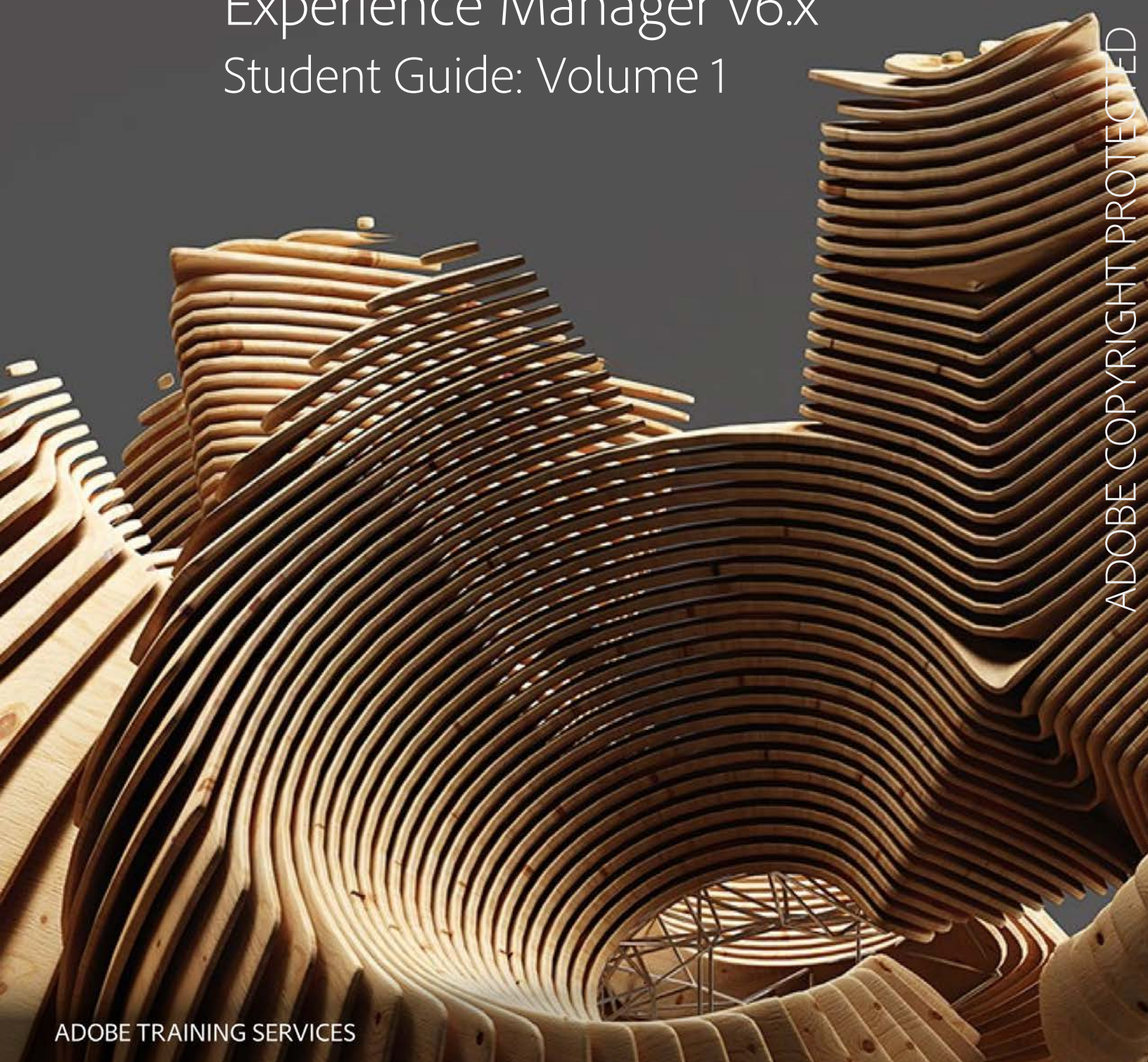
Adobe Experience Manager

Extend and Customize Adobe Experience Manager v6.x

Student Guide: Volume 1



ADOBE TRAINING SERVICES

Extend and Customize Adobe Experience Manager v6.x

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

November 22, 2016

Contents

CHAPTER ONE: BASICS OF THE ARCHITECTURAL STACK

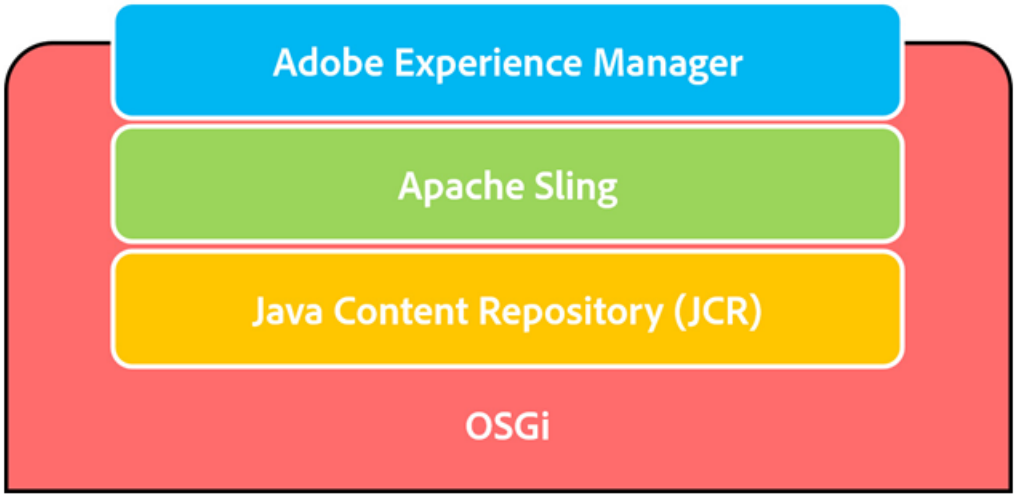
What is Adobe Experience Manager?

- Adobe Experience Manager is a web-based client-server system for building, managing, and deploying commercial websites and related services.
- A number of infrastructure-level and application-level functions are combined into a single integrated package.

Basics of the Architecture Stack

Adobe Experience Manager is a Java web application, and is based on technologies such as Open Service Gateway Initiative (OSGi), Java Content Repository (JCR), and Apache Sling.

The following diagram is a high-level view of the architecture stack.



Introduction to the Granite Platform

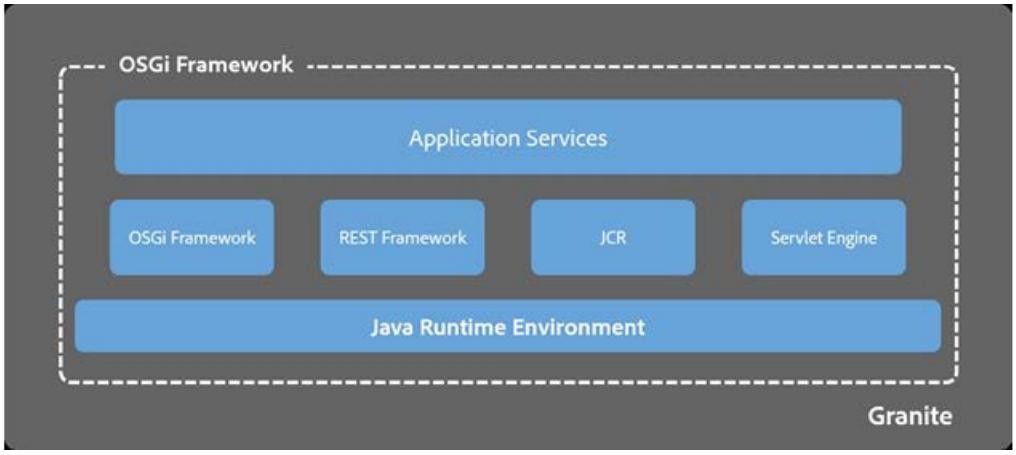
- Granite is the technical foundation on which Adobe Experience Manager is built
- Granite provides the following components:
  - ✓ An application launcher
  - ✓ An OSGi framework into which everything is deployed
  - ✓ A number of OSGi compendium services to support building applications
  - ✓ A comprehensive Logging Framework providing various logging APIs

- ✓ The CRX Repository (Content Repository eXtreme) implementation of the JCR API specification
- ✓ The Apache Sling framework

- The platform includes:
  - ✓ Application Runtime - Felix
  - ✓ Content Repository - CRX
  - ✓ Web Application Framework - Sling

OSGi enables a collaborative and modular environment, where each application may be built and implemented as a small bundle. Each of these bundles is a collection of tightly coupled, dynamically loadable classes, JAR files, and configuration files that explicitly declare their external dependencies.

All content is stored in the content repository, and hence backup is done at the repository level. OSGi runtime hosts Java applications that can access the repository using the JCR API. As part of the application runtime, you get Apache Sling, a RESTful web application framework that exposes the full repository content using HTTP and other protocols.



Apache Felix is an open source implementation of the OSGi that the Adobe Experience Manager framework makes use of. It provides a dynamic runtime environment, where code and content bundles can be loaded, unloaded, and reconfigured at runtime.

Introduction to the Java Content Repository

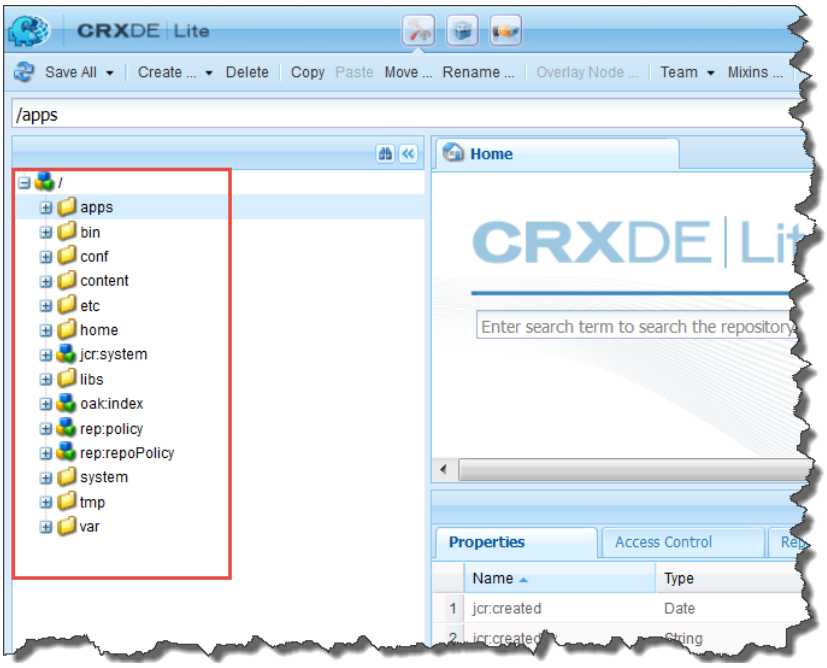
The JCR, specifically JSR-283, is a database that supports structured and unstructured content, versioning, and observation. All data pertaining to Adobe Experience Manager such as HTML, CSS, JS/Java, Images, and Videos are stored in the JCR object database. It is built with Apache Jackrabbit Oak. The Adobe implementation of JSR-283 is the Content Repository eXtreme (CRX), and the version of CRX that comes with Adobe Experience Manager supports JCR 2.x.

In addition to CRX, Adobe Experience Manager can also work with other JCR repositories such as Apache Jackrabbit and with a number of non-JCR data stores through connectors. As Adobe Experience Manager is built on top of this standard, it is capable of pulling content not just from its built-in CRX repository, but from any JCR-compliant source, such as a third-party repository (for example, Jackrabbit) or a connector that exposes legacy storage through JCR.

Advantages of using JCR

1. JCR provides a generic application data store for structured and unstructured content. While file systems provide excellent storage for unstructured, hierarchical content, and databases provide storage for structured data, JCR provides the best of both data storage architectures.
2. It supports namespaces. Namespaces prevent naming collisions among items and node types that come from different sources and application domains. JCR namespaces are defined with a prefix, delimited by a single colon (:); for example, jcr:title.

Understanding the Repository Structure



/apps – all custom templates, components, and any other definitions related to your site is stored here. When you inherit foundation components, it is done from the libs folder. Always copy the components from libs to apps, and then modify it in apps. By doing this, no code will be lost when an upgrade is done to Adobe Experience Manager. You just need to make the updates to the apps folder. **NOTE**: Best practice is to use a feature known as Sling Resource Merger, and just duplicate the required /libs folder structure (empty folders) under /apps. Make modifications to the node or property, wherever the changes are required.

/libs –All libraries, and definitions that belong to the code of Adobe Experience Manager are stored here. It includes out-of-the-box components, templates, and other features such as search or replication. It is also referred to as the foundation components. Avoid making any modifications to any of the components in libs.

/content – all content of your website is stored in this folder.

/conf – holds all the configuration files. For 6.2, it is used to store the dynamic templates and content policies.
/etc – contains all resources related to utilities and tools

/**home** – contains all information related to users and groups.

/tmp – This is a temporary working area

/var – This folder contains files that change and are updated by the system, such as audit logs, statistics, and event handling.

/oak:index – This node contains Jackrabbit Oak index definitions. Each node specifies the details of one index. Standard indexes for the Adobe Experience Manager application are visible and additional custom indexes can be created.

Introduction to Apache Sling

Apache Sling is a web application framework for content-centric applications, and uses a Java Content Repository, such as Apache Jackrabbit or CRX to store and manage content.

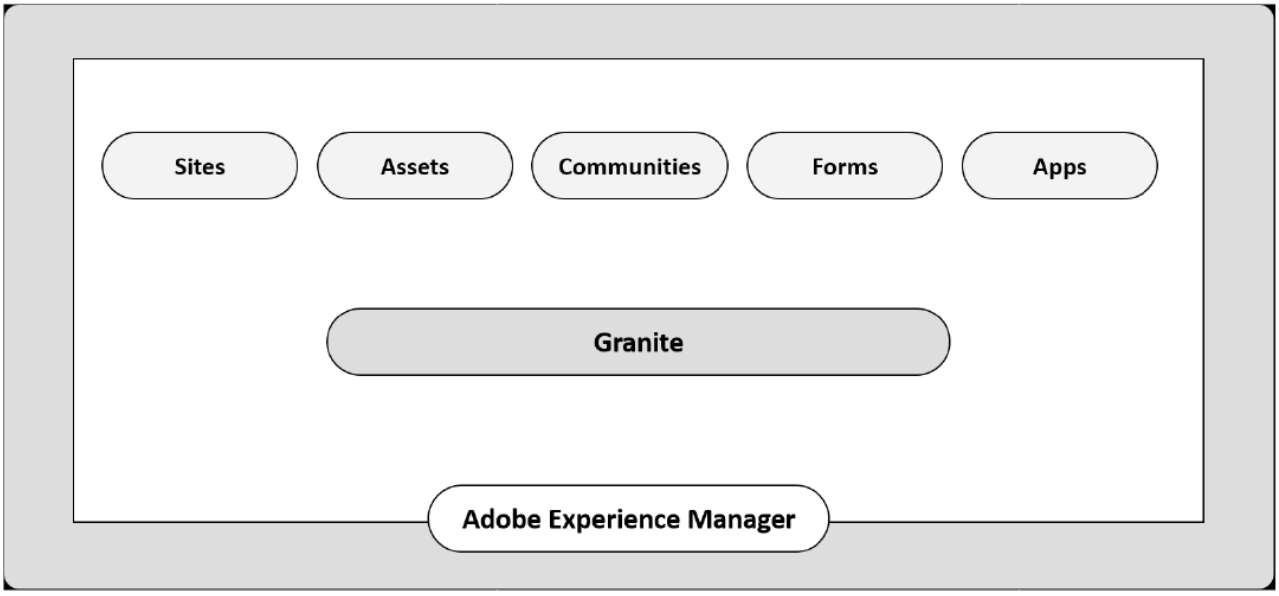
- It is based on REST principles
- Its applications are built as a series of OSGi bundles
- It is resource-oriented, and usually map into JCR nodes

The embedded Apache Felix OSGi framework and console provides a dynamic runtime environment, where code and content bundles can be loaded, unloaded, and reconfigured at runtime. Sling makes it easy to implement simple applications, while providing an enterprise-level framework for more complex applications.

Sling assumes that 'Everything is a Resource'. That is, Sling is resource oriented, where the resources have URLs, and each resource is mapped into a JCR node. A request URL is first resolved to a resource, and then based on the resource, it selects the Servlet or script to handle that request. Servlets and scripts are handled uniformly in that they are represented as resources themselves and are accessible by a resource path. This means that every script, Servlet, filter, error handler, and so on is available from the ResourceResolver just like normal content—providing data to be rendered on request.

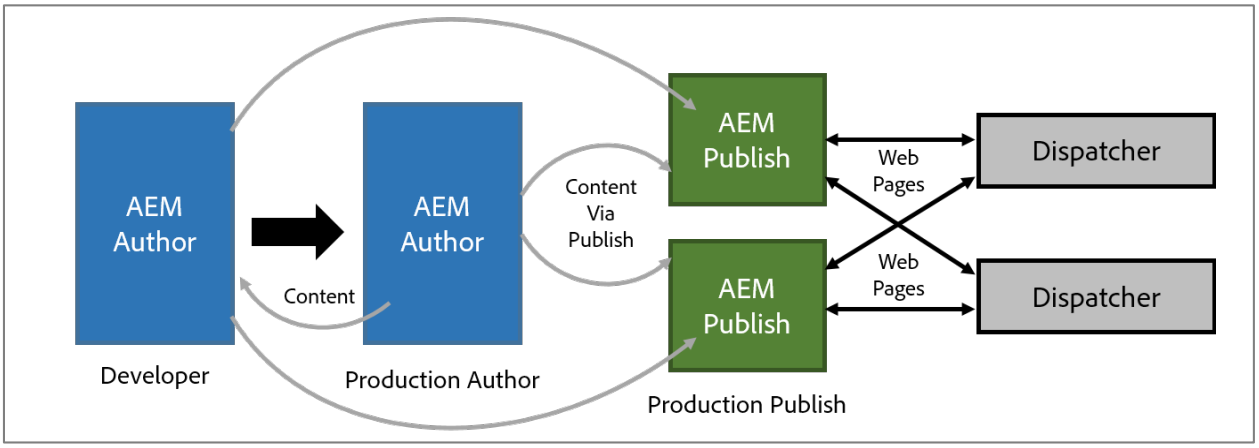
The Functional Building Blocks of Adobe Experience Manager

The Adobe Experience Manager application modules such as Sites, Assets, Communities, Forms, and Apps, sit on top of the Adobe Experience Manager shared framework, a framework known as Granite. This framework includes all the application layer functionality such as mobile functionality, multisite manager, taxonomy management, and workflow. These functionalities are shared among all the application modules. In addition to these functionalities, these Adobe Experience Manager applications share the same infrastructure and UI framework, and are very tightly integrated with each other. All of this is encapsulated within the OSGi container.



