Remember Me

fork and edit tutorial (https://github.ibm.com/MFPSamples/DevCenter/tree/master/tutorials/en/foundation/7.1/advanced-topics/remember-me.html) | report issue (https://github.ibm.com/MFPSamples/DevCenter/issues/new)

Overview

In this tutorial, you implement a **Remember Me** feature in a hybrid application where, when trying to access a protected resource, the user is presented with a login screen. The login screen includes a check box labeled **Remember Me**. If the user does not select the check box, the session remains open for a short period of time (2 hours, for example). If the user selects the check box, the client is trusted for a longer period (2 weeks, for example).

To better understand this advanced tutorial, first make sure to understand the following prerequisites:

- Java Adapters (../../server-side-development/java-adapter/)
- Authentication Concepts (../../authentication-security/authentication-concepts/)
- Custom Authentication Tutorial (../../authentication-security/custom-authentication/)
- Custom Authentication video blog (file:////home/travis/build/MFPSamples/DevCenter/_site/blog/2015/05/29/ibm-mobilefirst-platform-foundation-custom-authenticators-and-login-modules/)
- Cloudant-as-a-service (https://docs.cloudant.com/)

Agenda

- Architecture
- authenticationConfig.xml
- Adapter
- Cloudant
- RememberedClient.java
- Challenge handler
- Logout
- Sample application

Architecture

Important: This is only one of many possible ways to build such a scenario.
You might need to adapt it to your specific needs.

The simple scenario without the **Remember Me** feature is similar to the Custom Authentication tutorial:

1. The client tries to access a protected resource.
2. The authenticator sends a challenge to the client.
3. The client answers the challenge with a user name and password.
4. The authenticator extracts the credentials.
5. The Login Module validates and creates a user identity.
6. The client has access to the protected resource.

With the **Remember Me** check box, the steps become:

1. The client tries to access a protected resource.
2. The authenticator looks up the client ID in a remote data storage (Cloudant).
3. If the client is found, the Login Module creates a user identity and the client has immediate access to the protected resource.
4. If the client is not found, the authenticator sends the challenge to the client.
5. The client answers the challenge with a username, password, and remember-me check.
6. The authenticator validates the credentials and saves the client ID to the remote data storage (Cloudant).
7. The Login Module creates a user identity.
8. The client has access to the protected resource.

Note: Cloudant is used here as the remote storage. However, you can easily change this default choice to your preferred database or to any other type of remote storage.

authenticationConfig.xml

Login Module

The `login module` is usually responsible for checking the user credentials and creating the `user identity`. However, in this example you need only the `login module` to create the `user identity`, while the rest of the logic happens in the `authenticator`. The standard `NonValidatingLoginModule` is good enough to achieve this.

```
<loginModules><br />
 <!-- Use expirationInSeconds to set the session timeout when NOT using rememberMe  -->
 <loginModule name="CustomLoginModule" expirationInSeconds="30">
  <className>com.worklight.core.auth.ext.NonValidatingLoginModule</className>
 </loginModule>
</loginModules>
```

In previous versions of IBM MobileFirst Platform Foundation, the logged-in state of clients was entirely dependent on an HTTP session. After logging in, the end user remained logged in only as long as the HTTP session was alive. If either the client restarted, or the HTTP session ended, the end user was automatically logged out.
In version 7.1 and later, the session-independent mode decouples the link between the logged-in state and HTTP sessions, so that interaction between client and MobileFirst Server is no longer session-dependent. This feature enables you to take down a server without breaking user sessions.

The `expirationInSeconds` parameter is a built-in setting to determine how long the user session is valid. In this example, the value of this parameter is the default expiration when the user does **not** select **Remember Me**. To make testing easier, the expiration value is set to 30 seconds here, but in a real application you would probably want to use a longer expiration time, such as one hour...

Realm

```
<realms>
 <realm name="CustomRealm" loginModule="CustomLoginModule">
  <className>com.sample.CustomAuthenticator</className>
  <!-- Set the number of days to remember when the user checks rememberMe -->
  <parameter name="rememberMeExpirationInDays" value="3"/>
 </realm>
</realms>
```

The `realm` uses the `loginModule` that was defined in the Login Module section and the class that you will define in the Custom authenticator section.
A custom parameter called `rememberMeExpirationInDays` defines how long the client is remembered if that user selects **Remember Me**.

