Introduction to PicketLink

JBUG, London

JBoss by Red Hat

Peter Škopek, pskopek@redhat.com, twitter: @pskopek

Apr 30, 2014

Abstract

PicketLink is project offering multiple solutions to identity management and security for Java applications.

Section 1
Welcome

Agenda

Section 2

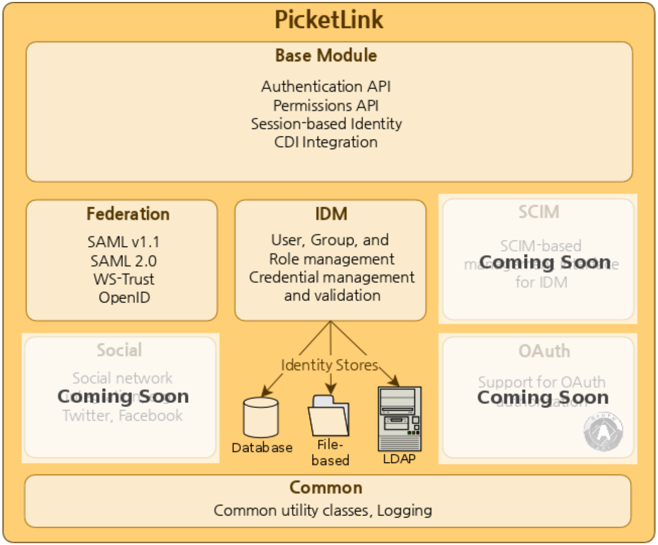
PicketLink Overview

What is PicketLink?

PicketLink is an Application Security Framework for Java EE applications. It provides features for authenticating users, authorizing access to the business methods of your application, managing your application's users, groups, roles and permissions. Areas of usage:

- Java EE Application Security
- Identity Management
- Identity Federation
- Social Login
- Mobile Applications Security
- REST Applications Security

Areas of Usage



Java EE Application Security

- Authentication
- Authorization and Permissions API
- CDI Based Integration
- REST Security

Identity Management

- Built-in Identity Stores for Databases and LDAP
- Rich and Extensible API
- Multitenancy

Identity Federation

- SAML (v2.0 and v1.1)
- OAuth2
- XACML v2
- OpenID
- Security Token Server (STS) for WS-Trust

Social Login

- Facebook Connect
- Twitter Login
- Google+ Login

Section 3

Java EE Application Security

PicketLink API

Identity - central PicketLink API bean has following methods:

- boolean isLoggedIn();
- AuthenticationResult login() throws AuthenticationException;
- void logout();
- boolean hasPermission(Object resource, String operation);
- boolean hasPermission(Class<?>resourceClass, Serializable identifier, String operation);
- Account getAccount();

Please note that Identity bean is marked with @Named annotation, which means that its methods may be invoked directly from the view layer (if the view layer, such as JSF, supports it) via an EL expression.

Authentication Example in JSF application

```
@Named
@RequestScoped
public class AuthController {
    @Inject
    private FacesContext facesContext;

    @Inject
    private Conversation conversation;

    @Inject
    private Identity identity;

    public void login() {

        AuthenticationResult result = identity.login();
        if (AuthenticationResult.FAILED.equals(result)) {
            facesContext.addMessage(null, new FacesMessage(
                    "Authentication was unsuccessful.  Please check your username and password "
                            + "before trying again."));
        }
        else {

            User user = getCurrentUser();
            if (user == null) {
                user = createUser(getCurrentUserName());
            }
            conversation.begin();
        }
    }
}
```

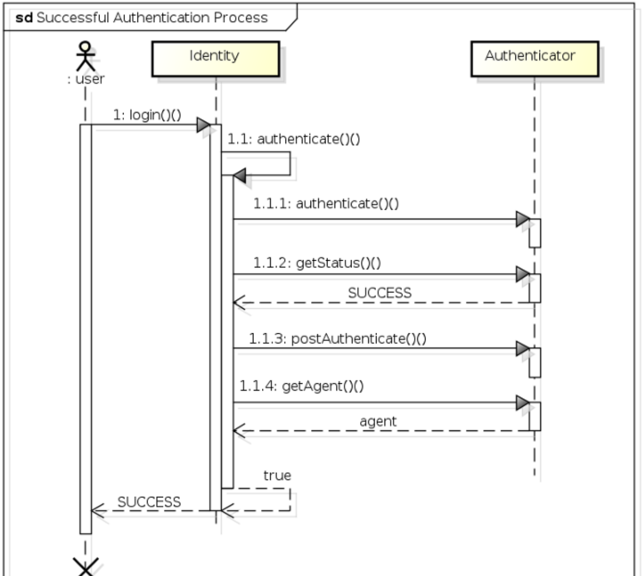
Authentication Example in JSF application

```
<h:form rendered="#{not identity.loggedIn}">
    <h:panelGrid columns="2" >
        <h:outputLabel value="Username" />
        <h:inputText value="#{loginCredentials.userId}" />

        <h:outputLabel value="Password" />
        <h:inputSecret value="#{loginCredentials.password}" />

        <h:commandButton value="Log in" action="#{authController.login}" />
    </h:panelGrid>
</h:form>
```