CHAPTER TWO: INSTALLATION

1.1 Adobe Experience Manager Workflow



In Adobe Experience Manager terminology, an "instance" is a copy of Adobe Experience Manager running on a server. Adobe Experience Manager installations usually involve at least two instances, typically running on separate machines:

- Author: An Adobe Experience Manager instance used to create, upload, and edit content, and administer the website. After content is ready to go live, it is replicated to the Publish Instance.
- Publish: An Adobe Experience Manager instance that serves the published content to the public.

NOTE: In a basic sense, the author and publish instances are the same software stack but two different run modes. Run modes are covered later in this course.

These instances are identical in terms of installed software. They are differentiated only by their configuration.

In addition, most installations use a Dispatcher:

- Dispatcher: A static web server (Apache httpd, Microsoft IIS, and so on) augmented with the Adobe Experience Manager Dispatcher module. It caches webpages produced by the Publish instance to improve performance.

1.2 Prerequisites

To install Adobe Experience Manager, you need:

- Adobe Experience Manager installation and startup JAR file
- Valid Adobe Experience Manager license key properties file
- JDK version 1.8 (http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html)
- Approximately 4 GB of free space per instance
- Approximately 4 GB of RAM (at the **very minimum!)**

The Adobe Experience Manager installation and startup JAR file is also known as the "quickstart" file. You use the file to install Adobe Experience Manager. Once installed, the file is referred to as the Adobe Experience Manager startup file. During installation, you will notice the JAR file creates a root folder called **crx-quickstart**.

You will also need to set environment variables as part of setting up your JDK 1.8.

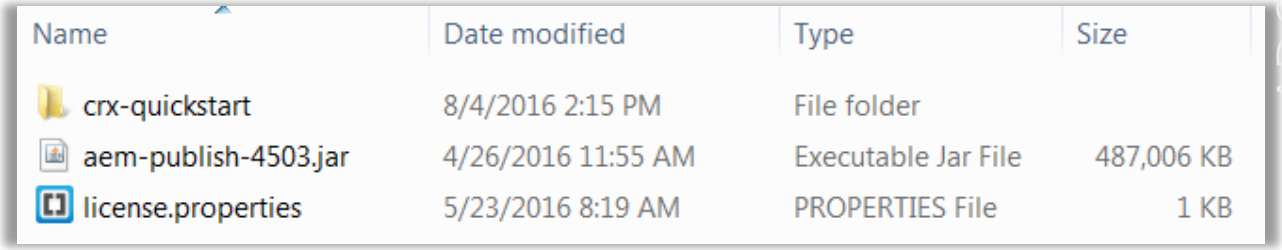
1.3 Installing Adobe Experience Manager on Your System

In general, when you want to install Adobe Experience Manager on your system, you would follow this procedure:

1. Create a specific folder structure for your Adobe Experience Manager instance.
    a. Author instance
    For Windows: C:/adobe/AEM/author
    For Mac OS or *x: /opt/adobe/AEM/author OR /Applications/AEM/author
    b. Publish instance
    For Windows: C:/adobe/AEM/publish
    For Mac OS or *x: /opt/adobe/AEM/publish OR /Applications/AEM/publish
3. Add the **AEM Quickstart JAR** file along with the license.properties file to the folder, which you created earlier.
4. Rename the jar file to include the run mode as well as the port number. That is, rename the file to the format: aem-<run mode>-<port number>.jar
For example,
Author instance: `aem-author-4502`
Publish instance: `aem-publish-4503`

    The first time you double-click the jar file, Adobe Experience Manager will be installed on your system, creating a root folder called `crx-quickstart`, which serves as your repository.

A sample folder structure for an Author instance is shown below.



NOTE: The Adobe Experience Manager quickstart file is renamed for installation purposes. When running for the first time, the quickstart file will notice that it has to install Adobe Experience Manager. By renaming the file, you use a convention of passing instance name (Webpathcontext) and port number through the file name so that no user interaction is needed during the installation process. If no port number is provided in the file name, Adobe Experience Manager will select the first available port from the following list in this specific order: 1) 4502, 2) 8080, 3) 8081, 4) 8082, 5) 8083, 6) 8084, or a random port.

Perform Task - Start an Adobe Experience Manager Author Instance, from the Lab Activity section.

Perform Task - Start an Adobe Experience Manager Publish Instance, from the Lab Activity section.

1.4 Starting an Adobe Experience Manager Instance

There are many ways of starting an Adobe Experience Manager instance, two of which are—graphical and by command line. The latter is more powerful because the user has the possibility of providing additional performance-tuning parameters to the Java Virtual Machine (JVM).

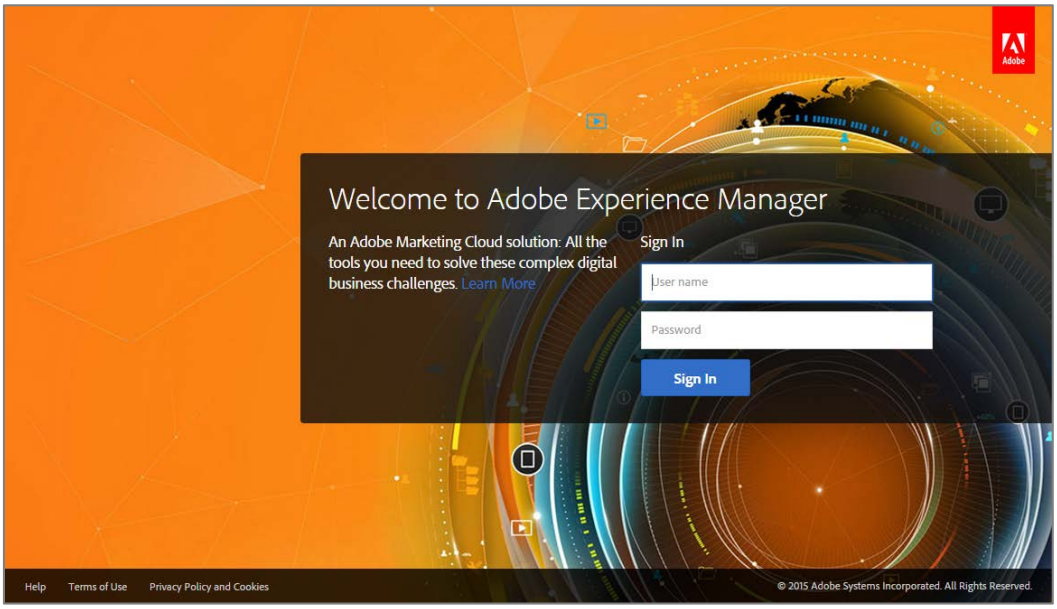
1.4.1 Using the JAR FILE to Start an Adobe Experience Manager Instance

In a Windows or Mac OS environment, you can double-click the `aem-author-4502.jar` file to start an Author instance (or the `aem-publish-4503.jar` file for a Publish instance).

- Installation will take approximately 5-7 minutes, depending on your system's capabilities.
- A dialog box will pop up similar to the following (this is known as the GUI):

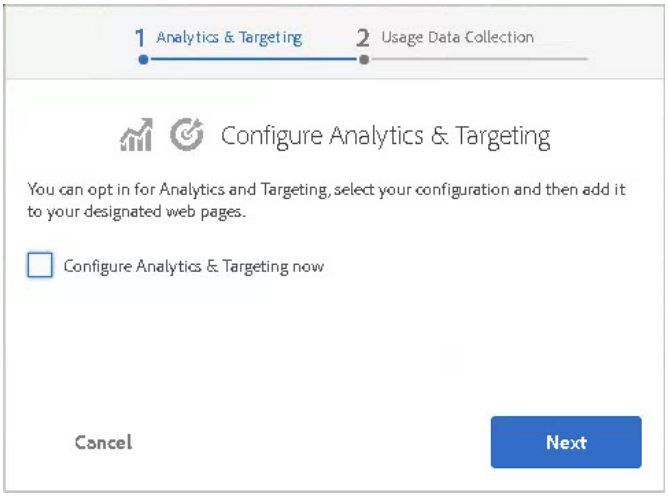
After Adobe Experience Manager starts, your default browser will open automatically, pointing to Adobe Experience Manager's start URL (where the port number is the one you defined on installation).

2. In the Sign In area that displays, enter the default administrator's credentials (`admin/admin`), and then click **Sign In**.

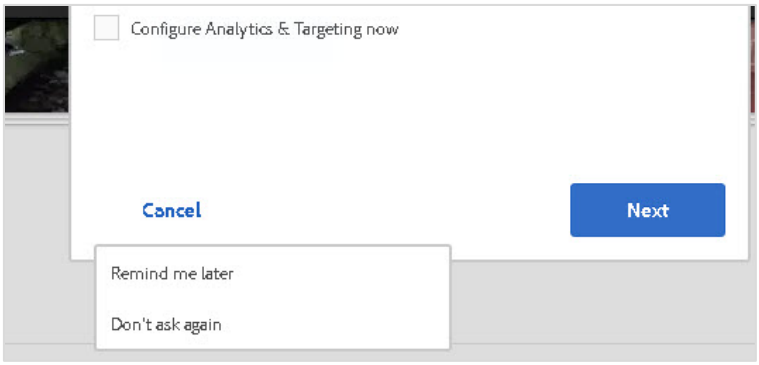


If this is the first time you are logging in, a wizard displays, asking if you want to configure Analytics & Targeting now.

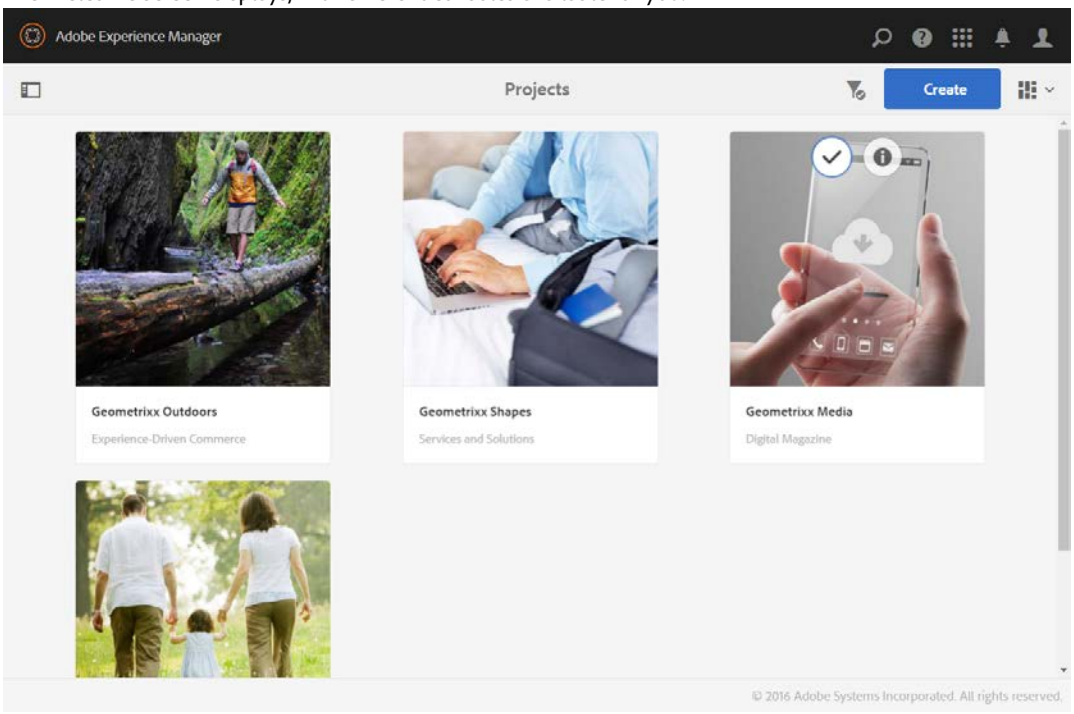
3. Click **Cancel**.



4. A hover menu opens, asking if you want to be reminded to complete the Configure Analytics & Target wizard later, or not ask again (which means it will never open again when you first log in). For this class, click Don't ask again.



5. The Welcome screen displays, with different consoles available for you.



1.4.2 Using the Command Line to Start Adobe Experience Manager Author Instance

Prior to the installation, you may want to know which parameters are available to configure quickstart. Enter the following command to display a complete list of optional parameters:

```
java -jar aem-author-4502.jar -h
```

The Adobe Experience Manager quickstart installer will show all available command-line options without starting the server. In addition, you need to tune the JVM used for running Adobe Experience Manager. Tuning the JVM is an important and delicate task and requires a more realistic environment in terms of resources (hardware, operating system, and so on) and workload (content, requests, and so on). For now, it will be enough to know that you can start your instance (Author or Publish) using the following parameters:

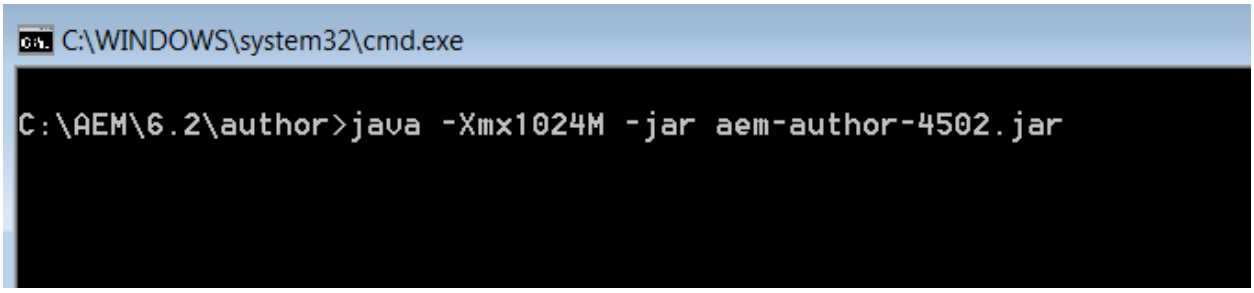
-Xms --> assigns the initial heap size

Default value	64 MB for a JVM running on 32-bit machines, or 83 MB for 64-bit machines
Recommended	Specific to physical memory available and expected traffic
Syntax	-Xms512m (sets the initial heap size to 512 MB)

-Xmx --> assigns the maximum size the heap can grow

Default value	64 MB for a JVM running on 32-bit machines, or 83 MB for 64-bit machines
Recommended	Specific to physical memory available and expected traffic, but should be equal or greater than the initial size. To run Adobe Experience Manager, it is recommended to allocate at least 1024 MB of heap size.
Syntax	-Xmx1024m (sets the maximum size for the heap. In the example, we are letting it grow to 1024 MB; however, in production, this should be higher because Adobe Experience Manager consumes a lot of resources).

You can now install and start Adobe Experience Manager from the command line together with increasing the Java heap size, which will improve performance. See the image below for an example of the command line.



You can control the way Adobe Experience Manager is installed by defining properties via file name.

Perform Task - Start and install Adobe Experience Manager using command line, from the Lab Activity section.

1.4.3 Using the Command Line to Start the Adobe Experience Manager Publish Instance

In your command prompt, navigate to the directory **/adobe/AEM/publish**, and enter the following command to install the publish instance:

```
java -jar aem-publish-4503.jar
```

Chapter 2 Lab Activity - I

Overview

Unlike many other applications, you can install Adobe Experience Manager using a quickstart, self-extracting JAR file and a license.properties file containing the license key. When you double-click the JAR file for the first time, everything you need is automatically extracted and installed.

In some scenarios, you need to start the Adobe Experience Manager instance using the command line, where you provide the argument during the instance start-up.

To begin, you need to configure and install two servers:

Author on localhost port number 4502

Publish on localhost port number 4503

Content creators and developers use Author server to create and manage the content. Visitors access the website and interact with it on the Publish instance.

NOTE: The port numbers would differ in a VILT environment.

Pre-requisites

You need the following to complete the tasks in this module:

Adobe Experience Manager quickstart JAR file

license.properties file

NOTE: In a VILT class, the installation would have already been performed.

Steps

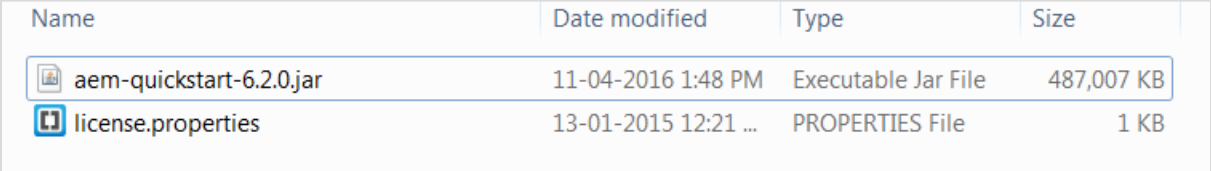
1. Task - Start an Adobe Experience Manager Author instance

   Create a folder structure on your file system where you will store, install, and start Adobe Experience Manager. For example:

   a. Windows: C:/adobe/AEM/author

   b. MacOS X: /Applications/adobe/AEM/author or *x: /opt/adobe/AEM/author

Copy the `aem-quickstart-6.2.0.jar` and `license.properties` files from USB contents.

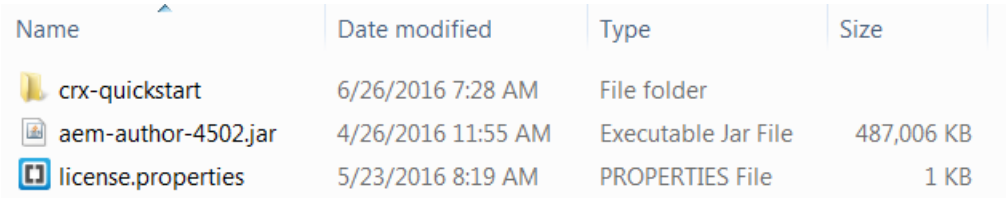


Rename the `aem-quickstart-6.2.0.jar` file to `aem-author-4502.jar`:

c. aem = Application

d. author = Web Content Management (WCM) mode it will run in (in this case, Author)

e. 4502 = Port it will run in (any available port is acceptable)



In a Windows or MacOS X environment, double-click the `aem-author-4502.jar` file. Installation will take approximately 5–7 minutes depending on your system's capabilities. After the Adobe Experience Manager Author instance has started successfully, the start-up GUI window/dialog will change to something similar to the following:

In addition, after Adobe Experience Manager starts, your default browser will automatically open to Adobe Experience Manager's start URL (where the port number is the one you defined on installation) and ask for your sign in credentials. For example: http://localhost:4502:

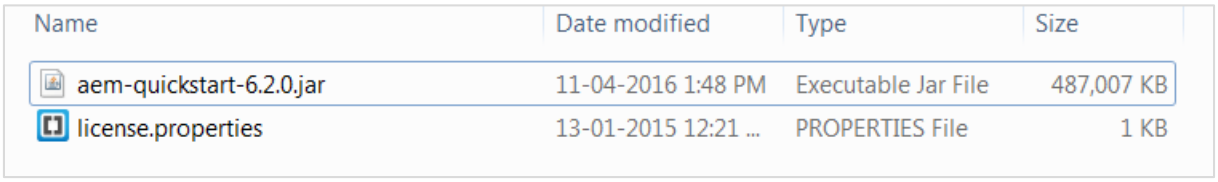
2. **Task - Start an Adobe Experience Manager Publish instance**

Create a folder structure on your file system where you will store, install, and start the Adobe Experience Manager. For example:

Windows: C:/adobe/AEM/publish

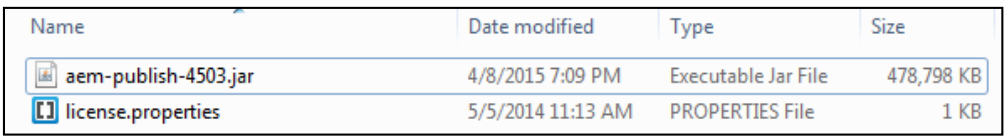
MacOS X: /Applications/adobe/AEM/publish or *x: /opt/adobe/AEM/publish

Copy the `aem-quickstart-6.2.0` JAR and `license.properties` files from USB contents.



