Custom Authentication in hybrid applications

fork and edit tutorial (https://github.ibm.com/MFPSamples/DevCenter/tree/master/tutorials/en/foundation/7.1/authentication-security/custom-authentication/custom-authentication-hybrid-applications.html) | report issue (https://github.ibm.com/MFPSamples/DevCenter/issues/new)

Overview

This is a continuation of Custom Authentication (../).

Creating client-side authentication components

The application consists of two main div elements:

The *AppDiv* element is used to display the application content.

The AuthDiv element is used for authentication forms.

When authentication is required, the application hides *AppDiv* and shows *AuthDiv*. When authentication is complete, it does the opposite.

AppDiv

```
<div id="AppDiv">
  <input type="button" id="getSecretDataButton" value="Call protected adapter proc" onclick="getSecret Data()" />
  <input type="button" class="appButton" value="Logout" onclick="WL.Client.logout('CustomAuthenticato rRealm',{onSuccess: WL.Client.reloadApp})" />
  <div id="ResponseDiv"></div>
</div></div></div></div></div></div></div></div></div></div></div></div></div></div></div></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber></tiber>
```

Buttons are used to call the *getSecretData* procedure and to log out.

AuthDiv

AuthDiv is styled with display:none because it must not be displayed before the server requests the authentication.

Challenge Handler

Use WL.Client.createChallengeHandler to create a challenge handler object. Supply a realm name as a parameter.

var customAuthenticatorRealmChallengeHandler = WL.Client.createChallengeHandler("CustomAuthentic
atorRealm");

The *isCustomResponse* function of the challenge handler is called each time a response is received from the server.

It is used to detect whether the response contains data that is related to this challenge handler. It must return **true** or **false**.

```
customAuthenticatorRealmChallengeHandler.isCustomResponse = function(response) {}
```

If *isCustomResponse* returns true, the framework calls the *handleChallenge* function. This function is used to perform required actions, such as hide application screen and show login screen.

```
customAuthenticatorRealmChallengeHandler.handleChallenge = function(response){}
```

In addition to the methods that the developer must implement, the challenge handler contains functionality that the developer might want to use:

- *submitLoginForm* to send collected credentials to a specific URL. The developer can also specify request parameters, headers, and callback.
- *submitSuccess* to notify the framework that the authentication finished successfully. The framework then automatically issues the original request that triggered the authentication.
- *submitFailure* to notify the framework that the authentication completed with a failure. The framework then disposes of the original request that triggered the authentication

Note: Attach each of these functions to its object. For example: myChallengeHandler.submitSucces()

isCustomResponse

If the challenge JSON block contains the authStatus property, return true, otherwise return false.

```
customAuthenticatorRealmChallengeHandler.isCustomResponse = function(response) {
   if (!response || !response.responseJSON) {
      return false;
   }
   if (response.responseJSON.authStatus)
      return true;
   else
      return false;
};
```

If the *authStatus* property equals "required", show the login form, clean up the password input field, and display the error message if applicable.

if *authStatus* equals "complete", hide the login screen, return to the application, and notify the framework that authentication completed successfully.

```
customAuthenticatorRealmChallengeHandler.handleChallenge = function(response){
   var authStatus = response.responseJSON.authStatus;
   if (authStatus == "required"){
      $('#AppDiv').hide();
      $('#AuthDiv').show();
      $("#AuthInfo").empty();
      $('#AuthPassword').val(");
      if (response.responseJSON.errorMessage){
        $("#AuthInfo").html(response.responseJSON.errorMessage);
   } else if (authStatus == "complete"){
      $('#AppDiv').show();
      $('#AuthDiv').hide();
      customAuthenticatorRealmChallengeHandler.submitSuccess();
   }
 };
4
                                                                                 •
```

Clicking the **login** button triggers the function that collects the user name and password from HTML input fields and submits them to server. You can set request headers here and specify callback functions.

```
$('#AuthSubmitButton').bind('click', function () {
    var reqURL = '/my_custom_auth_request_url';
    var options = {};
    options.parameters = {
        username : $('#AuthUsername').val(),
        password : $('#AuthPassword').val()
    };
    options.headers = {};
    customAuthenticatorRealmChallengeHandler.submitLoginForm(reqURL, options, customAuthenticator RealmChallengeHandler.submitLoginFormCallback);<br/>});
```

Clicking the **cancel** button hides *AuthDiv*, shows *AppDiv* and notifies the framework that authentication failed.

```
$('#AuthCancelButton').bind('click', function () {
   $('#AppDiv').show();
   $('#AuthDiv').hide();
   customAuthenticatorRealmChallengeHandler.submitFailure()
;
});
```

The **submitLoginFormCallback** function checks the response for the containing server challenge once again. If the challenge is found, the *handleChallenge* function is called again.

```
customAuthenticatorRealmChallengeHandler.submitLoginFormCallback = function(response) {
    var isLoginFormResponse = customAuthenticatorRealmChallengeHandler.isCustomResponse(response);
    if (isLoginFormResponse){
        customAuthenticatorRealmChallengeHandler.handleChallenge(response);
    }
};
```

Worklight Protocol

If your custom authenticator uses <code>WorklightProtocolAuthenticator</code>, some simplifications can be made:

- Create the challenge handler using WL.Client.createWLChallengeHandler instead of WL.Client.createChallengeHandler. Note the WL in the middle.
- You no longer need to implement <code>isCustomResponse</code> as the challenge handler will automatically check that the realm name matches.
- handleChallenge will receive the challenge as a parameter, not the entire response object.
- Instead of submitLoginForm, use submitChallengeAnswer to send your challenge response as a JSON.
- There is no need to call submitSuccess or submitFailure as the framework will do it for you.

For an example that uses <code>WorklightProtocolAuthenticator</code>, 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/).

Sample application

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/CustomAuth/tree/release71) the MobileFirst project.