Security test

Because this example uses a Java adapter, you do not really need to define a security test (Java adapters use scopes instead of tests), but just in case it is necessary to protect other resources, create a simple security test.

```
<securityTests>
 <customSecurityTest name="CustomAuthSecurityTest">
  <test realm="wl_antiXSRFRealm" />
  <test realm="wl_deviceNoProvisioningRealm" isInternalDeviceID="true" /
>
  <test realm="CustomRealm" isInternalUserID="true" />
 </customSecurityTest>
</securityTests>
```

The security test uses some standard realms, and uses the `CustomRealm` as a user realm.

Adapter code

Create a simple Java adapter with a single `hello` method. Protect it with the realm that you created in the Realm section.

```
@GET
@Path("/hello")
@Produces("application/xml")
@OAuthSecurity(scope="CustomRealm")
public String hello(){
 //log message to server log
    logger.info("Logging info message...");
 return "Hello from the Java REST adapter";
}
```

Cloudant

Storing in Cloudant is optional, you can choose any database you want for this part.

cloudant-client

Because this example uses Cloudant, client code is needed to connect to the Cloudant instance. Use the official java-cloudant (https://github.com/cloudant/java-cloudant/releases) client by putting `cloudant-client-1.0.1-.jar` in the `lib` folder of the server.

Cloudant credentials

1. Create a Cloudant.com (http://www.cloudant.com) account if you don't have one already (free trial).
2. Create a new database (*rememberme*), generate a new set of API key and password for this database, and grant it read/write access.

worklight.properties

You need to store the Cloudant credentials somewhere. An option is the `worklight.properties` file, which you can access from the custom authenticator.

```
###########
#  Cloudant credentials for RememberMe sample
###########
cloudant.domain=myCloudantUser
cloudant.key=myAPIKey
cloudant.password=myAPIPassword
```

RememberedClient.java

Cloudant uses JSON notation to store documents. To make it easier in Java, create a simple `POJO (https://en.wikipedia.org/wiki/Plain_Old_Java_Object)` to represent a "remembered client id".

```
public class RememberedClient {
 private String username, _id, _rev;
 private Date expiration;
//insert getters and setters
}
```

For convenience, this class also includes a getter/setter pair to map `ClientID` to the `_id` property.

```
public String getClientID() {
 return get_id();
}
public void setClientID(String clientID) {
 set_id(clientID);
}
```

CustomAuthenticator.java

The authenticator extends `WorklightProtocolAuthenticator` to inherit the simplifications provided by the *Worklight protocol*.

```
public class CustomAuthenticator extends WorklightProtocolAuthenticator {
}
```

init

This example uses the `init` method to load the configurations from the `authenticatorConfig.xml` and from the `worklight.properties` files. The CloudantClient and database are initialized here, too.

```
@Override
public void init(Map<String, String> options) throws MissingConfigurationOptionException {
 logger.info("CustomAuthenticator :: Initializing. options :: " + options.toString());
 super.init(options);
 String tempExpiration = options.remove(PROPERTY_EXPIRATION_DAYS);
 if(tempExpiration == null){
  throw new MissingConfigurationOptionException(PROPERTY_EXPIRATION_DAYS);
 }
 rememberMeExpirationInDays = Integer.parseInt(tempExpiration);
 String cloudantDomain = WorklightConfiguration.getInstance().getStringProperty("cloudant.domain");
 String cloudantKey = WorklightConfiguration.getInstance().getStringProperty("cloudant.key");
 String cloudantPassword = WorklightConfiguration.getInstance().getStringProperty("cloudant.password")
;
 cloudant = new CloudantClient(cloudantDomain,cloudantKey,cloudantPassword);
 db = cloudant.database(CLOUDANT_DB, false);
}
```

Cloudant helper methods

Create methods to create, read, and delete remembered clients from the *cloudant* database.

- The `rememberClient` method creates a new document in the database that stores the `clientID`, associated `username`, and expiration `date`.
- The `getRememberedClient` method looks up a Cloudant document by `ClientID`. If the client is not known, the method returns `null`. If the client is expired, the method deletes it and returns `null`.
- The `forgetClient` method deletes a client from Cloudant. It has 2 implementations, one of which is static so that the method can be called from outside the authenticator.
  See the sample code for implementation details.

generateChallenge

The `generateChallenge` method is a helper method to create a JSON challenge that the client can interpret in its challenge handler.

```
private AuthenticationResult generateChallenge(String errorMessage){
 logger.info("CustomAuthenticator :: generateChallenge");
 AuthenticationResult authenticationResult = AuthenticationResult.createFrom(AuthenticationStat
us.CLIENT_INTERACTION_REQUIRED);
 JSONObject challengeObj = new JSONObject();
 challengeObj.put("authStatus", "credentialsRequired");
 challengeObj.put("errorMessage", errorMessage);
 authenticationResult.setJson(challengeObj);
 return authenticationResult;
}
```

processRequest

The `processRequest` method is where the main logic happens. This is the method that the framework calls to access a protected resource.
The `processRequest` method first checks whether the client exists in the database.