Rename the `aem-quickstart-6.2.0.jar` file to `aem-publish-4503.jar`

- aem = Application
- publish = WCM mode it will run in
- 4503 = Port it will run in

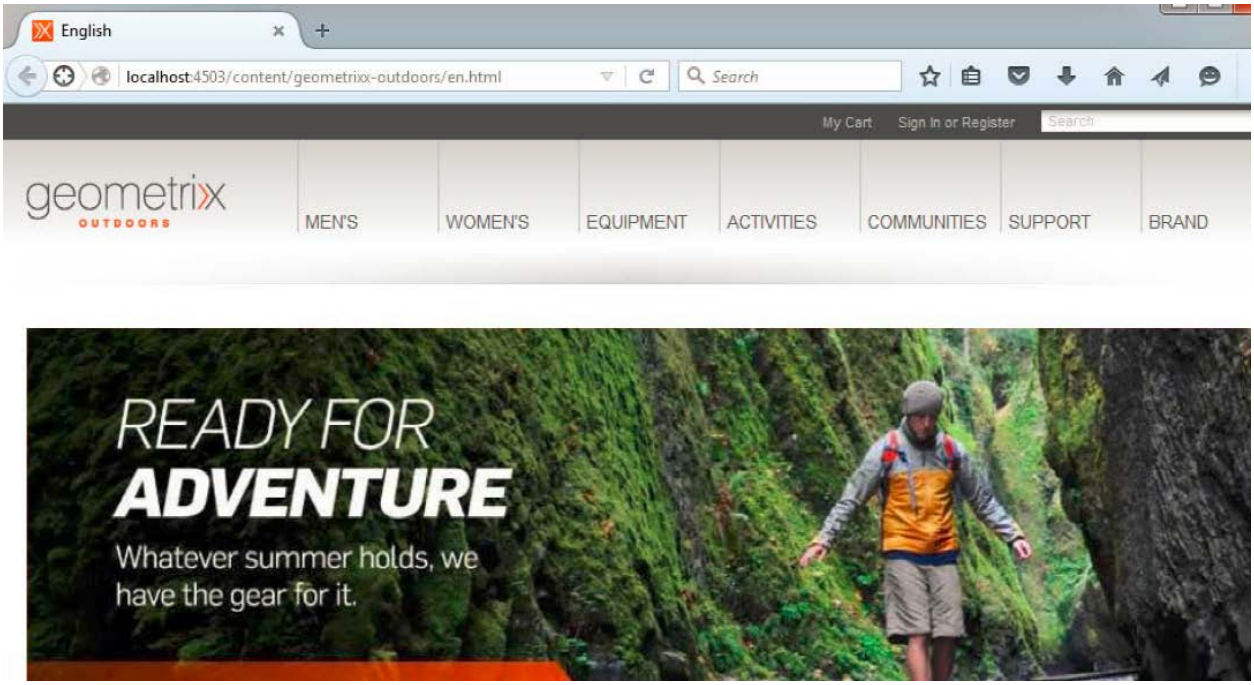


Double-click the `aem-publish-4503.jar` file to start the Publish instance.

After Adobe Experience Manager Publish instance has started successfully, the start-up screen will change to something similar to the following:

In addition, the Adobe Experience Manager login page opens from your default browser (where the port number is the one you defined on installation). For example: http://localhost:4503

The Geometrixx Outdoors site appears in the browser once the Publish instance is up and running. If you are wondering why it shows this, it is based on the root mapping set out of the box for AEM:



You have now successfully installed and started Adobe Experience Manager Author and Publish instances on localhost. To start Adobe Experience Manager in the future, double-click the renamed `aem-quickstart-6.2.0.jar` file; for example, `aem-author-4502.jar`.

3. Task - Start and install Adobe Experience Manager using command line [optional]

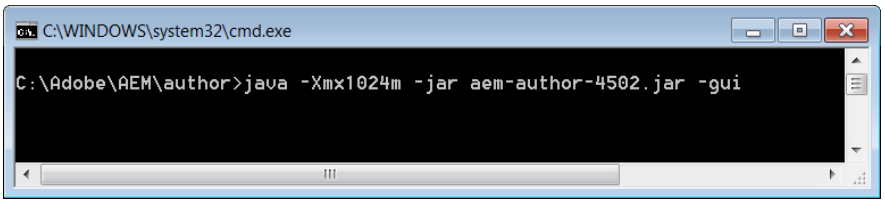
You already have an author instance and a publish instance running. Perform this task only when necessary or as an add-on exercise to try out beyond this class.

This is a powerful method because the user can provide additional performance-tuning parameters to the Java Virtual Machine (JVM). On Windows, MacOS X, or *x, you can install or start Adobe Experience Manager from the command line, while increasing the Java heap size, which improves performance.

A typical command line start will have the following:

```
java -Xmx1024m -jar aem-author-4502.jar -v
```

This example below starts AEM author runmode with a specific memory allocation to the JVM and the GUI window "on":



In your command prompt, navigate to the Adobe\AEM\author directory, and use the following command to install Adobe Experience Manager, without installing the Geometrixx sites:

```
java -jar aem-author-4502.jar -r author, nosamplecontent -gui
```

TIP: To open a directory in Windows Explorer in a command-line, select the directory, hold down the **Shift** key, and right-click. Then, you will see an option to open that directory in a command-line window.

Summary

You should now be able to:

- install and run the Adobe Experience Manager Author instance
- install and run the Adobe Experience Manager Publish instance

CHAPTER THREE: LEVERAGING THE DEVELOPER TOOLSET

Overview

This chapter contains information and hands-on exercises that will help you discover and learn about the development tools available in Adobe Experience Manager, such as CRXDE Lite, Eclipse, Maven, and VLT. You will also learn to use the packaging tools to build your project packages for distribution.

Objectives

By the end of this chapter, you will be able to:

- use the developer tools available in Adobe Experience Manager to build your site
- use Eclipse, Maven, and VLT for development

The Web Console

The Web Console in Adobe Experience Manager is based on the Apache Felix Web Management Console. It is used to manage various bundles and configurations. Any changes made through this console are automatically applied to the running system, without the need to restart the instance.

You can access the console through this link: http://localhost:4502/system/console

This opens the Adobe Experience Manager Web Consoles Bundles home page – the URL above will change to: http://localhost:4502/system/console/bundles

The Web Console's toolbar features the following selections, but at this point, we are only interested in the OSGi tab:

- Main
- OSGi
- Sling
- Status
- Web Console

The Web Console has the following major selections under the OSGi tab.

- Bundles—used for installing and managing bundles.
- Components—used for managing and controlling the status of components required for Adobe Experience Manager.
- Configuration—used for configuring the OSGi bundles, and is the underlying mechanism for configuring Adobe Experience Manager parameters.

The following selections are available on the OSGi tab, but we will not be using them for the class as most of our labs are concentrated on the first three options—Bundles, Components, and Configuration.

- Events—used for the recording of events that occur.
- Log Service—used for recording messages' output to the Web Console.
- Package Dependencies—used for recording packages and class names.
- Services—used for displaying and recording services being used by the client.

Developing with CRXDE Lite

CRXDE Lite is embedded into Adobe Experience Manager and allows you to perform common development and administration tasks in the browser. As it is embedded in the server and is always available, CRXDE Lite is often the preferred tool for administrators and developers for working with nodes and properties in the JCR/CRX repository. It gives quick and direct access to the repository for observation, configuration, and development.

With CRXDE Lite, you can create a project, create and edit files (like .jsp and .java), folders, templates, components, dialogs, nodes, properties, and bundles. CRXDE Lite is recommended for most repository-level administration tasks as well as many lightweight development tasks.

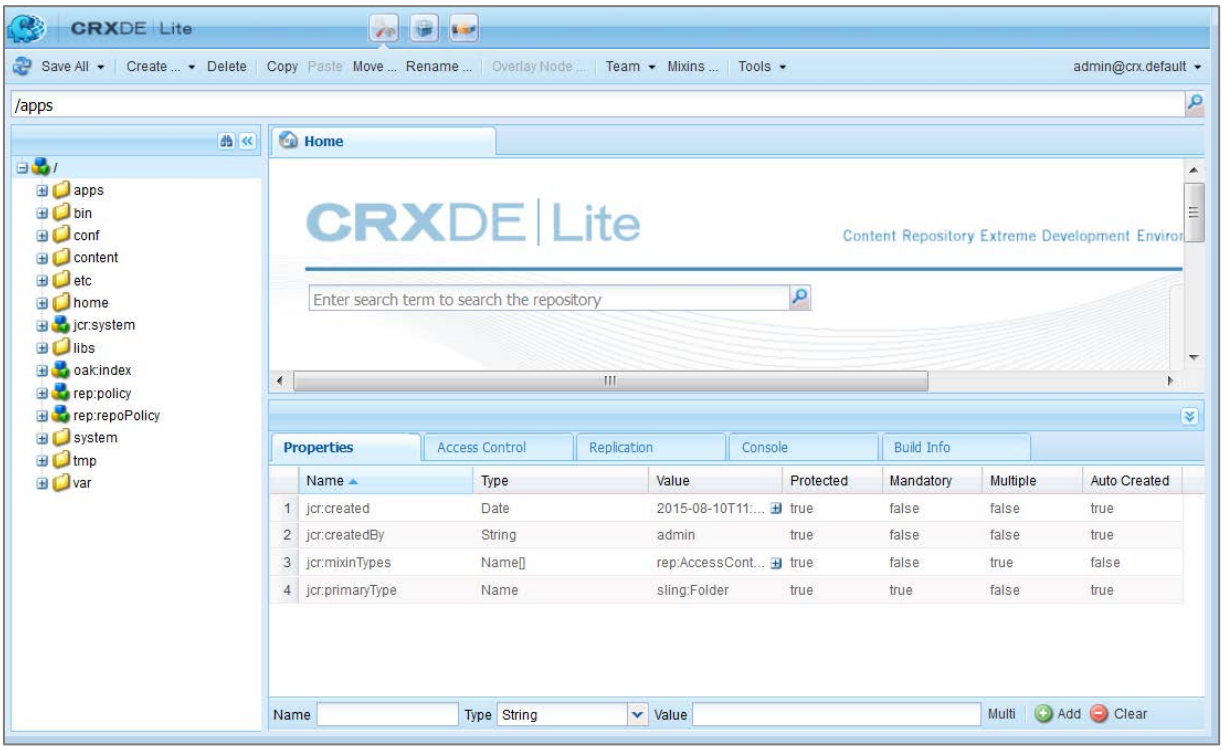
- **Explorer pane:** Displays a tree of all the nodes in the repository. You can perform the following actions on a node in the tree:

- Select the node and view its properties in the **Properties** tab. Examine all the JCR properties of various nodes.
- Right-click the node and perform an action on it, such as, renaming the node, creating a new node, creating a folder, creating a file, and so on. You can see some of the common types of folders and nodes that will be used in the training class.

- **Edit pane:** Double-click a file in the **Explorer** pane to display its content; for example, a .jsp or a .html file. You can then modify it and save the changes.
- **Properties tab:** Displays the properties of the node that you selected. You can add new properties or delete existing ones.

You can access this console through the following link: http://localhost:4502/crx/de/index.jsp

NOTE: You will use this interface frequently in the training. It is recommended to use this interface when there is no direct access to the Adobe Experience Manager/CRX server.



It is recommended that you use Eclipse or Brackets for application development. You can use CRXDE Lite as a window to the content repository.

Working with Packages

Throughout this training, you will be working with packages; either by creating your own or installing an available package. These packages allow you to import and export repository content. A package may contain:

- Page-related content
- Project-related content
- Vault meta information such as filter definitions and import configuration information
- Other package information such as package settings, package filters, package screenshots, and package icons

You would commonly use packages for any of the following:

- New functionality installation
- Content transfer between instances
- Repository content backup
- Content export to the local file system

There are two ways you can work with packages:

Package Manager—can be used to import or export content, transfer content between instances, and back up repository content. Using filters, you can create a package to contain page content or project-related content. You can access this console through the following link: http://localhost:4502/crx/packmgr/index.jsp With the Package Manager, you can perform the following common tasks:

a. Create, build, and download content packages

b. Upload and install packages

c. Modify existing packages

d. View package information

Package Share—a centralized server that contains both public and private packages available across all instances. Public packages may include hotfixes, new functionality, documentation, and so on.

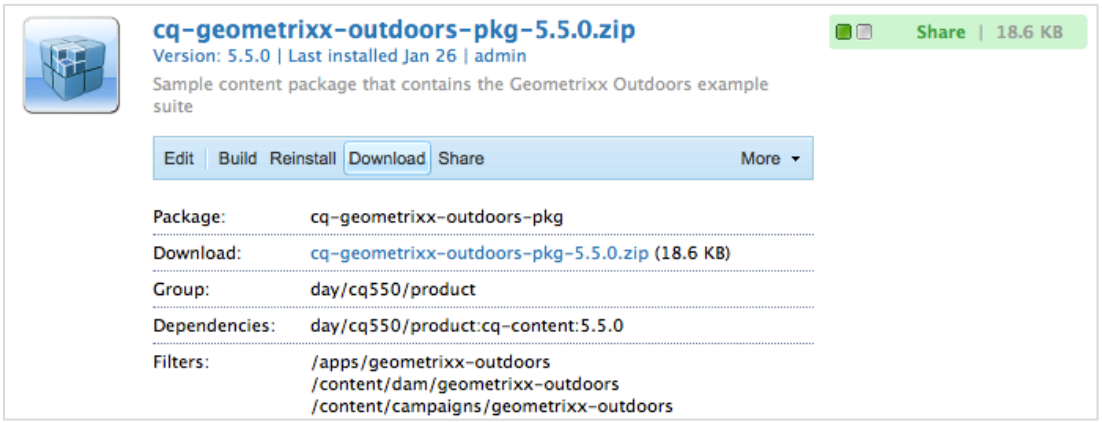
Creating and Building New Packages

Packages are created by applying filters and rules that will extract content from the repository. You can do this using the Package Manager available in Adobe Experience Manager. Once you define a package, you can build it. Building a package results in the creation of a .zip file that can be downloaded to your local file system. You can test the contents of the package before building it. There are several options available for a package. Some of them are:

- **Rebuild**—rebuilds if there is a change in the repository content.
- **Edit**—edits filters or rules applied to the package.
- **Test**—performs a dry run of the installation.
- **Rewrap**—recreates the package with additional information such as thumbnails and icons. An example of when this is required is when you might have to share this package on Package Share.

Downloading packages to your file system

Packages are in the form of a .zip file that you can download to your file system. To download a package, select the package from the list and click the download link available when the package details are expanded. Once downloaded, you can unzip them to retrieve the contents of the package.



Typically, a content package contains the following folders:

- jcr_root: This folder represents the root node of the repository, and contains the actual content of the package.
- META-INF: This folder contains metadata regarding node definitions and also contains the filter.xml file, which gives directions to FileVault about which paths to include.

Perform Task - Installing a content package, from the Lab Activity section.

Perform Task - Create, Build, and Download a Package, from the Lab Activity section.

Using the Package Share

Package Share is a centralized server where public and private packages are made available. These packages may be hotfixes, feature sets, updates, or Adobe Experience Manager content generated by other users. You can search, download, and install any package either to your instance or to your local file system.

Within the Package Share, you have access to the following:

- Adobe packages provided by Adobe
- Shared packages provided by others companies and made public by Adobe
- Your company packages that are private

You can access the Package Share with this link: localhost:4502/crx/packageshare/index.html

NOTE:

- You need to have an Adobe ID to use the Package Share. For more information, refer to the online documentation on Package Share.
- You can directly access Package Share through the following link, which opens a Windows Security dialog where you must enter your Adobe Experience Manager Cloud Management credentials to log in: https://www.adobeaemcloud.com/content/packageshare.html

Chapter 3 Lab Activity - II

Scenario

You are assigned to a project that requires your project artifacts like code, components, and so on, with several members of a team. To do this, you need to configure new development tools apart from the in-built Adobe Experience Manager capabilities.

Challenge

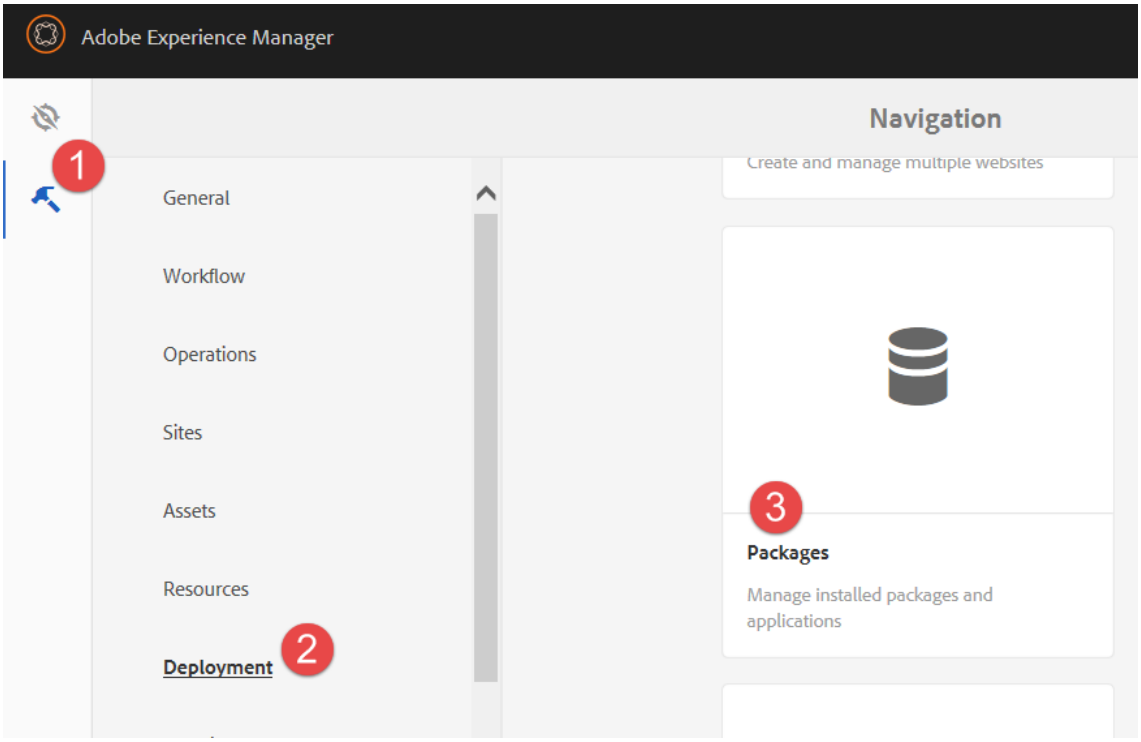
Identify the code editor for easy use, catering to developers comfortable with simple HTML editors. Identify the developer tool available in Adobe Experience Manager that enables packaging of developer artifacts for distribution.

