# Device Enrollment

fork and edit tutorial (https://github.ibm.com/MFPSamples/DevCenter/tree/master/tutorials/en/foundation/7.0/advanced-topics/device-enrollment.html) | report issue (https://github.ibm.com/MFPSamples/DevCenter/issues/new)

## Overview

Device enrollment consists of registering a specific device as a trusted device from a customer's perspective. From an enrolled device, a customer might want to see some basic information without needing to log in, more secure information would require a PIN code, while changes to the account might require full credentials.

**Prequisite:** This advanced tutorial requires good understanding of IBM MobileFirst Platform Foundation security concepts such as security tests, realms, challenge handlers, etc. Review the tutorials and IBM knowledge center if you are not comfortable with those concepts.

This tutorial covers the following topics:

- Example flow
- Username/password security test
- Enrollment
- PIN code protection
- Sample application

## Example flow

Here is the application flow for this example:

1. When the application is opened, it sends a request to the server to check whether this device is currently enrolled.
2. If the device is not recognized, the enrollment process is started.
3. To enroll, the user first needs to log in with standard credentials.
4. After authentication, the user is prompted to set a PIN code for quick access to sensitive information.
5. After successful enrollment, the user has access to the bank account balance.
6. Closing the application and reopening it shows the balance page immediately without the user having to log in again.
7. For access to a button to see detailed transactions, the user must enter the defined PIN code each time.
8. By using a Logout button, the user can start everything from the beginning.

## Username/password security test

For a username/password security test, the sample application uses a simple adapter-based realm.

### Login module

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

This simple `NonValidatingLoginModule` module lets the adapter-based authenticator handle the validation.

## Realm

```
<realm name="UsernamePasswordRealm" loginModule="UsernamePasswordModule">
   <className>com.worklight.integration.auth.AdapterAuthenticator</className>
   <parameter name="login-function" value="MyAdapter.UsernamePasswordRequired" /
>
</realm>
```

By using `AdapterAuthenticator`, you can write the login logic in an adapter. The `login-function` parameter is used to trigger a challenge.

## Security test

```
<customSecurityTest name="UsernamePasswordAuthentication">
   <test realm="wl_directUpdateRealm" step="1" />
   <test isInternalDeviceID="true" realm="wl_deviceNoProvisioningRealm" step="1" /
>
   <test realm="wl_antiXSRFRealm" step="1" />
   <test isInternalUserID="true" realm="UsernamePasswordRealm" step="1" />
</customSecurityTest>
```

`UsernamePasswordRealm` is used as the internal user ID, while the built-in `wl_deviceNoProvisioningRealm` helps get the device ID automatically from the device. The direct update and anti-XSRF realms are additional internal realms that are not specific to this scenario.

## Adapter code

```
function UsernamePasswordRequired() {
   return {
      usernamePasswordRequired: true,
   };
}
```

This procedure notifies the client that the given resource requires username/password authentication. You

must implement a challenge handler to react to this flag.

```
function verifyUsernamePassword(username, password) {
    if (username && password && username == password) {
        WL.Server.setActiveUser("UsernamePasswordRealm", {
            'userId': username
        });
        return {
            success: true
        };
    } else {
        return {
            success: false,
            usernamePasswordRequired: true,
            error: 'username and password must be the same and not empty'
        };
    }
}
```

This procedure checks the validity of the user name and password and, if they are validated, sets the current user in the session. In this example, a "correct" username/password combination is any 2 strings that are exactly the same. In the real world, of course, the combination is verified by a database or a back-end service.

## Challenge handler

```
var usernamePasswordRealm = WL.Client.createChallengeHandler("UsernamePasswordRealm");
usernamePasswordRealm.isCustomResponse = function(response) {
    return response && response.responseJSON &&
response.responseJSON.usernamePasswordRequired;
};
```

The challenge handler needs to be registered. The `usernamePasswordRequired` check in the response determines whether the challenge needs to be handled.

```
usernamePasswordRealm.handleChallenge = function(response) {
    $('.section').hide();
    $('#loginBody').show();
    if (response.responseJSON.error) {
        alert(response.responseJSON.error);
    }
};
```

If the challenge needs to be handled, simply show an HTML login form (and show errors if any). The submit button of this form is linked to the `login()` function below.

```
function login() {
    usernamePasswordRealm.submitAdapterAuthentication(
{
        adapter: 'MyAdapter',
        procedure: 'verifyUsernamePassword',
        parameters: [$('#username').val(), $('#password').val()
]
    }, {
        onSuccess: function(response) {
            $('.section').hide();
            usernamePasswordRealm.submitSuccess();
        }
    });
}
```

The `login` function uses the `submitAdapterAuthentication` API (which is the same as the `invokeProcedure` API) to send the entered user name and password for verification against the adapter. Upon success, use `submitSuccess` to let the application continue its flow.