Authentication Process



Authenticator

It is a bean handling authentication.

```
@PicketLink
public class SimpleAuthenticator extends BaseAuthenticator {

    @Inject DefaultLoginCredentials credentials;

    @Override
    public void authenticate() {
        if ("jsmith".equals(credentials.getUserId()) &&
                "abc123".equals(credentials.getPassword())) {
            setStatus(AuthenticationStatus.SUCCESS);
            setAccount(new User("jsmith"));
        } else {
            setStatus(AuthenticationStatus.FAILURE);
            FacesContext.getCurrentInstance().addMessage(null, new FacesMessage(
                    "Authentication Failure - The username or password you provided were invalid."));
        }
    }
}
```

Authenticator

The first thing we can notice about the above code is that the class is annotated with the @PicketLink annotation.
This annotation indicates that this bean should be used for the authentication process.
The next thing is that the authenticator class extends something called BaseAuthenticator. Which is abstract base class provided by PicketLink which makes it easier for custom authenticator to just implement authenticate() method.

Credentials

Credentials are something that provides evidence of a user's identity. PicketLink has extensive support for a variety of credential types, and also makes it relatively simple to add custom support for credential types that PicketLink doesn't support out of the box itself.

- UsernamePasswordCredentials
- TOTPCredentials
- X509CertificateCredentials
- DigestCredentials

Custom Authenticator and DefaultLoginCredentials

The DefaultLoginCredentials bean is provided by PicketLink as a convenience, and is intended to serve as a general purpose Credentials implementation suitable for a variety of use cases. It supports the setting of a userId and credential property, and provides convenience methods for working with text-based passwords. It is a request-scoped bean and is also annotated with @Named so as to make it accessible directly from the view layer.

```
public class VerySimpleAuthenticator extends BaseAuthenticator {

    @Inject DefaultLoginCredentials credentials;

    // code snipped
}
```

Permissions API

The Permissions API is a set of extensible authorization features that provide capabilities for determining access privileges for application resources.

```
public interface Permission {

    Object getResource();

    Class<?> getResourceClass();

    Serializable getResourceIdentifier();

    IdentityType getAssignee();

    String getOperation();
}
```

Permissions API

Each permission instance represents a specific resource permission, and contains three important pieces of state:

- assignee - identity to which the permission is assigned
- operation - string value that represents the exact action that the assigned identity is allowed to perform
- reference - to the resource (if known), or a combination of a resource class and resource identifier. This value represents the resource to which the permission applies.

Checking permissions for the current user

The primary method for accessing the Permissions API is via the Identity bean, which provides the following two methods for checking permissions for the currently authenticated user:

- `boolean hasPermission(Object resource, String operation);`
- `boolean hasPermission(Class<?> resourceClass, Serializable identifier, String operation);`

Checking permissions for the current user - example 1

```
@Inject Identity identity;

public void deleteAccount(Account account) {
    // Check the current user has permission to delete the account
    if (identity.hasPermission(account, "DELETE")) {
        // Logic to delete Account object goes here
    } else {
        throw new SecurityException("Insufficient privileges!");
    }
}
```

Checking permissions for the current user - example 2

When you don't have a reference to the resource object, but you have it's identifier value (for example the primary key value of an entity bean). It is more efficient to not a resource when don't actually need it.

```
@Inject Identity identity;

public void deleteCustomer(Long customerId) {
    // Check the current user has permission to delete the customer
    if (identity.hasPermission(Customer.class, customerId, "DELETE")) {
        // Logic to delete Customer object goes here
    } else {
        throw new SecurityException("Insufficient privileges!");
    }
}
```

Restricting resource operations

For many resource types it makes sense to restrict the set of resource operations for which permissions might be assigned. classOperation option can be set to true if the permission applies to the class itself, and not an instance of a class.

```
@Entity
@AllowedOperations({
    @AllowedOperation(value = "CREATE", classOperation = true),
    @AllowedOperation(value = "READ"),
    @AllowedOperation(value = "UPDATE"),
    @AllowedOperation(value = "DELETE")
})
public class Country implements Serializable {
```

