Octopus Framework

Rudy De Busscher

Version 0.4, ??/??/2018

Table of Contents

Release notes

0.4

1. Integration with Keycloak (Client Credentials for Java SE, AuthorizationCode grant for Web, AccessToken for JAX-RS)
2. Supported for Hashed Passwords (MessageDigest ones and PBKDF2)
3. Support for MP rest Client and Providers available to add tokens for MP JWT Auth and Keycloak
4. Logout functionality for Web.
5. Authentication events.
6. More features for JAX-RS integration (authorization violations on JAX-RS resource [no need for CDI or EJB], correct 401 return messages, …)
7. Support for default user filter (no need to define user filter before authorizationFilter)

Important changes

1. Primary Principal is always a *be.atbash.ee.security.octopus.subject.UserPrincipal* instance.
2. NamedPermission interface extends now Permission.

This means that enums implementing NamedPermission must implement implies. When there is no hierarchy between the values, the following method can be used (for *DemoPermission* enum)

```
public boolean implies(Permission permission) {
    return permission instanceof DemoPermission && permission.equals(this);
}
```

1. *be.atbash.ee.security.octopus.subject.PrincipalCollection* is now a class and longer an interface.
2. All filters used within filter chain definition (securedURLs.ini) must be an implementation of *be.atbash.ee.security.octopus.filter.AdviceFilter*

0.3

1. Authorization parts implementation (tags like securedComponent, interceptors for EJB/CDI, filters for URLs)
2. AuthorizationToken to be able to extract AuthorizationInfo from token (MicroProfile, Offline, …)
3. Basic support for authorization annotation in Java SE (with *MethodAuthorizationChecker.checkAuthorization();*)
4. Declarative permissions for FXML views in JavaFX (POC)

0.2

1. Split into different modules (Core, JSON, Non-Web [Java SE, …], Web [JSF, JAX-RS])
2. Octopus-jwt-support for handling JSON supporting plain, JWS and JWE.
3. Octopus-json is optimized smart-json code
4. MicroProfile JWT Auth for Rest (POC)
5. OfflineToken for standalone Java SE (POC)

0.1

1. POC integration Apache Shiro into Octopus
2. Use of CDI as much as possible.

Modules

List of Maven modules

Artefact	SE, CDI, EE	info
be.atbash.ee.security:octopus-core	SE, CDI	All Octopus classes usable in Java SE and Java EE environment.
be.atbash.ee.security:octopus-common-se	SE, CDI	All Octopus classes Specific for Java SE
be.atbash.ee.security:octopus-se-standalone	SE, CDI	Specific for Java SE CLI programs
be.atbash.ee.security:octopus-token-generator	SE, CDI	Contains class to generate the Offline Token (for SE usage).
be.atbash.ee.security:octopus-javafx	SE (JavaFX)	Integration within FXML views.
be.atbash.ee.security:octopus-keycloak-se	SE	Integration of Keycloak with CLI programs (pure Java SE or JavaFX).
be.atbash.ee.security:octopus-common-web	EE (Web)	All Octopus classes Specific for Java EE (Web - Servlets)
be.atbash.ee.security:octopus-jsf7	EE (JSF)	Specific for JSF
be.atbash.ee.security:octopus-keycloak	EE (JSF)	Integration of Keycloak specific for JSF
be.atbash.ee.security:keycloak	EE (JSF)	Integration with Keycloak (including SSO)
be.atbash.ee.security:octopus-rest	EE (JAX-RS)	Specific for JAX-RS
be.atbash.ee.security:octopus-mp-rest-client	SE (JAX-RS), EE (JAX-RS)	Support for MP Rest Client with MP JWT auth

Artefact	SE, CDI, EE	info
be.atbash.ee.security:octopus-mp	EE (JAX-RS)	Support for MP JWT Auth tokens server side.
be.atbash.ee.security:octopus-keycloak-rest	EE (JAX-RS)	Support for Keycloak Access Tokens server side
be.atbash.ee.security:octopus-keycloak-rest-client	SE (JAX-RS), EE (JAX-RS)	Support for MP Rest Client with Keycloak Access Token

octopus-utilities contains for the moment the JavaFX app to maintain JWK files.

Authentication

Authentication ways

Octopus allows different methods for the authentication of the other party.

1. Octopus is able to verify if the user-supplied credentials (user name - password combination) is valid. For example Database and File based storages.
2. Octopus passes the user-supplied credentials (user name - password combination) to an external source for verification. For example LDAP.
3. Authentication is 'externalised' and application is contacted with a token. Examples are Google OAuth2, CAS, SAML, Keycloak, Octopus SSO, etc.

Octopus verifies

In this case, we need to supply the password using the AuthenticationInfoBuilder to Octopus. The defaults matchers (There is also support for hashed password, which is recommended of course) are able then to verify if the password matches.

External validation

In case we can't supply Octopus the password, but user has entered credentials in our application, we can ask for an external authentication and supply a correct *Matcher* which calls the external validation. For LDAP, their is a Octopus supplied one which can be configured.

External authentication

When the user enters the credentials in an external system and the verification also happens there, we need special handling for receiving the token which identifies the authenticated user.

Summary

In the below table, one can see which of the 3 options applies to your situation.

