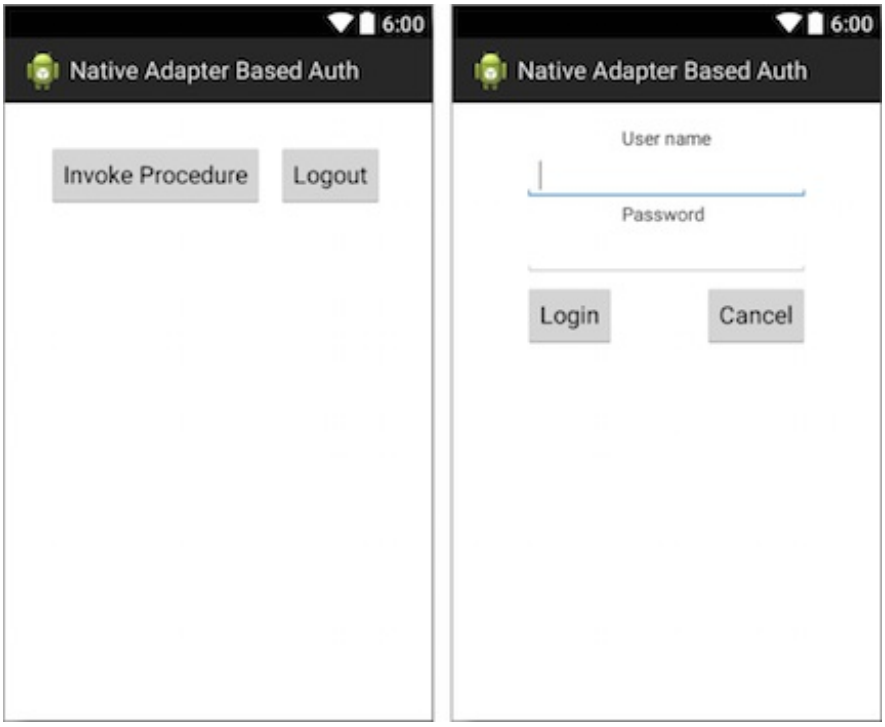
# Adapter-based authentication in native Android applications

## Overview

This tutorial explains how to implement the client-side of adapter-based authentication in native Android.
**Prerequisite:** Make sure that you read the Adapter-based authentication (../) tutorial first.

## Implementing the client-side authentication

- Create a native Android application and add the MobileFirst native APIs as explained in the Configuring a native Android application with the MobileFirst Platform SDK (../../../hello-world/configuring-a-native-android-application-with-the-mfp-sdk/) tutorial.
- Add an activity which handles and presents a login form.




### Challenge Handler

- Create a `MyChallengeHandler` class as a subclass of `ChallengeHandler`.

```
public class AndroidChallengeHandler extends ChallengeHandler
```

- Call the `super` method:

```
public AndroidChallengeHandler(String realm) {
 super(realm);
}
```

- Add an implementation of the following `ChallengeHandler` methods to handle the form-based challenge:
  1. **`isCustomResponse` method:**
     The `isCustomResponse` method is invoked each time a response is received from the MobileFirst Server. It is used to detect whether the response contains data that is related to this challenge handler. It must return either `true` or `false`.

     ```
     public boolean isCustomResponse(WLResponse response) {
      try {
       if(response!= null &&
         response.getResponseJSON()!=null &&
         !response.getResponseJSON().isNull("authStatus") &&
         response.getResponseJSON().getString("authStatus") != "")
     {
        return true;
       }
      } catch (JSONException e) {
       e.printStackTrace();
      }
      return false;
     }
     ```

  2. **`handleChallenge` method:**
     If `isCustomResponse` returns `true`, the framework calls the `handleChallenge` method. This function is used to perform required actions, such as hiding the application screen and showing the login screen.

     ```
     public void handleChallenge(WLResponse response){
      cachedResponse = response;
      try {
       if(response.getResponseJSON().getString("authStatus").equals("credentialsRequired"))
     {
        MainAdapterBasedAuth.setMainText("handleChallenge->credentialsRequired");
        Intent login = new Intent(parentActivity, LoginAdapterBasedAuth.class);
        parentActivity.startActivityForResult(login, 1);
       }
       else if(response.getResponseJSON().getString("authStatus").equals("complete")){
        submitSuccess(cachedResponse);
       }
      } catch (JSONException e) {
       // TODO Auto-generated catch block
       e.printStackTrace();
      }
     }
     ```

  3. **`onSuccess` and `onFailure` methods:**
     At the end of the authentication flow, `onSuccess` or `onFailure` will be triggered
     Call the `submitSuccess` method in order to inform the framework that the authentication process completed successfully and for the `onSuccess` handler of the invocation to be called.
     Call the `submitFailure` method in order to inform the framework that the authentication

process failed and for the `onFailure` handler of the invocation to be called.

```
public void onFailure(WLFailResponse response) {
 submitFailure(response);
}
public void onSuccess(WLResponse response) {
 submitSuccess(response);
}
```

### submitLoginForm

When the user taps to submit the credentials, call the `submitLoginForm` method to send the credentials to the adapter procedure.
For example, in here we implemented a `submitLogin method` that called by the `MainActivity` after the login process is completed.

```
public void submitLogin(int resultCode, String userName, String password, boolean back){
 if (resultCode != Activity.RESULT_OK || back) {
  submitFailure(cachedResponse);
 } else {
  Object[] parameters = new Object[]{userName, password};
  WLProcedureInvocationData invocationData = new WLProcedureInvocationData("AuthAdapter", "subm
itAuthentication");
  invocationData.setParameters(parameters);
  WLRequestOptions options = new WLRequestOptions();
  options.setTimeout(30000);
  submitAdapterAuthentication(invocationData, options);
 }
}
```

## The Main Activity

In the sample project, in order to trigger the challenge handler we use the `WLClient invokeProcedure` method.
The protected procedure invocation triggers MobileFirst Server to send the challenge.

