Adapter-based authentication

Overview

Adapter-based authentication enables you to implement the entire authentication logic, including validation of the credentials, in a JavaScript adapter.

This tutorial implements an adapter-based authentication mechanism that relies on a user name and a password.

This tutorial covers the following topics:

- Configuring the authenticationConfig.xml file
- Creating the server-side authentication components
- Protecting a Java adapter
- Protecting a JavaScript adapter
- Creating client-side authentication components

Configuring the authenticationConfig.xml file

Realm

Add an authentication realm to the `realm` section of the `authenticationConfig.xml` file.

```
<realm loginModule="AuthLoginModule" name="AuthRealm">
   <className>com.worklight.integration.auth.AdapterAuthenticator</className
>
   <parameter name="login-function" value="AuthAdapter.onAuthRequired"/>
   <parameter name="logout-function" value="AuthAdapter.onLogout"/>
</realm>
```

This realm uses the `AuthLoginModule` login module, which is defined in the LoginModule section.
Using the `com.worklight.integration.auth.AdapterAuthenticator` class means that the server-side part of the authenticator is defined in the adapter.
Whenever the MobileFirst authentication framework detects an attempt to access a protected resource, an adapter function that is defined in a `login-function` parameter is called automatically.
When logout is detected (explicit or session timeout), a `logout-function` is called automatically.
In both cases, the parameter value syntax is `adapterName.functionName`.

LoginModule

Add a login module to the `loginModules` section of the `authenticationConfig.xml` file and call it `AuthLoginModule`.

```
<loginModule name="AuthLoginModule">
   <className>com.worklight.core.auth.ext.NonValidatingLoginModule</className
>
</loginModule>
```

Using a `NonValidatingLoginModule` class name means that no additional validation is performed by the MobileFirst platform, and the developer takes responsibility for the validation of credentials within the adapter.
Because all authentication-related actions are done in the adapter code, using `NonValidatingLoginModule` is mandatory for adapter-based authentication.