# Enrollment

Enrolled devices need to be saved in a database of trusted devices that are associated with their user account.

## Login module

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

This non-validating login module leaves the responsibility of validating the data to you.

## Realm

```
<realm name="EnrollmentRealm" loginModule="EnrollmentModule">
    <className>com.worklight.integration.auth.AdapterAuthenticator</className
>
    <parameter name="login-function" value="MyAdapter.enrollmentRequired" />
 </realm>
```

Again, an `AdapterAuthenticator` object is used to check the enrollment.

In the adapter, this procedure is defined to let the client know that a given resource requires enrollment.

```
function enrollmentRequired() {
    return {
        enrollmentRequired: true
    };
}
```

## Security test

```
<customSecurityTest name="EnrolledDevice">
    <test realm="wl_directUpdateRealm" step="1" />
    <test isInternalDeviceID="true" realm="wl_deviceNoProvisioningRealm" step="1" /
>
    <test realm="wl_antiXSRFRealm" step="1" />
    <test isInternalUserID="true" realm="EnrollmentRealm" step="1" />
 </customSecurityTest>
```

Similar to the previous security test, this security test uses some predefined realms in addition to our `EnrollmentRealm` realm.

## Storage

In your application, you might want to use an external database or a back-end service to safely store the list of enrolled devices. This sample includes a Java class that emulates storage of JSON objects for a given ID string. The implementation of this class is not important and its public API features 3 methods: `getInfo(id)`, `setInfo(id, info)`, `removeInfo(id)`, where `info` is an arbitrary JSON object.

In the adapter code, a reference to this class is declared:

```
var enrolledDevices = com.ibm.sample.EnrolledDeviceStore;
```

## Enroll device

```
function enroll(pin) {
    var deviceId = WL.Server.getCurrentDeviceIdentity().deviceId
;
    var userId = WL.Server.getCurrentUserIdentity().userId;
    enrolledDevices.setInfo(deviceId, {
        'userId': userId,
        'pin': pin
    });
    WL.Server.setActiveUser("EnrollmentRealm", {
        'userId': userId
    });
    return {
        'success': true
    };
}
```

For a device to be enrolled, the current device ID and the currently logged-in user need to be retrieved. Together with a PIN code provided by the user, this information is saved in the enrolled devices storage.

Because only logged-in users can enroll, this procedure is protected by the username/password realm.

```
<procedure name="getTransactions" securityTest="PinCode"/>
```

## Check enrollment

```
function checkEnrollment() {
    var deviceId = WL.Server.getCurrentDeviceIdentity().deviceId
;
    var enrollInfo = enrolledDevices.getInfo(deviceId);
    if (enrollInfo) {
        //Clean up if any
        WL.Server.setActiveUser("EnrollmentRealm", null);
        //Log into the enrollment realm
        WL.Server.setActiveUser("EnrollmentRealm", {
            'userId': enrollInfo.userId
        });<br />
    }
}
```

This adapter procedure retrieves the current device ID and checks it against the enrolled list. If the device is enrolled, the user is logged in to the enrollment realm.

## Remove device

To make testing easier, a procedure is added so that you can remove the current device from the list of devices and log out from all realms.

```
function removeDevice() {
    var deviceId = WL.Server.getCurrentDeviceIdentity().deviceId
;
    enrolledDevices.removeInfo(deviceId);
    WL.Server.setActiveUser(enrollmentRealm, null);
    WL.Server.setActiveUser(pinCodeRealm, null);
    WL.Server.setActiveUser(userPwdRealm, null);
}
```

