Custom Authenticator and Login Module

Overview

This tutorial covers how to create a custom `authenticator` and a `login module`.
The following topics will be covered:

- 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.

Jump to:

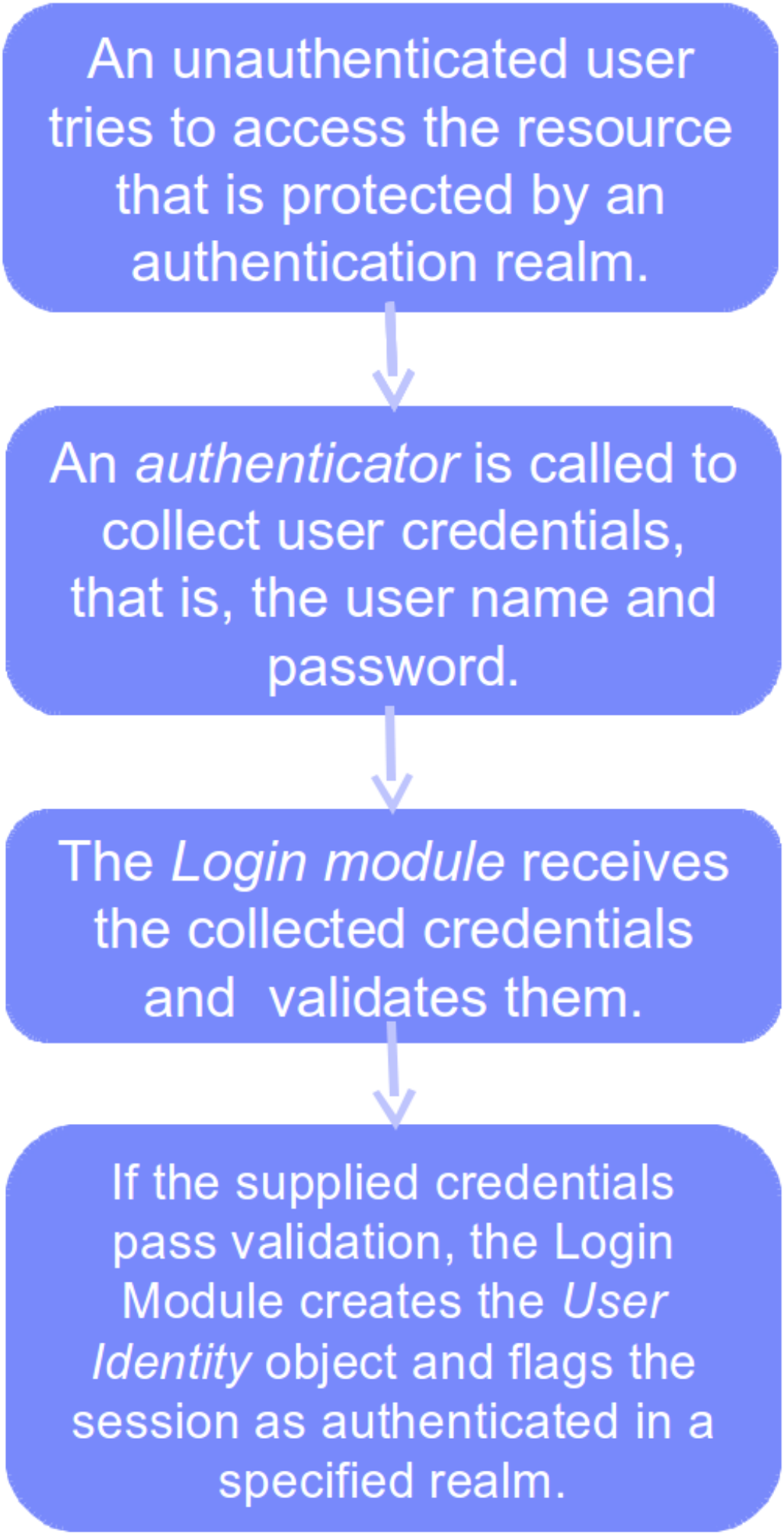
Introduction

The authentication process can be interactive (for example, using 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.


An unauthenticated user tries to access the resource that is protected by an authentication realm.
An *authenticator* is called to collect user credentials, that is, the user name and password.
The *Login module* receives the collected credentials and validates them.
If the supplied credentials pass validation, the Login Module creates the *User Identity* object and flags the session as authenticated in a specified realm.

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 default provided ones do not match your requirements.
In previous tutorials, form-based authentication was implemented using a non-validating login module. Adapter-based authentication was also implemented without having to add login modules, and credentials validation was manually ran.