Credential entry	Credential verification	Type
Application	Application	Octopus Verifies

Credential entry	Credential verification	Type
Application	External	External validation
External	External	External authentication

AuthenticationInfoProvider

The interface *be.atbash.ee.security.octopus.authc.AuthenticationInfoProvider* needs to be implemented by the developer if he want to supply some authentication information to Octopus in response to an AuthenticationToken.

The AuthenticationToken is the data created in response of a authentication request from the end user. This can be a UsernamePasswordToken in the case our application has shown a login form to the end user for this information. But the AuthenticationToken can also be a token from the external system in response of a successfull authentication in that system.

As developer you can use an implementation of this interface to retrieve the password from your database for instance. Depending on the environment (CDI based or not), the instance must be configured differently.

1. CDI environment → Define the implementation as CDI bean by annotating it as Application scoped bean (since no user specific information is kept, this is the best scope)
2. non CDI environment → Define the class through the Service loader mechanism (create a file src/main/resources/META-INF/services/be.atbash.ee.security.octopus.authc.AuthenticationInfoProvider which contains the fully qualified name of the implementation class)

In both cases, the method *getAuthenticationInfo* should return null when the user name cannot be found (but maybe can be resolved through another AuthenticationInfoProvider, see ???) or an instance of AuthenticationInfo which can be created through the AuthenticationInfoBuilder.

AuthenticationInfoBuilder

With the AuthenticationInfoBuilder, we can create an instance of AuthenticationInfo which provides Octopus the necessary information to decide if the user can be authenticated.

1. principalId(Serializable) : Required, uniquely identifies the user. It can later be used to determine the permissions for the user.
2. name(String) : Optional, defines the *full name* of the user.
3. userName(String) : Optional, defines the user name to identify the user.
4. password(Object) : Optional (password or token is required), defines the password known for the user internally (can be the hashed format, see ???)
5. salt(byte[]) : Optional (recommended for password usage), defines the salt when creating the hashed version of the password.
6. externalPasswordCheck() : Optional, indicates that Ocoptus can’t verify the user and that an external system must perform this (for ex LDAP)

7. token(ValidatedAuthenticationToken) : Optional (password or token is required), indicates the token received from the external system which identifies the user.

AuthenticationToken

addUserInfo() :

The AuthenticationToken represent the user supplied information to decide if the user is allowed access to the application.

When the information (like user and password) is requested by the application itself (by using a login form), the type is a UsernamePasswordToken.

But the type can also describe a token which identifies the user by the external system (in case of the above described external authentication scenario) These tokens implement the interface *ValidatedAuthenticationToken* which is a marker for Octopus that it is a token which doesn't need to be validated. (The external system has performed already a successful validation and assembled the token) It is not the *raw* token which the external party has send to us, but it is already the processed (for example payload of JWT) data and is already validated.

These token can also implement *AuthorizationToken* interface. This is the case when the token also contain authorization information like roles and permissions.

The interface only requires one method, which returns the class name of the logic which will retrieve the authorization info from the token.

Logout

If you want to perform a logout within a Web environment, just call

```
securityContext.logout();
```

on an injected OctopusWebSecurityContext

```
@Inject
private OctopusWebSecurityContext securityContext;
```

or within a JSF environment, you should call

```
loginBean.logout()
```

Since we need to perform a redirect to the logout page (or the main page if no specific page is defined)

An alternative is to define a (virtual) URL which performs the logout. It does the same thing as calling the *loginBean* method. For example define the following entry with the *securedURLs.ini* file.

```
/doLogout = logout
```

When a URL *doLogout* is called, it will logout the subject (of course, the URL can be freely chosen but make sure it is anonymously accessible.)

The following steps are performed during logout.

1. Call all registered *AuthenticationListener*, method onLogout()
2. The default *AuthenticationListener* fires the CDI event LogoutEvent so it becomes easier to react on a logout
3. Remove Principal information from AuthenticationCache and AuthorizationCache
4. Remove Session information (if Session is used to store information about Principal)
5. Remove Principal information and set current Subject as unauthenticated.
6. Redirect to logout Page (if logout sequence started from loginBean)

Events

There are a few CDI event generated depending on the authentication process. These events can be used for your own logic (last login, number of invalid attempts, …)

To get notified when someone is successful logged in, you can define the following method on a CDI bean.

```
public void onSuccess(@Observes LogonEvent logonEvent) {
}
```

LogonEvent

This event is thrown when a user is successful logged in into the application.

1. logonEvent.getInfo() : The *AuthenticationInfo* associated with this login.
2. logonEvent.getAuthenticationToken() : The *AuthenticationToken* used to grant the user access. In case it is a UsernamePasswordToken, the sensitive information (like password and remote host) is already cleared.
3. logonEvent.getUserPrincipal() : The *UserPrincipal* created for the user in response of the the successful authentication.

LogonFailureEvent

This event is thrown when the user is denied access based on the presented credentials (wrong password, expired JWT token, …)

1. logonFailureEvent.getAuthenticationToken() : The *AuthenticationToken* used to grant the user access. In case it is a UsernamePasswordToken, the sensitive information (like password and

remote host) is still present.

2. logonFailureEvent.getException() : The exception thrown because of the denied access.

LogoutEvent

The event is thrown just **before** the user is effectively logged out of the system.

1. logonEvent.getUserPrincipal() : The *UserPrincipal* of the user which is in the process of being logged out.

