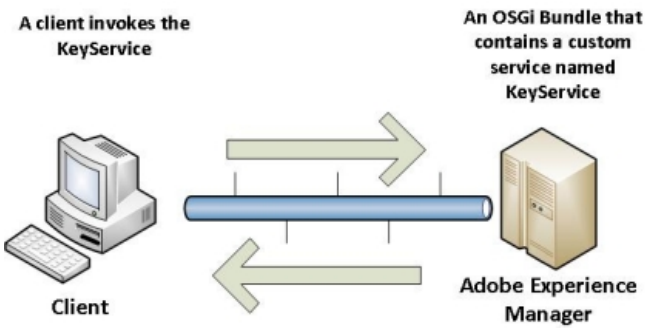
Adobe CQ Help /
Creating your first AEM Service using an Adobe Maven Archetype project

Article summary

Summary	Discusses how to create your first Adobe Experience Manager (AEM) custom service by using Maven and an Adobe Archetype project. The name of the service created in this development article is *KeyService*. This article is meant for new AEM developers whom want to learn about creating custom AEM services. A special thank you to Kalyan Venkat, a member of the AEM community for contributing towards this article. For information about Kalyan, click here.
Digital Marketing Solution(s)	Adobe Experience Manager (Adobe CQ)
Audience	Developer (beginner)
Required Skills	Java, Maven, HTML
Tested On	Adobe CQ 5.5, Adobe CQ 5.6

Introduction

In some business use cases, you create an OSGi bundle when creating an Adobe Experience Manager (AEM) application. Although there are different ways to create an OSGi bundle, a recommended way is to use Maven and the Adobe Maven Archetype. This development article walks you through creating a basic OSGi bundle that contains a simple service named *KeyService*. All this service does is accept an input value and sets a key value. It also exposes a method that returns the key value and the value can be displayed within an AEM web page.



An client invokes the AEM custom service named KeyService

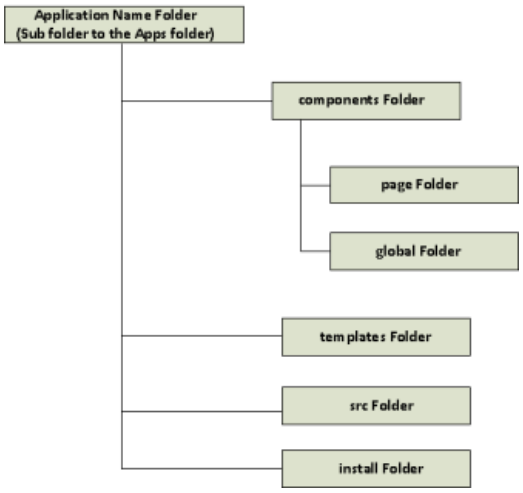
To create an Adobe CQ application that interacts with the custom KeyService, perform these tasks:

1. Create an Adobe CQ application folder structure.
2. Create a template on which the page component is based.
3. Create a render component that uses the template.
4. Setup Maven in your development environment.
5. Create an Adobe CQ archetype project.
6. Add Java files that represent the KeyService to the Maven project.
7. Modify the Maven POM file.
8. Build the OSGi bundle using Maven.
9. Deploy the bundle to Adobe CQ.
10. Modify the render component to invoke the KeyService.
11. Create a site that contains a page that lets an AEM user view data returned by the KeyService.

Create a CQ application folder structure

To the top

Create an Adobe CQ application folder structure that contains templates, components, and pages by using CRXDE Lite.



A CQ application folder structure

The following describes each application folder:

- **application name**: contains all of the resources that an application uses. The resources can be templates, pages, components, and so on.
- **components**: contains components that your application uses.
- **page**: contains page components. A page component is a script such as a JSP file.
  global: contains global components that your application uses.
- **template**: contains templates on which you base page components.
- **src**: contains source code that comprises an OSGi component (this development article does not create an OSGi bundle using this folder).
- **install**: contains a compiled OSGi bundles container.

To create an application folder structure:

1. To view the CQ welcome page, enter the URL http://[host name]:[port] into a web browser. For example, http://localhost:4502.
2. Select CRXDE Lite.
3. Right-click the apps folder (or the parent folder), select Create, Create Folder.
4. Enter the folder name into the Create Folder dialog box. Enter `firstOSGI`.
5. Repeat steps 1-4 for each folder specified in the previous illustration.
6. Click the Save All button.