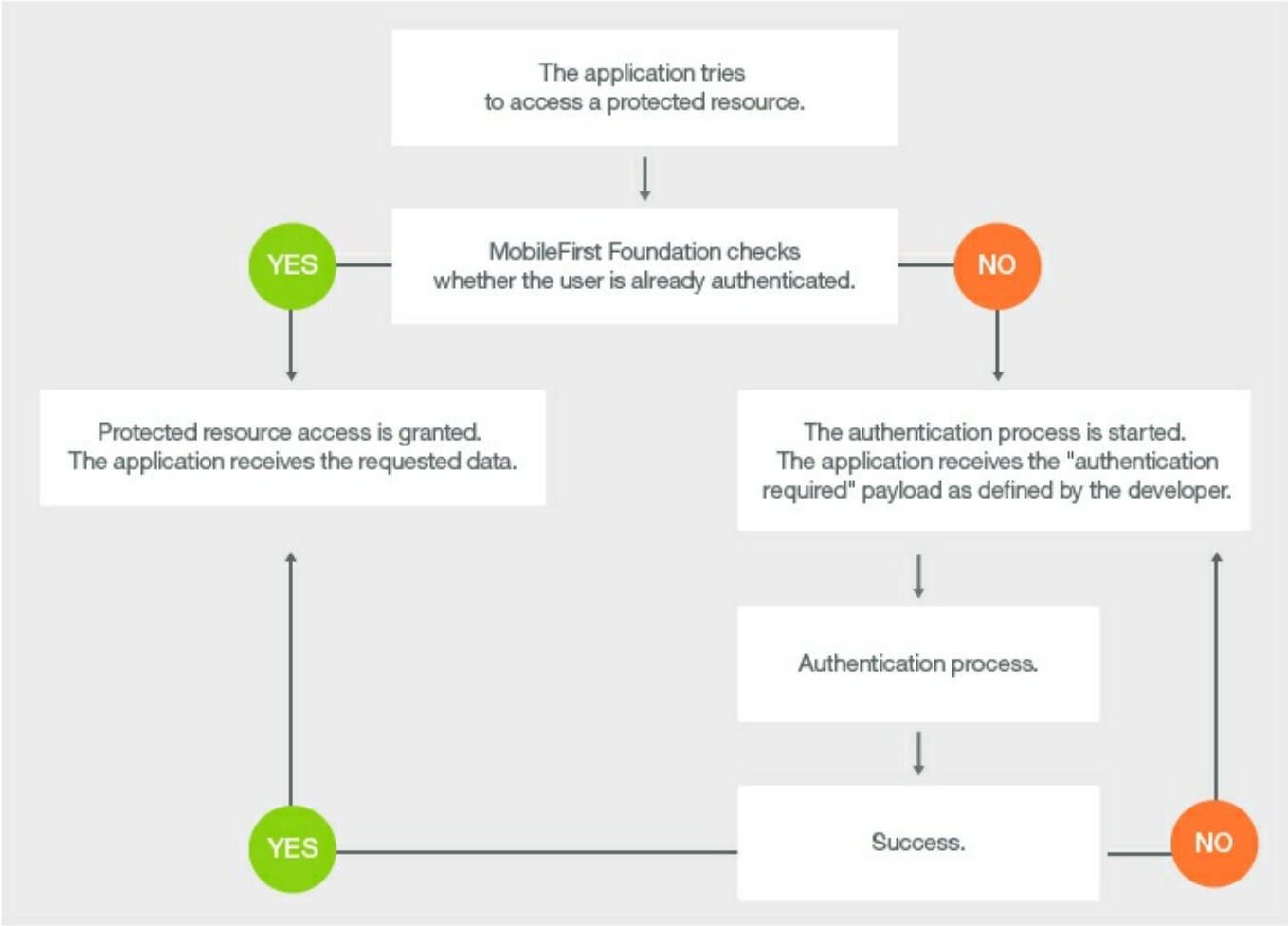
Security tests

Add security tests to the `securityTests` section of the `authenticationConfig.xml` file.
You must use this security test to protect the adapter procedure, so use the `customSecurityTest` element.
Remember the security test names. You must use them in subsequent steps.
Note: If you use Java adapters, this step is not required.

```
 <customSecurityTest name="AuthSecurityTest">
    <test isInternalUserID="true" realm="AuthRealm"/
>
 </customSecurityTest>
```

Creating the server-side authentication components

The following diagram illustrates the adapter-based authentication process:



