Using the MobileFirst Server to authenticate external resources

Overview

By using MobileFirst Server to authenticate external resources, you can use Single Sign-On (SSO) between IBM MobileFirst Platform Foundation and external services.

This feature protects those services through the MobileFirst Security Framework.

MobileFirst Server acts as an authorization server and issues an access token that can be validated by the external service.

Client applications request the access token from MobileFirst via token endpoint and send it to the external services.

The scope of the access token is a security test that is defined inside a MobileFirst project.

Each scope has a timeout property that determines the lifetime of the token. For example, if the token timeout is configured to be 30 seconds, the token remains valid for 30 seconds after MobileFirst Server has issued it. After that time, the token is rejected by the external service and a new token must be requested and issued.

- MobileFirst authentication by using an access token over OAuth 2.0
- MobileFirst project configuration
- External service configuration
- Use of the client-side API

MobileFirst authentication by using an access token over OAuth 2.0

The OAuth 2.0 authorization framework enables an application from independent software vendors (also called third-party application) to obtain limited access to an HTTP service.

The implementation uses three roles of the OAuth protocol:

- **Resource Server**: third-party server
  The server that hosts the protected resources. It can accept, and respond to, protected resource requests by using access tokens.
- **Client**: app
  An application that requests protected resources.
- **Authorization Server**: MobileFirst Server
  The server that issues access tokens to the client after it has successfully authenticated the resource owner and obtained authorization.

Overview of the Resource Server component

The **Resource Server** is an external server that hosts the available resource.

One use-case is that of services that are deployed on a cloud, such as **MbaaS**. But this flow is not restricted to that case and works with any third- party server.

The **Token lib** library is provided for use with the **Resource Server**.

The public key that is necessary to verify the token must be configured for this component.

Java and `node.js` libraries are provided for offline validation.

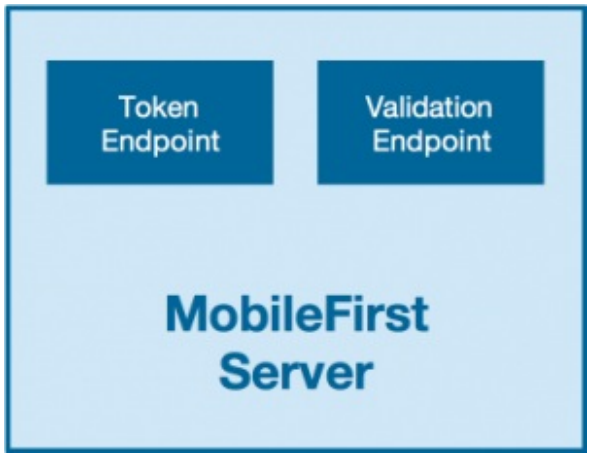
Overview of the client component

The **External SDK** is used to access the resource server. It must be able to attach a header to the request.

The **MobileFirst SDK** exposes the following API for both hybrid and native code:

1. Request the access token from **MobileFirst Server**.
2. Get the last access token (local).
3. Analyze **Resource server** error response to obtain the required scope.

Overview of the MobileFirst Server component



The **Validation Endpoint** is a REST API exposed under the path `/oauth/validation.s`

It can be used by **Resource Server** or by the reverse proxy for online validation.

The **Token Endpoint** is a REST API exposed under the path `/oauth/token`.

MobileFirst Server uses the authentication infrastructure to issue an access token for the requested scope (MobileFirst security test).

Overview of token format

```
{
   version: "1.0 (of token)",
   scope: "<the security test that the token authenticated>",
   expiration: "<time in msec since epoch>",
   data: {
      user_id: "<authenticated user, for example, shachor@il.ibm.com>"
,
      device_id: "<device id as known by MobileFirst server>",
      application_id: "<identity of the app>"
   }
}
```