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 authenticationConfig.xml

In the `realms` section of **authenticationConfig.xml**, define a realm called **`CustomAuthenticatorRealm`**. Make sure that it uses **`CustomLoginModule`**. Specify `MyCustomAuthenticator` as the class name, which will be implemented later.

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

In the `loginModules` section, add a **`loginModule`** called **`CustomLoginModule`**.
Specify `MyCustomLoginModule` as the class name, which will be implemented later.

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

In the `securityTests` section, add a security test.
Later, this security test will be used to protect the adapter procedure. Therefore, use a `customSecurityTest` element. Remember the security test name because it will used later.

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

Creating a custom Java authenticator

The `WorkLightAuthenticator` API includes the following methods:

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

The `init` method of the `authenticator` 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.

```
AuthenticationResult processRequest(HttpServletRequest request, HttpServletResponse response, bool
ean isAccessToProtectedResource)
```

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

```
AuthenticationResult processAuthenticationFailure(HttpServletRequest request, HttpServletResponse re
sponse, String errorMessage)
```

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

```
AuthenticationResult processRequestAlreadyAuthenticated(HttpServletRequest request, HttpServletRes
ponse response)
```

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

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

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

```
Boolean changeResponseOnSuccess (HttpServletRequest request, HttpServletResponse response)
```

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

```
WorkLightAuthenticator clone()
```

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

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

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

Add the `authenticationData` map to your authenticator to hold the credentials information.
This object is retrieved and used by a login module.

```
private Map<String, Object> authenticationData = null;
```

You must add a dependency on server runtime libraries to use server-related classes, for example, **HttpServletRequest**.

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

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 MissingConfigurationOptionException {
 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();
   otherAuthenticator.authenticationData = new HashMap<String, Object>(authenticationData)
;
   return otherAuthenticator;
}
```

processRequest

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

```
public AuthenticationResult processRequest(HttpServletRequest request, HttpServletResponse respons
e, boolean isAccessToProtectedResource) throws IOException, ServletException {
 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.length() > 0){
   authenticationData = new HashMap<String, Object>();
   authenticationData.put("username", username);
   authenticationData.put("password", 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 enter username and p
assword\"}");
   return AuthenticationResult.createFrom(AuthenticationStatus.CLIENT_INTERACTION_REQUIRED);
  }
 }
 if (!isAccessToProtectedResource)
  return AuthenticationResult.createFrom(AuthenticationStatus.REQUEST_NOT_RECOGNIZED);
 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 might 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, HttpServletResponse respons
e, boolean isAccessToProtectedResource) throws IOException, ServletException {
```

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 make sure that this request is an authentication request. It is recommended to have a different URL component in every `authenticator`.

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

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

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

The `authenticator` checks the credentials for basic validity, creates an `authenticationData` object, and returns `SUCCESS`.

`SUCCESS` 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){
 authenticationData = new HashMap<String, Object>();
 authenticationData.put("username", username);
 authenticationData.put("password", password);
 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 username and pa
ssword\"}");
 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_RECOGNIZED);
```

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_REQUIRED);
```

The `authenticator`'s `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");
 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 `false` is returned.
Use it to notify a client application about the authentication success.

```
public boolean changeResponseOnSuccess(HttpServletRequest request, HttpServletResponse respon
se) throws IOException {
 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 request, HttpServ
letResponse response) throws IOException, ServletException {
 return AuthenticationResult.createFrom(AuthenticationStatus.REQUEST_NOT_RECOGNIZED);
}
```

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, HttpServletRespo
nse 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\":\"" + errorMessage + "\"}");
 return AuthenticationResult.createFrom(AuthenticationStatus.CLIENT_INTERACTION_REQUIRED);
}
```

Creating a custom Java login module

The **WorkLightAuthLoginModule** API includes the following methods:

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

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

```
boolean login(Map<String, Object> authenticationData)
```

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

```
UserIdentity createIdentity(String loginModule)
```

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

```
void logout()
void abort()
```

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

```
WorkLightLoginModule clone()
```

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

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

```
public class MyCustomLoginModule implements WorkLightAuthLoginModule {
```

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

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

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 MissingConfigurationOptionException {
}
```

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. The `login` method returns `true` if the credentials are valid.

If the validation fails, the `login` method can either return `false` or throw a `RuntimeException`. 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("12345")
)
  return true;
 else
  throw new RuntimeException("Invalid credentials");
}
```

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 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, customAttributes, PASS
WORD);
 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;
}
```