Steps

1. **Task - Install a package in Adobe Experience Manager**
   Navigate to the Projects console, or click the following link: http://localhost:4502

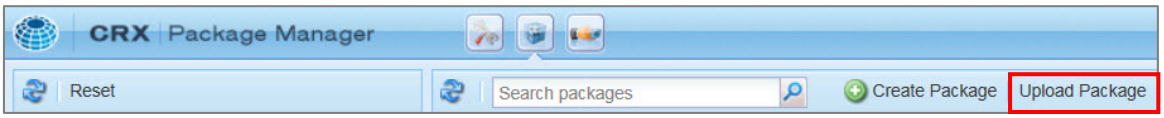
   Click Adobe Experience Manager in the upper-left corner.

   Open the Package Manager of CRXDE Lite:



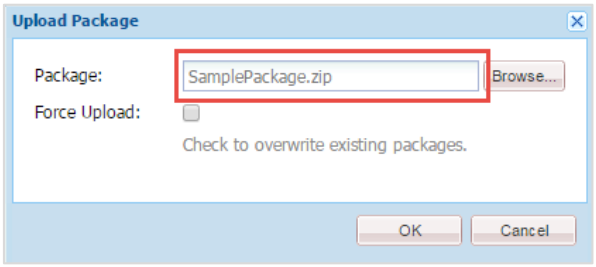
   a. As an alternative, click the following link to open the page directly: http://localhost:4502/crx/packmgr/index.jsp

Click Upload Package.

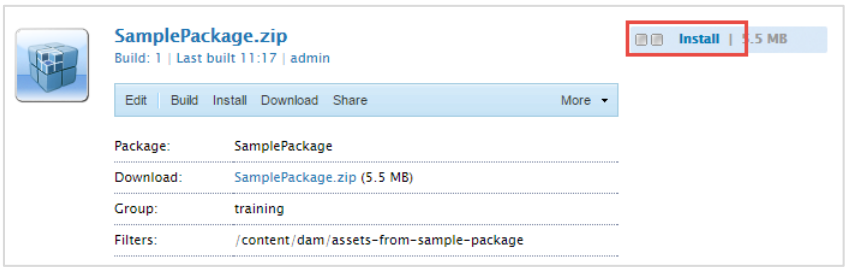


In the **Upload Package** dialog box, click **Browse**, and select the `SamplePackage.zip` package from the task files provided to you then click **Open**.

Once the Upload Package dialog box pre-populates with the `SamplePackage.zip` filename, click **OK**.

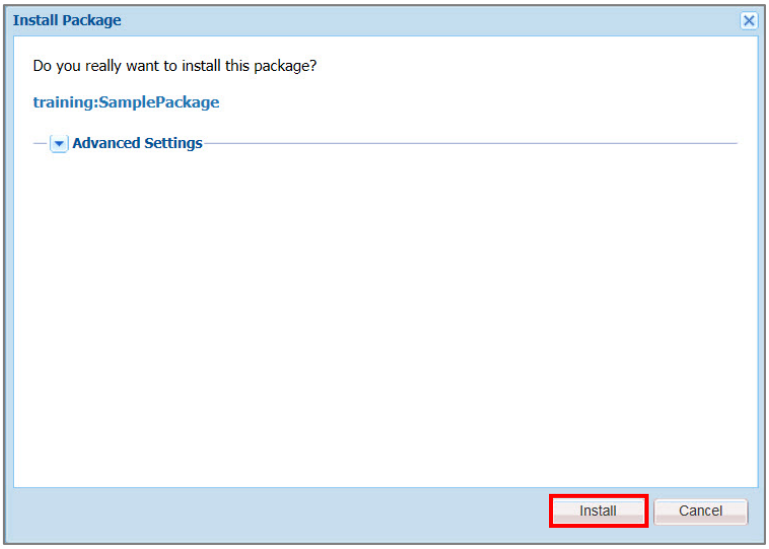


After the package is uploaded, click **Install**.



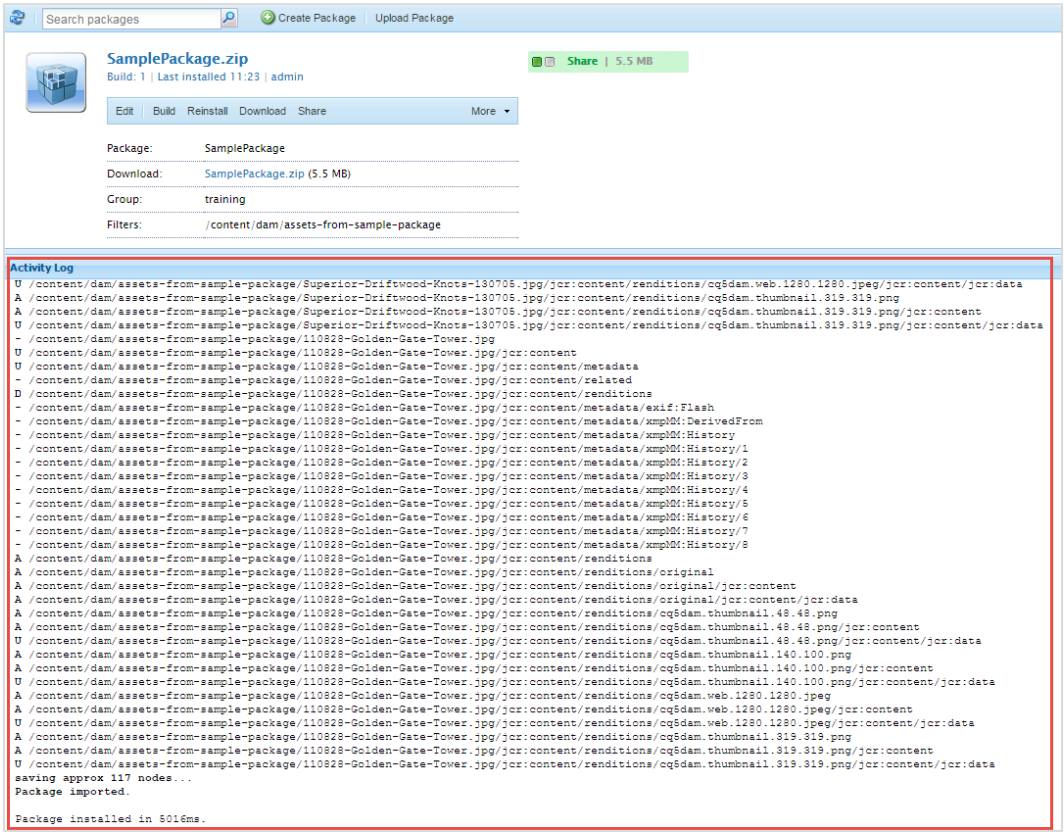
A dialog box opens, asking if you want to install the package. You can ignore the **Advanced Settings** drop-down.

In the Install Package dialog box, click Install.

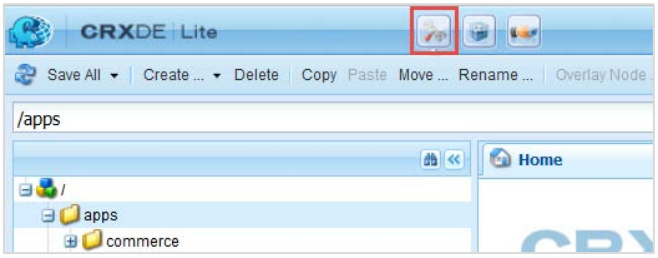


Check the **Activity Log** below the SamplePackage.zip information. You can see the content that was added from the package. The Activity Log will also tell you how many "nodes" (assets, components) were saved and how

long it took to install the package.

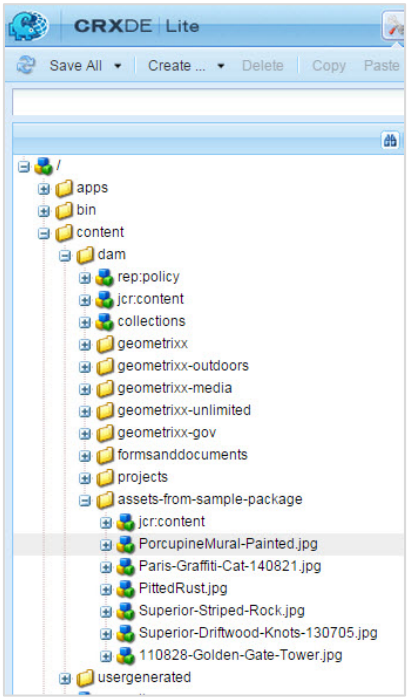


Go back to CRXDE Lite by clicking the **Develop** button at the top.



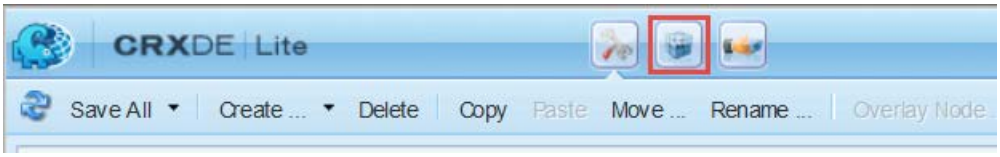
Check the **/content/dam/ assets-from-sample-pacakge** folder in CRXDE Lite to see the changes reflected there – the **assets-from-sample-pacakge** is the result of you uploading the SamplePackage.zip file. The contents of the

package are added to the repository.

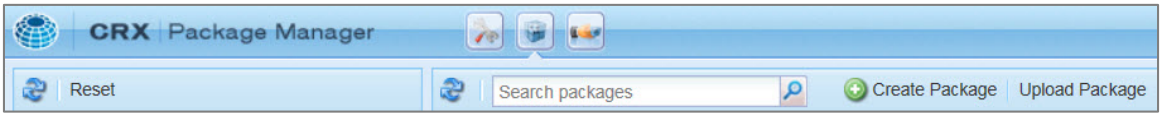


2. Task - Create, Build, and Download a Package

Navigate back to the **Package Manager** (or go to http://localhost:4502/crx/packmgr/index.jsp):

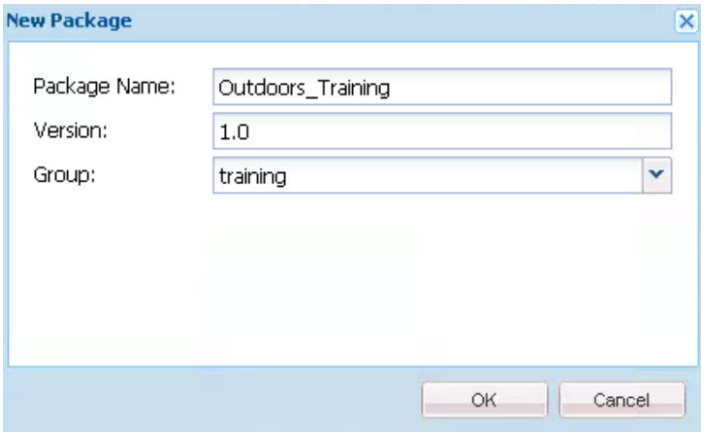


Click Create Package.

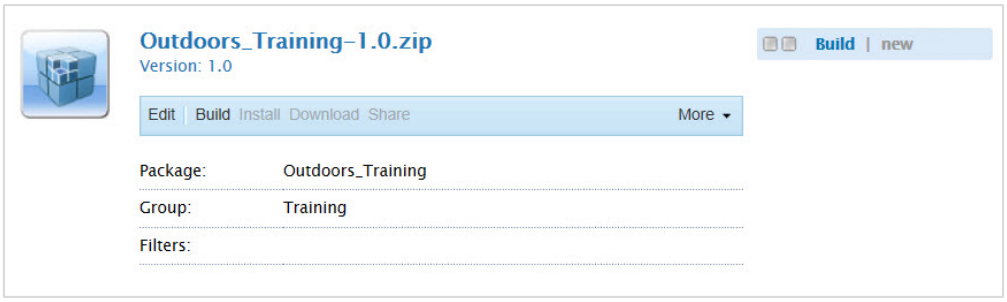


In the **New Package** dialog box, enter the following details:

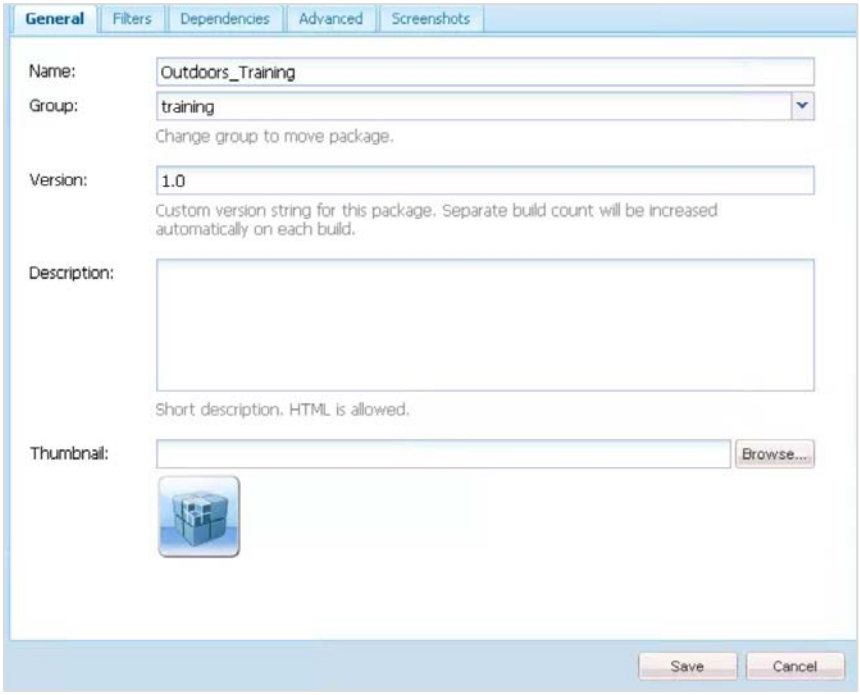
Property	Value
Package Name	Outdoors_Training
Version	1.0
Group	training



Click **OK**. The Outdoors_Training-1.0.zip is created.



Click **Edit** on the newly created package. The **Edit Package** dialog box opens.



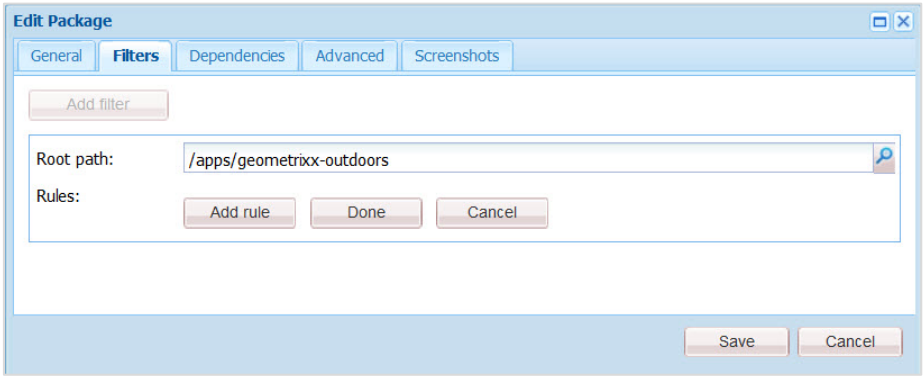
To add filters to the package, click the **Filters** tab and then click **Add Filter**.

For the **Root path:**

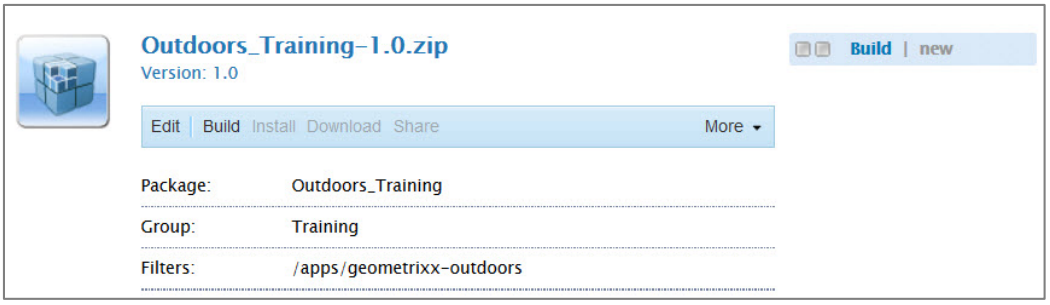
a. Click the magnifying glass ( ) then click **apps.**

b. Select the **geometrixx-outdoors** project then click **OK**.

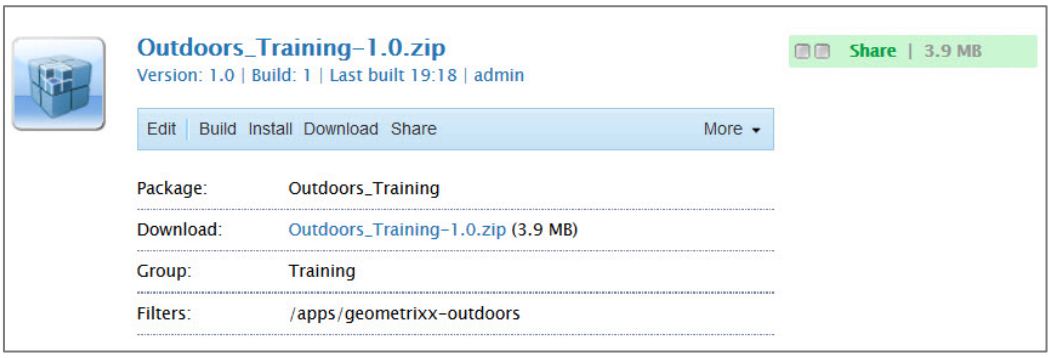
c. Click **Done**, and then click **Save**.



Click **Build** to build the package, and then click **Build** again in the confirmation dialog box. The Activity Log once again logs all events associated with building the **Outdoors-Training–1.0.zip**. The package is now ready for download.



Click **Download** to download the package. In the Browser in the lower-left corner, you will see the zip has downloaded.



Scenario Conclusion

To ensure you could successfully develop and share artifacts with the team, you have packaged the developer artifacts.

Configuring the Development Environment

For this training course, you will use the following development tools:

- Maven: This tool is used to build and deploy bundles to the JCR.
- Eclipse: This tool is used to edit code, build OSGi bundles, and Unit tests.
- FileVault: This tool is used to perform synchronization of the content with the server.

