

# Octopus Framework

Rudy De Busscher

Version 0.4, ??/??/2018

# Table of Contents

Release notes .....	1
0.4 .....	1
0.3 .....	1
0.2 .....	1
0.1 .....	1
Modules .....	1
Authentication .....	2
Authentication ways .....	2
AuthenticationInfoProvider .....	3
AuthenticationInfoBuilder .....	3
AuthenticationToken .....	4
Authorization .....	4
Specify authorization info .....	4
Interceptors .....	4
Voters .....	6
Custom voters .....	7
NamedPermissions .....	7
Custom tags .....	8
Filter (web based projects) .....	11
Java SE .....	12
Java EE .....	12
JAX-RS .....	12
Java SE .....	13
OfflineToken .....	13
Configuration .....	13
Core .....	13
Keycloak configuration .....	15

# Release notes

## 0.4

1. Integration with Keycloak (Client Credentials for Java SE, AuthorizationCode grant for Web, AccessToken for JAX-RS)

## 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

# Modules

List of Maven modules

Artefact	SE, CDI, EE	info
be.atbash.json:octopus-json-smart	SE	Very light-weight JSON reader/writer (based on json-smart). Can be externalized if needed.
be.atbash.ee.security:octopus-jwt-support	SE, CDI	Java Beans to/from JSON/JWS/JWE. Can be externalized if needed. octopus-keys can be extracted from it.
be.atbash.ee.security:octopus-core	SE, CDI	All Octopus classes usable in Java SE and Java EE environment.

Artefact	SE, CDI, EE	info
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-common-web	EE (Web)	All Octopus classes Specific for Java EE (Web - Servlets)
be.atbash.ee.security:octopus-rest	EE (JAX-RS)	Specific for JAX-RS
be.atbash.ee.security:octopus-jsf7	EE (JSF)	Specific for JSF
be.atbash.ee.security:octopus-mp	EE (JAX-RS)	Support for MP JWT Auth tokens

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, there 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
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 Octopus 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.

## 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.DemoPermissionCheck
```

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 controls 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 of the 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 controls 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 performed

### **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.

# **Java EE**

## **JAX-RS**

### **Core**

FIXME

## MP Auth token

Creation of the token can be done using the `be.atbash.ee.security.octopus.token.MPJWTTokenBuilder` class.

Maven artefact `be.atbash.ee.security:octopus-mp` contains the *mpUser* filter.

# 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

# Configuration

## Core

### 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.

## 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.

## **Keycloak configuration**

### **keycloak.file**

default : **/keycloak.json**

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