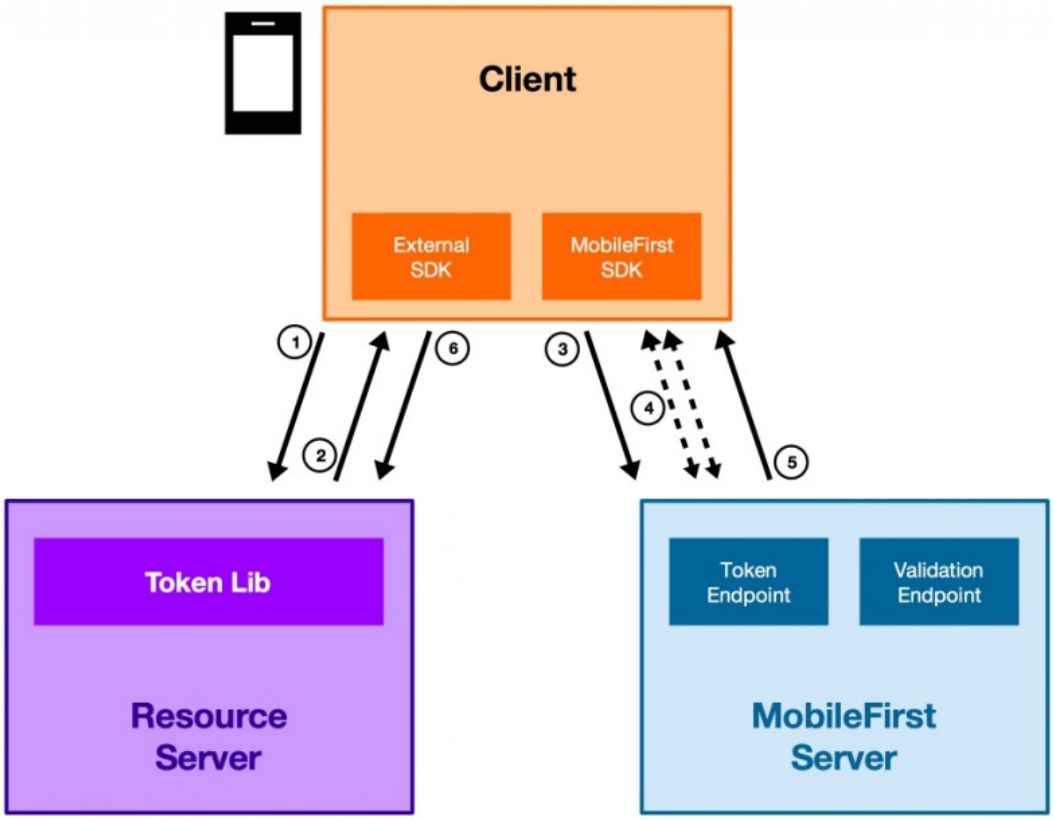
The token is signed by the MobileFirst Server instance.

Clarification for the `data` field:

- The `user_id` field is added only if the security test has a user realm.
- The `device_id` field is added only if the security test has a device realm.
- The `application_id` field is **always** added

Quick Flow:

The client requests a token from the authorization server, receives it, and with that token, accesses the protected resource on the resource server.



1. **The application developer attempts to access a protected resource on a remote server.**

   In case the user already has a previous token, the user adds an **"Authorization"** header with the token as value.

   > **NOTE:**
   > The flow that is demonstrated uses a provided **Token lib**. Another possibility is to use a custom filter with the **Validation Endpoint**.

2. **If the request does not have a token, or the token is not valid, Resource Server returns 401/403 via the supplied lib.**

   The response has the **"WWW- Authenticate"** header with the following format:

   Bearer scope = ""

   The is the security test that protects the resource.

3. **The application developer parses the response and obtains the access token by using MobileFirst SDK.**

   The **MobileFirst SDK** is called with the optional scope parameter.

   If the scope is not specified, the security test of the application is used.

4. **If some realms are not authenticated yet, MobileFirst Server sends challenges back to the client and the client responds.**

   Challenges have the same flow as the standard authentication.

The scope is a two-step security test:

1. `wl_authenticityRealm`
2. `wl_deviceAutoProvisioningRealm`

5. **After all relevant realms are authenticated, MobileFirst Server returns an access token to the client.**

   The **MobileFirst Server** returns a JSON object with the following format:

```
{
   scope: "<actual scope *>",
   token_type: "Bearer",
   access_token: "<encrypted token string>
"
}
```

* different than given scope only if default scope requested

6. **The application developer now resends the original request, adding an "Authorization" header with the retrieved token.**

   After it has validated, the **Token lib** updates the client Context object with the following data from the access token: `app_id`, `user_id`, `device_id`

   The **Token lib validates** the token in this order:

   1. Check signature.
   2. Check expiration.
   3. Check scope.

MobileFirst project configuration - Configuring the scope for the access token

The scope of an access token must be a predefined security test in a MobileFirst project. You configure it in this file: `/server/conf/authenticationConfig.xml`.

