Authentication - create a new authentication method

This tutorial shows, how to add a new authentication method to CzechIdM. The authentication will be typically done by some external authority, such as OpenAM, OAuth, Facebook, Google etc.

Supported authentication methods are implemented in several Authenticators. So you need to create a new authenticator. After you install the authenticator, users will be allowed to authenticate to CzechIdM (using CzechIdM login page) by the new authentication method.

Something different holds for SSO. If you want the users to come to CzechIdM and be immediately logged in without the need to fill in any credentials (or be redirected to some other login page of e.g. Facebook), you need to implement a new IdmAuthenticationFilter (see SSO).

A combination of both situations is possible, e.g. the OpenAM module supports both SSO (if the user already has OpenAM token) and authentication against OpenAM through CzechIdM login page.

Create a new authenticator

Step 1 - Choose the module

Your authenticator must belong to some backend module. You can create a new module, or choose an existing one. The authenticator will be called during authentication process only when the module is enabled.

If you decide to create a new module, please follow the tutorial.

Step 2 - Create the Authenticator class

Now you create a new Spring component, which implements the Authenticator interface. You should extend AbstractAuthenticator which has already some common methods implemented.

After you create the new class, you should:

- Add the `@Component` annotation and specify a unique name for the component.
- Add the annotation `@Enabled` and specify the module descriptor of the chosen module.
- Implement the interface method `getModule()` by returning the module name.
- Implement the interface method `getName()` by returning some name for your authenticator (this name will be seen in logs during the authentication process).
- Implement the interface method `getOrder()`. First decide, whether your authenticator should be called before, or after the core authenticator (i.e. CzechIdM local authentication). If you want your authenticator to be called before, return something like `DEFAULT\_AUTHENTICATOR\_ORDER - 10`.
- Implement the interface method `getExceptedResult()`. Depending on the type of your environment, you should decide whether the new authentication method must be successful

always, or only for some users. For supported types see Authenticator's result type.

After this, your class would look similar to the following code (the example is taken from the OpenAM module):

```
package eu.bcvsolutions.idm.openam.authentication.impl;

// ... imports

@Component("openAMAuthenticator")
@Enabled(OpenAMModuleDescriptor.MODULE_ID)
@Description("OpenAM authenticator, which authenticates users against
OpenAM.")
public class OpenAMAuthenticator extends AbstractAuthenticator implements
Authenticator {

    private static final org.slf4j.Logger LOG =
org.slf4j.LoggerFactory.getLogger(OpenAMAuthenticator.class);

    private static final String AUTHENTICATOR_NAME = "openam-authenticator";

    @Override
    public String getName() {
        return OpenAMAuthenticator.AUTHENTICATOR_NAME;
    }

    @Override
    public String getModule() {
        return EntityUtils.getModule(this.getClass());
    }

    @Override
    public int getOrder() {
        return DEFAULT_AUTHENTICATOR_ORDER - 10;
    }

    @Override
    public LoginDto authenticate(LoginDto loginDto) {
        // TODO add implementation
        return null;
    }

    @Override
    public AuthenticationResponseEnum getExceptedResult() {
        return AuthenticationResponseEnum.SUFFICIENT;
    }
}
```

Step 3 - Implement the authentication method

Now you must add the main thing - the implementation of the method `public LoginDto authenticate(LoginDto loginDto)`.

Fix Me! Since 10.7, the method validate should be also implemented.

The LoginDto input parameter contains username and password filled by the user. Note that the password is of the type GuardedString. When you want to get the real string value of the password, use the method `asString()`, but be sure you don't write this value anywhere in plaintext (particularly not in the log)!

If the credentials are not correct, the method `authenticate` should return NULL. If some other authentication error occurs (an error which should be logged), you should throw an exception.

If you successfully validate user name and password with your authentication method, you will need to find IdM identity and create JWT token, which will be used for following requests of the user. For getting the user by username, use the IdmIdentityService. For creating the JWT token and setting it into the output `LoginDto`, use the JwtAuthenticationService. Those services should be autowired by Spring.

The example implementation of autowiring the services and implementing the `authenticate` method follows. Note that it's recommended to implement your authentication method in a separate service (here `OpenAMAuthenticationService`) to keep the code clean and concise.

```
    private final IdmIdentityService identityService;

    private final JwtAuthenticationService jwtAuthenticationService;

    private final OpenAMAuthenticationService openAMAuthenticationService;

    @Autowired
    public OpenAMAuthenticator(IdmIdentityService identityService,
            JwtAuthenticationService jwtAuthenticationService,
            OpenAMAuthenticationService openAMAuthenticationService) {
        super();

        Assert.notNull(identityService);
        Assert.notNull(jwtAuthenticationService);
        Assert.notNull(openAMAuthenticationService);
        //
        this.identityService = identityService;
        this.jwtAuthenticationService = jwtAuthenticationService;
        this.openAMAuthenticationService = openAMAuthenticationService;
    }

    // ... other implementation

    @Override
    public LoginDto authenticate(LoginDto loginDto) {
        String username = loginDto.getUsername();
```

```
        GuardedString password = loginDto.getPassword();

        if (username == null || password == null) {
            LOG.warn("Could not authenticate against OpenAM, username and
password must be set");
            return null;
        }

        String tokenId =
openAMAuthenticationService.loginUserAndGetToken(username, password);

        if (tokenId == null) {
            LOG.info("User [{}] couldn't be authenticated against OpenAM.",
username);
            return null;
        }

        LOG.info("Identity with username [{}] was authenticated by OpenAM
and got token [{}]", username, tokenId);

        // ... other implementation

        IdmIdentityDto identity = identityService.getByUsername(username);

        if (identity == null) {
            throw new IdmAuthenticationException(MessageFormat.format(
                    "Check identity can login: The identity [{0}] either
doesn't exist or is deleted.", username));
        }

        return
jwtAuthenticationService.createJwtAuthenticationAndAuthenticate(
                loginDto, identity, OpenAMModuleDescriptor.MODULE_ID);
    }
```

