Custom device provisioning

Overview

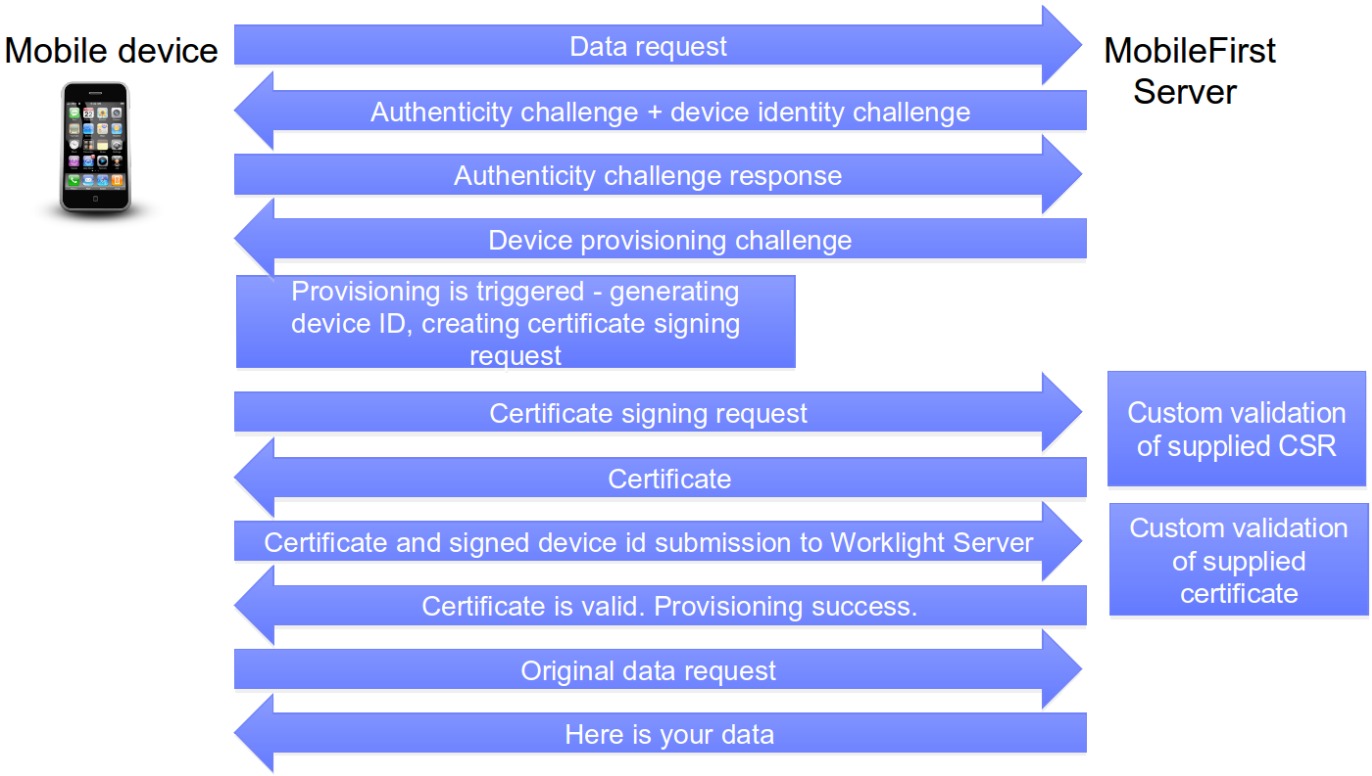
In this tutorial, enabling and configuring custom device provisioning will be covered.
Custom device provisioning is an extension of auto device provisioning.
With Custom device provisioning you can validate:

- Certificate Signing Request during initial provisioning flow.
- Certificate during every application start.

It is vital to gain a solid understanding of the topics that are discussed in the Device Provisioning Concepts tutorial (../../authentication-security/device-provisioning-concepts/) because this tutorial is fully based on those concepts.