Note: You have to click the Save All button when working in CRXDELite for the changes to be made.

Create a template

To the top

You can create a template by using CRXDE Lite. A CQ template enables you to define a consistent style for the pages in your application. A template comprises of nodes that specify the page structure. For more information about templates, see http://dev.day.com/docs/en/cq/current/developing/templates.html.

To create a template, perform these tasks:

1. To view the CQ welcome page, enter the URL http://[host name]:[port] into a web browser. For example, http://localhost:4502.
2. Select CRXDE Lite.
3. Right-click the template folder (within your application), select Create, Create Template.
4. Enter the following information into the Create Template dialog box:

- **Label**: The name of the template to create. Enter `keyTemplate`.
- **Title**: The title that is assigned to the template.
- **Description**: The description that is assigned to the template.
- **Resource Type**: The component's path that is assigned to the template and copied to implementing pages. Enter *firstOSGI/components/page/keyTemplate*.
- **Ranking**: The order (ascending) in which this template will appear in relation to other

templates. Setting this value to 1 ensures that the template appears first in the list.

5. Add a path to Allowed Paths. Click on the plus sign and enter the following value: */content(/.*)?*.
6. Click Next for Allowed Parents.
7. Select OK on Allowed Children.

To the top

Create a render component that uses the template

Components are re-usable modules that implement specific application logic to render the content of your web site. You can think of a component as a collection of scripts (for example, JSPs, Java servlets, and so on) that completely realize a specific function. In order to realize this functionality, it is your responsibility as a CQ developer to create scripts that perform specific functionality. For more information about components, see http://dev.day.com/docs/en/cq/current/developing/components.html.

By default, a component has at least one default script, identical to the name of the component. To create a render component, perform these tasks:

1. To view the CQ welcome page, enter the URL http://[host name]:[port] into a web browser. For example, http://localhost:4502.
2. Select CRXDE Lite.
3. Right-click */apps/firstOSGI/components/page*, then select Create, Create Component.
4. Enter the following information into the Create Component dialog box:

- **Label**: The name of the component to create. Enter `keyTemplate`.
- **Title**: The title that is assigned to the component.
- **Description**: The description that is assigned to the template.

5. Select Next for Advanced Component Settings and Allowed Parents.
6. Select OK on Allowed Children.
7. Open the slingTemplateJCR.jsp located at:
/apps/firstOSGI/components/page/keyTemplate/keyTemplate.jsp.
8. Enter the following JSP code.

```
<html>
<head>
<title>Hello World !!!</title>
</head>
<body>
<h1>Hello Key Service!!!</h1>
<h2>This page will invoke the Custom Key Service</h2>
</body>
</html>
```

To the top

Setup Maven in your development environment

You can use Maven to build an OSGi bundle that contains a Sling Servlet. Maven manages required JAR files that a Java project needs in its class path. Instead of searching the Internet trying to find and download third-party JAR files to include in your project's class path, Maven manages these dependencies for you.

You can download Maven 3 from the following URL:

http://maven.apache.org/download.html

After you download and extract Maven, create an environment variable named `M3_HOME`. Assign the Maven install location to this environment variable. For example:

```
C:\Programs\Apache\apache-maven-3.0.4
```

Set up a system environment variable to reference Maven. To test whether you properly setup Maven, enter the following Maven command into a command prompt:

```
%M3_HOME%\bin\mvn -version
```

This command provides Maven and Java install details and resembles the following message:

```
Java home: C:\Programs\Java64-6\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 7", version: "6.1", arch: "amd64", family: "windows"
```

Note: It is recommended that you use Maven 3.0.3 or greater. For more information about setting up Maven and the Home variable, see: Maven in 5 Minutes.

Next, copy the Maven configuration file named settings.xml from [install location]\apache-maven-3.0.4\conf\ to your user profile. For example, C:\Users\scottm\.m2\.

You have to configure your settings.xml file to use Adobe's public repository. For information, see Adobe Public Maven Repository at http://repo.adobe.com/.

The following XML code represents a settings.xml file that you can use.