Logout

There are various levels of authentication and thus also various scenarios related to logout.

Java SE

The logout in a Java SE environment is important when you are using third party authenticators like Keycloak, OAuth2 or OpenId Connect servers and so on.

When we authenticated against them, these remote systems started a session for the logged in user; We the java SE programs exits, it is thus important that we also do a logout with these external systems.

So always perform a logout of the subject when the programs stops.

```
SecurityUtils.getSubject().logout();
```

The registered **RemoteLogoutHandler** performs then the logout at the external system.

Authorization

Specify authorization info

Authorization info will be retrieved by the Octopus framework by calling implementations of *be.atbash.ee.security.octopus.authz.AuthorizationInfoProvider*.

The method *getAuthorizationInfo* needs to supply the authorization info (permissions and roles) for the user.

Interceptors

```
@RequiresPermissions
```

Can be used to protect the execution of an EJB method. User (subject) must have the permission before method is executed.

```
String[] value()
```

Supply the permission(s) wildcard or named permission. See Permission chapter.

```
Class<? extends NamedPermission>[] permission()
```

Supply the permission to check as class instance.

```
Combined combined() default Combined.OR
```

When multiple permissions are supplied, must they all be satisfied or only one (the default)

```
@RequiresRoles
```

Can be used to protect the execution of an EJB method. User (subject) must have the role before method is executed.

```
String[] value()
```

Supply the role name(s). See Permission chapter.

within EJB

Create a ejb-jar.xml with the following content to protect all methods within all EJB beans

```
    <interceptors>
        <interceptor>
            <interceptor-
class>be.atbash.ee.security.octopus.interceptor.OctopusInterceptor</interceptor-class>
        </interceptor>
    </interceptors>
    <assembly-descriptor>
        <interceptor-binding>
            <ejb-name>*</ejb-name>
            <interceptor-
class>be.atbash.ee.security.octopus.interceptor.OctopusInterceptor</interceptor-class>
        </interceptor-binding>
    </assembly-descriptor>
```

The interceptor can also be added manually to the beans by putting the *@OctopusInterceptorBinding* annotation.

within CDI

Activate the CDI interceptor to add interceptor to CDI beans by defining the config parameter

```
cdi.interceptor.enabled=true
```

Define the regex of *ApplicationScoped* CDI beans which needs to have the *OctopusInterceptor* within the file */resources/octopusInterceptor.config* or the file defined by the parameter *cdi.interceptor.configfile*

```
be.atbash.ee.security.octopus.jsf.*
```

Activate the Octopus interceptor within the _beans.xml file

```
    <interceptors>
        <class>be.atbash.ee.security.octopus.interceptor.OctopusInterceptor</class>
    </interceptors>
```

Another option is adding the *@OctopusInterceptorBinding* annotation to those beans which needs to be verified. Beware that the activation of the interceptor within *beans.xml* is also required in this case.

Voters

Voters for a certain permission or role can be created.

```
@Inject
@RequiresPermissions("order:read:*")
private GenericPermissionVoter orderReadVoter;
```

or for a role

```
@Inject
@RequiresRoles("admin")
private GenericRoleVoter adminRoleVoter;
```

Things you can do with a voter

```
voter.checkPermission(AccessDecisionVoterContext, Set<SecurityViolation>);
```

Verifies the permission and add violations to the Set<SecurityViolation> parameter. AccessDecisionVoterContext supplies context

```
voter.verifyPermission();
```

returns true if the current user /subject has the required permission checked by the voter.

These voters can be created programmatically in those environments where no CDI inject is available.

```
GenericPermissionVoter.createInstance(String);
```

or

```
GenericRoleVoter.createInstance(ApplicationRole)
```

Custom voters

When the default checks on permissions or not enough. It can be that more complex logic is required or that multiple checks must be combined.

```
@ApplicationScoped
@Named
public class CustomVoter extends AbstractGenericVoter {
}
```

Typically the injection of voters is performed within these custom voters.

NamedPermissions

Using type safe enums for permissions names can be handy for small to medium sized applications. For large scale or Self-Contained Systems, it is probably no the best way.

The idea is that you specify the name of the permission using an Enum, something like.

```
public enum DemoPermission implements NamedPermission {
    BASIC_PERMISSION, ADVANCED_PERMISSION
}
```

These names (like *BASIC_PERMISSION* can be used within JSF custom tags or the *namedFilter* filter.

For EJB, you have the possibility to create a special annotation which allow you to define the authorization requirements.

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface DemoPermissionCheck {
    DemoPermission[] value();
}
```

In order make this work, you need to provide Octopus with the information of these custom constructs by specifying the following configuration values within the *octopusConfig.properties* file.

```
namedPermission.class=be.atbash.ee.security.octopus.jsf.security.DemoPermission
namedPermissionCheck.class=be.atbash.ee.security.octopus.jsf.security.DemoPermissionCh
eck
```

Now, EJB methods can be secured (authorized) by using

```
    @DemoPermissionCheck(DemoPermission.BASIC_PERMISSION)
    public String doBasicAction() {
        return "Basic Action Method called";
    }
```

Custom tags

Custom tags are created to perform declarative authorization on JSF components. These are defined in the namespace

```
xmlns:sec="http://www.atbash.be/secure/octopus"
```

<sec:securedComponent>

Defines if a certain JSF component can be viewed (is rendered) for the user/subject.

When defined within another JSF tag (without the for attribute) it controles the parent. With the for attribute one can define the JSF component on which it operates.