- If the request finds the client, it sends a `SUCCESS` result to the framework.

```
@Override
public AuthenticationResult processRequest(HttpServletRequest request,
  HttpServletResponse response, boolean isAccessToProtectedResource)
  throws IOException, ServletException {
 logger.info("CustomAuthenticator :: processRequest");
 if (!isAccessToProtectedResource){
  logger.info("CustomAuthenticator :: !isAccessToProtectedResource");
  return AuthenticationResult.createFrom(AuthenticationStatus.REQUEST_NOT_RECOGNIZED)
;
 }
 String clientID = AuthenticationContext.getCurrentClientId();
 logger.info("CustomAuthenticator :: clientID = " + clientID);
 RememberedClient client = getRememberedClient(clientID);
 if(client != null){
  logger.info("CustomAuthenticator :: found client");
  username = client.getUsername();
  password = null; //Using non-validating
  logger.info("CustomAuthenticator :: SUCCESS from ClientID");
  return AuthenticationResult.createFrom(AuthenticationStatus.SUCCESS);
 }
```

- If the client was not found and the current request does not already contain a challenge response, the request generates a challenge and sends it.

```
JSONObject challengeResponse = (JSONObject) getChallengeResponse(request);
if (null == challengeResponse){
 return generateChallenge("Please enter username and password");
}
```

- If the current request already contains the response from the challenge, it verifies the credentials (in this example, no real verification happens). It then saves the client in the database and returns `SUCCESS`.

```
username = (String) challengeResponse.get("username");
password = (String) challengeResponse.get("password");
rememberMe = (Boolean) challengeResponse.get("rememberMe");
logger.info("CustomAuthenticator :: rememberMe = " + rememberMe);
if (null == username || null == password || username.length() == 0 || password.length() == 0){
 return generateChallenge("Username and password cannot be blank");
} else {
 //The login module is non-validating. Do the validation here
 //TODO: password validation against your database
 if(rememberMe){
  //If all is good, backup the user/clientID
  this.rememberClient(clientID, username);
 }
 logger.info("CustomAuthenticator :: SUCCESS");
 return AuthenticationResult.createFrom(AuthenticationStatus.SUCCESS);
}
```

getAuthenticationData

The `getAuthenticationData` method is called by the framework to provide login credentials to the `login module`. In this example, the `login module` is non-validating, therefore the password can be ignored. The `login module` is used only to create a `user identity` in the framework.

```
@Override
public Map<String, Object> getAuthenticationData() {
   logger.info("CustomAuthenticator :: getAuthenticationData");
   Map<String, Object> authData = new HashMap<String, Object>()
;
   authData.put(USERNAME_KEY, username);
   authData.put(PASSWORD_KEY, password);

   return authData;
}
```

Challenge handler

Because the `authenticator` uses the *Worklight protocol*, use the `createWLChallengeHandler` method to create the challenge handler.

```
var challengeHandler = WL.Client.createWLChallengeHandler("CustomRealm");
```

The challenge handler hides and shows the login parts of the application when required. It uses the `submitChallengeAnswer` method to send the credentials and the **Remember Me** checkbox. See the sample code for implementation details.

The client side can be adapted to a native environment.

Logout

The standard way of logging out of a realm in JavaScript is `WL.Client.logout('CustomRealm')`. However, after the client is logged out of the realm, the `clientID` is still stored in Cloudant and will be recognized immediately during the next request. This implementation could lead to unwanted behavior whereby it would be virtually impossible to log out.

To work around this issue, clear the `clientId` from the remote storage before logging out.

1. In your Java adapter, add a method that uses the static `forgetClient` method that you created in Cloudant helper methods.

```
@POST
@Path("/forgetMe")
@OAuthSecurity(scope="CustomRealm")
public Response forgetMe(){
   CustomAuthenticator.forgetClient(AuthenticationContext.getCurrentClientId())
;
   return Response.ok().build();
}
```

2. From the client code, make sure that you call this method before using the standard logout feature.

```
function logout(){
   busyIndicator.show();
   var resourceRequest = new WLResourceRequest("/adapters/Content/forgetMe", WLResource
Request.POST, 30000);
   resourceRequest.send().then(
      function(){
         WL.Client.logout('CustomRealm', {onSuccess:WL.Client.reloadApp});
      }
   );
}
```

Sample application

To run the sample, make sure that you download the cloudant JAR file and set up the credentials in the `worklight.properties` file.

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/RememberMe/tree/release71) the sample application.