Step 4 - Implement the tests

You should implement unit tests that would cover your new authenticator. You should have tests for your service, which implements the authentication logic. For the authenticator tests, you can mock the outputs of the services and test only the logic contained in the authenticator.

Example:

```
package eu.bcvsolutions.idm.openam.authentication.impl;

// ... imports

public class OpenAMAuthenticatorTest extends AbstractUnitTest {

    @Mock
```

```
    private IdmIdentityService identityService;

    @Mock
    private JwtAuthenticationService jwtAuthenticationService;

    @Mock
    private OpenAMAuthenticationService openAMAuthenticationService;

    private OpenAMAuthenticator openAMAuthenticator;

    @Before
    public void init() {
        openAMAuthenticator = new OpenAMAuthenticator(identityService,
jwtAuthenticationService,
                openAMAuthenticationService);
    }

    @Test
    public void testAuthenticateSuccess() {

        LoginDto loginDto = OpenAMTestUtil.createLoginDto();
        IdmIdentityDto idmIdentityDto = OpenAMTestUtil.createIdentityDto();

when(openAMAuthenticationService.loginUserAndGetToken(loginDto.getUsername()
, loginDto.getPassword())).thenReturn(TEST_TOKEN_ID);

when(identityService.getByUsername(TEST_USERNAME)).thenReturn(idmIdentityDto
);

        LoginDto responseLoginDto = new LoginDto(loginDto);
responseLoginDto.setAuthenticationModule(OpenAMModuleDescriptor.MODULE_ID);
when(jwtAuthenticationService.createJwtAuthenticationAndAuthenticate(eq(logi
nDto), eq(idmIdentityDto),
eq(OpenAMModuleDescriptor.MODULE_ID))).thenReturn(responseLoginDto);

        LoginDto resultDto = openAMAuthenticator.authenticate(loginDto);

verify(openAMAuthenticationService).loginUserAndGetToken(loginDto.getUsernam
e(), loginDto.getPassword());

        verify(identityService).getByUsername(TEST_USERNAME);

        Assert.assertNotNull(resultDto);
        Assert.assertEquals(OpenAMModuleDescriptor.MODULE_ID,
resultDto.getAuthenticationModule());
        Assert.assertEquals(TEST_USERNAME, resultDto.getUsername());

    }

    @Test
    public void testAuthenticateInvalidPassword() {
```

```
        LoginDto loginDto = OpenAMTestUtil.createLoginDto();

when(openAMAuthenticationService.loginUserAndGetToken(loginDto.getUsername()
, loginDto.getPassword())).thenReturn(null);

        LoginDto resultDto = openAMAuthenticator.authenticate(loginDto);

verify(openAMAuthenticationService).loginUserAndGetToken(loginDto.getUsernam
e(), loginDto.getPassword());

        Assert.assertNull(resultDto);
    }

    @Test
    public void testAuthenticateMissingIdentity() {

        LoginDto loginDto = OpenAMTestUtil.createLoginDto();

when(openAMAuthenticationService.loginUserAndGetToken(loginDto.getUsername()
, loginDto.getPassword())).thenReturn(TEST_TOKEN_ID);

        when(identityService.getByUsername(TEST_USERNAME)).thenReturn(null);

        Exception ex = null;
        try {
            openAMAuthenticator.authenticate(loginDto);
        } catch (IdmAuthenticationException e) {
            ex = e;
        }
        // Exception was returned because of non-existing identity
        Assert.assertNotNull(ex);

verify(openAMAuthenticationService).loginUserAndGetToken(loginDto.getUsernam
e(), loginDto.getPassword());
        verify(identityService).getByUsername(TEST_USERNAME);

    }
}
```

Step 5 - Build and install

Finally, build the module and install it to CzechIdM. Make sure your module is enabled in the Configuration.

Congratulations, your new authenticator is ready to use!

Create a new authentication filter for SSO

Step 1 - Choose the module

Your authentication filter must belong to some backend module. You can create a new module, or choose an existing one. The filter will be called during authentication process only when the module is enabled.

If you decide to create a new module, please follow the tutorial.

Step 2 - Create the IdmAuthenticationFilter class

Now you create a new Spring component, which implements the IdmAuthenticationFilter interface.

