Implementing the challenge handler in JavaScript (Cordova, Web) applications

Overview

When trying to access a protected resource, the server (the security check) will send back to the client a list containing one or more **challenges** for the client to handle.
This list is received as a `JSON object`, listing the security check name with an optional `JSON` of additional data:

```
{
  "challenges": {
    "SomeSecurityCheck1":null,
    "SomeSecurityCheck2":{
      "some property": "some value"
    }
  }
}
```

The client should then register a **challenge handler** for each security check.
The challenge handler defines the client-side behavior that is specific to the security check.

Creating the challenge handler

A challenge handler is responsible for handling challenges sent by the MobileFirst server, such as displaying a login screen, collecting credentials and submitting them back to the security check.

In this example, the security check is `PinCodeAttempts` which was defined in Implementing the CredentialsValidationSecurityCheck (../security-check). The challenge sent by this security check contains the number of remaining attempts to login (`remainingAttempts`), and an optional `errorMsg`.

Use the `WL.Client.createSecurityCheckChallengeHandler()` API method to create and register a challenge Handler:

```
PinCodeChallengeHandler = WL.Client.createSecurityCheckChallengeHandler("PinCodeAttempts");
```

Handling the challenge

The minimum requirement from the `createSecurityCheckChallengeHandler` protocol is to implement the `handleChallenge()` method, that is responsible for asking the user to provide the credentials. The `handleChallenge` method receives the challenge as a `JSON` Object.

> Learn more about the `createSecurityCheckChallengeHandler` protocol in the user documentation.

In this example, a prompt is displayed asking to enter the PIN code:

```
PinCodeChallengeHandler.handleChallenge = function(challenge) {
    var msg = "";

    // Create the title string for the prompt
    if(challenge.errorMsg != null) {
        msg =  challenge.errorMsg + "\n";
    } else {
        msg = "This data requires a PIN code.\n";
    }

    msg += "Remaining attempts: " + challenge.remainingAttempts;

    // Display a prompt for user to enter the pin code
    var pinCode = prompt(msg, "");

    if(pinCode){ // calling submitChallengeAnswer with the entered value
        PinCodeChallengeHandler.submitChallengeAnswer({"pin":pinCode});
    } else { // calling cancel in case user pressed the cancel button
        PinCodeChallengeHandler.cancel();
    }
};
```

If the credentials are incorrect, you can expect the framework to call `handleChallenge` again.

Submitting the challenge's answer

Once the credentials have been collected from the UI, use `createSecurityCheckChallengeHandler`'s `submitChallengeAnswer()` to send an answer back to the security check. In this example `PinCodeAttempts` expects a property called `pin` containing the submitted PIN code:

```
PinCodeChallengeHandler.submitChallengeAnswer({"pin":pinCode});
```

Cancelling the challenge

In some cases, such as clicking a "Cancel" button in the UI, you want to tell the framework to discard this challenge completely.
To achieve this, call:

```
PinCodeChallengeHandler.cancel();
```

Handling failures

Some scenarios may trigger a failure (such as maximum attempts reached). To handle these, implement `createSecurityCheckChallengeHandler`'s `handleFailure()`.
The structure of the JSON object passed as a parameter greatly depends on the nature of the failure.

```
PinCodeChallengeHandler.handleFailure = function(error) {
    WL.Logger.debug("Challenge Handler Failure!");

    if(error.failure && error.failure == "account blocked") {
        alert("No Remaining Attempts!");
    } else {
        alert("Error! " + JSON.stringify(error));
    }
};
```

Handling successes

In general successes are automatically processed by the framework to allow the rest of the application to continue.

Optionally you can also choose to do something before the framework closes the challenge handler flow, by implementing `createSecurityCheckChallengeHandler`'s `handleSuccess()`. Here again, the content and structure of the `success` JSON object depends on what the security check sends.

In the `PinCodeAttemptsCordova` sample application, the success does not contain any additional data.

Registering the challenge handler

In order for the challenge handler to listen for the right challenges, you must tell the framework to associate the challenge handler with a specific security check name.
This is done by creating the challenge handler with the security check like this:

```
someChallengeHandler = WL.Client.createSecurityCheckChallengeHandler("the-securityCheck-name");
```

Sample applications

The **PinCodeWeb** and **PinCodeCordova** projects use `WLResourceRequest` to get a bank balance.
The method is protected with a PIN code, with a maximum of 3 attempts.

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/PinCodeWeb/tree/release80) the Web project.
Click to download (https://github.com/MobileFirst-Platform-Developer-Center/PinCodeCordova/tree/release80) the Cordova project.
Click to download (https://github.com/MobileFirst-Platform-Developer-Center/SecurityCheckAdapters/tree/release80) the SecurityAdapters Maven project.

Web sample usage

Make sure you have Node.js installed.

1. Register the application in the MobileFirst Operations Console.
2. Start the reverse proxy by running the commands: `npm install` followed by: `npm start`.
3. Use either Maven, MobileFirst CLI or your IDE of choice to build and deploy the available **ResourceAdapter** and **PinCodeAttempts** adapters (../../../adapters/creating-adapters/).
4. In the MobileFirst Console → PinCodeWeb → Security, map the `accessRestricted` scope to the `PinCodeAttempts` security check.
5. In a browser, load the URL http://localhost:9081/sampleapp (http://localhost:9081/sampleapp).

Cordova Sample usage

1. Use either Maven or MobileFirst CLI to build and deploy the available **ResourceAdapter** and **PinCodeAttempts** adapters (../../../adapters/creating-adapters/).
2. From a **Command-line** window, navigate to the project's root folder and:
   - Add a platform by running the `cordova platform add` command.
   - Registering the application: `mfpdev app register`.
3. Map the `accessRestricted` scope to the `PinCodeAttempts` security check:
   - In the MobileFirst Operations Console, under **Applications → PIN Code → Security → Scope-Elements Mapping**, add a scope mapping from `accessRestricted` to `PinCodeAttempts`.
   - Alternatively, from the **Command-line**, navigate to the project's root folder and run the command: `mfpdev app push`.

     > Learn more about the mfpdev app push/push commands in the Using MobileFirst CLI to manage MobilefFirst artifacts (../../../using-the-mfpf-sdk/using-mobilefirst-cli-to-manage-mobilefirst-artifacts) tutorial.

4. Back in the command-line:
   - Run the Cordova application by running the `cordova run` command.