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
Licensed to the Apache Software Foundation (ASF) under one
or more contributor license agreements.  See the NOTICE file
distributed with this work for additional information
regarding copyright ownership.  The ASF licenses this file
to you under the Apache License, Version 2.0 (the
"License"); you may not use this file except in compliance
with the License.  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing,
software distributed under the License is distributed on an
"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
KIND, either express or implied.  See the License for the
specific language governing permissions and limitations
under the License.
-->

<!--
 | This is the configuration file for Maven. It can be specified at two levels:
 |
 |  1. User Level. This settings.xml file provides configuration for a single u
 |                 and is normally provided in ${user.home}/.m2/settings.xml.
 |
 |                 NOTE: This location can be overridden with the CLI option:
 |
 |                 -s /path/to/user/settings.xml
 |
 |  2. Global Level. This settings.xml file provides configuration for all Mave
 |                 users on a machine (assuming they're all using the same Mave
 |                 installation). It's normally provided in
 |                 ${maven.home}/conf/settings.xml.
 |
 |                 NOTE: This location can be overridden with the CLI option:
 |
 |                 -gs /path/to/global/settings.xml
 |
 | The sections in this sample file are intended to give you a running start at
 | getting the most out of your Maven installation. Where appropriate, the defa
 | values (values used when the setting is not specified) are provided.
 |
 |-->
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://mav
  <!-- localRepository
   | The path to the local repository maven will use to store artifacts.
   |
   | Default: ~/.m2/repository
  <localRepository>/path/to/local/repo</localRepository>
  -->

  <!-- interactiveMode
   | This will determine whether maven prompts you when it needs input. If set
   | maven will use a sensible default value, perhaps based on some other setti
   | the parameter in question.
   |
   | Default: true
  <interactiveMode>true</interactiveMode>
  -->

  <!-- offline
   | Determines whether maven should attempt to connect to the network when exe
   | This will have an effect on artifact downloads, artifact deployment, and o
   |
   | Default: false
  <offline>false</offline>
  -->

  <!-- pluginGroups
   | This is a list of additional group identifiers that will be searched when
   | when invoking a command line like "mvn prefix:goal". Maven will automatica
   | "org.apache.maven.plugins" and "org.codehaus.mojo" if these are not alread
```

```
  |-->
<pluginGroups>
  <!-- pluginGroup
   | Specifies a further group identifier to use for plugin lookup.
  <pluginGroup>com.your.plugins</pluginGroup>
  -->
</pluginGroups>

<!-- proxies
 | This is a list of proxies which can be used on this machine to connect to
 | Unless otherwise specified (by system property or command-line switch), th
 | specification in this list marked as active will be used.
 |-->
<proxies>
  <!-- proxy
   | Specification for one proxy, to be used in connecting to the network.
   |
  <proxy>
    <id>optional</id>
    <active>true</active>
    <protocol>http</protocol>
    <username>proxyuser</username>
    <password>proxypass</password>
    <host>proxy.host.net</host>
    <port>80</port>
    <nonProxyHosts>local.net|some.host.com</nonProxyHosts>
  </proxy>
  -->
</proxies>

<!-- servers
 | This is a list of authentication profiles, keyed by the server-id used wi
 | Authentication profiles can be used whenever maven must make a connection
 |-->
<servers>
  <!-- server
   | Specifies the authentication information to use when connecting to a par
   | a unique name within the system (referred to by the 'id' attribute belov
   |
   | NOTE: You should either specify username/password OR privateKey/passphra
   |       used together.
   |
  <server>
    <id>deploymentRepo</id>
    <username>repouser</username>
    <password>repopwd</password>
  </server>
  -->

  <!-- Another sample, using keys to authenticate.
  <server>
    <id>siteServer</id>
    <privateKey>/path/to/private/key</privateKey>
    <passphrase>optional; leave empty if not used.</passphrase>
  </server>
  -->
</servers>

<!-- mirrors
 | This is a list of mirrors to be used in downloading artifacts from remote
 |
 | It works like this: a POM may declare a repository to use in resolving cer
 | However, this repository may have problems with heavy traffic at times, so
 | it to several places.
 |
 | That repository definition will have a unique id, so we can create a mirro
 | repository, to be used as an alternate download site. The mirror site will
 | server for that repository.
 |-->
<mirrors>
  <!-- mirror
   | Specifies a repository mirror site to use instead of a given repository.
   | this mirror serves has an ID that matches the mirrorOf element of this r
   | for inheritance and direct lookup purposes, and must be unique across th
   |
  <mirror>
    <id>mirrorId</id>
    <mirrorOf>repositoryId</mirrorOf>
    <name>Human Readable Name for this Mirror.</name>
    <url>http://my.repository.com/repo/path</url>
  </mirror>
   -->
</mirrors>

```