NOTE:

- VLT is also known as the FileVault tool. We will discuss this tool more later in this Chapter (Using FileVault).
- The maven-sling-plugin uses VLT to synchronize bundles with the JCR.
- In this course, we will use VLT in this chapter for content synchronization and to migrate content from a legacy system (in a later chapter). The content and exercises involving VLT included in this student guide are for illustration purposes only.

The following sections guide you to install and configure Maven and Eclipse and have your development environment ready to start working on your projects.

Working with Maven

Maven is a tool that you can use to build and manage Java-based projects. It specifies how software is built, as well as its dependencies. Maven has a central repository from which it can dynamically download Java libraries and plugins, and store them in a local cache. This cache can also be updated with artifacts created by local projects. Public repositories can also be updated.

Maven projects are configured using a Project Object Model (POM) stored in a pom.xml file.

1..1 Advantages of Using Maven

The main objective of Maven is to allow a developer to comprehend the complete state of a development effort in a minimal amount of time. With Maven, you have the following advantages:

- Makes the build process easier
- Provides a uniform build system
- Provides quality project information
- Provides guidelines for best practices development
- Allows transparent migration to new features

1..2 Contents of a POM File

POM files are xml files that contain the identity and structure of the project, build configuration, and other dependencies. Being located in the root folder of your project, it identifies the project as a Maven project. A typical POM file includes:

- **General project information:** This includes the project name, website URL, and the organization. It can also include a list of developers and contributors along with the license for a project.
- **Build settings:** This section includes the directory settings of source and tests, plugins, plugin goals, and site-generation parameters.
- **Build environment:** This consists of profiles that can be activated for use in different environments. The build environment customizes the build settings for a specific environment and is often supplemented by a custom settings.xml file in the Maven repository.

  A Maven repository is a directory that stores all project files, including JAR files, plugins, artifacts, and other dependencies.
- **POM relationships or dependencies:** There may be interdependencies between projects, with their POM settings being inherited from parent POMs. These details are described in the POM relationships section, and include:
  - groupId
  - artifactId
  - Version
  - Dependencies

Maven project POMs extend a POM called the Super POM, which is located `in ${M2 _ HOME}/lib/maven-xxx-uber.jar` in a file named `pom-4.0.0.xml` under the `org.apache.maven.project` package. This defines the central Maven repository; a single remote Maven repository has an ID value of central. All Maven clients are configured to read from this repository by default, and it is also the repository of the default plugins. These can be overridden by a custom `settings.xml` file.

- **Snapshot versions:** Snapshot versions are indicated by the string `'-SNAPSHOT'`. These are used for projects under active development. If a project depends on a software component that is under active development, you can depend on a snapshot release, and Maven will periodically attempt to download the latest snapshot from the repository when you run a build.
- **Property references:** These are used to refer to properties from another part of the POM file, Java system properties, or implicit environment variables. It supports three environment variables:
  - env
  - project
  - settings
- **Plugins:** Plugins provide functionalities such as compiling source, packaging a WAR file, or running JUnit tests. These plugins are retrieved from the Maven repository. If new functionality is added to a plugin, you can make use of it by updating the version number of the plugin in a single POM configuration file.

1..3 The UberJar

The UberJar is a special JAR file provided by Adobe. The main goal of using the UberJar is to reduce the number of dependencies. Earlier, for every API being used, a separate and individual dependency had to be added to the project. This is avoided by using the UberJar.

The UberJar file contains:

- all public Java APIs exposed by Adobe Experience Manager.
- limited external libraries—all public APIs available in Adobe Experience Manager which comes from Apache Sling, Apache Jackrabbit, Apache Lucene, Google Guava, and two libraries used for image processing.
- interfaces and classes exported by an OSGi bundle in Adobe Experience Manager.
- MANIFEST.MF file with the correct package export versions for all exported packages.

1..3.1 Using the UberJar

To use the UberJar, you need to add the following elements to your pom.xml file.

- Dependency element: to add the actual dependency to your project

```
<dependency>
  <groupId>com.adobe.aem</groupId>
  <artifactId>uber-jar</artifactId>
  <version>6.2.0</version>
  <classifier>obfuscated-apis</classifier>
  <scope>provided</scope>
</dependency>
```

- Repository element

```
<repositories>
    <repository>
        <id>adobe-public-releases</id>
        <name>Adobe Public Repository</name>

<url>https://repo.adobe.com/nexus/content/groups/public/</url>
        <layout>default</layout>
    </repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>adobe-public-releases</id>
        <name>Adobe Public Repository</name>

<url>https://repo.adobe.com/nexus/content/groups/public/</url>
        <layout>default</layout>
    </pluginRepository>
</pluginRepositories>
```

If you are already using a Maven Repository Manager such as Sonatype Nexus, Apache Archiva, or JFrog Artifactory, add the appropriate configuration to your project to reference this repository manager; and add Adobe's Maven Repository to your repository manager.

1..4 Installing and Configuring Maven

The following exercises provide instructions for installing and configuring Maven on a Windows system as well as on a Mac.

Perform the following Tasks from the Lab Activity section:

- Install and Configure Maven on a MAC machine
- Install and Configure Maven on a Windows machine

Installing and Configuring Eclipse

Eclipse is an open source Integrated Development Environment (IDE) that is used to edit your project source locally on your file system. While working with Adobe Experience Manager projects, once you install Eclipse, you will also need to install the following plugins:

- Maven Integration for Eclipse (M2E)—a Maven plugin
- AEM plugin for eclipse

NOTE: Maven comes embedded in Eclipse with the M2E plugin.

Perform the Task – Install and Configure Eclipse, from the Lab Activity section.

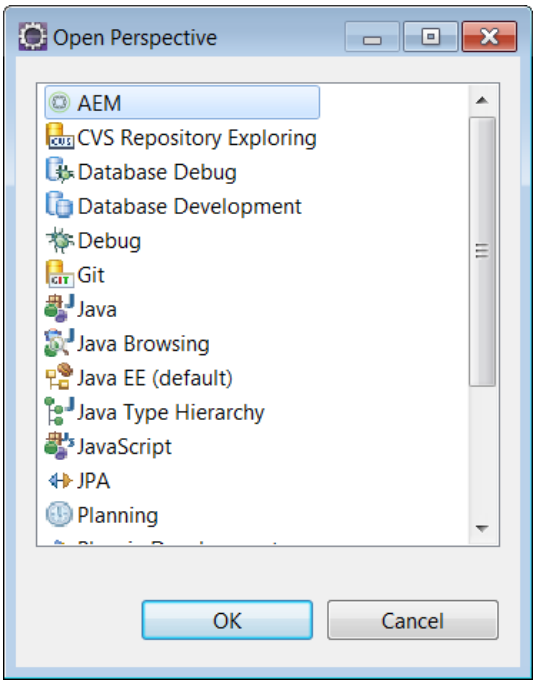
1..5 Setting up the AEM Plugin for Eclipse

The AEM plugin for Eclipse is a plugin based on the Eclipse plugin for Apache Sling. It has the following benefits:

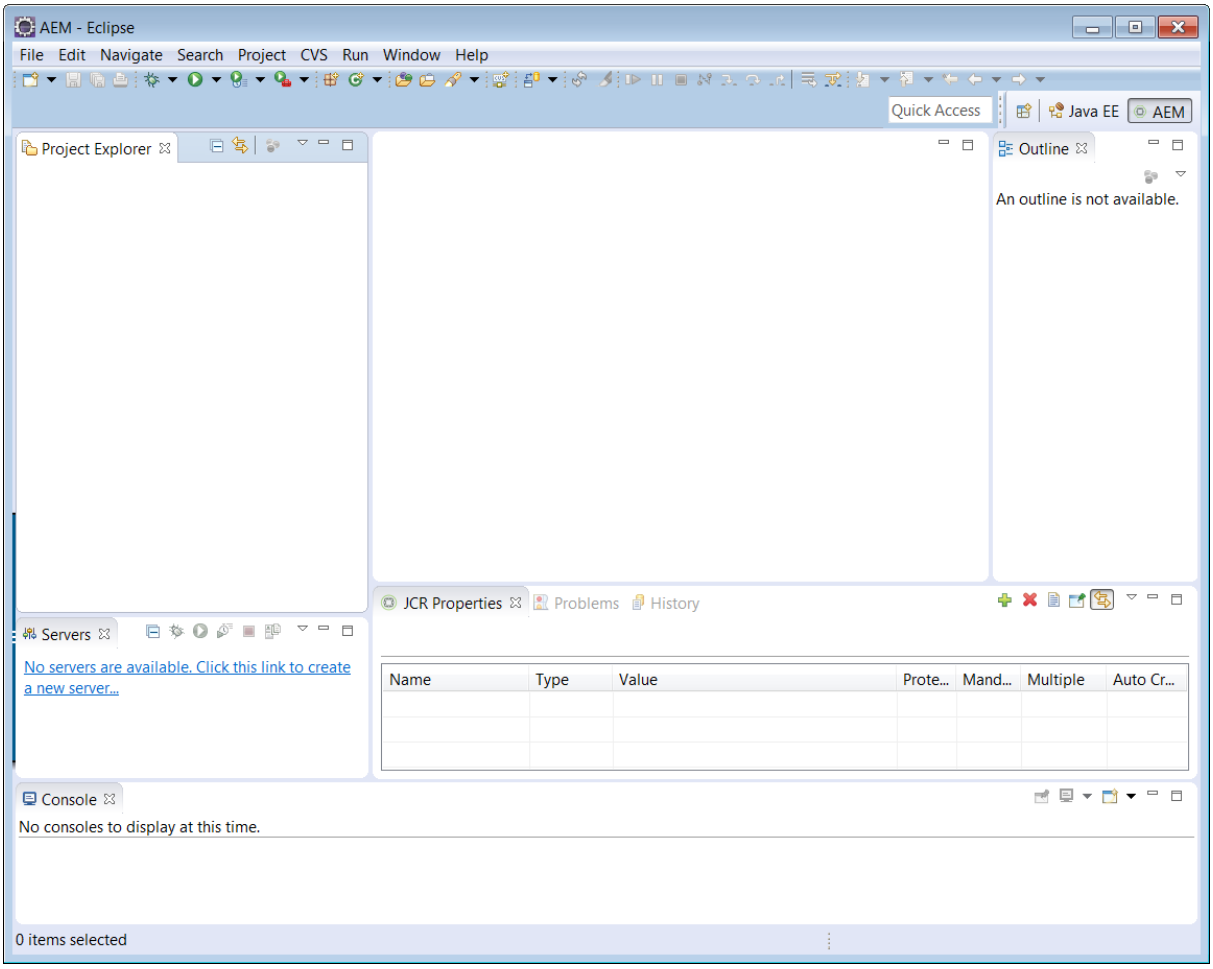
- Synchronizes both content and OSGi bundles
- Supports debugging and code hot-swapping
- Includes a project creation wizard to simplify bootstrapping of Adobe Experience Manager projects
- Enables edition of JCR properties
- Seamlessly integrates with Adobe Experience Manager instances through Eclipse Server Connector

1..5.1 Configuring the AEM Perspective

The AEM perspective offers a complete control over all your AEM projects and instances. To open the AEM perspective, navigate to Windows > Open Perspective, and select AEM then click OK.



Now, the AEM perspective displays as shown below.

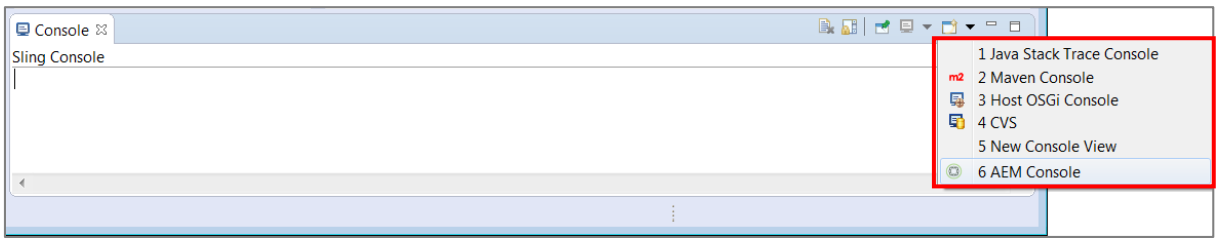


The AEM perspective makes available to you, the AEM Console, as well as the JCR properties view, where you can add and modify nodes and properties in your Adobe Experience Manager project.

1..5.2 Configuring the AEM Console

The console displays the progress of repository synchronization. That is, you can see the progress of all check in and check out of nodes and properties to and from the repository.

To configure the AEM console, in the **Console** tab, click the **Open Console** icon, and select **AEM Console**. If the Console tab is not visible, select **Window** > **Show View** > **Console** then click **Open Console** and select **AEM Console**.



1..5.3 *Working with JCR*

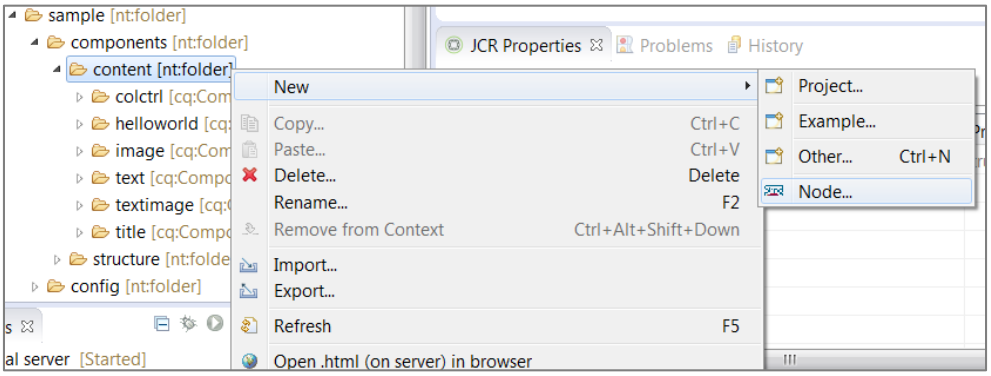
With the AEM perspective, you can do the following manipulations to the JCR:

- Add a new node.
- Add or edit properties of a node.

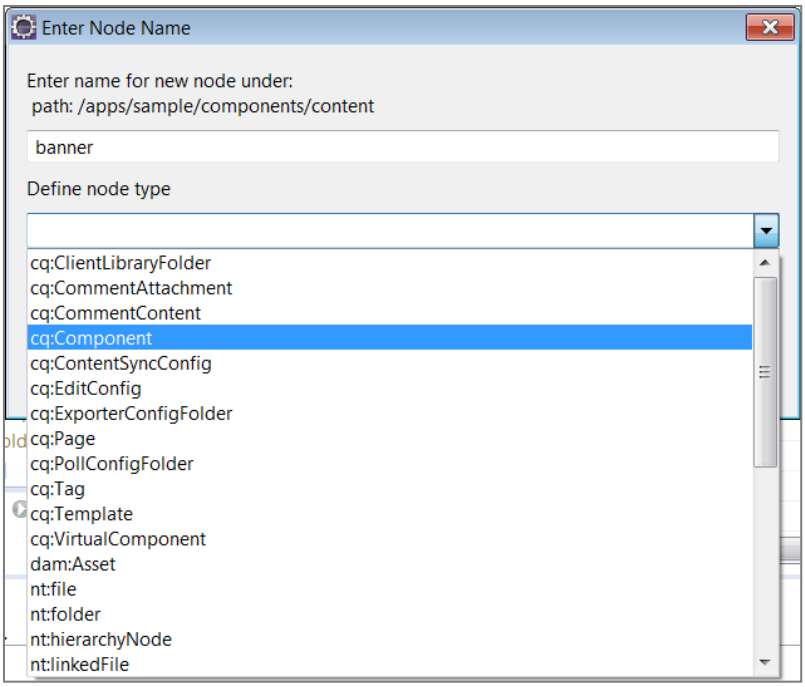
Adding a New Node

You can add a new node to the repository by following these steps:

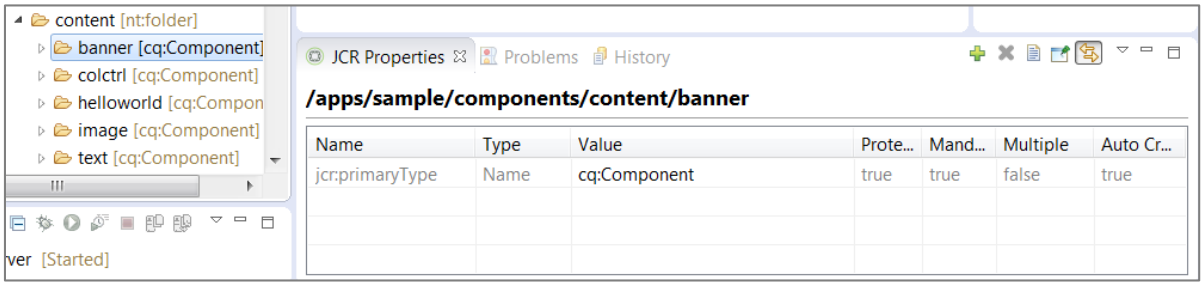
Right-click the parent node in the Project Explorer, select **New**, and click **Node...**



Give a name for the node, and select the type of node.

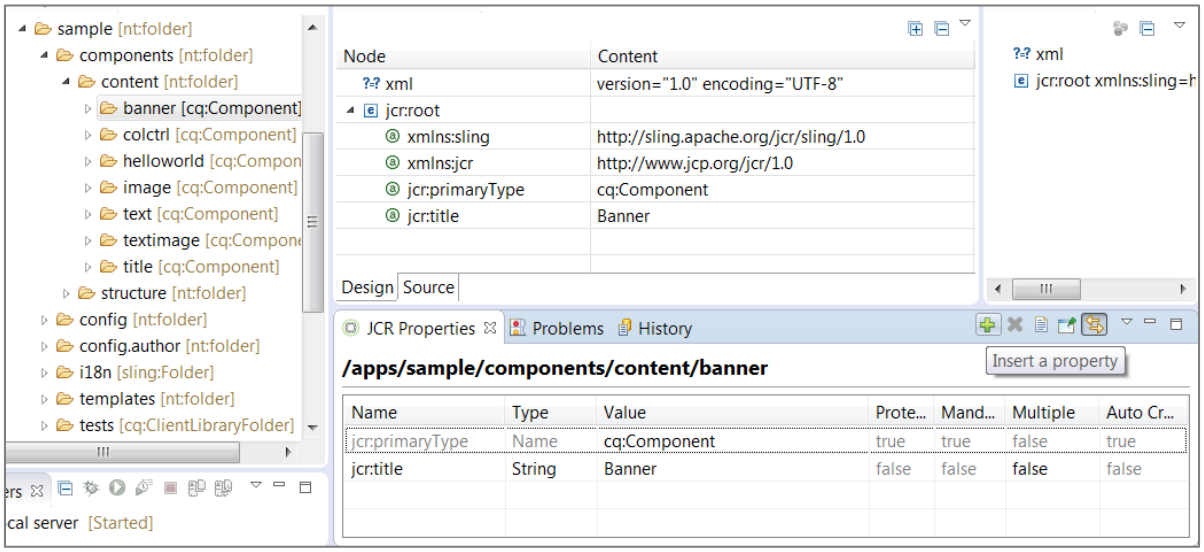


When you click on a node in the Project Explorer, you can check its properties in the JCR Properties view as shown below.