```
permission
```

Supply the permission(s) wildcard or named permission. See Permission chapter.

```
role
```

Supply the role name(s). See Permission chapter.

```
voter
```

Supply the names of the custom voter(s)

Combination of the 3 above attributes is allowed

```
not
```

Inverts the result of the check

```
combined (true/false)
```

Do all checks need to be pass on the user/subject or is only 1 enough.

```
for
```

Specifies the id of one (or more) JSF components for which the authorization check is performed.

<sec:securedListener>

Defines the possibility to execute a method when the authorization checks ofr te user are positive based on the supplied permission, role and/or voter. The Java method can update the component to allow correct styling based on the permissions of the users.

When defined within another JSF tag (without the for attribute) it controles the parent. With the for attribute one can define the JSF component on which it operates.

```
listener
```

Defines the EL expression of the method which needs to be executed. The EL expression must point to a Java method with a parameter of type UIComponent and has no return (void)

```
permission
```

Supply the permission(s) wildcard or named permission. See Permission chapter.

```
role
```

Supply the role name(s). See Permission chapter.

```
voter
```

Supply the names of the custom voter(s)

Combination of the 3 above attributes is allowed

```
not
```

Inverts the result of the check

```
combined (true/false)
```

Do all checks need to be pass on the user/subject or is only 1 enough.

```
for
```

Specifies the id of one (or more) JSF components for which the authorization check is performed.

<sec:securePage>

This is an alternative for the usage of the filter definition with the securedURLs.ini file. We can specify the authorization checks (using permission, role and voter) in order that the page is visible for the end user. If (s)he has no permission, the unauthorized page will be shown.

This tag can be placed anywhere on the page, but for optimal performance, it should be in the beginning of the page and within the <h:body> parent.

```
permission
```

Supply the permission(s) wildcard or named permission. See Permission chapter.

```
role
```

Supply the role name(s). See Permission chapter.

```
voter
```

Supply the names of the custom voter(s)

Combination of the 3 above attributes is allowed

```
not
```

Inverts the result of the check

```
combined (true/false)
```

Do all checks need to be pass on the user/subject or is only 1 enough.

Filter (web based projects)

anon

Every one can access the page, no checks perfomed

authcBasic

type : Authenticating

Requires BASIC authentication.

user

type : Authenticating

Requires authenticated user (?? TODO real difference between authenticated and remembered) When no authenticated user is available, a redirect to the loginURL is performed.

userRequired

type : Authorization

Requires authenticated user but no redirect to the login is performed but the unauthorized page is shown.

namedPermission / np

type : Authorization

Subject must have all the named permissions defined in the config.

namedPermission1 / np1

type : Authorization

Subject must have one of the named permissions defined in the config.

namedRole / nr

type : Authorization

Subject must have all the named roles defined in the config.

namedRole1 / nr1

type : Authorization

Subject must have one the named roles defined in the config.

rate

type :

Limit the number of requests for a certain path.

noSessionCreation (rest module)

type :

It makes sure that no session will be created by the framework

mpUser (MicroProfile module)

type : Authenticating

Retrieves authentication information from the Bearer header formatted accordingly to the MP JWT Auth spec.

Java SE

Methods can be annotated with authorization checks, like @RequiresPermission, and authorization checks are performed by calling the method

```
    @RequiresPermissions("demo:offline:*")
    public String checkPermission() {
        MethodAuthorizationChecker.checkAuthorization();
```

Since we are running in plain Java SE, we have no interceptors available to perform these checks automatically.

Filters

There are basically 3 types of filters within the system.

Authenticating filters

These filters extract information from the request and determine the principal based on that information. Examples are

1. authcBasic → BASIC authentication
2. mpUser → MicroProfile JWT auth token

3. authcKeycloak → AccessToken of Keycloak

When such a filter is available within the chain and the request doesn't define the required and correct information, an response with status 401 is returned.

All these filters extend from be.atbash.ee.security.octopus.filter.authc.AuthenticatingFilter

User filters

These filters are typically used to determine if the user is authenticated and if not, a redirect is performed to some kind of login form where the user can enter his credentials. This form can be defined within the application (when using LDAP, SPI, ...) or externally when integrating with Keycloak, Google Oauth2, CAS, etc ...)

All these filters extends from be.atbash.ee.security.octopus.authc.AbstractUserFilter which is defined within the JSF module. JSF is for the moment the only supported web framework where the user is able to interact with the application.

The predefined filters defined within Octopus are

1. user → redirect to /login.xhtml page or URL defined within config parameter *loginPage*
2. userKeycloak → redirect to Keycloak Login page (Keycloak integration)

Authorization filters

These filters determine is the user has the required permission, role, the customer voter allows access, ...

They assume that there is already an authenticated user / principal (because an anonymous user can't be assigned some permissions)

When no authenticated (or remembered) user / principal is detected, the response depends on the framework which handles the request and is encapsulated by the implementations of the accessDeniedHandler.

On a JSF request, the default user filter is retrieved from the configuration (parameter *user.filter.default*) and a redirect to the *login page* is performed (after the current request information is saved). This allows filter chain definitions without the need to always specify the user filter name.