```
  <!-- profiles
   | This is a list of profiles which can be activated in a variety of ways, ar
   | the build process. Profiles provided in the settings.xml are intended to p
   | specific paths and repository locations which allow the build to work in t
   |
   | For example, if you have an integration testing plugin - like cactus - tha
   | your Tomcat instance is installed, you can provide a variable here such th
   | dereferenced during the build process to configure the cactus plugin.
   |
   | As noted above, profiles can be activated in a variety of ways. One way -
   | section of this document (settings.xml) - will be discussed later. Another
   | relies on the detection of a system property, either matching a particular
   | or merely testing its existence. Profiles can also be activated by JDK ver
   | value of '1.4' might activate a profile when the build is executed on a JD
   | Finally, the list of active profiles can be specified directly from the co
   |
   | NOTE: For profiles defined in the settings.xml, you are restricted to spec
   |       repositories, plugin repositories, and free-form properties to be us
   |       variables for plugins in the POM.
   |
   |-->
  <profiles>
    <!-- profile
     | Specifies a set of introductions to the build process, to be activated u
     | mechanisms described above. For inheritance purposes, and to activate pr
     | or the command line, profiles have to have an ID that is unique.
     |
     | An encouraged best practice for profile identification is to use a consi
     | for profiles, such as 'env-dev', 'env-test', 'env-production', 'user-jdk
     | This will make it more intuitive to understand what the set of introduce
     | to accomplish, particularly when you only have a list of profile id's fo
     |
     | This profile example uses the JDK version to trigger activation, and pro
    <profile>
      <id>jdk-1.4</id>

      <activation>
        <jdk>1.4</jdk>
      </activation>

      <repositories>
        <repository>
          <id>jdk14</id>
          <name>Repository for JDK 1.4 builds</name>
          <url>http://www.myhost.com/maven/jdk14</url>
          <layout>default</layout>
          <snapshotPolicy>always</snapshotPolicy>
        </repository>
      </repositories>
    </profile>
    -->

    <!--
     | Here is another profile, activated by the system property 'target-env' w
     | which provides a specific path to the Tomcat instance. To use this, your
     | might hypothetically look like:
     |
     | ...
     | <plugin>
     |   <groupId>org.myco.myplugins</groupId>
     |   <artifactId>myplugin</artifactId>
     |
     |   <configuration>
     |     <tomcatLocation>${tomcatPath}</tomcatLocation>
     |   </configuration>
     | </plugin>
     | ...
     |
     | NOTE: If you just wanted to inject this configuration whenever someone s
     |       anything, you could just leave off the <value/> inside the activat
     |
    <profile>
      <id>env-dev</id>

      <activation>
        <property>
          <name>target-env</name>
          <value>dev</value>
        </property>
      </activation>

      <properties>
        <tomcatPath>/path/to/tomcat/instance</tomcatPath>
      </properties>
```

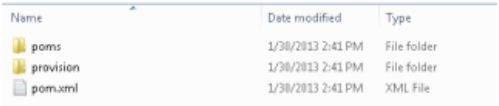
```
    </profile>
    -->


<profile>

                <id>adobe-public</id>

                <activation>

                    <activeByDefault>true</activeByDefault>

                </activation>

                <repositories>

                  <repository>

                    <id>adobe</id>

                    <name>Nexus Proxy Repository</name>

                    <url>http://repo.adobe.com/nexus/content/groups/public/</ur

                    <layout>default</layout>

                  </repository>

                </repositories>

                <pluginRepositories>

                  <pluginRepository>

                    <id>adobe</id>

                    <name>Nexus Proxy Repository</name>

                    <url>http://repo.adobe.com/nexus/content/groups/public/</ur

                    <layout>default</layout>

                  </pluginRepository>

                </pluginRepositories>

            </profile>

</profiles>

  <!-- activeProfiles
   | List of profiles that are active for all builds.
   |
  <activeProfiles>
    <activeProfile>alwaysActiveProfile</activeProfile>
    <activeProfile>anotherAlwaysActiveProfile</activeProfile>
  </activeProfiles>
  -->
</settings>
```

Sometimes Maven reads the settings.xml file from under its install folder. For example, instead of reading the settings.xml file from `user\m2`, it reads from `apache-maven-3.2.1\.m2`. Make sure that Maven is reading the settings.xml from the correct location. Otherwise, Maven will not successfully build an AEM Maven Archetype project.

Create an Adobe CQ archetype project

To the top

You can create an Adobe CQ archetype project by using the Maven archetype plugin. In this example, assume that the working directory is C:\AdobeCQ.



Default files created by the Maven archetype plugin

To create an Adobe CQ archetype project, perform these steps:

1. Open the command prompt and go to your working directory (for example, C:\AdobeCQ).

2. Run the following Maven command:

```
mvn archetype:generate -DarchetypeGroupId=com.day.jcr.vault -
DarchetypeArtifactId=multimodule-content-package-archetype -
DarchetypeVersion=1.0.0 -DarchetypeRepository=adobe-public-releases
```

3. When prompted for additional information, specify these values:

- **groupId**: com.adobe.cq
- **artifactId**: key
- **version**: 1.0-SNAPSHOT
- **package**: com.adobe.cq
- **appsFolderName**: adobe-training
- **artifactName**: Custom Service Bundle
- **packageGroup**: adobe training
- **confirm**: Y

4. Once done, you will see a message like:

```
[[INFO] Total time: 14:46.131s
[INFO] Finished at: Wed Mar 27 13:38:58 EDT 2013
[INFO] Final Memory: 10M/184M
```

5. Change the command prompt to the generated project. For example: C:\AdobeCQ\key. Run the following Maven command:

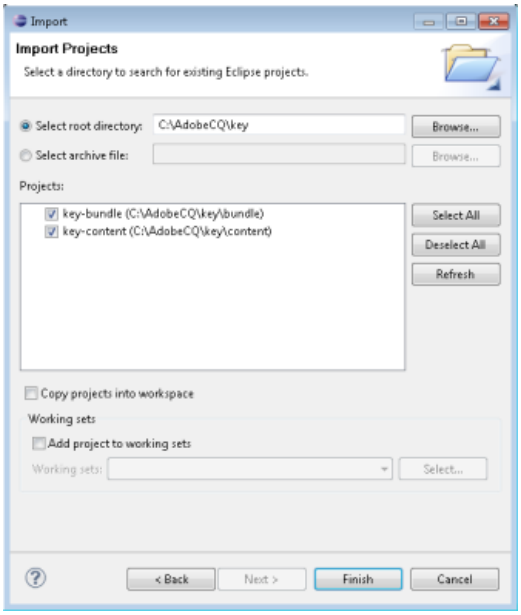
```
mvn eclipse:eclipse
```

After you run this command, you can import the project into Eclipse as discussed in the next section.

Add Java files to the Maven project using Eclipse

To the top

To make it easier to work with the Maven generated project, import it into the Eclipse development environment, as shown in the following illustration.



The Eclipse Import Project dialog

Eclipse is not used to build the project or even compile it. In this use case, you use Eclipse to view and edit the Java files and the project POM file. Do not worry about the errors that are displayed in the Eclipse IDE. You are using Maven to build the OSGi bundle from the command line, not Eclipse. So if you are getting an error messages such as:

```
cannot be resolved to a type
```

do not worry about them. When you update the POM file and build the OSGi bundle using Maven from the command line, the project builds the OSGi bundle despite the presence of these errror message.

However, if you are interested in resolving these error messages in Eclipse, you will have to install the Maven Eclipse plugin. For information, see Guide to using Eclipse with Maven 2.x.

Then you will have to right click in the Project and select Maven, Update project. However, this step is not required to build the OSGi bunlde. You can simply ignore the error messages.

The next step is to add the following files to the `com.adobe.cq` package:

- A Java interface named `KeyService`.
- A Java class named `KeyServiceImp` that implements the `KeyService` interface.

Note: Make sure that you work in the Eclipse project named key-bundle. You can delete key-content from Eclipse. Also be sure to work in the scr/main/java folder.

KeyService interface

The following code represents the `KeyService` interface. This interface contains two methods named `setKey` and `getkey`. The implementation logic for these methods is located in the `KeyServiceImp` class.

