Custom authentication

Overview

This tutorial describes how to create a custom *authenticator* and a *login module*, and define a *realm*. More specifically, you learn:

- How to implement a custom authenticator that collects the username and password by using a request to a predefined URL.
- How to implement a custom login module that checks credentials that are received from the authenticator.
- How to define a realm that uses your custom authenticator and login module.
- How to use this realm to protect resources.

This tutorial covers the following topics:

- Introduction
- Configuring the authenticationConfig.xml file
- Creating a custom Java authenticator
- WorklightProtocolAuthenticator
- Creating a custom Java login module
- Protecting a JavaScript adapter
- Protecting a Java adapter
- Creating the client-side authentication components

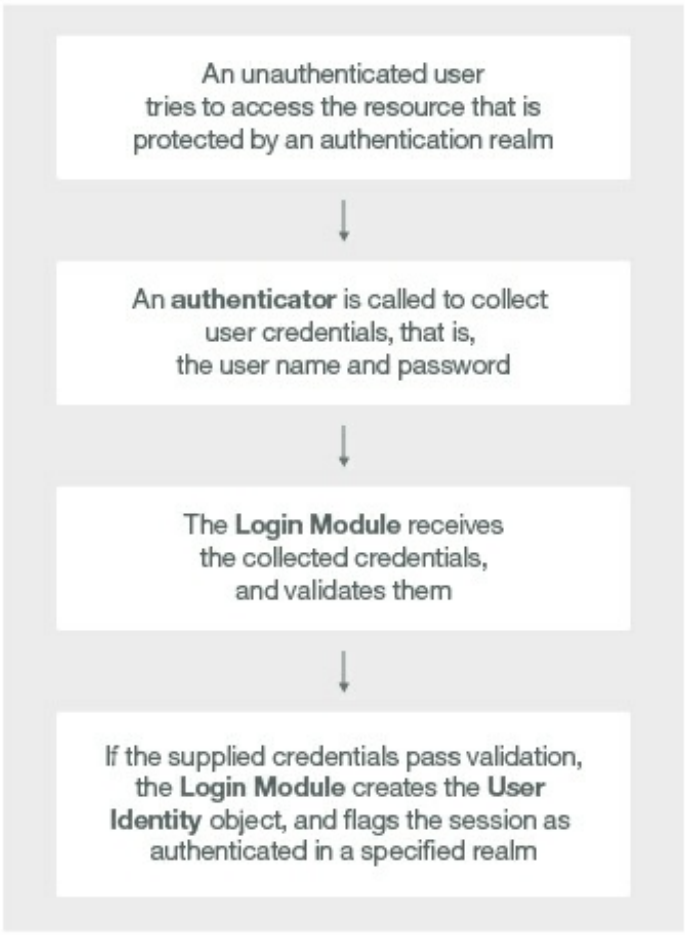
Introduction

The authentication process can be interactive (for example, based on a username and password), or non-interactive (for example, header-based authentication).
This process can involve a single step (a simple user name/password form) or multiple steps (it might have to add a challenge after it issued the first password).

An authentication realm includes the class name of an authenticator and a reference to a login module.

- An authenticator is an entity that collects user information, such as a login form.
- A login module is a server entity that validates the retrieved user credentials and builds the user identity.

Authentication settings such as realms, authenticators, and login modules are configured in the `authenticationConfig.xml` file that is generated for every MobileFirst project.



The authenticator, login module, and user identity instances are stored in a session scope. Therefore they exist as long as the session is alive.

You can write custom login modules and authenticators when the ones that are provided by default do not match your requirements.
In previous tutorials, form-based authentication was implemented in the form of a non-validating login module. Adapter-based authentication was also implemented without the developer having to add login modules, and credentials were valided manually.

In some cases, though, when credentials validation cannot be run at the adapter level and requires more complex code, you can implement an extra login module.
For example: When credentials validation must be customized for a specific enterprise, or when more information must be retrieved from each client request, such as `cookie`, `header`, or `user-agent`.

