

How to configure a built-in RDBMS identity store

The simplest built-in identity store is the database store, which is configured via the `@DataBaseIdentityStoreDefinition` annotation. Listing 4 shows a sample configuration for a built-in database store.

Listing 4. Configuring an RDBMS identity store

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

The configuration options in Listing 4 should be familiar if you've ever configured a database definition. One thing you should note is the priority setting of 10. This value is used in cases where multiple identity stores have been implemented. It's used to determine the order of iteration, which I will discuss in more detail soon.

There are nine possible parameters you can use to configure your database. You can review them in the Javadoc for `DatabaseIdentityStoreDefinition`.

How to configure a built-in LDAP identity store

The LDAP configuration has far more configuration options than the RDBMS option. If you are experienced with LDAP configuration semantics, the configuration options will be familiar to you. Listing 5 shows a subset of the options for configuring an LDAP identity store.

Listing 5. Configuration for an LDAP identity store

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

See the `LdapIdentityStoreDefinition` Javadoc to view the 24 possible parameters for configuring an LDAP identity store.

Develop a custom identity store

If neither of the built-in identity stores satisfies your requirements, then you might use the `IdentityStore` interface to develop a custom solution. The `IdentityStore` interface comes with four methods and all have default implementations. Listing 6 shows the signature for each of these methods.

Listing 6. IdentityStore's four methods

```
default CredentialValidationResult validate(Credential)
default Set<String> getCallerGroups(CredentialValidationResult)
default int priority()
default Set<ValidationType> validationTypes()
```

All methods in the `IdentityStore` interface are marked `default`, so it isn't obligatory to provide implementations. Two key methods are called by default, and a third is used for cases where you've configured multiple identity stores:

- **`validate()`** determines if the given `Credential` is valid and returns a `CredentialValidationResult`.
- **`getCallerGroups()`** returns a `Set` of group names that the caller is associated with and aggregates them with groups already listed in the `CredentialValidationResult` instance.

- **`getPriority()`** comes into play when more than one `IdentityStore` is defined. The lower the value the higher the priority. Equal priorities have undefined behavior.
- **`validationTypes()`** returns a set of `ValidationTypes` which determine which method/s (`validate()` and/or `getCallerGroups()`) have been implemented.

An invocation of the `validate()` method determines if the given `Credential` is valid and returns a `CredentialValidationResult`. Various methods on the returned `CredentialValidationResult` instance provide details about the caller's LDAP-distinguished name, unique identity store ID, result status, identity store ID, `Principal`, and group membership.

> **Note**: *Result status* is important for determining the behavior of the `IdentityStoreHandler` when more than one `IdentityStoreHandler` has been configured; status options are `NOT_VALIDATED`, `INVALID`, or `VALID`.

Implementing validate() and getCallerGroups()

The `validate()` and `getCallerGroups()` methods are used to validate a caller's `Credential` or get their group information. Either or both methods can be used by a data store implementation. The methods that are actually implemented are declared by the `validationTypes()` method.

This feature allows you the flexibility to specify one identity store to perform authentication, while another is tasked with authorization. The `validationTypes()` method returns a set of `ValidationTypes`, which can contain `VALIDATE` or `PROVIDE_GROUPS` or both. The `VALIDATE` constant signifies that the `validate()` method has been implemented, and `PROVIDE_GROUPS` signifies that the `getCallerGroups()` method has been implemented. If both are returned then both methods have been implemented.

> **Note**: An `IdentityStore` should not maintain state, nor should it have any knowledge of the caller's current progress in the authentication process. Logically, it does not make sense for the store to track a user's authentication state.

Handling multiple identity stores

The `IdentityStoreHandler` is used in scenarios that require handling multiple `IdentityStore` implementations. It provides one method called `validate()`, which has the same signature as the method of the same name on the `IdentityStore` implementation. The idea is to allow multiple identity stores to effectively operate as a single `IdentityStore`.

The `validate()` method on the `IdentityStoreHandler` performs a query of the identity stores using the following logic:

1. Call the `validate()` method of the identity stores in accordance with the capabilities declared by the `validationTypes()` method. Methods are called in the order determined by the `getPriority()` method.
    - If a `VALID` status result is returned, then no further identity stores need be interrogated. In that case logic jumps to Step 2.
    - If the status is `INVALID`, then this status is remembered for later use and the `IdentityStoreHandler` continues interrogating the remaining identity stores.

2. If only an `INVALID` status is returned then `INVALID` is return; otherwise `NOT_VALIDATED` is returned.
3. If a `VALID` result is returned and the identity store declares the `PROVIDE_GROUPS` validation type, then the `IdentityStoreHandler` will start collecting the caller group membership, which it does by aggregating the caller groups returned in the `CredentialValidationResult` object.
    - All `IdentityStores` that declare only the `PROVIDE_GROUPS` validation type are interrogated by calling the `getCallerGroups()` method. The returned list of group names is aggregated with the set of accumulated groups.
4. Once all `IdentityStores` have been interrogated, a `CredentialValidationResult` is constructed with a `VALID` status and the list of caller groups, and is returned.