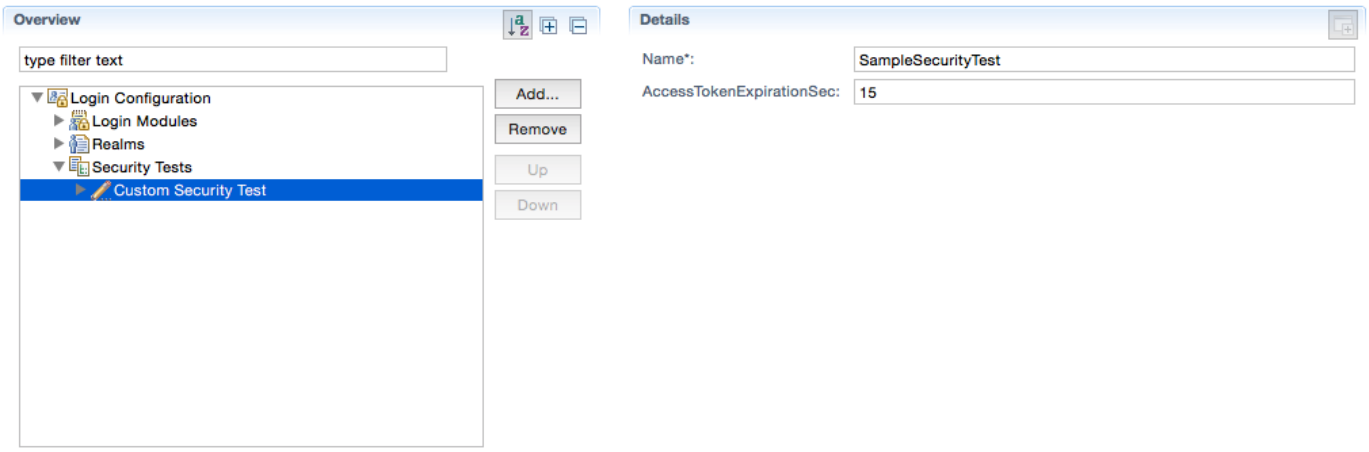
The default lifetime for each token is 60 seconds. You can override this timeout by adding the AccessTokenExpirationSec attribute to the security test.

For example, if you want to configure a security test, called SampleSecurityTest, with a lifetime of 15 seconds, edit the `authenticationConfig.xml` as follows

From the source view:

```
<securityTests>
  <customSecurityTest name="SampleSecurityTest" AccessTokenExpirationSec="15"
>
    <test realm="SampleRealm" isInternalUserID="true"/>
  </customSecurityTest>
</securityTests>
```

From the design view:



MobileFirst project configuration - Keystore

To use this feature, preferably use or create your own keystore and configure the MobileFirst Server to use it.

For an example in an unrelated context, see Configuring device auto provisioning in the product documentation.

Using the default MobileFirst Server keystore is **NOT** secure!

External service configuration

For your external service to accept the access token, you must add a validation library to your service, which must be able to validate the token with either online or offline validation.

Two libraries are provided for this purpose:

- `worklight-access-token-validator.jar` - Java lib
- `worklight-access-token-validator.tgz` - node.js module

For MobileFirst Server installation - You can find the libraries in: `/MobileFirstServer/external-server-libraries`

For the MobileFirst Studio - When you create a new project, you can find the libraries in: `/externalServerLibraries`

External service configuration - Using Java

The purpose of this module is to enable offline validation of access tokens that are generated by MobileFirst Server for Java web applications.

Validation of access tokens that are generated by MobileFirst Server is also possible for `node.js` servers.

To use the Java library, you need two files:

- `Certificate` - For the associated sample, the certificate that has been exported from the MobileFirst Server Keystore. You can use the Java keytool. In production, preferably use your own keystore as explained in MobileFirst Project Configuration - Keystore.
- `worklight-access-token-validator.jar`

External service configuration - Using Java: servlet filter

Add the `worklight-access-token-validator.jar` file to the class path of your web application and use the filter class `com.worklight.security.WLAccessTokenValidationFilter` as shown below.

Example of a filter definition:

- **Filter name** = `FilterName` "Choose an name you want for the filter"
- **URL** = `/some/protected/url` "The prefix for all the resources that you wish to protect"
- **Scope [Optional]** = `securityTestName` "The name of the security test, as defined in the authenticationConfig.xml file, which is needed to authenticate against to gain access to the protected resources. If the scope is not specified, the filter accepts any valid token that is provided by MobileFirst Server."
- **CertificatePath** = `certificateLib/WorklightServerCertificate.cert` "The path to the certificate of MobileFirst Server, relative to the WEB-INF folder."