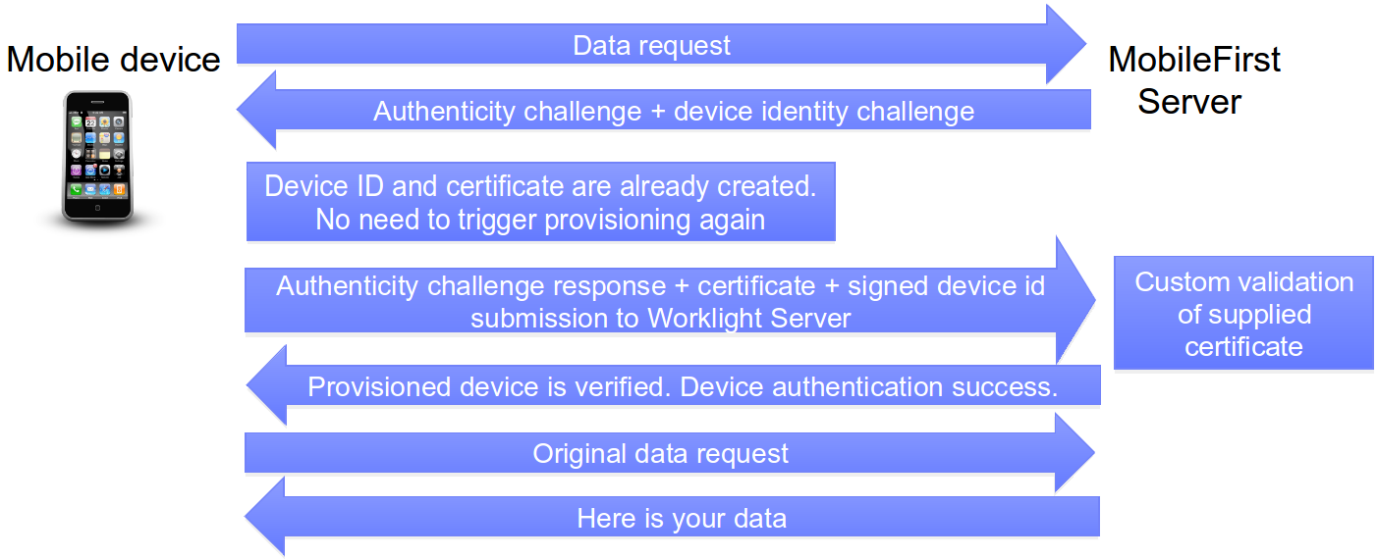
Understanding custom device provisioning

First Application Start



Authenticity check is a proprietary IBM MobileFirst Platform Foundation technology that makes sure application is the authentic one and was not modified by anyone.
The main difference between auto and custom provisioning is two functions that you can perform custom validation of CSR during the provisioning process and custom validation of certificate during each application start.

Subsequent Application Starts



keystore

By default, the MobileFirst Server uses its internal keystore to issue a certificate.
You can tell the server to use your own keystore by adjusting the **worklight.properties** file.
Note: The *wl.ca.keystore.path* property value can be either relative to the */server/* folder of the MobileFirst project or absolute to the file system.

```
##############################################################################################################
#    Worklight Default Certificate (For device provisioning)
##############################################################################################################
# You can change the default behavior with regard to CA certificates. You can also implement custom provisioning.
# If you want to change the auto-provisioning mechanism to use different granularity (application, device or group) o
r a
# different list of pre-required realms, you can create your own customized authenticator, login module and challeng
e handler.
# For more information, see the "Custom Authenticator and Login Module" Getting Started training module.
#The path to the keystore, relative to the server folder in the Worklight Project, for example: conf/my-cert.jks
#wl.ca.keystore.path=
#The type of the keystore file. Valid values are jks or pkcs12.
#wl.ca.keystore.type=
#The password to the keystore file.
#wl.ca.keystore.password=
#The alias of the entry where the private key and certificate are stored, in the keystore.
#wl.ca.key.alias=
#The password to the alias in the keystore.
#wl.ca.key.alias.password=
##############################################################################################################
#    Worklight SSL keystore
##############################################################################################################
#SSL certificate keystore location.
ssl.keystore.path=conf/default.keystore
#SSL certificate keystore type (jks or PKCS12)
ssl.keystore.type=jks
#SSL certificate keystore password.
ssl.keystore.password=worklight
```

Configuring authenticationConfig.xml

Realm

Start by adding a realm that is named **CustomDeviceProvisioningRealm** to the *authenticationConfig.xml* file.
Use **CustomDeviceProvisioningLoginModule**.
Use the auto provisioning authenticator *className* parameter.
Add a **validate-csr-function** parameter.
The value of this parameter points to an Adapter function that performs CSR validation.

```
<realm name="CustomDeviceProvisioningRealm" loginModule="CustomDeviceProvisioningLoginModule">
  <className>com.worklight.core.auth.ext.DeviceAutoProvisioningAuthenticator</className>
  <parameter name="validate-csr-function" value="ProvisioningAdapter.validateCSR" />
</realm>
```