Interrogation in practice

Now let's look at a scenario that requires interrogating multiple identity stores. In this scenario, IdentityStore 1 connects to an RDBMS, while IdentityStore 2 and IdentityStore 3 connect to an LDAP container.

In Figure 1 the identity store handler iterates over `IdentityStore` instances in priority order, calling the `validate()` method on each instance until it finds a `CredentialValidationResult` that returns a `VALID` status. This happens on interrogating IdentityStore 2. The handler stops the iteration and starts the second iteration to collect the caller's groups.

Figure 1. IdentityStoreHandler's first interrogation of identity stores

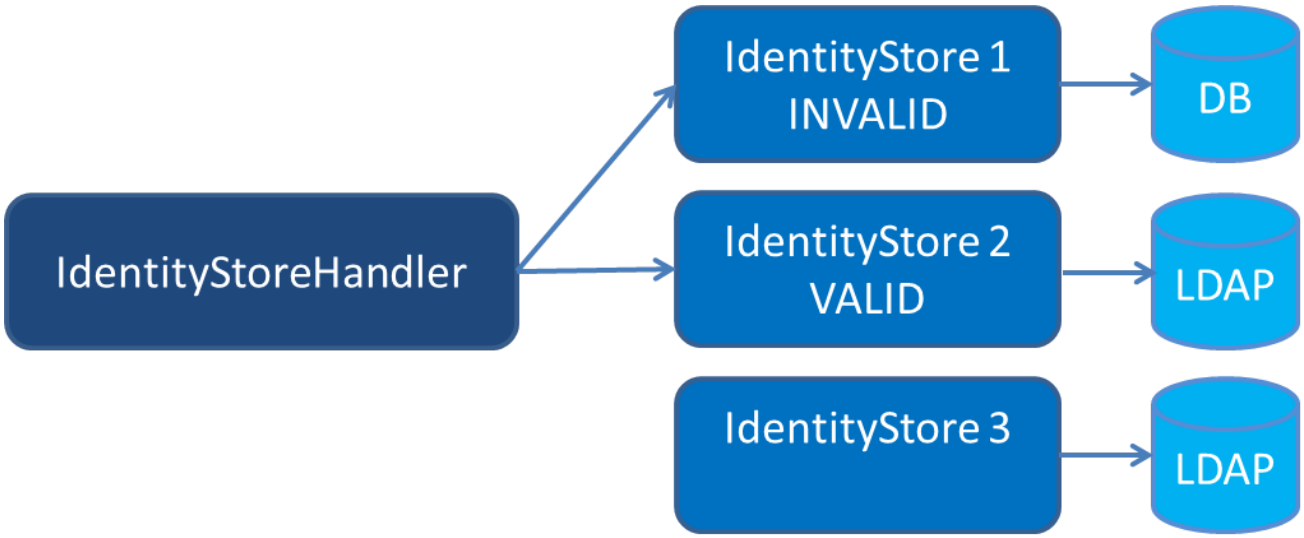
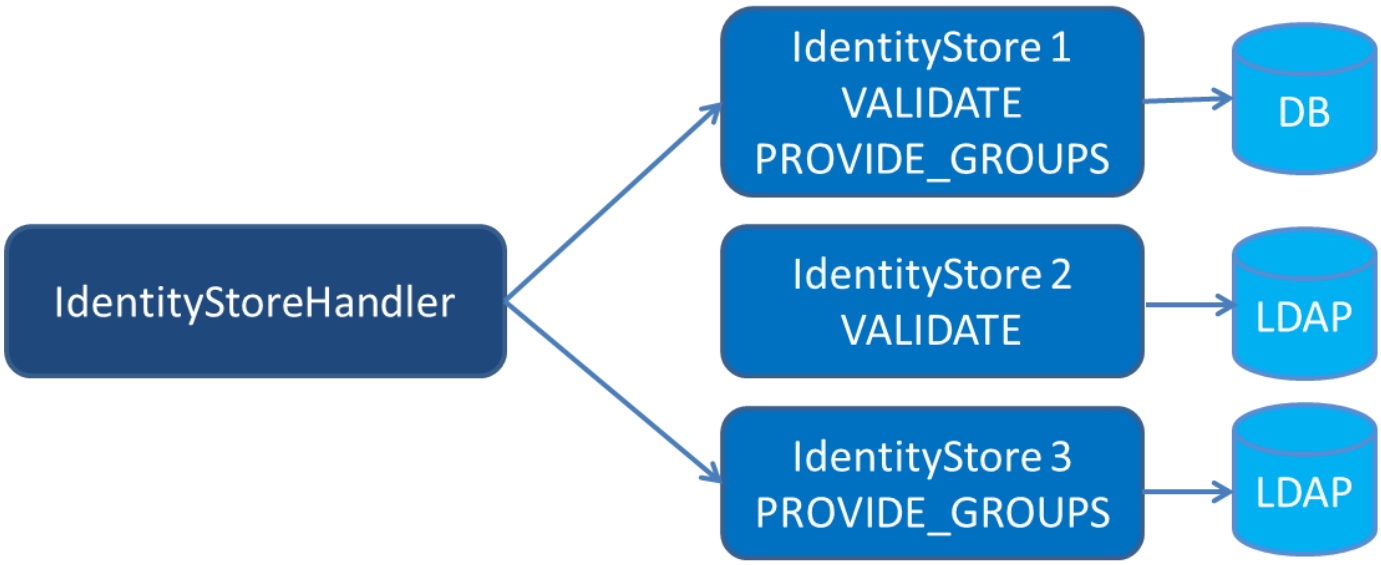