```
/pages/hr/** = np[hr:read:*]
```

On a JAX-RS request, a response with status 401 is returned (as we have no possibility the ask for credentials of the current user).

Java EE

JAX-RS

Core

The authorization annotations (like @NamedPermissions which check if the user has a certain permission) can be used on a JAX-RS control.

Normally, these annotation are only picked up by an interceptor when placed on an EJB bean (when interceptor is configured in *ejb-jar.xml*) or an application Scoped CDI bean (when configuration parameter cdi.interceptor.enabled has the value *true*.

These annotations are also picked up by Octopus on JAX-RS controller classes (without the need to define them as EJB or CDI bean) when the configuration parameter rest.interceptor.enabled is set to *true* (default value is *false*).

At that moment, a ContainerRequestFilter enforces the authorization defined by these annotations. And also throws an exception when an JAX-RS endpoint is called without any permission requirements (no annotation on method or class)

With the following example

```
@Path("/hello")
@javax.inject.Singleton
public class HelloController {
```

```
@Inject
private UserPrincipal principal;
```

```
@GET
@RequiresUser
public String sayHello() {
    return "Hello " + principal.getName();
}
```

```
    @Path("/protectedPermission1")
    @RequiresPermissions("demo:read:*")
    @GET
    public String testPermission1() {
        return "Has permission demo:read:*";
    }
}
```

The *sayHello* endpoint just requires a user, the *testPermission1* needs a user which has the permission *demo:read:** . How the user authentication is enforced is not defined by the annotations, but can be done by specifying a filter within the *securedURLs.ini* file (filters like *mpUser*,

authcKeycloak, etc ...)

The *javax.inject.Singleton* enforces that only 1 instance is created for all the requests it will be serving.

MP Auth token

Octopus has support for the MicroProfile JWT auth token, https://github.com/eclipse/microprofile-jwt-auth.

JWT token, compliant with the MP JWT AUth specification, can be used to authenticate and authorize the user within a an Octopus protected application.

For an authenticated user, a JWT token can be created using the information available to Octopus (like user name and permissions)

Setup endpoints

When you want to protect some URLs (endpoints) with the MicroProfile JWT auth token, add following dependency to your projects Maven project file.

```
<dependency>
    <groupId>be.atbash.ee.security</groupId>
    <artifactId>octopus-mp</artifactId>
    <version>0.4</version>
</dependency>
```

This dependency needs to be added to the other dependencies required for using Octopus in a Rest environment (see)

Specify the filter to protect the URLs within the *securedURLs.ni* (or configured name) file.

```
/data/** = noSessionCreation, mpUser
```

Configure the location where the cryptographic keys can be found for the verification of the RSA based signing. Define within *octopusConfig.properties* the parameter and value

```
keys.location=classpath:test.jwks
```

See ??? for more information about reading cryptographic keys.

Define the expected audience for received JWT tokens. Define within *octopusConfig.properties* the parameter and value

```
mp.aud=Octopus Rest MP
```

Setup clients

Octopus contains support for calling MicroProfile JWT auth token protected endpoints.

Add the required maven dependencies

```
<dependency>
    <groupId>be.atbash.ee.security</groupId>
    <artifactId>octopus-mp-rest-client</artifactId>
    <version>0.4</version>
</dependency>
```

```
<dependency>
    <groupId>org.apache.deltaspike.modules</groupId>
    <artifactId>deltaspike-partial-bean-module-impl</artifactId>
    <version>${deltaspike.version}</version>
</dependency>
```

Other dependencies may be required, depending on the other features of Octopus you might want to use. For example the *octopus-js7* dependency to support the JSF view.

Warning

Deltaspike 1.8.1 is required as a minimum due to the usage of for example DeltaSpikeProxyInvocationHandler within the Atbash Rest client.

Define the interface which describes the remote endpoint

```
@Path("/hello")
@ApplicationScoped
@RegisterRestClient
@RegisterProvider(MPRestClientProvider.class)
public interface HelloService {
```

```
@GET
String sayHello();
```

```
}
```

By specifying the *@RegisterRestClient* annotation, a CDI bean is created that can be injected and used to call the remote endpoint. The annotation @RegisterProvider is used to add a specific provider to the generated client so that the MicroProfile JWT auth compatible JWT token is added to the header of the request. The information of the current user (as defined by the UserPrincipal, is used to populate the claims of the JWT.

The JWT needs some values for claims issues, audience, etc ... These can be specified within the configuration file *octopusConfig.properties* (or equivalent when another name is defined)

```
mp.iss=Octopus example
mp.aud=Octopus Rest MP
mp.exp=1m
keys.location=classpath:test.jwks
```

We also need to define where the endpoint is located, this is also defined within the configuration file like this for the above example.

```
be.atbash.ee.security.rest.HelloService/mp-rest/url=http://localhost:8080/rest-mp/data
```

For more information on the format, look at the MicroProfile JWT Auth specification document.

Calling the endpoint becomes then very easy, like this.