Restricting resource operations on Class

One can check if the current user has permission to actually create a new Country bean. In this case, the permission check would look something like this:

```
@Inject Identity identity;

public void createCountry() {
    if (!identity.hasPermission(Country.class, "CREATE")) {
        throw new SecurityException(
          "Current user has insufficient privileges for this operation.");
    }

    ....
}
```

Section 4
Identity Management

Overview

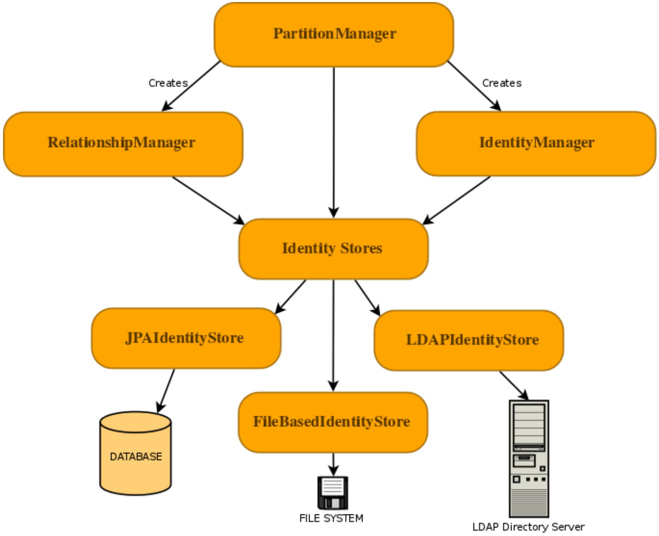
PicketLink Identity Management is a fundamental module of PicketLink, with all other modules building on top of the IDM component to implement their extended features.

- Provides rich and extensible API for managing the identities (such as users, groups and roles) of your applications and services
- Supports a flexible system for identity partitioning
- Provides the core Identity Model API classes (see below) upon which an application's identity classes are defined to provide the security structure for that application

Core Concepts

- PartitionManager
  is used to manage identity partitions, which are essentially containers for a set of identity objects
- IdentityManager
  is used to manage identity objects within the scope of a partition
- RelationshipManager
  is used to manage relationships - a relationship is a typed association between two or more identities
- IdentityStore
  provides the backend store for identity persistency
  - JPAIdentityStore
  - LDAPIdentityStore
  - FileBasedIdentityStore

Schema


PartitionManager
Creates
Creates
RelationshipManager
IdentityManager
Identity Stores
JPAIdentityStore
LDAPIdentityStore
FileBasedIdentityStore
DATABASE
FILE SYSTEM
LDAP Directory Server

Section 5

PicketLink Federation

PicketLink Federation - Overview

- The PicketLink Federation project provides the support for Federated Identity and Single Sign On type scenarios.
- PicketLink provides support for technologies
  - SAML (v2.0 and v1.1)
  - OAuth2
  - XACML v2
  - OpenID
  - Security Token Server (STS) for WS-Trust
  - OAuth2
- Integration with following servers is supported
  - WildFly 8
  - JBoss AS7
  - JBoss Application Server v5.0 onwards
  - Apache Tomcat v5.5 onwards

Section 6
PicketLink WildFly Subsystems

General

The PicketLink Subsystem extends WildFly Application Server to introduce some new capabilities, providing a infrastructure to deploy and manage PicketLink deployments and services.

- Minimal configuration for deployments. Part of the configuration is done automatically with some hooks for customizations.
- Minimal dependencies for deployments. All PicketLink dependencies are automatically set from modules.
- Configuration management using JBoss Application Server Management API. It can be managed in different ways: HTTP/JSON, CLI, Native DMR, etc.
- example located at

  `docs/examples/configs/standalone-picketlink.xml`

Federation Subsystem

- A rich domain model supporting the configuration of PicketLink Federation deployments and Identity Management services.
- No need to provide picketlink.xml deployment descriptor
- Cetralized configuration

Identity Managemet Subsystem

Subsystem parses the configuration, automatically build a org.picketlink.idm.PartitionManager and expose it via JNDI for further access.

- Externalize and centralize the IDM configuration for deployments
- Define multiple configuration for identity management services
- Expose the PartitionManager via JNDI for further access
- If using CDI, inject the PartitionManager instances using the Resource annotation
- If using CDI, use the PicketLink IDM alone without requiring the base module dependencies

Bibliography

PicketLink Documentation Site
`http://docs.jboss.org/picketlink/2/latest/reference/html/`

Security Assertion Markup Language (SAML) v2.0
`https://www.oasis-open.org/standards#samlv2.0`

PicketLink Web Site
`http://www.picketlink.org/`

The end.

Thanks for listening.