Login Module

Add **CustomDeviceProvisioningLoginModule**.
Use the auto provisioning login module *className* parameter.
Add a **validate-certificate-function** parameter.
The value of this parameter points to an Adapter function that performs certificate validation.

```
<loginModule name="CustomDeviceProvisioningLoginModule">
  <className>com.worklight.core.auth.ext.DeviceAutoProvisioningLoginModule</className>
  <parameter name="validate-certificate-function" value="ProvisioningAdapter.validateCertificate" /
>
</loginModule>
```

Security Test

Create a **mobileSecurityTest**.
Add a mandatory *testAppAuthenticity* test.
Add a mandatory *testDeviceId* test.
Specify *provisioningType="custom"*.
Specify *realm="CustomDeviceProvisioningRealm"*.

```
<mobileSecurityTest name="CustomDeviceProvisioningSecurityTest">
  <testAppAuthenticity/>
  <testDeviceId provisioningType="custom" realm="CustomDeviceProvisioningRealm" /
>
</mobileSecurityTest>
```

Implementing server-side components

Create an adapter that is named **ProvisioningAdapter**.
Add two functions with following signatures to the JavaScript file of the adapter.

- *validateCSR (clientDN, csrContent)* – this function is invoked only during initial device provisioning. It is used to check whether the device is authorized to be provisioned. Once the device is provisioned, this function is not invoked again.

- *validateCertificate (certificate, customAttributes)* – this function is invoked every time that the mobile application establishes a new session with the MobileFirst server. It is used to validate that the certificate that the application/device possesses is still valid and that the application/device is allowed to communicate with server.

These functions are called internally by the MobileFirst authentication framework. Therefore, do not declare them in the XML file of the adapter XML file.

validateCSR

```
function validateCSR(clientDN, csrContent){
 WL.Logger.info("validateCSR :: clientDN :: " + JSON.stringify(clientDN));
 WL.Logger.info("validateCSR :: csrContent :: " + JSON.stringify(csrContent));
 var activationCode = csrContent.activationCode;
 var response;
 // This is a place to perform validation of csrContent and update clientDN if required.
 // You can do it using adapter backend connectivity
 if (activationCode === "mobilefirst"){
  response = {
   isSuccessful: true,
   clientDN: clientDN + ",CN=someCustomData",
   attributes: {
    customAttribute: "some-custom-attribute"
   }
  };
 } else {
  response = {
   isSuccessful: false,
   errors: ["Invalid activation code"]
  };
 }
<p> return response;<br />
}<br />
```

csrContent.activationCode is a custom property that you add to CSR on the client side.
Adapter functionality, for example access http web services, can be used to validate CSR information. For simplicity, the *activationCode* is checked whether it is equal to a predefined hardcoded string.
If CSR validation is successful, the *validateCSR* function returns a *clientDN* (note that it can be modified with more custom data). In addition, it is possible to specify custom attributes to be saved in certificate. Once *isSuccessful:true* is returned from the *validateCSR* function, the server generates a certificate and return it to the application.
If CSR validation fails, you must return *isSuccessful:false* and supply an error message.

validateCertificate

```
function validateCertificate(certificate,customAttributes){
 WL.Logger.info("validateCertificate :: certificate :: " + JSON.stringify(certificate));
 WL.Logger.info("validateCertificate :: customAttributes :: " + JSON.stringify(customAttributes));
 // Additional custom certificate validations can be performed here.
 return {
  isSuccessful: true
 };
}
```

You can perform certificate validations according to your custom rules here. Adapter functionality, for example access http web services, can be used to validate the certificate. If the certificate is valid, you must return *isSuccessful:true*.
Returning *isSuccessful:false* means that application cannot operate and the only thing that can be done is to reinstall the application so it can be provisioned again.

Implementing client-side components

Create an application, add iPhone/iPad/Android environment to it.