```
1  package com.adobe.cq;
2
3  public interface KeyService {
4
5      public void setKey(int val);
6      public String getKey();
7
8  }
```

KeyServiceImpl class

The `KeyServiceImpl` class uses the following Apache Felix SCR annotations to create the OSGi component:

- **@Component** - defines the class as a component
- **@Service** - defines the service interface that is provided by the component

These Apache Felix SCR annotations are required in order to create an AEM service based on the Java logic located in the `KeyService` interface. For information about Apache Felix SCR annotations, see http://felix.apache.org/documentation/subprojects/apache-felix-maven-scr-plugin/scr-annotations.html.

A class member named `key` is created in the `KeyServiceImpl` class. The `setKey` method assigns a value to this class member. Likewise, the `getKey` value returns its value. The following code represents the `KeyServiceImpl` class.

```
1  package com.adobe.cq;
2
3  import org.apache.felix.scr.annotations.Component;
4  import org.apache.felix.scr.annotations.Service;
5
6
7  //This is a component so it can provide or consume services
8  @Component
9
10 @Service
11 public class KeyServiceImpl implements KeyService{
12
13
14     //Define a class member named key
15     private int key = 0 ;
16
17
18     //A basic setter method that sets key
19     public void setKey(int val)
20     {
21         //Set the key class member
22         this.key = val ;
23
24     }
25
26     //A basic getter that gets key
27     public String getKey()
28     {
29         //return the value of the key class member
30
31         //Convert the int to a String to display it within an AEM web page
32         String strI = Integer.toString(this.key);
33         return strI;
34     }
35
36
37 }
```

Note: Notice that there is another class in the `com.adobe.cq` package named `SimpleDSComponent`. To create the `KeyService`, you do not have to further modify this class. However, it has to be part of the build, so don't delete this class. This class contains methods, such as `activate`, that are invoked by the OSGi service container. You can add application logic to these methods. For example, you can add logging application logic to the activate method that is called when the OSGi bundle is placed into an active state.

Modify the Maven POM file

To the top

Modify the POM file visible from the Eclipse IDE, as shown in the following illustration.

The POM file located in the bundles folder

Notice that this POM file contains required dependencies such as `org.apache.felix.scr` and `org.apache.felix.scr.annotations` in the `dependencies` section. Copy the following XML code to your POM file.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.or
    <modelVersion>4.0.0</modelVersion>
    <!-- ======================================================================
    <!-- P A R E N T P R O J E C T D E S C R I P T I O N -->
    <!-- ======================================================================
    <parent>
        <groupId>com.adobe.cq</groupId>
        <artifactId>key</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <!-- ======================================================================
    <!-- P R O J E C T D E S C R I P T I O N -->
    <!-- ======================================================================

    <artifactId>key-bundle</artifactId>
    <packaging>bundle</packaging>
    <name>Custom Service Bundle Bundle</name>

    <!-- ======================================================================
    <!-- B U I L D D E F I N I T I O N -->
    <!-- ======================================================================
    <build>

        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-scr-plugin</artifactId>
                <executions>
                    <execution>
                        <id>generate-scr-descriptor</id>
                        <goals>
                            <goal>scr</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>com.adobe.cq.key-bundle</Bundle-Syr
                    </instructions>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.sling</groupId>
                <artifactId>maven-sling-plugin</artifactId>
                <configuration>
                    <slingUrl>http://${crx.host}:${crx.port}/apps/Custom Service
                    <usePut>true</usePut>
                </configuration>
```

```
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.compendium</artifactId>
        </dependency>
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.core</artifactId>
        </dependency>

        <dependency>
            <groupId>org.apache.felix</groupId>
            <artifactId>org.apache.felix.scr.annotations</artifactId>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
        </dependency>
    </dependencies>
</project>
```

Note: The KeyService uses basic Java application logic and does not use additional Java APIs. However, in more advanced use cases, you use other Java APIs. In this situation, you have to ensure that those APIs are specified in the POM file's `dependencies` section.

To the top

Build the OSGi bundle using Maven

To build the OSGi component by using Maven, perform these steps:

1. Open the command prompt and go to the C:\AdobeCQ\key folder.
2. Run the following maven command: `mvn clean install`.
3. The OSGi component can be found in the following folder: C:\AdobeCQ\key\bundle\target. The file name of the OSGi component is *key-bundle-1.0-SNAPSHOT.jar*.