```
@ApplicationScoped
public class SomeService {
```

```
@Inject
@RestClient
private HelloService helloService;
```

```
    public void doGreeting() {
       String greeting = helloService.sayHello("Rudy");
    }
}
```

The *@RestClient* is a Qualifier so that the special created CDI bean is used for injection at that point.

JSF

[comment]: # (FIXME keycloak.adi also contain JAX-RS info. Is Auth methods section more appropriate or do we need to define it twice ?)

include::keycloak.adi

Java SE

OfflineToken

Offline token can be used for standalone Java SE programs.

A token can be generated which will be only valid for a certain computer.

Besides the Processor Id and the first disk UUID, also a pass phrase is required (when multiple users are using the program on the same laptop/desktop.)

Steps (example flow, final programs not created yet)

1. Program **LocalSecret** (*examples/local-secret*) generates the token which is user dependent for a certain machine(Standalone program run by the end-user)
2. Program **CreateOfflineTokenFile** (*examples/se-cli*) generates the offline token (here stored within the *<user_home>/octopus.offline.token* file)
3. Program **SecuredCLI** uses the offline token to authenticate/authorize using Octopus.

include::keycloak-se.adi

MP Rest Client

Using the Atbash Rest client for calling an endpoint protected by the JWT auth specification, can also be used from plain Java SE. This means that it is also possible to call some JAX-RS endpoint quite easily from JavaFX.

Setup clients

The same dependencies can be used within the Java SE environment, as follows.

Add the required maven dependencies

```
<dependency>
    <groupId>be.atbash.ee.security</groupId>
    <artifactId>octopus-mp-rest-client</artifactId>
    <version>0.4</version>
</dependency>
```

```
<dependency>
    <groupId>org.apache.deltaspike.modules</groupId>
    <artifactId>deltaspike-partial-bean-module-impl</artifactId>
    <version>${deltaspike.version}</version>
</dependency>
```

Warning

Deltaspike 1.8.1 is required as a minimum due to the usage of for example DeltaSpikeProxyInvocationHandler within the Atbash Rest client.

Define the interface which describes the remote endpoint

```
@Path("/hello")
public interface HelloService {
```

```
@GET
String sayHello();
```

```
}
```

No additional annotations are required since we are not using the cdi integration features.

Configuration wise, we need to define the values of the claims (like issuer and audience) of the JWT and the cryptographic key to use. The following keys are thus required within the configuration file.

```
mp.iss=Octopus example
mp.aud=Octopus Rest MP
mp.exp=1m
keys.location=classpath:test.jwks
```

Calling the endpoint can be performed then using the folliwing code

```
HelloService helloService = AbstractRestClientBuilder.newBuilder()
        .baseUrl(new URL("http://localhost:8080/rest-mp/data"))
        .register(MPRestClientProvider.class)
        .build(HelloService.class);
System.out.println(helloService.sayHello());
```

We also register the *MPRestClientProvider* provider. It looks at the current user from Octopus Security Context and creates the required JWT to add into the authorization header.

Manual creation of JWT

In the case you don't ike to use the Atbash Rest client or just needs a JWT which is compliant with the MicroProfile JWT Auth specification, you can use the **be.atbash.ee.security.octopus.token.MPJWTTokenBuilder** class.

Authentication methods

General information

The user, or in general the other part as it can also be another process, will be represented by an AuthenticationToken. These token can represent a user name and password or a token from an external authentication mechanism like Keycloak or OAuth2.

For these tokens, Octopus requires an instance of AuthenticationInfo in order to verify the user (credentials) and have information about the user (like name).

See [Authentication ways](#Authentication-ways) for more information about creating an AuthenticationInfo instance.

SPI

With the SPI, you are can integrate any kind of storage for the authentication methods. In fact, all methods are implemented through this SPI and thus it is the fundamental integration point of Octopus with the retrieval of AuthenticationInfo data.

Just return the AuthenticationInfo instance corresponding to the AuthenticationToken.

Hashed password

Using *Hashed password* is a specific version of the SPI where you supply the expected password in a hash way and give all the information related to the hash (like salt, algorithm name, iterations, etc ..)

The following snippet is the standard way how you can define the logic for using a hashed password.

```
            AuthenticationInfoBuilder authenticationInfoBuilder = new
AuthenticationInfoBuilder();
            authenticationInfoBuilder.principalId(theId).name(theName);
            authenticationInfoBuilder.salt(salt);
            authenticationInfoBuilder.password(hashedPassword);
            return authenticationInfoBuilder.build();
```

- *theId*: A required Serializable value which indicates the unique identification of the user/remote process.
- *theName*: An optional description of the user which can be used to display on the screen
- *salt*: The salt which is used for generating the hash, see below
- *hashedPassword*: The String representation (in Hex or Base64, see below) of the hashed password.

The *salt* is required to pass to the *AuthenticationInfoBuilder* as this will be the trigger to use the hashing logic of Octopus. Although that it is possible to generate hashes without any salt it is bot allowed within octopus because it is less safe.

The parameter of the *salt* method is Object. The conversion to a *ByteSource* is performed by the *ByteSourceCreator*.

Other variables are defined by the configuration

- *hashAlgorithmName* : The hash algorithm name (like sha-256) or Key derivation function (like PBKDF2)
- *hashEncoding* : The String encoding of the hashedPassword, HEX or BASE64
- *hashIterations* : Number of iterations in the hashing.

Hash utilities

The class be.atbash.ee.security.octopus.crypto.hash.SaltHashingUtil contains some utility methods related to hashing. It is a CDI bean or can be retrieved in any environment using SaltHashingUtil.getInstance().

When you need to store data for a new user, you need to have a salt and store the password in his hashed format.

A convenient method for generating a new salt if provided by the nextSalt() method.