Application Descriptor

Add security test that is created in previous steps to protect created environment.

```
<iphone applicationId="CustomDeviceProvisioning" bundleId="com.CustomDeviceProvisioning" securityTest="C
ustomDeviceProvisioningSecurityTest" version="1.0">
  <worklightSettings include="false" />
  <security>
    <encryptWebResources enabled="false" />
    <testWebResourcesChecksum enabled="false" ignoreFileExtensions="png, jpg, jpeg, gif, mp4, mp3" />
  </security>
</iphone>
```

In case it is required, configure your application for Application Authenticity test as described in the Application Authenticity Protection (../../authentication-security/application-authenticity-protection/) training module.

HTML

The *AppBody* element holds application content.
The *ProvBody* element holds device provisioning-related content.
Note the *connectToServerButton* in *AppBody*.

```
<div id="wrapper">
  <div id="AppBody">
    <p id="beforeProv">
      Tap the connect button to authenticate using custom device provisioning.
    </p>
    <input type="button" id="connectToServerButton" class="appButton" value="Connect to MobileFirst Server" /
>
    <p id="provisioningError" style="display: none"></p>
  </div>
  <div id="ProvBody" style="display: none">
    <p id="provisioningError">
      <input id="provisioningCode" placeholder="Enter code" type="text" />
      <input type="button" id="submitProvCodeButton" class="formButton" value="Send" />
    </p>
  </div>
</div>
```

main.js

Add listener to **connectToServerButton**.
Use the *WL.Client.connect()* API to connect to the MobileFirst Server.

```
function wlCommonInit(){
 $("#connectToServerButton").click(function(){
  WL.Client.connect({
    onFailure: onConnectFailure
  });
 });
}
```

Challenge Handler

Add a *CustomDeviceProvisioningRealmChallengeHandler.js* file and reference it in the main HTML file.
Device provisioning challenge handler requires following methods to be implemented:

- *handler.createCustomCsr (challenge)* – This method is responsible for returning custom properties that are added to CSR. Here you add a custom *activationCode* property, which is used in the adapter's *validateCSR* function. This method is asynchronous to allow collecting custom properties via native code or separate flow.
- *handler.processSuccess(identity)* – This method is invoked when certificate validation is successfully completed by using the *validateCertificate* adapter function that you implemented earlier.
- *handler.handleFailure()* – This method is invoked when certificate validation fails *(isSuccessful:false* is returned from *validateCertificate* function).

```
var customDevProvChallengeHandler =
 WL.Client.createProvisioningChallengeHandler("CustomDeviceProvisioningRealm");
```

Create device provisioning challenge handler by using the *WL.Client.createProvisioningChallengeHandler()* API. Specify realm name as parameter.

```
customDevProvChallengeHandler.createCustomCsr = function(challenge){
 WL.Logger.debug("createCustomCsr :: " + JSON.stringify(challenge));
 $("#AppBody").hide();
 $("#ProvBody").show();
 $("#provisioningCode").val("");
 if (challenge.error) {
  $("#provisioningError").html(new Date() + " " + challenge.error);
 } else {
  $("#provisioningError").html(new Date() + " Enter activation code.");
 }
 $("#submitProvCodeButton").click(function(){
  var customCsrProperties = {
   activationCode: $("#provisioningCode").val()
  };
  customDevProvChallengeHandler.submitCustomCsr(customCsrProperties, challenge)
;
 });
};
```

When the MobileFirst Server triggers device provisioning, the *createCustomCsr* function is invoked. Use it to manipulate your UI, for example to hide the application screen and show device provisioning-related components. You can use information that is returned in the authentication challenge, for example, error messages.
When required custom properties are collected, invoke the *submitCustomCsr()* API. Adding custom properties to CSR is optional. If you do not want to add custom properties, supply empty JSON object as a parameter.

```
customDevProvChallengeHandler.processSuccess = function(identity) {
 WL.Logger.debug("processSuccess :: " + JSON.stringify(identity));
 $("#connectToServerButton").hide();
 $("#AppBody").show();
 $("#ProvBody").hide();
 $("#AppBody").html("Device authentication with custom device provisioning was successfully completed.");
};
```

processSuccess function is called each time the certificate successfully passes validation. You can use it for UI manipulations.

```
customDevProvChallengeHandler.handleFailure = function(){
 WL.Logger.debug("handleFailure");
 $("#AppBody").show();
 $("#ProvBody").hide();
 $("#wrapper").text("MobileFirst Server rejected your device. You will need to re-install the application and perfor
m device provisioning again.");
};
```

handleFailure function is called each time that the certificate fails validation. You can use it for UI manipulations and to notify the user that the application cannot connect to server.

Sample application

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