Implementing the challenge handler in Android applications

Overview

This tutorial is a continuation of **credentials validation**'s challenge handler (../../credentials-validation/android) tutorial. Make sure to read it first.

The challenge handler will be modified to fit the `UserAuthSecurityCheck` created in the matching security check tutorial (../security-check), and will demonstrate a few additional features such as the preemptive `login` API, `logout` and `obtainAccessToken`.

Creating the challenge handler

Create a Java class that extends `WLChallengeHandler`:

```
public class UserAuthChallengeHandler extends WLChallengeHandler {

}
```

Add a constructor method:

```
public UserAuthChallengeHandler(String securityCheck) {
    super(securityCheck);
}
```

Register the challenge handler:

```
WLClient client = WLClient.createInstance(this);
client.registerChallengeHandler(myUserAuthChallengeHandler);
```

Handling the challenge

In this example, the challenge sent by the `UserAuthSecurityCheck` is the same sent by `PinCodeAttempts`: the number of remaining attempts to login (`remainingAttempts`), and an optional `errorMsg`.

Your `handleChallenge` method is responsible for collecting the username and password from the user.

Submitting the credentials

Once the credentials have been collected from the UI, use the `WLChallengeHandler`'s `submitChallengeAnswer(JSONObject answer)` method to send an answer back to the security check. In this example `UserAuthSecurityCheck` expects keys called `username` and `password`. Optionally, it also accepts a boolean `rememberMe` key that will tell the security check to remember this user for a longer period. You can collect this boolean value from a checkbox in the login form.

`credentials` is a JSONObject containing `username`, `password` and `rememberMe`:

```
submitChallengeAnswer(credentials);
```

Sometimes, you want to login a user without any challenge being received. For examples, showing a login screen as the first screen of the application, or showing a login screen after a logout, or a login failure, etc. We call those scenarios **preemptive** logins.

You cannot call the `submitChallengeAnswer` API if there is no challenge to answer. For those scenarios, the MobileFirst Platform SDK includes a different API:

```
WLAuthorizationManager.getInstance().login(securityCheckName, credentials, new WLLoginResponseListe
ner() {
    @Override
    public void onSuccess() {
        Log.d(securityCheckName, "Login Preemptive Success");

    }

    @Override
    public void onFailure(WLFailResponse wlFailResponse) {
        Log.d(securityCheckName, "Login Preemptive Failure");
    }
});
```

If the credentials are wrong, the security check will send back a **challenge**.

It is your responsibility to know when to use `login` vs `submitChallengeAnswer` based on your application needs. One way to achieve this is to define a boolean flag, for example `isChallenged`; set it to `true` when you reach `handleChallenge` and set it to `false` in any other cases (failure, success, initializing, etc).

Then when the user clicks the **Login** button, you can change which API to use dynamically:

```
public void login(JSONObject credentials){
    if(isChallenged){
        submitChallengeAnswer(credentials);
    }
    else{
        WLAuthorizationManager.getInstance().login(securityCheckName, credentials, new WLLoginRespons
eListener() {
            @Override
            public void onSuccess() {
                Log.d(securityCheckName, "Login Preemptive Success");

            }

            @Override
            public void onFailure(WLFailResponse wlFailResponse) {
                Log.d(securityCheckName, "Login Preemptive Failure");
            }
        });
    }
}
```

Obtaining an access token

Since this security check supports *remember me*, it would be useful to check if the user is currently logged in, during the application startup.

MobileFirst Platform SDK provides an API to ask the server for a valid token:

```
WLAuthorizationManager.getInstance().obtainAccessToken(scope, new WLAccessTokenListener() {
    @Override
    public void onSuccess(AccessToken accessToken) {
        Log.d(securityCheckName, "auto login success");
    }

    @Override
    public void onFailure(WLFailResponse wlFailResponse) {
        Log.d(securityCheckName, "auto login failure");
    }
});
```

If the user is already logged in or is in the *remembered* state, the API will trigger a success. If the user is not logged in, the security check will send back a challenge.

The `obtainAccessToken` API takes in a **scope**. The scope here can be the name of your **security check**.

> Learn more about **scope** here: Authentication Concepts (../../authentication-concepts)

Handling success and failure

As noted in the previous tutorial, `WLChallengeHandler` will call your `handleSuccess` or `handleFailure` methods upon success or failure of the challenge. You can choose to update your UI based on those events.

> Note that while `WLAuthorizationManager`'s `login()` API has its own `onSuccess` and `onFailure` methods, the relevant challenge handler's `handleSuccess` or `handleFailure` will **also** be called.
>
> Note that while `WLAuthorizationManager`'s `obtainAccessToken()` API has its own `onSuccess` and `onFailure` methods, the relevant challenge handler's `handleSuccess` or `handleFailure` will **also** be called.

Retrieving the authenticated user

The challenge handler's `handleSuccess` method receives a `JSONObject identity` as a parameter. If the security check sets an `AuthenticatedUser`, this object will contain the user's properties. You can use `handleSuccess` to save the current user:

```
@Override
public void handleSuccess(JSONObject identity) {
    super.handleSuccess(identity);
    isChallenged = false;
    try {
        //Save the current user
        SharedPreferences preferences = context.getSharedPreferences(Constants.PREFERENCES_FILE, C
ontext.MODE_PRIVATE);
        SharedPreferences.Editor editor = preferences.edit();
        editor.putString(Constants.PREFERENCES_KEY_USER, identity.getJSONObject("user").toString());
        editor.commit();
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

Here, `identity` has a key called `user` which itself contains a `JSONObject` representing the `AuthenticatedUser`:

```
{
  "user": {
    "id": "john",
    "displayName": "john",
    "authenticatedAt": 1455803338008,
    "authenticatedBy": "UserAuthSecurityCheck"
  }
}
```

Logout

The MobileFirst Platform SDK also provides a `logout` API to logout from a specific security check:

```
WLAuthorizationManager.getInstance().logout(securityCheckName, new WLLogoutResponseListener() {
    @Override
    public void onSuccess() {
        Log.d(securityCheckName, "Logout Success");
    }

    @Override
    public void onFailure(WLFailResponse wlFailResponse) {
        Log.d(securityCheckName, "Logout Failure");
    }
});
```

Samples

There are two samples associated with this tutorial:

- **PreemptiveLoginAndroid**: An application that always starts with a login screen, using the preemptive `login` API.
- **RememberMeAndroid**: An application with a *Remember Me* checkbox. The user can bypass the login screen the next time the application is opened.

Both samples use the same `UserAuthSecurityCheck` from the **SecurityCheckAdapters** project.