```
    byte[] salt = saltHashingUtil.nextSalt();
```

The length of this value is determined by the configuration parameter *saltLength*. And longer salts are safer then short salt values.

The hash is calculated by the hash method:

```
    String hashedPassword = saltHashingUtil.hash(password, salt);
```

Configuration

Octopus Core Configuration

hashAlgorithmName

default : **(none)**

Name of the MessageDigest algorithm when you use hashed passwords. examples are Md5 and Sha512.

saltLength

default : **0**

Number of bytes used when creating a salt for the hashing of passwords. 0 means that no salt is

used.

hashEncoding

default : **HEX**

Defines how the hashed passwords are encoded (HEX or BASE64) before they are compared to the supplied value which should be identically before access is granted. The value specified in the configuration file is case insensitive compared with the allowed values.

hashIterations

default : **1** or **1024**

Defines the number of iterations that are performed within the hashing algorithm.

The default value 1 is taken for the **'real'** hashing functions like SHA-256, 1024 is for the Key deriviation functions like PBKDF2.

cacheManager.class

default : **be.atbash.ee.security.octopus.cache.MemoryConstrainedCacheManager**

The class responsible for holding/managing the cache of the authentication and authorization data. The developer can supply a custom implementation of be.atbash.ee.security.octopus.cache.AbstractCacheManager when the cache needs different logic.

When the class has the javax.enterprise.context.ApplicationScoped annotation, it is instantiated as a CDI bean, otherwise a classic new is performed.

voter.suffix.permission

default : **PermissionVoter**

The suffix used to determine the CDI named bean which are created dynamically for each Named Permission. See VoterNameFactory.

voter.suffix.role

default : **RoleVoter**

The suffix used to determine the CDI named bean which are created dynamically for each Named Role. See VoterNameFactory.

voter.suffix.check

default : **AccessDecisionVoter**

The suffix used to determine the CDI named bean for the Custom check functionality. See VoterNameFactory and Custom check feature description.

authorization.dynamic

default : false

???

namedPermission.class

default : **(none)**

Defines the Enum class which enumerates all permissions. Within the demo example it is the class **be.c4j.demo.security.permission.DemoPermission**.

namedPermissionCheck.class

default : **(none)**

Defines the annotation which can be used on method and class level to define the security requirements.

customCheck.class

default : **(none)**

Defines the annotation class which can be used to use custom declared Permissions, mostly useful in the case where you want to extend the named permission with some additional information.

namedRole.class

default : **(none)**

Defines the Enum class which enumerates all named roles. It is the role counterpart of the namedPermission.class configuration option.

namedRoleCheck.class

default : **(none)**

Defines the annotations which can be used on method and class level to define the security requirements.

cdi.interceptor.enabled

default : **false**

If set active, CDI beans will also receive the Octopus interceptor and thus permissions checks will be performed for all 'external' called methods. The fully qualified name of the intercepted beans will be matched with the name patterns defined within the file referenced by the *cdi.interceptor.configfile* parameter.

cdi.interceptor.configfile

default : **classpath:octopusInterceptor.config**

File containing the patterns of the fully qualified class names of CDI beans which will be intercepted by the Octopus security interceptor.

Web Configuration

???

JSF Configuration

user.filter.default

default : **user**

When authorization filter encounters a non authenticated user, this filter is used to perform the redirect to the login page. The filter name point to a filter instance which implements AbstractUserFilter.

loginPage

default : **/login.xhtml**

The JSF page which is shown when the system needs to ask for the user credentials (and no 3th party is defined for integration)

logoutPage

default : /

The page which is shown when the is logged out. Make sure the page is anonymously accessible. By default, it is the page defined as welcome-file in the web.xml

allowPostAsSavedRequest

default : **true**

Is it allowed that during a POST to the server, the login page is shown. After the redirect to the login page, it is possible that beans has lost their state and that post isn't functioning properly.

logoutFilter.postOnly

default : **false**

When using the Logout filter, is it only active for a POST request (to avoid issues with the browser prefetch)

single.logout

default : **false**

When the user has used some authentication mechanism which supports SSO (like Keycloak, OAuth2, …) should a logout from the application mean also a logout from the SSO?

unauthorizedExceptionPage

default : **/unauthorized.xhtml**

The page which is shown when the user has some missing permissions.

primefaces.mobile.exclusion

default : **false**

Exclude the wrapping of the PrimeFaces mobile renderers (for compatibility reasons, will be removed in some future version)

session.hijacking.level (JSF Only)

default : **ON**

Determines the Session Hijack Protection level. It uses the IP Address and User-Agent header information and checks if the sessionId could be 'stolen'.

The default level *ON*, checks both properties, *PARTIAL* only the User-Agent header value and *OFF* disables the protection.

Keycloak configuration

keycloak.file

default : **/keycloak.json**

Defines the location of the JSON file for configuration of the Octopus Keycloak integration.

Octopus MicroProfile JWT Configuration (Core)

mp.iss

default : **none**

The issues claim set within the JWT created by the *MPJWTTokenBuilder* by default. When *MPRestClientProvider* is used in combination with the Rest client, this value is required.

mp.aud

default : **none**

The audience claim set within the JWT created by the *MPJWTTokenBuilder* by default. When *MPRestClientProvider* is used in combination with the Rest client, this value is required. And the expected audience value when a MP JWT token is validated.

mp.exp

default : **none**

The expiration setting used in the exp claim set within the JWT created by the *MPJWTTokenBuilder*. When *MPRestClientProvider* is used in combination with the Rest client, this value is required. The value has following format