Figure 2 represents the second iteration. The handler calls the `getCallerGroups()` method on each `IdentityStore` instance, declaring a validation type of `PROVIDE_GROUPS` only.

In this scenario, the only identity store fitting that specification is IdentityStore 3. The caller groups returned are combined with the set of group names returned by calling `getCallerGroups()` on the `CredentialValidationResult` instance returned by IdentityStore 2.

Figure 2. IdentityStoreHandler's second interrogation of identity stores



Once all the `IdentityStores` have been interrogated, a `CredentialValidationResult` is constructed with a `VALID` status and the list of caller groups is returned.

This simple example demonstrates how it is possible for a caller to be validated by one `IdentityStore`, and for a group membership list to be built from another.

Credentials with cookies

Just as you saw with the `HttpAuthenticationMechanism` in Part 2, it is fairly easy to develop a custom `IdentityStore` solution using cookies. The `RememberMeIdentityStore` is similar to the `IdentityStore` interface, but is intended to be used by the interceptor binding backing the `@RememberMe` annotation, rather than by an authentication mechanism.

The `RememberMeIdentityStore` is used to:

- Generate a "remember me" login token for a caller.
- Remember the caller associated with the "remember me" login token.
- Validate the login token if the caller returns, and re-authenticate the caller without requiring additional credentials.

The `validate()` method is passed the `RememberMeCredential` and validates it, while the `generateLoginToken()` method associates a token with the given groups and principal. If no login token is found for the caller, or if the login token has expired, then normal authentication takes place.

Conclusion to Part 3

The `IdentityStore` interface provides the long-awaited simplification needed to integrate external caller authentication and authorization mechanisms in your Java enterprise applications.

`IdentityStore` ensures portability across containers and servers, and makes it easy to communicate seamlessly with multiple identity stores.

If you don't need to implement a custom identity store, then just one annotation and a few connection details are enough to configure an LDAP container or RDBMS. Any Java EE 8 identity store will back the built-in `HttpAuthenticationMechanism`, so connecting LDAP logins to web users is extremely simple, requiring only a few annotations.

Stay tuned for the final article in this tutorial series, introducing the new `SecurityContext` interface.

Test your knowledge

1. Which of the following are used to configure built-in identity stores? (Select all that apply.)
    a. `@LdapIdentityStoreDefinition`
    b. `@DatabaseIdentityStoreDefinition`
    c. `@RdbmsIdentityStoreDefinition`
    d. `@DataBaseIdentityStoreDefinition`
    e. `@RememberMeIdentityStoreDefinition`
2. Which of the following `IdentityStore` interface methods have default implementations?
    a. Only `priority()` and `validationTypes()`.
    b. Only `priority()` and if not set the default priority is 100.
    c. Only `CredentialValidationResult()`, `priority()` and `validationTypes()`.
    d. All four interface methods.
    e. None of the interface methods.
3. Given multiple `IdentityStore` implementations, what is the default behaviour of the `IdentityStoreHandler` when a call to the `validate()` method returns `VALID`?
    a. It continues to interrogate the remaining identity stores before commencing with the second pass of the identity stores.
    b. It stops iteration and confirms the caller’s authorization by returning a `CredentialValidationResult` object.
    c. It restarts the iteration over the identity stores and calls the `getCallerGroups()` method.
    d. It calls the `getCallerGroups()` method on that identity store and constructs and returns a `CredentialValidationResult` object.
    e. None of the above
4. Which one of the following types are returned by calling the `getCallerGroups()` method on an `IdentityStore` instance?
    a. `List<String>`
    b. `Set<String>`
    c. `Map<Caller, String>`
    d. `Set<Group>`
    e. `List<Group>`
5. Which of the following statements about the `RememberMeIdentityStore` are true?
    a. `RememberMeIdentityStore` extends `IdentityStore`.
    b. It is intended to be used by the interceptor binding backing the `@RememberMe` annotation.
    c. It can be used to re-authenticate the caller without the need to provide additional credentials.

d. It is one of the three built in IdentityStore types.
e. If the "remember me" login token has expired, normal authentication takes place.

Check your answers.

Related topics

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