Configuring the authenticationConfig.xml file

1. In the `realms` section of the `authenticationConfig.xml` file, define a realm called `CustomAuthenticatorRealm`.
2. Make sure that it uses `CustomLoginModule`.
3. Specify `MyCustomAuthenticator` as the name of the class, which you implement later.

```
<realm name="CustomAuthenticatorRealm" loginModule="CustomLoginModule">
  <className>com.mypackage.MyCustomAuthenticator</className>
</realm>
```

4. In the `loginModules` section, add a `loginModule` instance called `CustomLoginModule`.

5. Specify `MyCustomLoginModule` as the class name, which you implement later.

```
<loginModule name="CustomLoginModule">
   <className>com.mypackage.MyCustomLoginModule</className
>
</loginModule>
```

6. In the `securityTests` section, add a security test.
   Later, you use this security test to protect a JavaScript adapter. Therefore, use a `customSecurityTest` element and remember the security test name.
   Note: If you use Java adapters, this step is not required.

```
<customSecurityTest name="AuthAdapter-securityTest">
   <test isInternalUserID="true" realm="CustomAuthenticatorRealm" /
>
</customSecurityTest>
```

Creating a custom Java authenticator

API overview

The `WorkLightAuthenticator` API includes the following methods:

- The `init` method is called when the authenticator instance is created. It takes the parameters that are specified in the definition of the realm in the `authenticationConfig.xml` file.

  ```
  void init(Map<String, String> options)
  ```

- The `processRequest` method is called for each request from an unauthenticated session.

  ```
  AuthenticationResult processRequest(HttpServletRequest request, HttpServletResponse respon
  se, boolean isAccessToProtectedResource)<
  ```

- The `processAuthenticationFailure` method is called if the login module returns a failure of credentials validation.

  ```
  AuthenticationResult processAuthenticationFailure(HttpServletRequest request, HttpServletRes
  ponse response, String errorMessage)
  ```

- The `processRequestAlreadyAuthenticated` method is called for each request from an already authenticated session.

```
AuthenticationResult processRequestAlreadyAuthenticated(HttpServletRequest request, HttpS
ervletResponse response)
```

- The `getAuthenticationData` method is used by a login module to get the credentials that are collected by an authenticator.

```
Map<String, Object> getAuthenticationData()
```

- The `changeResponseOnSuccess` method is called after authentication success. It is used to add data to the response after authentication is successful.

```
Boolean changeResponseOnSuccess (HttpServletRequest request, HttpServletResponse respo
nse)
```

- The `clone` method is used to create a deep copy of class members.

```
WorkLightAuthenticator clone()
```

Walkthrough

1. Create a `MyCustomAuthenticator` class in the `server\java` folder
   Make sure that this class implements the `WorkLightAuthenticator` interface.

   ```
   public class MyCustomAuthenticator implements WorkLightAuthenticator{
   ```

   The `WorkLightAuthenticator` interface extends the `java.io.Serializable` interface, and therefore so does the custom authenticator. This means that the custom authenticator should not have any nonserializable fields. Moreover, it is recommended to include a `serialVersionUID` value in accordance to the `java.io.Serializable` documentation.

2. Add the credentials fields to your authenticator to hold the credentials information.
   These objects will be filled to the authentication data map in `getAuthenticationData()` and used by a login module.

   ```
   private String username;
   private String password;
   ```

3. Add a dependency on the server runtime libraries to use server-related classes, for example, `HttpServletRequest`. In Eclipse:
   1. Right-click your MobileFirst project and select **Properties**.
   2. Select **Java Build Path → Libraries** and click **Add Library**.

3. Select **Server Runtime** and click **Next**.
   You see a list of server runtimes that are installed in your Eclipse IDE.
4. Select one and click **Finish**.
5. Click **OK**.

4. Add calls to the following methods as appropriate.
   - The `init` method is called when the authenticator is created. As its parameter, this method takes the map of options that is specified in a realm definition in the `authenticationConfig.xml` file.

     ```
     public void init(Map<String, String> options) throws MissingConfigurationOptionExceptio
     n {
        logger.info("MyCustomAuthenticator initialized");
     }
     ```

   - The `clone` method of the authenticator creates a deep copy of the object members.