After you create the new class, you should:

- Add the `@Component` annotation and specify a unique name for the component.
- Add the annotation `@Enabled` and specify the module descriptor of the chosen module.
- Add the annotation `@Order`. First decide, if your filter should be called before, or after the core JwtIdmAuthenticationFilter (i.e. CzechIdM local authentication). If the filter should do only SSO and shouldn't authorize every request, its order should be a positive number.

After this, your class would look similar to the following code (the example is taken from the OpenAM module):

```
package eu.bcvsolutions.idm.openam.authentication.filter;

// ... imports

@Order(10)
@Component("openAMIdmAuthenticationFilter")
@Enabled(OpenAMModuleDescriptor.MODULE_ID)
public class OpenAMIdmAuthenticationFilter implements
IdmAuthenticationFilter {

    private static final Logger LOG =
LoggerFactory.getLogger(OpenAMIdmAuthenticationFilter.class);

    @Override
    public boolean authorize(String token, HttpServletRequest req,
HttpServletResponse res) {
        // TODO add implementation
        return false;
    }
}
```

Step 4 - Specify headers and implement the authentication method

The filter will be called only when the HTTP request contains specified headers. You must know which header contains tokens for your authentication method and return its name in the overriden method `getAuthorizationHeaderName()`. For example, BasicIdmAuthenticationFilter uses the header "Basic" to hold user name and login in Base64. In the example, the token is in request cookies, so we specify the header name "Cookie".

You can also override the method `getAuthorizationHeaderPrefix()` to specify the prefix of the header value.

Now you must add the main thing - the implementation of the method `public boolean authorize(String token, HttpServletRequest req, HttpServletResponse res)`.

The input variable `token` is the value (without prefix), which was contained in the specified header. Note that for cookies, the value is concatenated from all request cookies, so you should rather use the whole HttpServletRequest to obtain the cookie you are interested in.

If the authentication is not successful, the method `authorize` should return false. Don't throw any exception even if any other authentication error occurs, because it would break the authentication filter chain completely.

If you successfully validate the user with your authentication method, you will need to find IdM identity and create JWT token, which will be used for following requests of the user. For getting the user, obtain their username from the external authority, and use the IdmIdentityService. Then create LoginDto with this user name. For creating the JWT token and setting it into the output `LoginDto`, use the JwtAuthenticationService. Those services should be autowired by Spring.

The example implementation of autowiring the services and implementing the interface methods follows. Note that it's recommended to implement your authentication method in a separate service (here `OpenAMTokenValidationService`) to keep the code clean and concise.

```
    @Autowired
    private OpenAMTokenValidationService openAMTokenValidationService;

    @Autowired
    private IdmIdentityService identityService;

    @Autowired
    private JwtAuthenticationService jwtAuthenticationService;

    @Override
    public boolean authorize(String token, HttpServletRequest req,
HttpServletResponse res) {
        try {
            String tokenId =
openAMTokenValidationService.retrieveTokenId(req);

            if (tokenId == null) {
                LOG.debug("No cookie for OpenAM");
                return false;
```

```
            }

            String userName =
openAMTokenValidationService.retrieveUserNameForToken(tokenId, true);

            if (userName == null) {
                // Remove invalid cookie so next requests won't need to try
validation again
                openAMTokenValidationService.removeTokenFromHeaders(res);

                LOG.info("TokenId for OpenAM is no longer valid, removing
invalid token from headers. User must authenticate first.");
                return false;
            }

            LOG.info("TokenId for OpenAM is valid for user [{}].",
userName);

            IdmIdentityDto identity =
identityService.getByUsername(userName);

            if (identity == null) {
                throw new IdmAuthenticationException(MessageFormat.format(
                        "Check identity can login: The identity "
                        + "[{0}] either doesn't exist or is deleted.",
                        userName));
            }

            LoginDto loginDto = createLoginDto(userName);

            LoginDto fullLoginDto =
jwtAuthenticationService.createJwtAuthenticationAndAuthenticate(loginDto,
                    identity, OpenAMModuleDescriptor.MODULE_ID);

            return fullLoginDto != null;

        } catch (IdmAuthenticationException e) {
            LOG.warn("Authentication exception raised during OpenAM
authentication: [{}].", e.getMessage());
        } catch (Exception e) {
            LOG.error("Exception was raised during OpenAM authentication:
[{}].", e.getMessage(), e);
        }

        return false;
    }

    private LoginDto createLoginDto(String userName) {
        LoginDto ldto = new LoginDto();
        ldto.setUsername(userName);
        return ldto;
```

```
    }

    @Override
    public String getAuthorizationHeaderName() {
        return "Cookie";
    }

    @Override
    public String getAuthorizationHeaderPrefix() {
        return "";
    }
```

Step 4 - Implement the tests

You should implement unit tests that would cover your new filter. You should have tests for your service, which implements the authentication logic. For the filter tests, you can mock the outputs of the services and test only the logic contained in the filter.

Step 5 - Build and install

Finally, build the module and install it to CzechIdM. Make sure your module is enabled in the Configuration.