Adding or Editing a Node Property

You can add a property by selecting the Insert a property icon from the toolbar. Once you add the details of the property, press Enter to save the changes. You can right-click on the property, and select Show in editor to view the underlying vault file in the editor window.

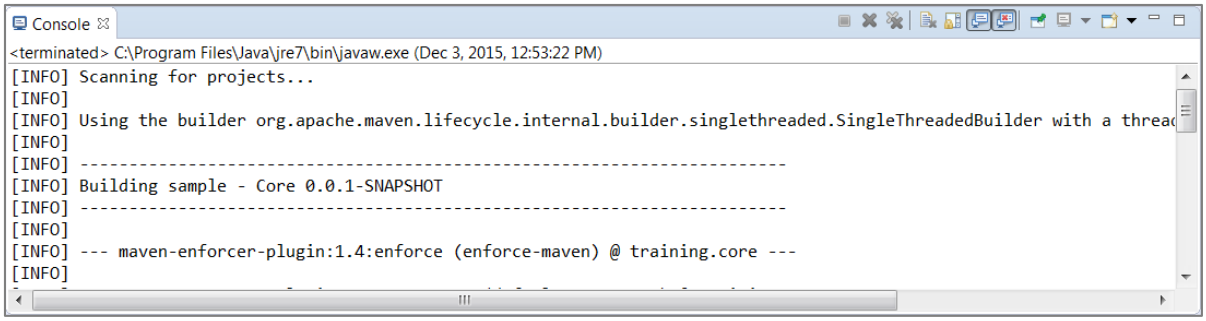


Perform the Task – Install and Configure Eclipse AEM Plugin

1..6 Building and Deploying Your Project Using Maven

Once you complete a part of your development, you may want to manually build and deploy your content package or OSGi bundle. To do this, you can select the project file from the Project Explorer, navigate to Run > Run As > Maven Install.

You can check the progress of the command in the Console.



For more information on the AEM plugin, see https://docs.adobe.com/docs/en/dev-tools/aem-eclipse.html

Setting up your project

You now have all your development tools ready, and it is time to set up your project. In the following sections, you will learn the following:

- How to configure the Maven archetype
- How to create an Adobe Experience Manager project using Maven archetypes
- How to use Maven to generate Eclipse project files

Creating an Adobe Experience Manager Project Using Maven Archetypes

Before you can create an Adobe Experience Manager project, you need to ensure that you have configured the Maven archetype. These archetypes are a type of template that can be used to simplify the project creation process.

Perform the Task – Configure the Maven Archetype for the Eclipse Plugin, from the Lab Activity section.

Now that you have everything set up, you can create your Adobe Experience Manager project.

Perform the Task – Create an Adobe Experience Manager project using the Maven archetypes, from the Lab Activity section.

Every Adobe Experience Manager project that you create has the following modules:

- **Core:** This is a Java bundle that contains all core functionality such as OSGi services, listeners, schedulers, as well as component-related Java code such as servlets or request filters.
- **Apps:** This module contains the /apps and /etc parts of the project, such as JS and CSS clientlibs, components, templates, runmode specific configs, as well as Hobbes tests.
- **Content:** This module contains sample content using the components from the apps module.
- **Tests:** This module is a set of Java bundles that contain JUnit test that are executed on the server-side.

- **Launcher:** This module contains code that deploys the tests bundle to the server and triggers the remote JUnit execution.

Once your project is set up, you can use the Export to Server and Import to Server options to synchronize content between Eclipse and the Adobe Experience Manager server.

Using Maven to generate Eclipse Project Files

Projects files are required so that the project can be correctly recreated in another workspace, in this case, Eclipse. These project files store project settings such as builder and project nature settings. These project files can be generated using Maven.

Open the command prompt from the root directory of your project that contains the POM file, and execute the following command:

```
mvn eclipse:eclipse -DdownloadSources=true -DdownloadJavadocs=true
```

On successful completion of the command, a .project file is created. To understand the command, take a look at its description below:

- eclipse:eclipse—tells Maven to generate the Eclipse project files.
- DdownloadSources—enables/disables the download of source attachments. When it is true, the remote repositories are checked for sources.
- DdownloadJavadocs—enables/disables the download of javadoc attachments. When it is true, the remote repositories are checked for javadocs.

Perform Task – Generate project files for Eclipse, from the Lab Activity section.

Using FileVault

The FileVault tool (VLT) is used to synchronize the content between CRX or Adobe Experience Manager and your local file system. This is achieved as VLT has a client-side code that issues HTTP commands to the JCR and a server-side code that outputs to the file system. VLT is a command line tool and can be used to perform normal repository operations such as check-in, check-out, as well as management and configuration operations.

Commonly Used FileVault Commands

The following are some of the most commonly used VLT commands.

Command	Description
vlt --version	Displays the version of the FileVault tool
vlt co	Performs a VLT check-out of the JCR repository to your local file system
vlt --force	Forces check-out to overwrite local files if they already exist
vlt up	Short for vlt update. Brings changes from the repository into the working directory
vlt ci	Short for vlt commit. Sends changes from your working copy to the repository

Installing and Configuring FileVault

VLT is available in the standard Adobe Experience Manager installation folder in the directory `/crx-quickstart/opt/filevault`.

Perform Task – Install and Configure VLT on your system, from the Lab Activity section.

Using FileVault to Synchronize Content with Server

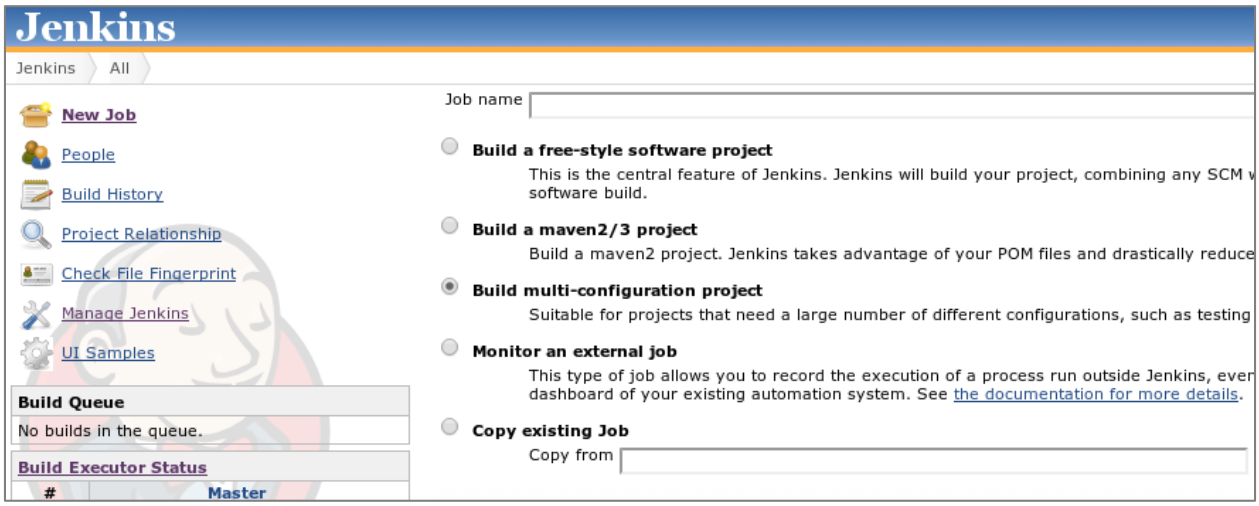
By performing a series of check-outs, updates, and commits, you can synchronize your changes between CRX and your local file system. To understand how content is mapped between the two, take a look at the following table:

CRX/Adobe Experience Manager	Local File System
nt:file	Rendered as **files**, with the data from `jcr:content` sub-nodes rendered as the contents of the file.
nt:folder	Rendered as **directories**
Any other	Rendered as **directories**, with their set of properties and values recorded in a special file called `.content.xml` within the directory of the node.

Perform Task – Use VLT to perform content synchronization

Collaborating with Teams

Most projects are a collaborative effort across various members of a team. Developers commit changes to their code base several times in a day. To make this effort run smoothly, you need to use an automated build and test suite that runs immediately after new deliveries are made. This type of Continuous Integration (CI) is possible using Jenkins.



You can use Jenkins to create jobs that are triggered as soon as new code is delivered. Members are immediately informed of any build breakages or regressions. By using Jenkins, you can collaborate very closely with your team, and ensure that your core functionality is intact.

Chapter 3 Lab Activity - III

Scenario

Your team uses a central repository for storing all development artifacts. This repository will have the latest working code that you can download into your local system. As such, at the end of the day, the code that you develop will be tested and pushed to the repository. The result of this process is that all teams can share code constantly, with any conflicts being detected at an early stage.

Challenge

You need to ensure that you have the appropriate plugins and tools that are required for building an application using Eclipse.

Pre-requisites

Eclipse Luna, Eclipse AEM plugin and VLT. These can be downloaded from their respective websites or used from the USB contents provided to you. Eclipse Luna is already integrated with Maven. Please use only Eclipse Luna provided in the USB contents.

1. Task - Install and Configure Eclipse

   Eclipse is open source software used to edit the project source locally on your file system. In this section, you will install Eclipse.

   IMPORTANT! Please use **Eclipse Luna** for this course. For this class, you must use the "Luna" version of Eclipse provided.

   Suitable versions of Eclipse Luna for Mac and Windows are provided in the USB contents.

   Extract files from the USB zip file to the destination directory of your choosing to begin the installation.

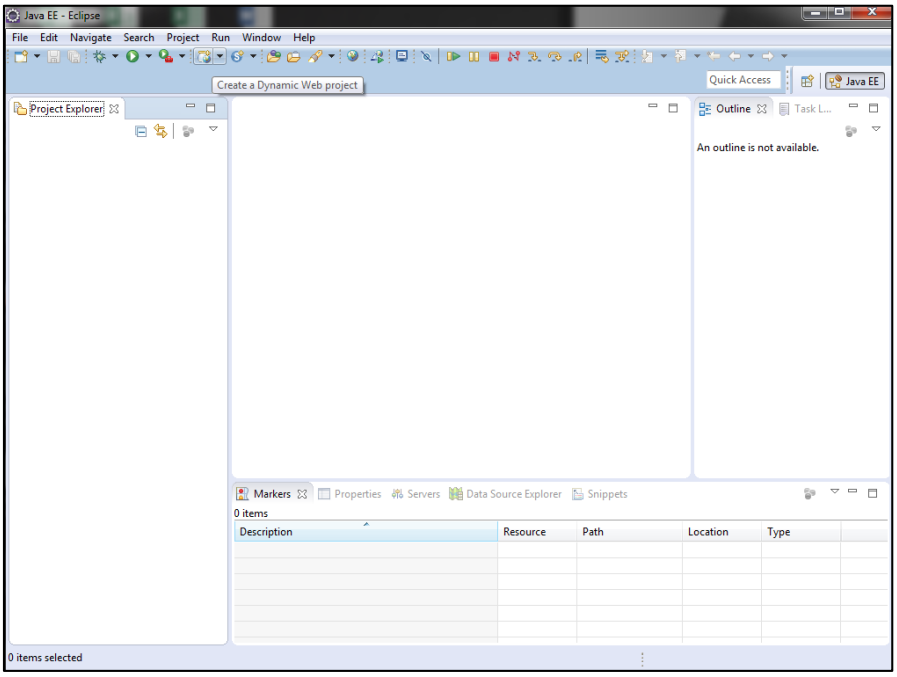
   Navigate to the directory into which you extracted the contents of the Eclipse installation zip file. For example, **C:\Program Files\Eclipse** on Windows, or **Applications/Eclipse** on the Mac. You may need to use the sudo command on the Mac to do this.

   Double-click **eclipse.exe** (or **eclipse.app**) to start Eclipse.

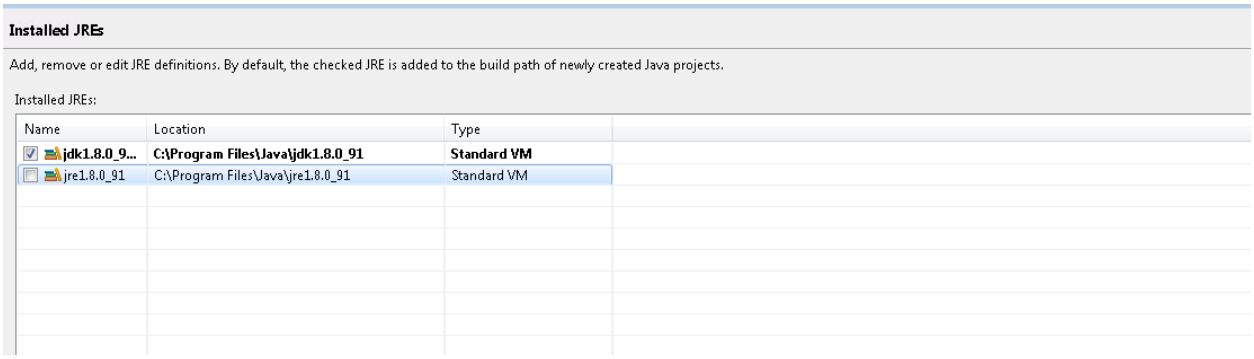
   NOTE: You can only open the Eclipse Luna application once. When open, Eclipse creates a workspace where you can perform your development tasks. You cannot have two Eclipse workspaces open at the same time. If you try to open Eclipse again, while Eclipse is already open, a "Workspace Unavailable" dialog box opens, indicating a workspace is already in use or cannot be created.

   Click the **Workbench** logo in the upper-right corner to close the welcome screen and go to the main workbench.

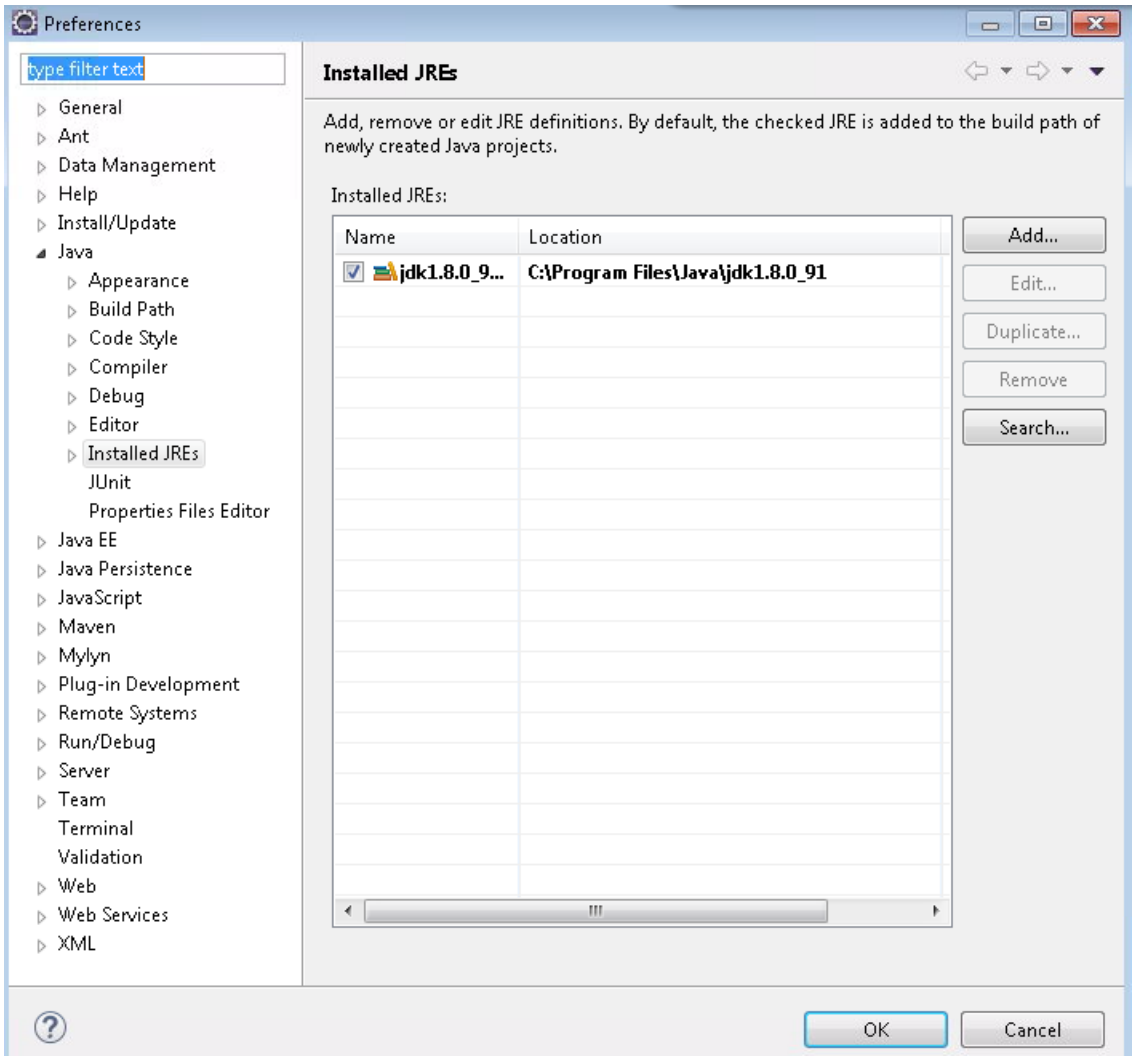


Set Eclipse's JRE

1. You need to ensure Eclipse Luna is using the correct JDK. This may be set to a JRE, not a JDK. To check this, click Window > Preferences > Java > Installed JREs. You should see something like this but if not, you need to provide the correct directory path to your JDK.

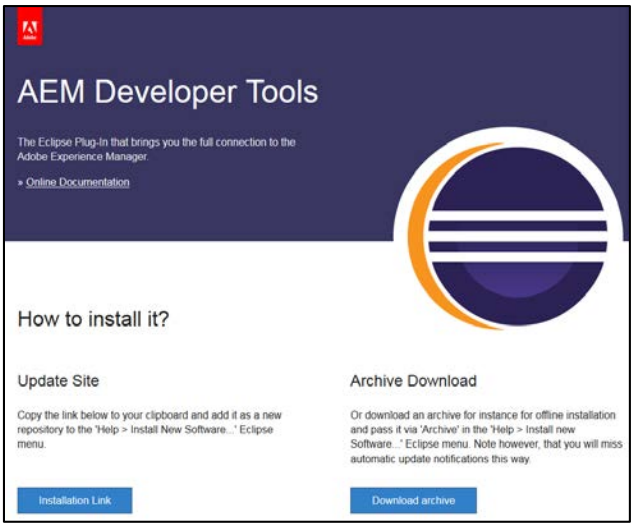


2. You should remove the reference to the jre if it exists. When done, your settings should appear as follows (remember to use JDK 1.8):



2. Task- Install and Configure Eclipse AEM Plugin

Open the URL: https://eclipse.adobe.com/aem/dev-tools/ or use the files provided in USB contents.