     ```
     public WorkLightAuthenticator clone() throws CloneNotSupportedException {
        MyCustomAuthenticator otherAuthenticator = (MyCustomAuthenticator) super.clone()
     ;
        return otherAuthenticator;
     }
     ```

   - The `processRequest` method is called for each unauthenticated request to collect credentials.

```
public AuthenticationResult processRequest(HttpServletRequest request, HttpServletRes
ponse response, boolean isAccessToProtectedResource) throws IOException, ServletEx
ception {
   if (request.getRequestURI().contains("my_custom_auth_request_url")){
      String username = request.getParameter("username");
      String password = request.getParameter("password");
      if (null != username && null != password && username.length() > 0 && password.l
ength() > 0){
         this.username = request.getParameter("username");
    this.password = request.getParameter("password");
         return AuthenticationResult.createFrom(AuthenticationStatus.SUCCESS);
      } else {
         response.setContentType("application/json; charset=UTF-8");
         response.setHeader("Cache-Control", "no-cache, must-revalidate");
         response.getWriter().print("{\"authStatus\":\"required\", \"errorMessage\":\"Please e
nter username and password\"}");
         return AuthenticationResult.createFrom(AuthenticationStatus.CLIENT_INTERACT
ION_REQUIRED);
      }
   }
   if (!isAccessToProtectedResource)
      return AuthenticationResult.createFrom(AuthenticationStatus.REQUEST_NOT_REC
OGNIZED);
   response.setContentType("application/json; charset=UTF-8");
   response.setHeader("Cache-Control", "no-cache, must-revalidate");
   response.getWriter().print("{\"authStatus\":\"required\"}");
   return AuthenticationResult.createFrom(AuthenticationStatus.CLIENT_INTERACTION_
REQUIRED);
}
```

The `processRequest()` method takes the `request`, `response`, and `isAccessToProtectedResource` arguments. The method can retrieve data from a request and write data to a response, and must return a specific `AuthenticationResult` status as described later.

Reminder: The authenticator collects the credentials for a login module; it **does not** validate them.

```
public AuthenticationResult processRequest(HttpServletRequest request, HttpServletRes
ponse response, boolean isAccessToProtectedResource) throws IOException, ServletEx
ception {
```

The application sends an authentication request to a specific URL. This request URL contains a *`my_custom_auth_request_url`* component, which is used by the authenticator to verify whether this request is an authentication request. It is recommended to have a different URL component in each authenticator.

```
if (request.getRequestURI().contains("my_custom_auth_request_url")){
```

The `authenticator` retrieves the username and password that are passed as request parameters.

```
this.username = request.getParameter("username");
this.password = request.getParameter("password");
```

The authenticator checks the credentials for basic validity, creates an `authenticationData` object, and returns `SUCCESS`.
The `SUCCESS` return value means only that the credentials were successfully collected; after that, the login module is called to validate the credentials.

```
if (null != username && null != password && username.length() > 0 && password.length(
) > 0){
   return AuthenticationResult.createFrom(AuthenticationStatus.SUCCESS);
}
```

If a problem occurs with the received credentials, the authenticator adds an error message to the response and returns `CLIENT_INTERACTION_REQUIRED`. The client must still supply authentication data.

```
else {
   response.setContentType("application/json; charset=UTF-8");
   response.setHeader("Cache-Control", "no-cache, must-revalidate");
   response.getWriter().print("{\"authStatus\":\"required\", \"errorMessage\":\"Please enter u
sername and password\"}");
   return AuthenticationResult.createFrom(AuthenticationStatus.CLIENT_INTERACTION_
REQUIRED);
}
```

The `isAccessToProtectedResource` argument specifies whether an access attempt was made to a protected resource.
If not, the method returns `REQUEST_NOT_RECOGNIZED`, which means that the authenticator treatment is not required, and can proceed with the request as is.

```
if (!isAccessToProtectedResource)
   return AuthenticationResult.createFrom(AuthenticationStatus.REQUEST_NOT_RECOG
NIZED);
```

If the request made to a protected resource does not contain authentication data, the authenticator adds an `authStatus:required` property to the response, and also returns a `CLIENT_INTERACTION_REQUIRED` status.

```
response.setContentType("application/json; charset=UTF-8");
response.setHeader("Cache-Control", "no-cache, must-revalidate");
response.getWriter().print("{\"authStatus\":\"required\"}");
return AuthenticationResult.createFrom(AuthenticationStatus.CLIENT_INTERACTION_RE
QUIRED);
```