In addition to the previous parameters, write the following code for the web.xml file of your external server:

```
<web-app ...>
  ...
  <filter>
      <filter-name>FilterName</filter-name>
      <filter-class>com.worklight.security.WLAccessTokenValidationFilter</filter-class
>
      <init-param>
         <param-name>worklightCertificateFile</param-name>
         <param-value>certificateLib/WorklightServerCertificate.cert</param-value>
      </init-param>
      <init-param>
         <param-name>scope</param-name>
         <param-value>securityTestName</param-value>
      </init-param>
  </filter>
  <filter-mapping>
      <filter-name>FilterName</filter-name>
      <url-pattern>/some/protected/url</url-pattern>
  </filter-mapping>
  ...
</web-app>
```

After successful validation, the filter updates the `ClientContext` object that the service can use to access user, application, or device identities that are contained in the access token.

Example of `ClientContext` usage:

```
ClientContext context = ClientContext.getInstance();
String appId = context.getApplication();
String userId = context.getUser();
String deviceId = context.getDevice();
```

Use of the client-side API: which methods are allowed

The MobileFirst `WL.Client` API provides built-in support for using MobileFirst access tokens for the following platforms:

- Hybrid - JavaScript™
- Android
- iOS

The methods included in this API provide the following services:

- Obtaining and caching a token for a specified scope
- Getting the last obtained access token
- Getting the required scope from the external service response

For more information, see also Using SSO between IBM MobileFirst Platform Foundation and external services in the product documentation.

Obtaining and caching a token for a specified scope

In JavaScript, the method is called **obtainAccessToken**.

The `WL.Client` instance requests a new token from the MobileFirst Server instance. To obtain the token, the client must be authenticated in all realms of the requested scope (which is represented by a security test in the MobileFirst Server `AuthenticationConfig.xml` configuration file). Thus, calling this method triggers an authentication sequence for all realms for which authentication is still required.

This method is asynchronous in all platforms. It does not return a value but, instead, triggers a response handler.

> **NOTE:** It is not necessary to parse the response from the server in the response handler. The token is automatically parsed and cached inside the `WL.Client` instance and can be retrieved by using the next method.

Getting the last obtained access token

In JavaScript, this method is called **getLastAccessToken**.

The `WL.Client` instance returns the last access token for a certain scope as a string. Alternatively, if no scope is specified, the last obtained token is returned. This capability is useful when an application is using one scope **only**.

Add a header named "Authorization" and for the header content, add "Bearer", followed by the token. For example, when issuing an Ajax request, you can write the following code:

```
var token = WL.Client.getLastAccessToken();
$.ajax({
    type: "GET",
    url: MY_URL,
    headers: {
        "Authorization": "Bearer " + token
    }
});
```

Getting the required scope from the external service response

In JavaScript, this method is called **getRequiredAccessTokenScope**.

When a request to the external service fails, the `WL.Client` instance can identify the cause of the failure.

If the failure is related to access token issues, the client returns the name of the scope that is required to access the service. In this case, it is necessary to obtain a new token for the returned scope. For example, the token does not match the required scope, or the token has expired.

If the error is not related to access token issues, the method returns `null`.

Use of the client-side API: JavaScript example

This JavaScript example shows how to use the client-side API for access to an external service.

```
function callProtectedRestAPI(retries) {
    // You must be able to call this method
    // recursively, because in some cases it is
    // necessary to obtain a new token and try a
    // second time.
    if (retries == 0) {
        return;
    }
    // Get the last obtained access token.
    // On the first call, the token can be null.
    var token = WL.Client.getLastAccessToken();
    var headersObject = (token != null) ? {
        "Authorization": "Bearer " + token
    } : {};
    $.ajax({
        type: "GET",
        url: MY_EXTERNAL_SERVER_URL,
        headers: headersObject
    }).done(function(response) {
        showResult(response);
    }).fail(function(response) {
        // Need to extract this header from
        // the response to know the scope.
        var header = response.getResponseHeader("WWW- Authenticate");
        var scope = WL.Client.getRequiredAccessTokenScope(response.status, header)
;
        if (scope != null) {
            // The failure is related to the access token. Get a new one.
            WL.Client.obtainAccessToken(scope, getTokenSuccess, getTokenFailure);
        } else {
            showErrorResult("request failed");
        }
    });
    function getTokenSuccess(response) {
        // Obtained a token. Try to access the external server one more time.
        callProtectedRestAPI(retries - 1);
    };
    function getTokenFailure(response) {
        showErrorResult(response);
    };
}
```

Sample application

Click to download (http://public.dhe.ibm.com/software/products/en/MobileFirstPlatform/docs/v630/MobileFirstAsAuthorizationServer.zip) the Studio project.