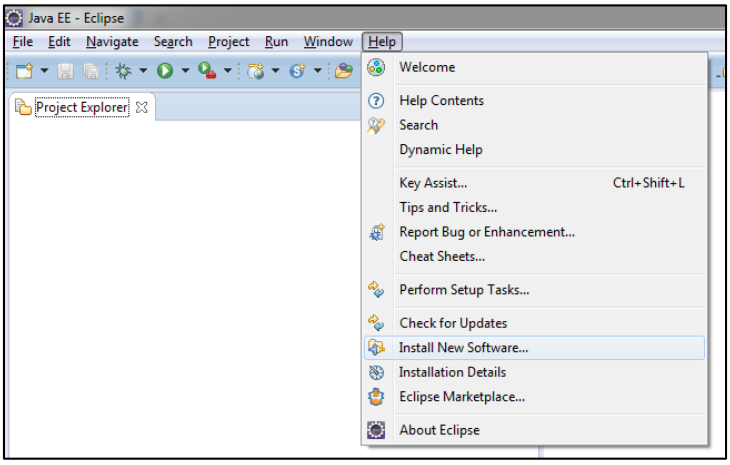
There are two ways to install the plugin:

Online: You will provide the link to install the plugin in Eclipse.
Offline: you will provide the downloaded plugin in Eclipse.

We will follow the offline method here. Please use the downloaded file from USB contents.

Open Eclipse Luna by double-clicking on **eclipse.exe** (or **eclipse.app**).

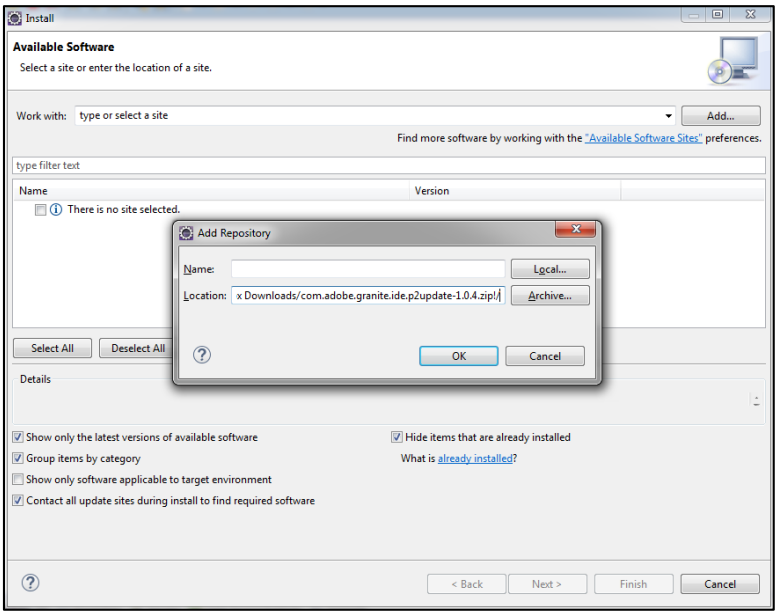
Click Help > Install New Software…



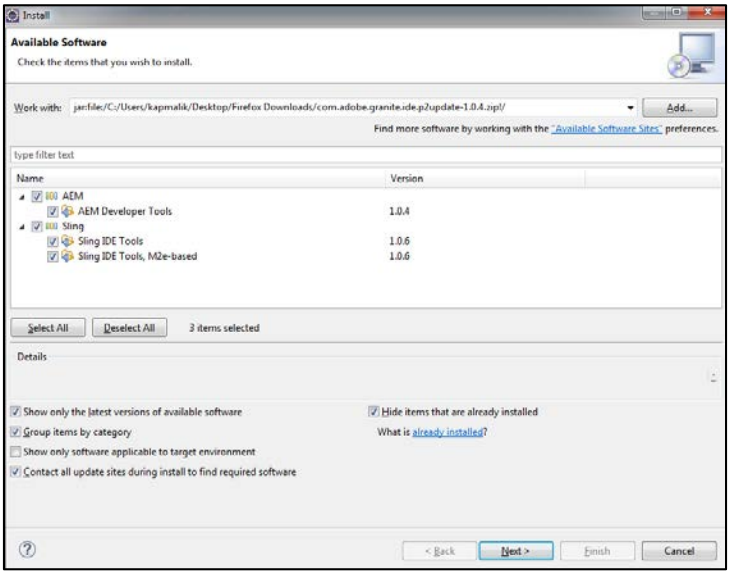
Click **Add…** In **Add Repository** popup, click **Archive**…; navigate to the repository archive and select the zip file (**com.adobe.granite.ide.p2update-1.0.4.zip**) provided for the plugin from USB content.

Tip: If you are having trouble locating the plugin, look for the Supported Applications folder in your USB content and navigate to the following path: **Supported Applications\Common\eclipse\plugins**

Click **OK**.



Select the Tools listed in the window by clicking **Select All** then click **Next**.



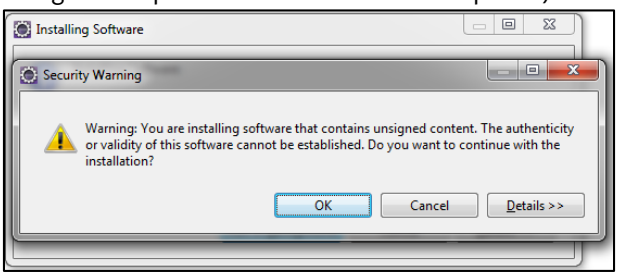
An Install Details window opens, allowing you to review the tools you are about to install.

Click **Next**, which opens the Review Licenses screen.

Click the radio button for "**I accept the terms of the license agreements**" then click **Finish**. An Installing Software dialog opens with a progress bar. The installation should only take a minute or two.

NOTE: Do not click Run in Background.

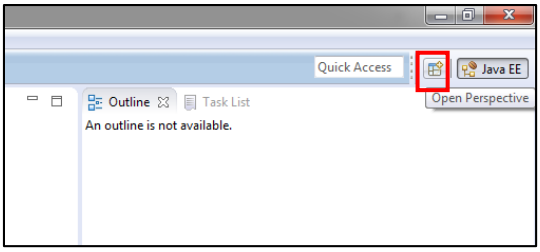
In case you receive security warning, click **OK** to continue installation. (After you click OK, the Installing Software dialog will re-open until the installation is completed.)



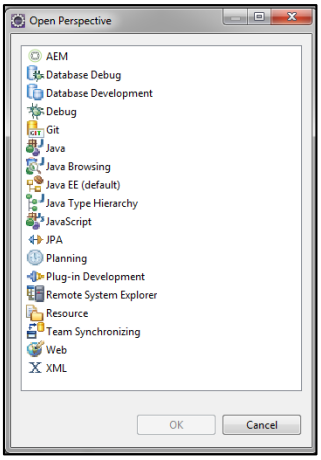
Click **Yes** to restart Eclipse to load the newly installed tools. This will only take about a minute. (If you see a dialog that asks if you want to keep the current location of your Eclipse install, click **OK**.) Eclipse opens.

Click **Workbench** in the upper-right corner again.

Notice the new **AEM** perspective available in Eclipse. To check this, click the **Open Perspective** icon as highlighted in the screenshot below.

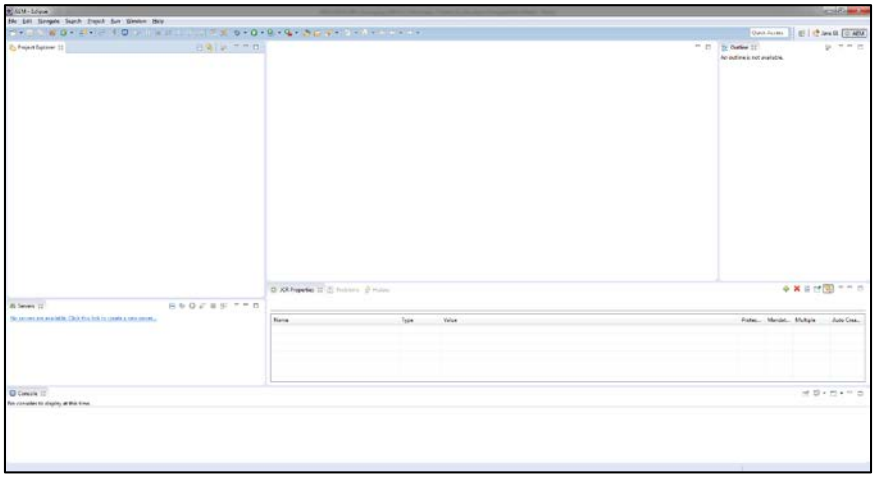


The AEM perspective is at the top:



Select **AEM** and click **OK**.

You will see a new perspective opened in Eclipse as follows:



You are now ready with the Eclipse and plugin.

At this point, you installed and configured Eclipse, and Eclipse AEM plugin. These tools will help you work on Adobe Experience Manager projects in a team environment.

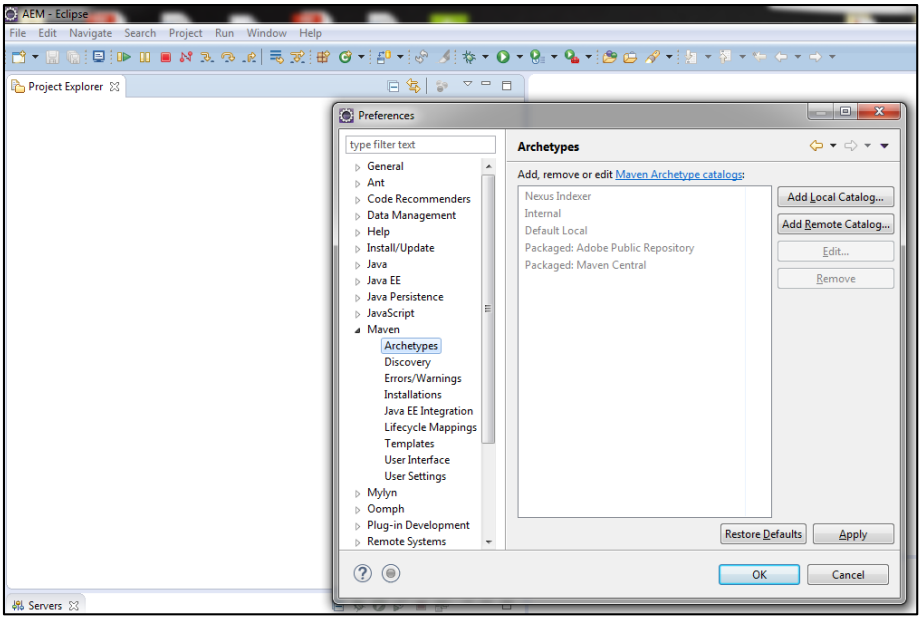
3. Task - Configure the Maven Archetype for the Eclipse Plugin

Archetype is a Maven project templating toolkit. An archetype is defined as an original pattern or model from which all other things of the same kind are made.

To configure Maven archetype, you need to have the AEM plugin installed for Eclipse, which you installed in the previous exercise.

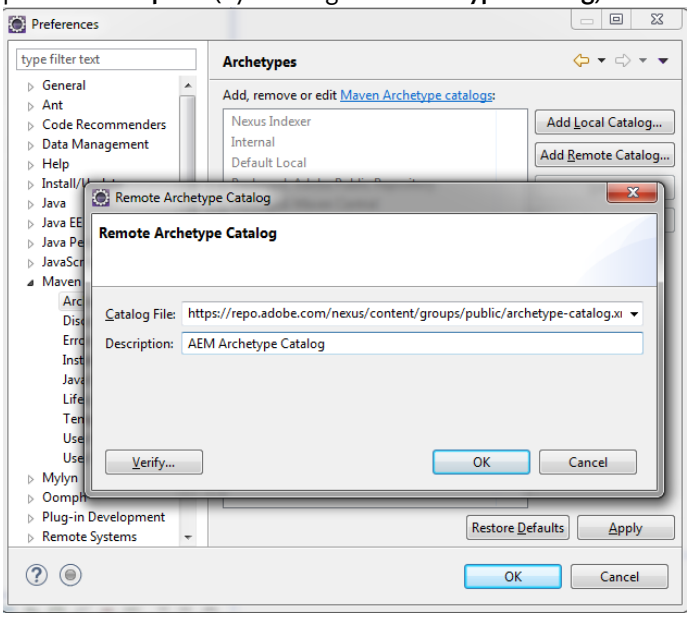
Complete the following steps to configure the Maven archetype:

Click Window > Preferences > Maven, and then select Archetypes.

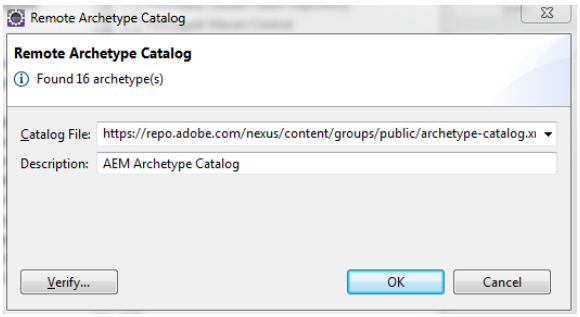


Click Add Remote Catalog…

Give the **Catalog File** url https://repo.adobe.com/nexus/content/groups/public/archetype-catalog.xml and provide **Description** (by entering **AEM Archetype Catalog**) then click **Verify**.



This checks and shows the number of archetypes in the catalog file provided. Eclipse displays the number of archetypes in the upper-left corner of the dialog box as shown. In this exercise, you will see 16 archetypes were verified.



Click **OK** > **Apply** > **OK**.

Now, the Maven archetypes are available as per our catalog used above. These archetypes will be used to create new projects in Eclipse. You will use these in the next section.

If there were no proxy issues, you may move on to the next section.

If there are any proxy issues (which Eclipse will indicate by not having found any archetypes) and the verification fails in the above step, you should use the **archetype-catalog.xml** from USB contents to add the catalog. Follow these steps:

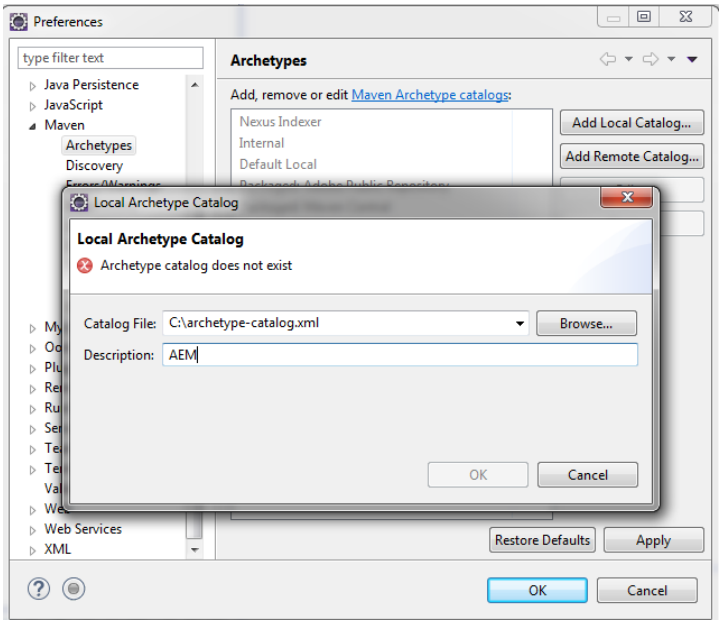
Manually Configuring the Maven Archetype:

Click Window > Preferences > Maven, and then select Archetypes.

Click **Add Local Catalog…** Browse for the **archetype-catalog.xml** file, select it, and click **Open**.

In the Description, enter **AEM**.

Click **OK**.

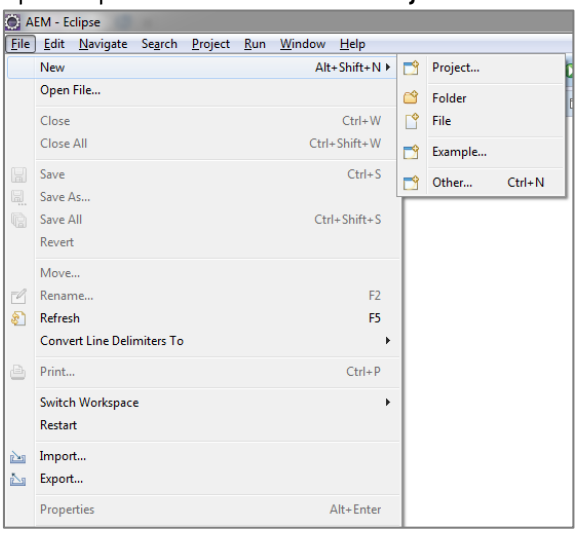


Click **Apply** then click **OK**.

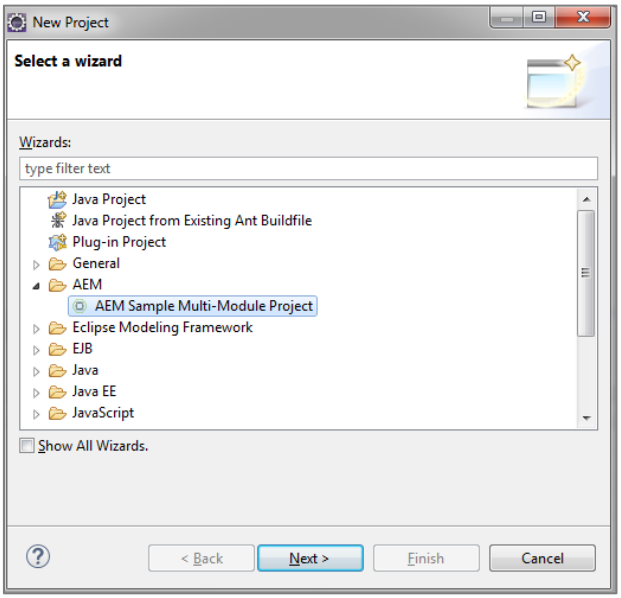
4. Task - Create an Adobe Experience Manager project using the Maven archetypes

We already installed the AEM plugin and configured archetype in Eclipse, now we need to make use of those and create an AEM project. To create a new project, follow the steps below:

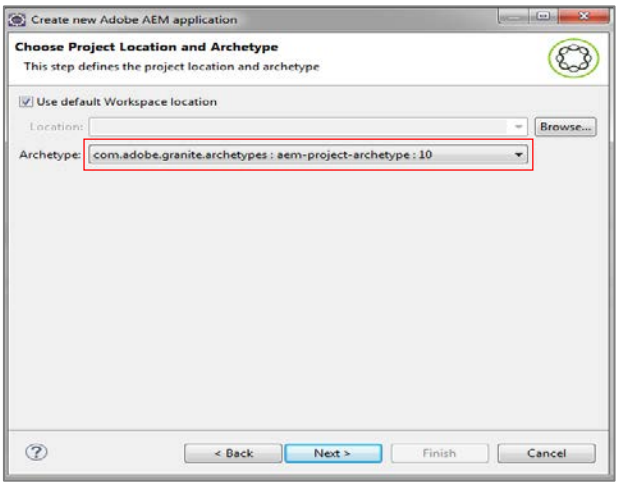
Open Eclipse and click **File** > **New** > **Project**.