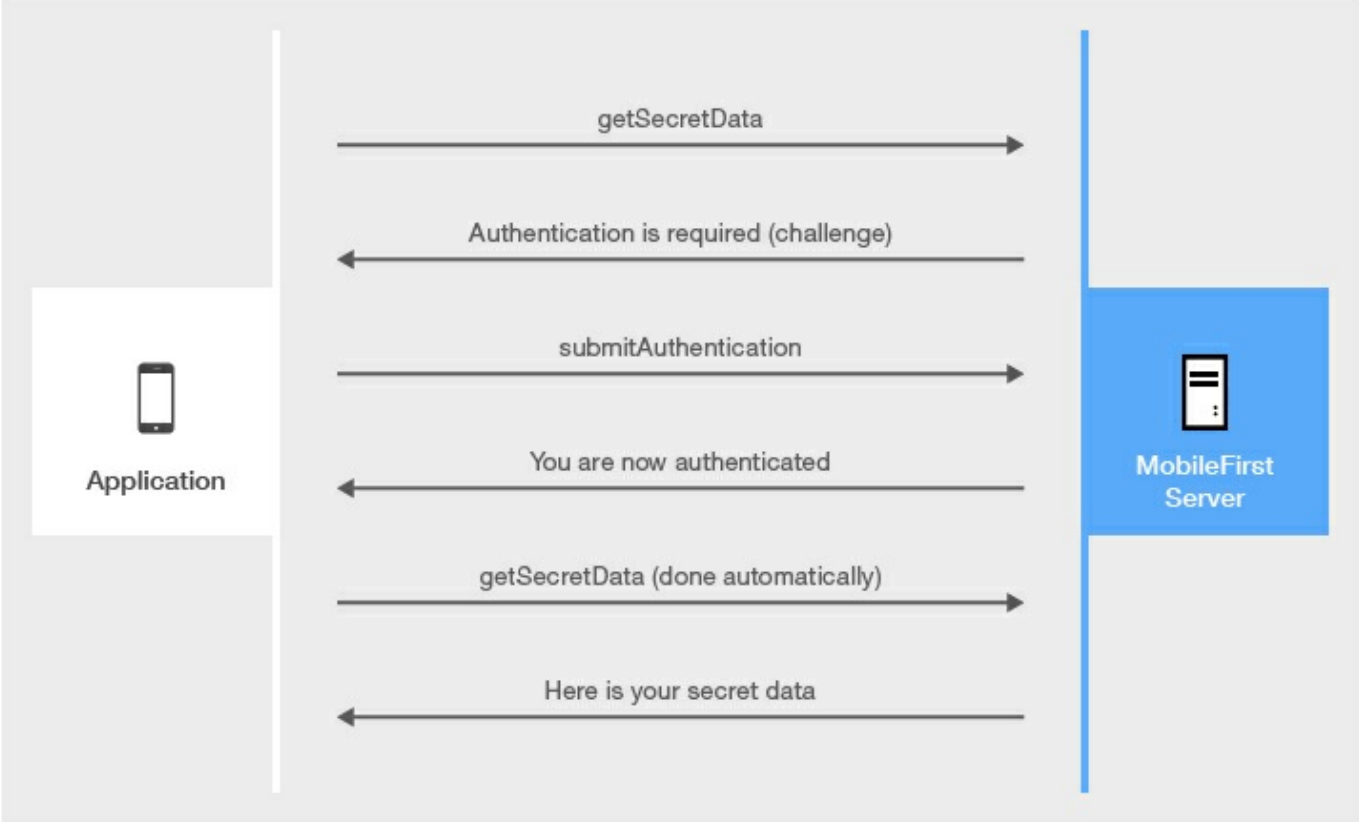
Adapter XML

Create an adapter that takes care of the authentication process. Name it `AuthAdapter`.
This adapter includes the following procedure:

```
<procedure name="submitAuthentication" securityTest="wl_unprotected"/>
```

The `submitAuthentication` procedure takes care of the authentication process. Note that when the challenge handler invokes the `submitAuthentication` call, it is responsible for handling all the possible responses. In particular, if the `submitAuthentication` call returns a challenge, it is passed to the invocation callback, and is not processed by the security framework.

Note: It is required for the "submit" procedure to be unprotected by using the `wl_unprotected` security test.
The following diagram shows the flow to implement:



onAuthRequired

Whenever the framework detects an unauthenticated attempt to access a protected resource, the `onAuthRequired` function is called, as defined in the `authenticationConfig.xml` file.

```
function onAuthRequired(headers, errorMessage){
    errorMessage = errorMessage ? errorMessage : null
;
    return {
        authRequired: true,
        errorMessage: errorMessage
    };
}
```

The returned object is a *custom* challenge object that is sent to the application.
This function receives the request headers and an optional `errorMessage` parameter. The object that is returned by this function is sent to the client application.
Note: In the sample, the `authRequired:true` property in the challenge handler detects that the server is requesting authentication.

submitAuthentication

The `submitAuthentication` function is called by a client application to validate the user name and password.

```
function submitAuthentication(username, password){
   if (username==="user" && password === "password"){
      var userIdentity = {
            userId: username,
            displayName: username,
            attributes: {
               foo: "bar"
            }
      };
      WL.Server.setActiveUser("AuthRealm", userIdentity);
      return {
         authRequired: false
      };
   }
   return onAuthRequired(null, "Invalid login credentials")
;
}
```

The user name and password are received from the application as parameters. In this sample, the credentials are validated against some hardcoded values, but any other validation mode is valid, for example by using SQL or web services.
If the validation passed successfully, the `WL.Server.setActiveUser` method is called to create an authenticated session for the `AuthRealm`, with user data stored in a `userIdentity` object. You can add your own custom properties to the user identity attributes.
An object (`{authRequired: false}`) is sent to the application, stating that the authentication screen is no longer required.
If credentials validation fails, an object that is built by the `onAuthRequired` function is returned to the application with a suitable error message.

onLogout

The `onLogout` function is defined in the `authenticationConfig.xml` file to be called automatically after a logout, for example to perform a cleanup. This step is optional.

```
function onLogout(){
   WL.Logger.debug("Logged out")
;
}
```

Protecting a Javascript adapter

Create a new adapter or add the following procedure to the XML code of your existing adapter:

```
<br />
<procedure name="getSecretData" securityTest="AuthSecurityTest"/><br />
```

For training purposes, the `getSecretData` function returns a hardcoded value. Keep in mind that `getSecretData` is protected by a security test, as defined in the adapter XML.

```
function getSecretData(){
   return {
      secretData: "Very very very very secret data"
   };
}
```

Protecting a Java adapter

The adapter responsible for authenticating the user needs to be a JavaScript-based adapter. However, you can still use adapter-based authentication to protect other resources such as Java adapters.

1. Create a Java adapter. Add a `getSecretData` method and protect it with the realm you created previously. In this module, the `getSecretData` procedure returns some hardcoded value:

   ```
   @GET
   @Produces("application/json")
   @OAuthSecurity(scope="AuthRealm")
   public JSONObject getSecretData(){
    JSONObject result = new JSONObject()
   ;
    result.put("secretData", "123456");
    return result;
   }
   ```

2. To set our new realm as the default user identity for the application, add this option in the application descriptor:

   ```
   <userIdentityRealms>AuthRealm</userIdentityRealms>
   ```

> To learn more about application descriptor properties, see the user documentation.

> Learn more in the "Implementing adapter-based authenticators" user documentation topic.