- The authenticator `getAuthenticationData` method is used by a login module to get collected credentials.
  After the authenticated session is established, all requests are transported through the `changeResponseOnSuccess` and `processRequestAlreadyAuthenticated` methods. You can use these methods to retrieve data from requests and to update responses.

```
public Map<String, Object> getAuthenticationData() {
   logger.info("getAuthenticationData");
   Map<String, Object> authenticationData = new HashMap<String, Object>()
;
   authenticationData.put("username", username);
   authenticationData.put("password", password);
   return authenticationData;
}
```

- The `changeResponseOnSuccess` method is called after credentials are successfully validated by the login module. You can use this method to modify the response before you return it to the client. This method must return `true` if the response was modified. Otherwise, it returns `false`.
  Use it to notify a client application about the authentication success.

```
public boolean changeResponseOnSuccess(HttpServletRequest request, HttpServletR
esponse response) throws IOException {<br />
   if (request.getRequestURI().contains("my_custom_auth_request_url")){
      response.setContentType("application/json; charset=UTF-8");
      response.setHeader("Cache-Control", "no-cache, must-revalidate");
      response.getWriter().print("{\"authStatus\":\"complete\"}");
      return true;
   }
   return false;
}
```

- The `processRequestAlreadyAuthenticated` method returns `AuthenticationResult` objects for authenticated requests.

```
public AuthenticationResult processRequestAlreadyAuthenticated(HttpServletRequest r
equest, HttpServletResponse response) throws IOException, ServletException {
   return AuthenticationResult.createFrom(AuthenticationStatus.REQUEST_NOT_RECOG
NIZED);
}
```

- If the login module returns an authentication failure, the `processAuthenticationFailure` method is called. This method writes an error message to a response body, and returns the `CLIENT_INTERACTION_REQUIRED` status.

```
public AuthenticationResult processAuthenticationFailure(HttpServletRequest request,
HttpServletResponse response,
      String errorMessage) throws IOException, ServletException {
   response.setContentType("application/json; charset=UTF-8");
   response.setHeader("Cache-Control", "no-cache, must-revalidate");
   response.getWriter().print("{\"authStatus\":\"required\", \"errorMessage\":\"" + errorMessa
ge + "\"}");
   return AuthenticationResult.createFrom(AuthenticationStatus.CLIENT_INTERACTION_
REQUIRED);
}
```

WorklightProtocolAuthenticator

Optionally, you can use the Worklight Protocol to simplify your authentication implementation.

The Worklight protocol can be used when the challenges and challenge responses are represented as JSON objects. In that case, the framework combines all the challenges that conform to the Worklight protocol in a single HTTP request, therefore reducing the number of trips between the server and the client.

For your Custom Authenticator to use the Worklight protocol, make it extend `WorklightProtocolAuthenticator` instead of `WorkLightAuthenticator`.

Note: the `WorklightProtocolAuthenticator` implementation is visible. In Eclipse, click the name and select **Open Declaration** to see its implementation.

`WorklightProtocolAuthenticator` provides you with default implementation for most of the required API, and you can choose which ones to override. The interface also provides a helper method called `getChallengeResponse` to extract the challenge response from the HTTP request. `WorklightProtocolAuthenticator` also knows the name of the realm it belongs to and you can retrieve it by using the `getRealmName` method.

> For an example that uses `WorklightProtocolAuthenticator`, see the Remember Me (../../advanced-topics/remember-me/) tutorial or this video blog post (file:////home/travis/build/MFPSamples/DevCenter/_site/blog/2015/05/29/ibm-mobilefirst-platform-foundation-custom-authenticators-and-login-modules/).

Creating a custom Java login module

API overview

The `WorkLightAuthLoginModule` API includes the following methods:

- The `init` method of the login module is called when the login module instance is created. This method takes the options that are specified in the login module definition of the `authenticationConfig.xml` file.

  void init(Map<String, String> options)

- The `login` method of the login module is used to validate the credentials that are collected by the authenticator.

  boolean login(Map<String, Object> authenticationData)

- The `createIdentity` method of the login module is used to create a `userIdentity` object after validation of the credentials succeeds.

  UserIdentity **createIdentity**(String loginModule)

- The `logout` and `abort` methods are used to clean up cached data after a logout or authentication aborts.

  void logout()
  void abort()

- The `clone` method is used to create a deep copy of the class members.