<v><unit>

- v : A positive number
- unit : s (seconds), m (minutes) or h (hours)

Examples

- 5s → JWT will have an expiration time of 5 seconds (so current time + 5 seconds)
- 20m → JWT will have an expiration time of 20 minutes (so current time + 20 minutes)
- 1h → JWT will have an expiration time of 1 hour (so current time + 1 hour)

Octopus MicroProfile JWT Configuration (Rest Client)

mp.key.id

default : **none**

The key id (looked up from the cryptographic keys read by the Key support) which will be used for the signing of the JWT. We no key Id is specified, a key which has a private part is chosen (but there can only be one)

Octopus Rest client providers

octopus.rest.client.providers

default : **none**

List of fully qualified class names which will be added as provider to the generated Rest Clients. This is the equivalent of specifying them with the @RegisterProvider annotation (but now for all clients and not individual specified). An alias is defined for a few Providers to make it easier to specify them

- *mp-authc* → Adds authentication and authorization information known within Octopus context to the header as MP JWT Auth token
- *keycloak-authc* → When available, adds the keycloak token as header.

OWASP

Session fixation

A possible way to fight against a session fixation attack is that the session is changed during the authentication of the end user.

Therefore, by default, the HTTPSession which is active during login (When using username - password, OAuth2, KeyCloak, CAS, …) is invalidated and a new one is created.

Because vital information is placed on the session for the correct functioning of Octopus, all session attributes are copied to the new session.

One can disable this invalidation and recreation of the session by means of the parameter session.invalidate.login

Session Hijacking

With a Session Hijack attack, someone got hold of your sessionId and use it to access the application with your credentials.

A protection is built-in Octopus and compares the IP address and the User agent information. If it detects a request with a different IP address and User-Agent then a previous request with the same SessionId, it blocks it.

The request where the anomaly is detected receives a response with status 401, a marker is placed on the other session for which there was a hijack attempt. This can be checked by the EL expression #{octopusUserInfoBean.sessionHijackDetected}

With the configuration parameter, session.hijacking.level we can control the level of the protection.

- *ON* (default value), the IP address and User-Agent header value must match for all requests with the same sessionId
- *PARTIAL*, only the User-Agent header value must match.
- *OFF*, no protection, only recommended when your application does not contain personal information or uses no permissions.

The *PARTIAL* value is required if your end users switch for example from a wired to a wireless internet connection (used in some companies) and the IP address is different on both systems. Or you have a mobile JSF application (with PrimeFaces mobile for example) with end user connection can change (between WIFI and 3G for example)

Single session

By default, a user can only be logged in once into the application. If the same user, this is determined by the principalId of the AuthenticationInfo see ???, is already logged in, the other session is invalidated (and thus a logout is performed) automatically.

This behaviour can be controlled by the configuration parameter session.single.

Catch OctopusUnexpectedException

Various parts of the code throws the OctopusUnexpectedException when some unexpected condition happens. Make sure you catch at a minimum these Exception (by means of an ExcetionHandler in JSF for example) so that no stacktrace is exposed to the client.

It is a good idea to route any exception to a custom page/JAX-RS endpoint response so that internal application information isn't exposed.

OWASP

Additional information can be found here

https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

Custom

Custom Filters

Only filters which are descendant of be.atbash.ee.security.octopus.filter.AdviceFilter can be by Octopus (like from within the *securedURLs.ini* file).

When you need to create a custom filter (first check the documentation about filters to see if the filter you need isn't available) you can start from some base classes. These classes are also used in the standard octopus filters.

Filters are a Web only concept, and thus are always dependent on the module *octopus-common-web*.

Custom user Filter

A user filter is typical a filter for authentication where the end user has the possibility to provide credentials when no authenticated user is detected.

Various filters are already available within octopus (KeyCloak, OAuth2, etc ...) but as a developer, you can create your custom version by extending be.atbash.ee.security.octopus.authc.AbstractUserFilter.

This Abstract user filter can be found within the *octopus_jsf7* module.

Custom authentication filter

The authentication filters who define the current user based on the information available on the Request (like an authorization header with a JWT) are available for various systems (like MicroProfile JWT, Keycloak, …)

This filters try ro authenticate the user based on the request info and in when this is not valid, a response with the appropriate status is returned.

These are mostly used for JAX-RS endpoints and when you need your custom filter you should start from the be.atbash.ee.security.octopus.filter.RestAuthenticatingFilter. (located in the *octopus-rest* module).

In some rare situation where you don't want the status 401 when the request can't be authenticated, you can start from the be.atbash.ee.security.octopus.filter.authc.AuthenticatingFilter which is located in the *octopus-common-web* module.

Authorization filter

An authorization filter checks if the user has specific permissions before access is granted to the URL. There are already various filters defined by default in octopus and with the CustomVoterFilter, almost any kind of authorization filter can be created by the deeloper.

In case you need another custom authorization filter, extend from the be.atbash.ee.security.octopus.filter.authz.AuthorizationFilter class.

The base class for this type of filters is defined within the *octopus-common-web* module as it can be used in JSF and JAX-RS environments.