- Create a `WLClient` instance and use the `connect` method to connect to the MobileFirst Server:

  ```
  final WLClient client = WLClient.createInstance(this);
  client.connect(new MyConnectionListener());
  ```

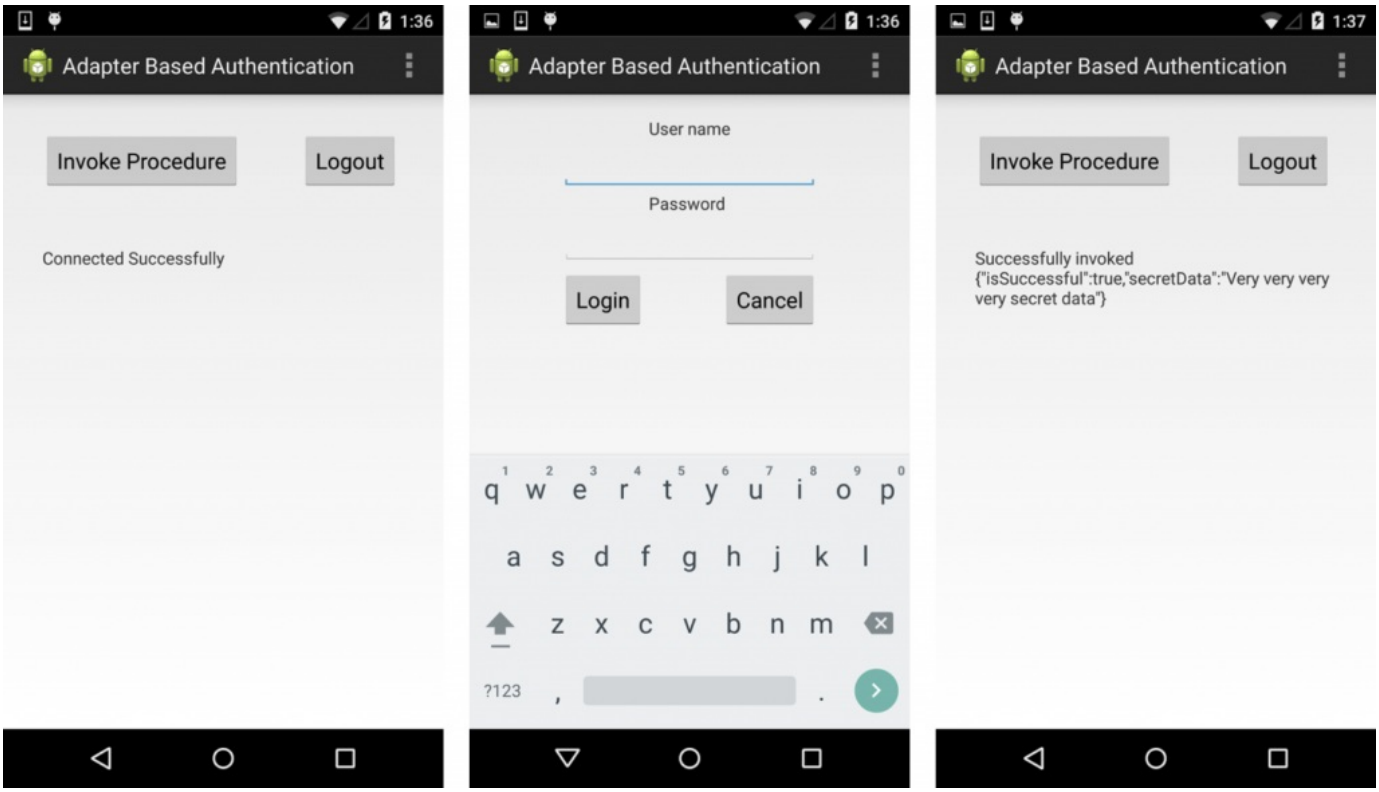
- In order to listen to incoming challenges, make sure to register the challenge handler by using the `registerChallengeHandler` method:

  ```
  challengeHandler = new AndroidChallengeHandler(this, realm);
  client.registerChallengeHandler(challengeHandler);
  ```

- Invoke the protected adapter procedure:

```
URI adapterPath = new URI("/adapters/AuthAdapter/getSecretData");
WLResourceRequest request = new WLResourceRequest(adapterPath,WLResourceRequest.GET);
request.send(new MyResponseListener());
```

## Sample application

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

- The `AdapterBasedAuth` project contains a MobileFirst native API that you can deploy to your MobileFirst server.
- The `AdapterBasedAuthAndroid` project contains a native Android application that uses a MobileFirst native API library.
- Make sure to update the `worklight.plist` file in the native project with the relevant server settings.