## Pre-emptive login and enrollment

While you could wait for a challenge, you can also write client logic to pre-emptively trigger a login or enrollment.
In this sample, when the application starts, the following method is called.

```
function checkEnrollment() {
    if (WL.Client.isUserAuthenticated("EnrollmentRealm")) {
        getBalance();
    } else {
        WL.Client.invokeProcedure({
            adapter: 'MyAdapter',
            procedure: 'checkEnrollment'
        }, {
            onSuccess: function(result) {
                busyInd.hide();
                if (WL.Client.isUserAuthenticated("EnrollmentRealm"))
{
                    getBalance();
                } else {
                    WL.Client.login("UsernamePasswordRealm", {
                        onSuccess: function() {
                            $('.section').hide();
                            $('#enrollBody').show();
                        }
                    });
                }
            },
            onFailure: function(error) {
                console.log(error);
            }
        });
    }
}
```

This code checks whether the device is already enrolled. If it is, the account balance is shown. If the device is not enrolled, the code triggers the username/password authentication. Upon success, it shows the actual enrollment form, which requires to set up a PIN code.
Submitting this form triggers the `setPinCode()` function, then allows the user to see the balance.

```
function setPinCode() {
    if ($('#newPin').val() != "" && $('#newPin').val() == $('#repeatPin').val())
{
        WL.Client.invokeProcedure({
            adapter: 'MyAdapter',
            procedure: 'enroll',
            parameters: [$('#newPin').val()]
        }, {
            onSuccess: function(response) {
                $('.section').hide();
                getBalance();
            }
        });
    } else {
        alert("Please enter the PIN code twice");
    }
}
```

Because device enrollment is triggered pre-emptively by the client, the challenge handler for the `EnrollmentRealm` should not occur. To catch a rare case, a simple challenge handler could reload the application when it happens.

```
var enrollmentRealm = WL.Client.createChallengeHandler("EnrollmentRealm");
enrollmentRealm.isCustomResponse = function(response) {
    return response && response.responseJSON && response.responseJSON.enrollmentRequired;
<
};
enrollmentRealm.handleChallenge = function(response) {
    //This should not happen, we enroll pre-emptively
    WL.Client.reloadApp();
}
```

# PIN code protection

Some resources can be protected with the PIN code. In this sample, the `getTransactions` procedure is protected by a PIN code.

```
<procedure name="getTransactions" securityTest="PinCode"/>
```

## Login module

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

## Realm

```
<realm name="PinCodeRealm" loginModule="PinCodeModule">
    <className>com.worklight.integration.auth.AdapterAuthenticator</className
>
    <parameter name="login-function" value="MyAdapter.pinCodeRequired" />
 </realm>
```

```
function pinCodeRequired() {
    return {
        pinCodeRequired: true
    };
}
```

## Security test

```
<customSecurityTest name="PinCode">
    <test realm="wl_directUpdateRealm" step="1" />
    <test isInternalDeviceID="true" realm="wl_deviceNoProvisioningRealm" step="1" /
>
    <test realm="wl_antiXSRFRealm" step="1" />
    <test realm="EnrollmentRealm" step="1" />
    <test isInternalUserID="true" realm="PinCodeRealm" step="2" />
</customSecurityTest>
```

Accessing a PIN-protected resource requires both `PinCodeRealm` and `EnrollmentRealm`.

## Challenge handler

```
var pinCodeRealm = WL.Client.createChallengeHandler("PinCodeRealm");
pinCodeRealm.isCustomResponse = function(response) {
    return response && response.responseJSON && response.responseJSON.pinCodeRequired
;
};
pinCodeRealm.handleChallenge = function(response) {
    $('.section').hide();
    $('#checkPinCode').show();
    if (response.responseJSON.error) {
        alert(response.responseJSON.error);
    }
};
```

When a resource is protected by a `PinCodeRealm` realm, the challenge handler is called. It shows a form that prompts the user to enter the valid PIN code. Clicking the button triggers the `verifyPinCode` function.

```
function verifyPinCode() {
    pinCodeRealm.submitAdapterAuthentication(
{
        adapter: 'MyAdapter',
        procedure: 'verifyPinCode',
        parameters: [$('#pin').val()]
    }, {
        onSuccess: function(response) {
            pinCodeRealm.submitSuccess();
        },
        onFailure: function(response) {
            alert("error");
        }
    });
}
```

This function sends the PIN code attempt to the adapter for verification.