To the top

Deploy the bundle to Adobe CQ

Once you deploy the OSGi bundle, you can invoke methods located in `KeyService` (this is shown later in this development article). After you deploy the OSGi bundle, you will be able to see it in the Adobe CQ Apache Felix Web Conole. In the following illustration, notice that details about the OSGi bundle are displayed. For example, the Symbolic Name is `com.adobe.cq.key-bundle`.

Apache Felix Web Console Bundles view

Deploy the OSGi bundle that contains `KeyService` to Adobe CQ by performing these steps:

1. Login to Adobe CQ's Apache Felix Web Console at http://server:port/system/console/bundles (default admin user = admin with password= admin).
2. Click the Bundles tab, sort the bundle list by Id, and note the Id of the last bundle.

3. Click the Install/Update button.
4. Browse to the bundle JAR file you just built using Maven. (C:\AdobeCQ\key\bundle\target).
5. Click Install.
6. Click the Refresh Packages button.
7. Check the bundle with the highest Id.
8. Click Active.
9. Your new bundle should now be listed with the status Active.
10. If the status is not Active, check the CQ error.log for exceptions.

To the top

Modify the keyTemplate JSP to invoke the KeyService

You can invoke the KeyService from an AEM JSP file. You create a `KeyService` instance by using the `sling.getService` method, as shown in the following example:

```
com.adobe.cq.KeyService keyService =
sling.getService(com.adobe.cq.KeyService.class);
```

Note: Do not specify the name of the implementation class in `sling.getService`. For example, the following JavaScript causes an exception:
com.adobe.cq.KeyServiceImpl keyService =
sling.getService(com.adobe.cq.KeyServiceImpl.class);

You pass the fully qualified name of the service to `sling.getService` method. After you create a `KeyService` object by using `sling.getService`, you can invoke the `getKey` and `setKey` methods exposed by the service. When you invoke the `setKey` method, pass an int value.

The following code represents the keyTemplate JSP file.

```
<%@include file="/libs/foundation/global.jsp"%>
<h1><%= properties.get("title", currentPage.getTitle()) %></h1>
<%

com.adobe.cq.KeyService keyService = sling.getService(com.adobe.cq.KeyService.c
keyService.setKey(10) ;

%>

<h2>This page invokes the AEM KeyService</h2>
<h3>The value of the key is: <%=keyService.getKey()%></h3>
```

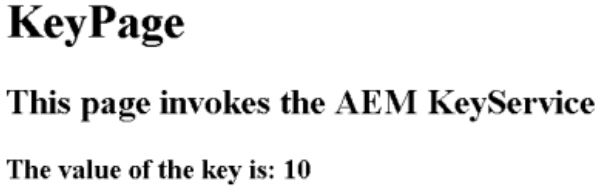
Modify the keyTemplate JSP file

1. To view the CQ welcome page, enter the URL: http://[host name]:[port] into a web browser. For example, http://localhost:4502.
2. Select CRXDE Lite.
3. Double-click /apps/firstOSGI/components/page/keyTemplate/keyTemplate.jsp.
4. Replace the JSP code with the new code shown in this section.
5. Click Save All.

To the top

Create a CQ web page that displays the client web page

The final task is to create a site that contains a page that is based on the keyTemplate (the template created earlier in this development article). Notice that the JSP (the JSP that is part of the keyTemplate component) invoked the `setKey` method and passed the value 10. The JSP also invoked the `getKey` value and displayed the return value in the AEM web page.



Create an AEM web page that displays data returned by the KeyService:

1. Go to the CQ WebSite page at *http://[host name]:[port]/siteadmin#/content*; for example,

http://localhost:4502/siteadmin#/content.

2. Select New Page.
3. Specify the title of the page in the Title field.
4. Specify the name of the page in the Name field.
5. Select *keyTemplate* from the template list that appears. This value represents the template that is created in this development article. If you do not see it, then repeat the steps in this development article. For example, if you made a typing mistake when entering in path information, the template will not show up in the New Page dialog box.
6. Open the new page that you created by double-clicking it in the right pane. The new page opens in a web browser. You will see the return value from the KeyService.

See Also

Congratulations, you have just created your first AEM custom service by using an Adobe Maven Archetype project. Now that you understand how to build a custom service using Maven and an Adobe Maven Archetype, you can build more advanced services. For example, you can build an AEM service that uses the JCR API to query data from the AEM JCR.

Please refer to the AEM community page for more articles that discuss how to build AEM services/applications by using an Adobe Maven Archetype project.