Form-based authentication in native iOS applications

Overview

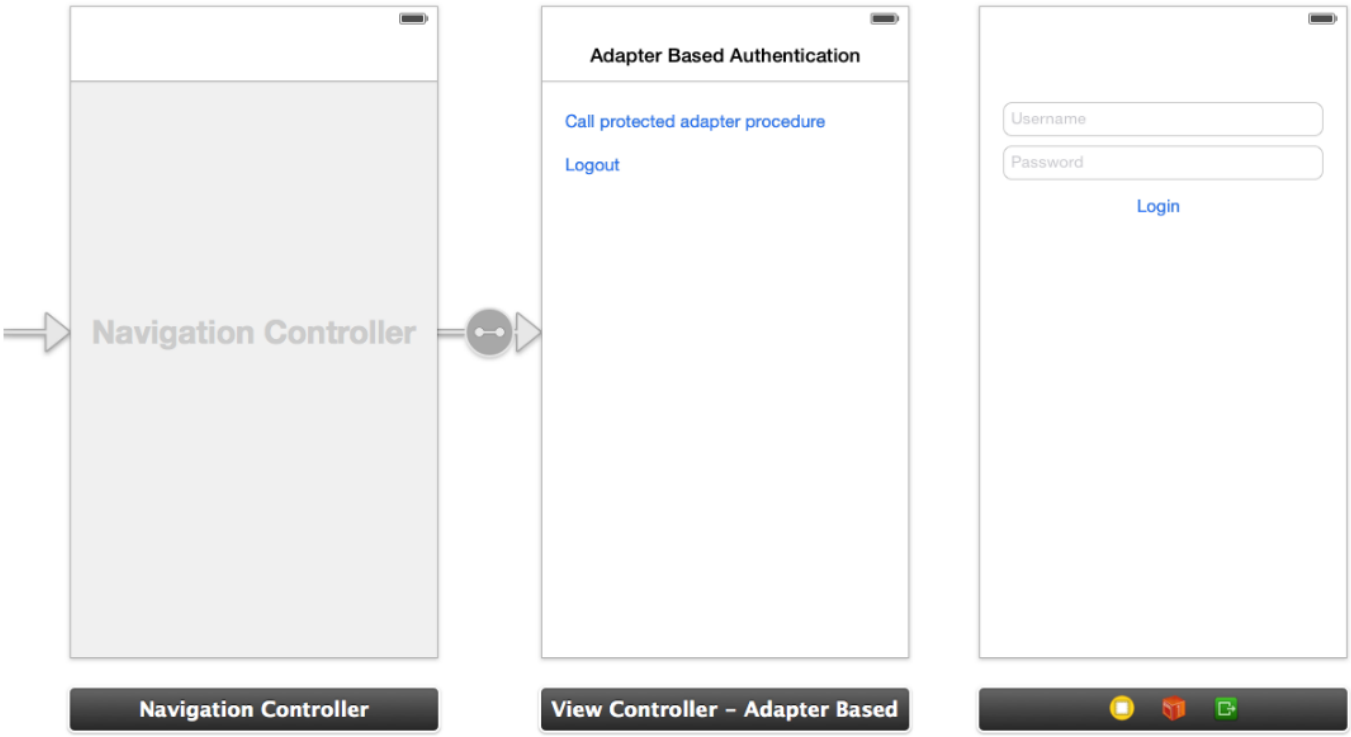
This tutorial explains how to implement the client-side of form-based authentication in native iOS.
Prerequisite: Make sure that you read the Form-based authentication (../) tutorial first.

Implementing the client-side authentication

Create a native iOS application and add the MobileFirst native APIs as explained in the Configuring a native iOS application with the MobileFirst Platform SDK (../../../hello-world/configuring-a-native-ios-application-with-the-mfp-sdk/) tutorial.

Storyboard

In your storyboard, add a View Controller containing a login form.



Challenge Handler

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

```
@interface MyChallengeHandler : ChallengeHandler
```

- Call the `initWithRealm` method:

```
@implementation MyChallengeHandler
//...
-(id)init:{
    self = [self initWithRealm:@"SampleAppRealm"]
;
    return self;
}
```

- 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`.
       The default login form that returns from the MobileFirst Server contains the `j_security_check` string. If the response contains the string, the challenge handler returns `true`.

       ```
       -(BOOL) isCustomResponse:(WLResponse *)response {
           if(response && response.responseText){
               if ([response.responseText rangeOfString:@"j_security_check" options:NSCaseIns
       ensitiveSearch].location != NSNotFound) {
                   NSLog(@"Detected j_security_check string - returns true");
                   return true;
               }
           }
           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.

       ```
       -(void) handleChallenge:(WLResponse *)response {
           NSLog(@"A login form should appear");
           LoginViewController* loginController = [self.vc.storyboard instantiateViewControllerWit
       hIdentifier:@"LoginViewController"];
           loginController.challengeHandler = self;
           [self.vc.navigationController pushViewController:loginController animated:YES];
       }
       ```

    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.

```
-(void) onSuccess:(WLResponse *)response {
    NSLog(@"Challenge succeeded");
    [self.vc.navigationController popViewControllerAnimated:YES]
;
    [self submitSuccess:response];
}
-(void) onFailure:(WLFailResponse *)response {
    NSLog(@"Challenge failed");
    [self submitFailure:response];
}
```

submitLoginForm

In your login View Controller, when the user taps to submit the credentials, call the `submitLoginForm` method to send the `j_security_check` string and the credentials to the MobileFirst Server.

```
@implementation LoginViewController
//...
- (IBAction)login:(id)sender {
    [self.challengeHandler submitLoginForm:@"j_security_check"
        requestParameters:@{@"j_username": self.username.text, @"j_password": self.password.text
}
        requestHeaders:nil
        requestTimeoutInMilliSeconds:0
        requestMethod:@"POST"];
}
@end
```

The Main ViewController

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:

  ```
  MyConnectListener *connectListener = [[MyConnectListener alloc] init];
  [[WLClient sharedInstance] wlConnectWithDelegate:connectListener];
  ```

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

```
[[WLClient sharedInstance] registerChallengeHandler:[[MyChallengeHandler alloc] initWithVie
wController:self] ];
```

- Invoke the protected adapter procedure:

```
NSURL* url = [NSURL URLWithString:@"/adapters/AuthAdapter/getSecretData"];
WLResourceRequest* request = [WLResourceRequest requestWithURL:url method:WLHttpM
ethodGet];
[request sendWithCompletionHandler:^(WLResponse *response, NSError *error) {
...
}];
```

Sample application

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/FormBasedAuth/tree/release71) the MobileFirst project.
Click to download (https://github.com/MobileFirst-Platform-Developer-Center/FormBasedAuthObjC/tree/release71) the Objective-C project.
Click to download (https://github.com/MobileFirst-Platform-Developer-Center/FormBasedAuthSwift/tree/release71) the Swift project.

- The `FormBasedAuth` project contains a MobileFirst native API that you can deploy to your MobileFirst server.
- The `FormBasedAuthObjC` and `FormBasedAuthSwift` projects contains a native iOS 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.