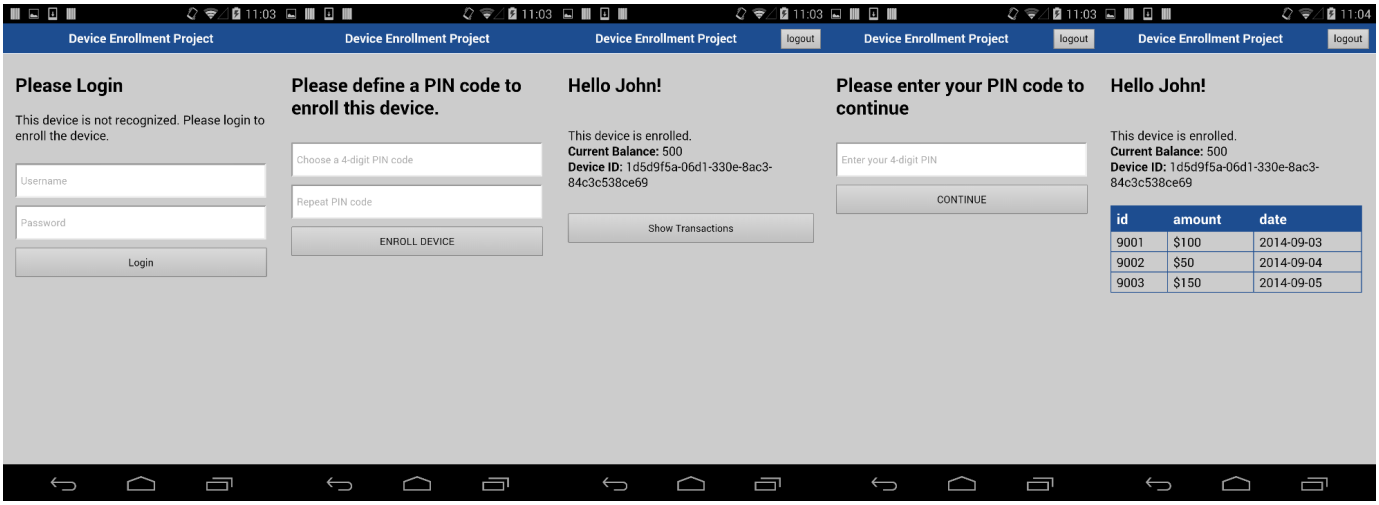
```
function verifyPinCode(pin) {
    var userId = WL.Server.getCurrentUserIdentity().userId;
    var deviceId = WL.Server.getCurrentDeviceIdentity().deviceId
;
    var enrollInfo = enrolledDevices.getInfo(deviceId);
    if (enrollInfo && enrollInfo.pin == pin) {
        WL.Server.setActiveUser("PinCodeRealm", null);
        WL.Server.setActiveUser("PinCodeRealm", {
            'userId': userId
        });
        return {
            success: true
        };
    }

    return {
        success: false,
        pinCodeRequired: true,
        error: "Wrong PIN Code"
    };
}
```

This procedure retrieves the enrollment information from the enrolled devices list and compares the set PIN code. A successful procedure means that the device was found in the list and that the PIN that was entered matches the stored one. When the procedure is successful, the state is saved in the session.

# Sample application

Click to download (http://public.dhe.ibm.com/software/products/en/MobileFirstPlatform/docs/v700/DeviceEnrollmentProject.zip) the Studio project.


11:03
Device Enrollment Project
Please Login
This device is not recognized. Please login to enroll the device.
Username
Password
Login
11:03
Device Enrollment Project
Please define a PIN code to enroll this device.
Choose a 4-digit PIN code
Repeat PIN code
ENROLL DEVICE
11:03
Device Enrollment Project
logout
Hello John!
This device is enrolled.
Current Balance: 500
Device ID: 1d5d9f5a-06d1-330e-8ac3-84c3c538ce69
Show Transactions
11:03
Device Enrollment Project
logout
Please enter your PIN code to continue
Enter your 4-digit PIN
CONTINUE
11:04
Device Enrollment Project
logout
Hello John!
This device is enrolled.
Current Balance: 500
Device ID: 1d5d9f5a-06d1-330e-8ac3-84c3c538ce69
id amount date
9001 $100 2014-09-03
9002 $50 2014-09-04
9003 $150 2014-09-05