  WorkLightLoginModule **clone**()

Walkthrough

1. Create a `MyCustomLoginModule` class in the `server\java` folder.
2. Make sure that this class implements the `WorkLightAuthLoginModule` interface.

   public class MyCustomLoginModule implements WorkLightAuthLoginModule **{**

   The `WorkLightAuthLoginModule` interface extends the `java.io.Serializable` interface, and therefore so does your custom login module. This means that it should not have any nonserializable fields. Moreover, it is recommended to include a `serialVersionUID` value in accordance to `java.io.Serializable` documentation.

3. Add two private class members, `USERNAME` and `PASSWORD`, to hold the user credentials.

```
private String USERNAME;
private String PASSWORD;
```

4. Add calls to the following methods as appropriate.
   - The `init` method is called when the login module instance is created. As its parameter, it takes the map of options that are specified in a login module definition in the `authenticationConfig.xml` file.

     ```
     public void init(Map<String, String> options) throws MissingConfigurationOptionExceptio
     n {
     }
     ```

   - The `clone` method of the login module creates a deep copy of the object members.

     ```
     public MyCustomLoginModule clone() throws CloneNotSupportedException {
        return (MyCustomLoginModule) super.clone();
     }
     ```

   - The `login` method is called after the authenticator returns the `SUCCESS` status. When called, the `login` method gets an `authenticationData` object from the authenticator. The `login` method retrieves the username and password that the `authenticator` previously stored.

     In this example, the login module validates the credentials against hardcoded values. You can implement your own validation rules.

     - If the credentials are valid, the `login` method returns `true`.
     - If validation fails, the `login` method can either return `false` or throw a `RuntimeException` exception. The exception string is returned to the authenticator as an `errorMessage` parameter.

       ```
       public boolean login(Map<String, Object> authenticationData) {
          USERNAME = (String) authenticationData.get("username");
          PASSWORD = (String) authenticationData.get("password");
          if (USERNAME.equals("user") && PASSWORD.equals("password")
       )
              return true;
          else
              throw new RuntimeException("Invalid credentials");<br />
       }<br />
       ```

   - The `createIdentity` method is called when the `login` method returns `true`. It is used to create a `UserIdentity` object. You can store your own custom attributes in it to use them later in Java or adapter code.

```
public UserIdentity createIdentity(String loginModule) {
   HashMap<String, Object> customAttributes = new HashMap<String, Object>();
   customAttributes.put("AuthenticationDate", new Date());
   UserIdentity identity = new UserIdentity(loginModule, USERNAME, null, null, customAt
tributes, PASSWORD);
   return identity;
}
```

The `UserIdentity` object contains user information. Its constructor is:

```
public UserIdentity(String loginModule,
      String name,
      String displayName,
      Set<String> roles
      Map<String, Object> attributes,
      Object credentials)
```

- The `logout` and `abort` methods are used to clean up class members after the user logs out or aborts the authentication flow.

```
public void logout() {
   USERNAME = null;
   PASSWORD = null
;
}
public void abort() {
   USERNAME = null;
   PASSWORD = null
;
}
```

Protecting a JavaScript adapter

1. Create an adapter and name it `AuthAdapter`.
2. Add a `getSecretData` procedure and protect it with the security test that you created when you configured the authenticationConfig.xml file.

   ```
   <procedure name="getSecretData" securityTest="AuthAdapter-securityTest"/>
   ```

   In this module, the `getSecretData` procedure returns some hardcoded value:

```
function getSecretData(){
    return {
        secretData: '123456'
    };
}
```

Protecting a Java adapter

1. Create a Java adapter.
2. Add a `getSecretData` method and protect it with the realm that you created when you configured the authenticationConfig.xml file.
   In this module, the `getSecretData` procedure returns some hardcoded value:

```
@GET
@Produces("application/json")
@OAuthSecurity(scope="CustomAuthenticatorRealm")
public JSONObject getSecretData(){
 JSONObject result = new JSONObject();
 result.put("secretData", "123456");
 return result;
}
```

3. To set the new realm as the default user identity for the application, in the application descriptor, add this option:

```
<userIdentityRealms>CustomAuthenticatorRealm</userIdentityRealms>
```

> To learn more about application descriptor properties, see the user documentation.

> To learn more about custom authenticators, see the topic about implementing custom authenticators, in the user documentation.

> Related content: Video blog post on Custom Authentication (file:////home/travis/build/MFPSamples/DevCenter/_site/blog/2015/05/29/ibm-mobilefirst-platform-foundation-custom-authenticators-and-login-modules/)