Select AEM > AEM Sample Multi-Module Project, and click Next.

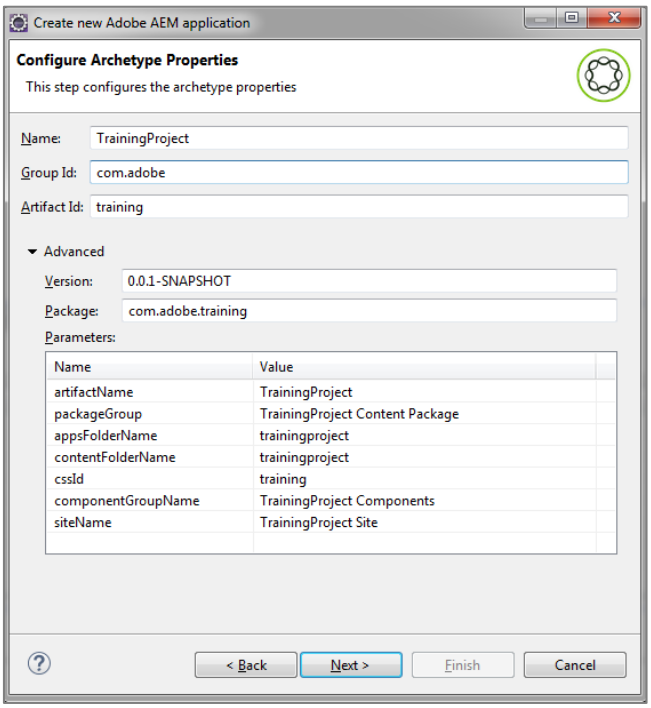


If not already pre-selected, select **com.adobe.granite.archetypes : aem=project-archetype:10** from the list then click **Next**.



Replace the project details with:

- Name: TrainingProject
- Group Id: com.adobe
- Artifact Id: training

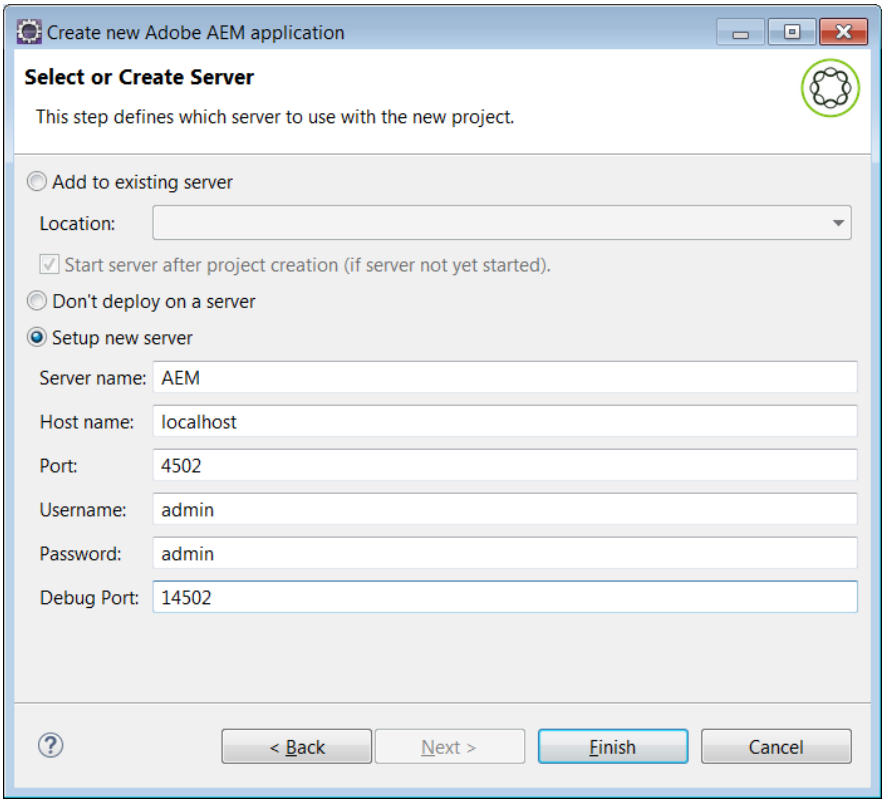


Click **Next**.

In the Create new Adobe AEM application dialog, leave the default as **Setup new server**.

Enter the following server details:

- **Server name**: AEM
- **Host name**: localhost
- **Port**: 4502
- **Username**: admin
- **Password**: admin
- **Debug Port**: 14502

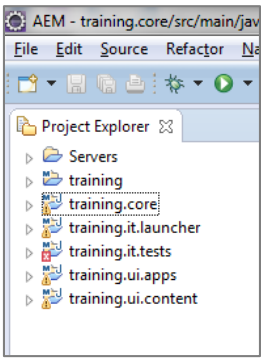


Click **Finish**.

NOTE: If you encounter errors during this step, try again and select "Don't deploy on a server". Some errors you can **ignore** are things like:

- Server not JEE compliant
- Not support 2.4

Be patient as **Eclipse Luna** will create the project as follows:



Note: If eclipse is restarted check whether the above projects are created or not. If projects are not created you may have to delete the workspace and start the process from Task 4 - Create an Adobe Experience Manager project using the Maven archetypes

Right-click the **training** project, go to **Maven**, and select **Download JavaDoc**.

Right-click the **training** project, go to **Maven**, and select **Download Sources**.

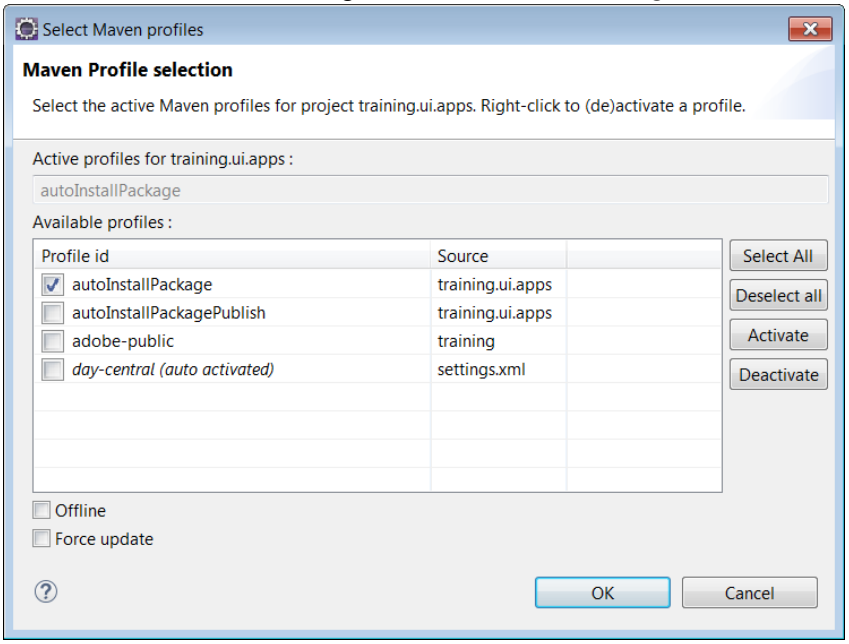
You are now going to build and deploy this project into Adobe Experience Manager. Note, first we need to associate two of the archetype's build profiles, for three of the project's sub-modules. These profiles are:

autoInstallPackage – which we will configure Eclipse to run for the **ui.content** and **ui.apps** directories. Using this profile, we can then either build against these sub-modules, or against the parent **training** directory instead. This has the advantage of building the java bundle, and dropping it into a content package- allowing the bundle to sit in the repository along with the contents of the **ui.apps** and **ui.content** nodes and properties.

autoInstallBundle – which we will configure Eclipse to run for the **training.core** directory. Running this profile (either directly on this directory, or on the parent training directory) will build and deploy the jar file into AEM only, not the content package structures.

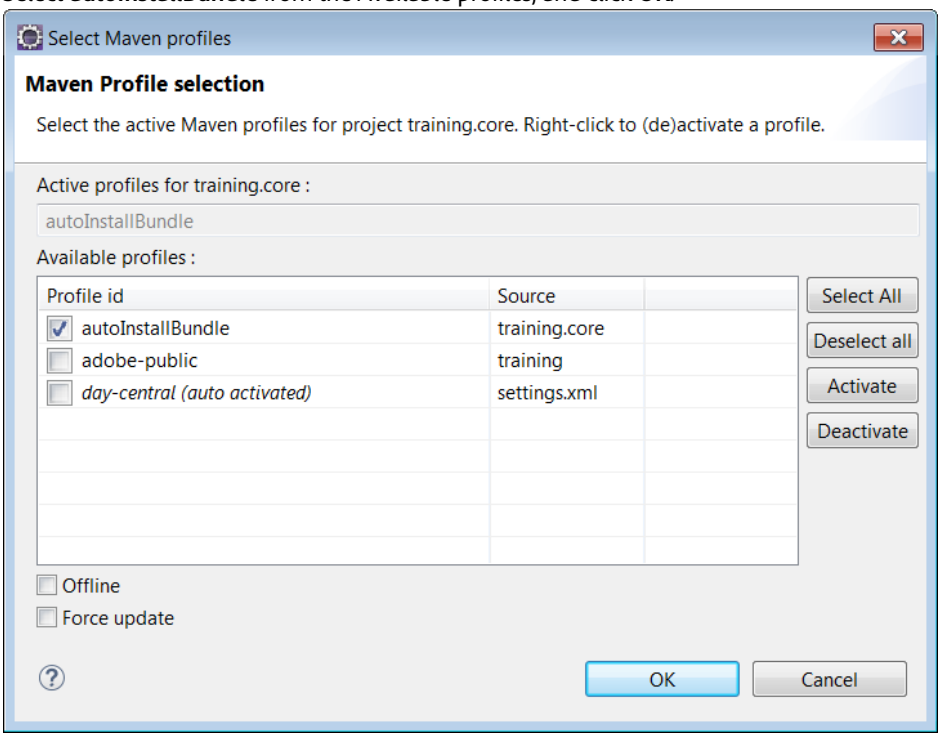
Right-click training.ui.apps, and go to **Maven** > **Select Maven Profiles...**

In the Select Maven Profiles dialog box, **select autoInstallPackage** from the Available profiles, and click **OK**.



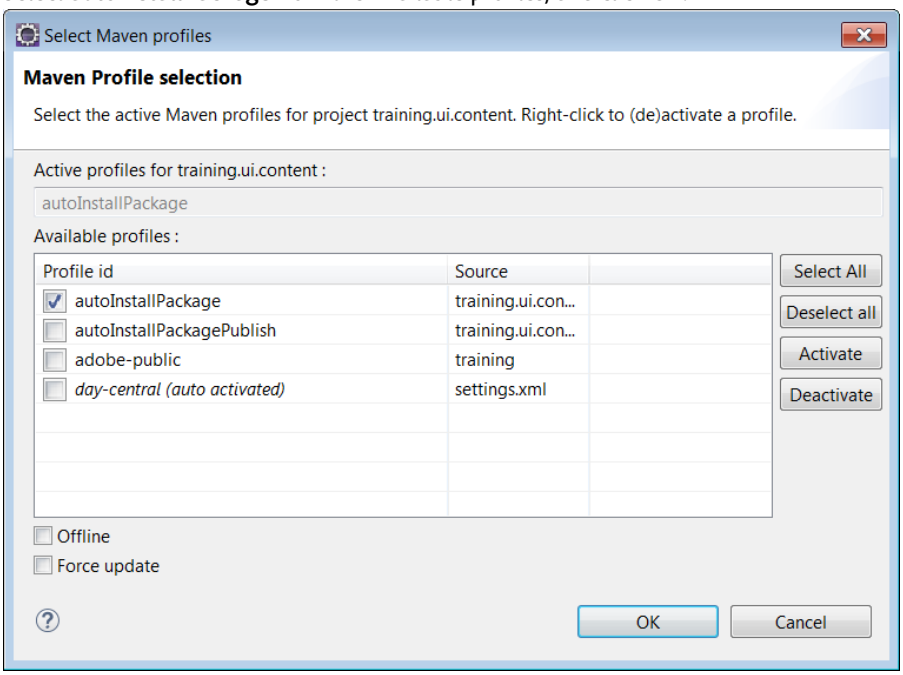
Right-click training.core, and go to Maven > Select Maven Profiles.

Select **autoInstallBundle** from the Available profiles, and click **OK**.



Right-click training.ui.content, and go to Maven > Select Maven Profiles.

Select **autoInstallPackage** from the Available profiles, and click **OK**.



Note : If you are facing issues with dependencies you may have to delete the .m2 directory to resolve dependencies. Locate your .m2 directory in your local file system, and delete it. You may want to perform a back-up of its contents if required. For example, in Windows, the .m2 folder would be located in **C:\Users\<user-name>.**

Fix the **javax.inject** version dependency issue

Note : training.core bundle does not resolve on AEM 6.2 due to different version of javax.inject. With AEM 6.2 the javax.inject package is exported with version 1.0.0 by org.apache.geronimo.specs.geronimo-atinject_1.0_spec.

Add the following dependencies to the core/pom.xml to make javax.inject work in AEM6.2 or replace the POM files given in the USB contents.

```
<Import-Package>javax.inject;version=0.0.0,*</Import-Package>
```

to the <instructions> block of the maven-bundle-plugin, like shown below:

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <!-- Import any version of javax.inject, to allow running on multiple versions of AEM -->
            <Import-Package>javax.inject;version=0.0.0,*</Import-Package>
            <Sling-Model-Packages>
                com.adobe.training.core
            </Sling-Model-Packages>
        </instructions>
    </configuration>
```

AEM Maven dependencies update through UberJar

"UberJar" JAR file contains all of the public Java APIs exposed by Adobe Experience Manager. It includes limited external libraries as well, specifically all public APIs available in AEM which come from the Apache Sling, Apache Jackrabbit, Apache Lucene, Google Guava, and two libraries used for image processing. The UberJar only contains API interfaces and classes, meaning that it only contains interfaces and classes which are exported by an OSGi bundle in AEM. It also contained a MANIFEST.MF file containing the correct package export versions for all of these exported packages, thus ensuring that projects built against the UberJar have the correct package import ranges.

Replace the dependency to the aem-api package by the latest uber-jar or replace the POM files given in the USB contents. In the parent POM, replace this block:

```
<dependency>
        <groupId>com.adobe.aem</groupId>
        <artifactId>aem-api</artifactId>
        <version>6.0.0.1</version>
        <scope>provided</scope>
</dependency>
```

By:

```
<dependency>
        <groupId>com.adobe.aem</groupId>
        <artifactId>uber-jar</artifactId>
        <version>6.2.0</version>
        <classifier>apis</classifier>
        <scope>provided</scope>
</dependency>
```

Save the file, and do the same for the other POMs (without <version> and <scope> blocks).
Note: this is only needed for the core POM, the dependency block can be removed for the other POMs.

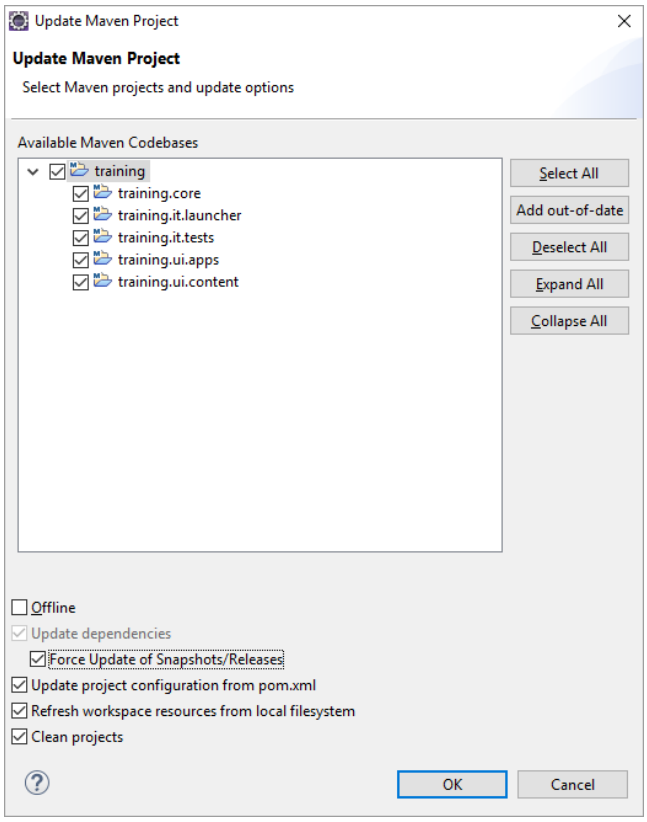
Replacing POM Files

Replace the following POM files in your Eclipse project with the files given in the USB Contents.

- Find and replace the contents of **training.it.tests POM** with **it-tests_pom.xml**
- Find and replace the contents of the **parent POM** with **parent_pom.xml**
- Find and replace the contents of **training.core POM** with **trainingcore_pom.xml**
- Find and replace the contents of **training.ui.apps POM** with **ui-apps_pom.xml**
- Find and replace the contents of **training.ui.content POM** with **ui-content_pom.xml**

Update the project

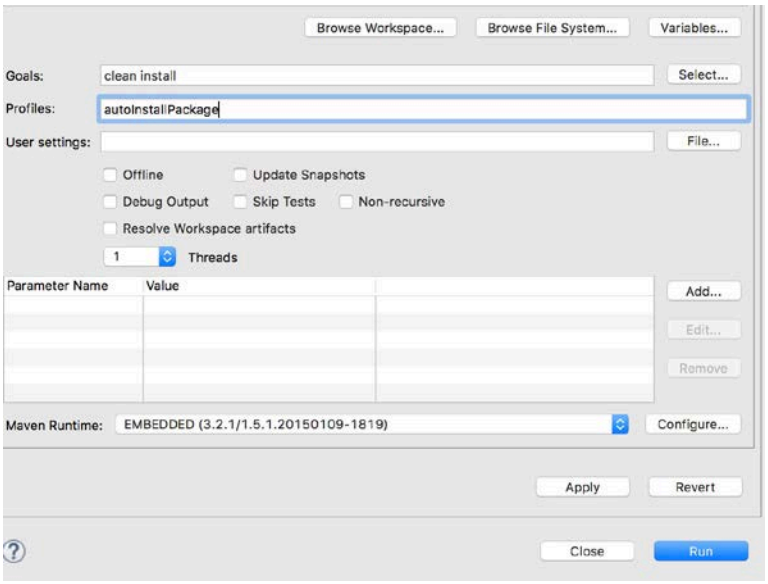
Right-click training, and go to **Maven** > **Update Project**. (or Alt-F5). Don't forget to check the box "Force Update of Snapshots/Releases" before clicking OK.



Once your project is set up, you can also use the Export to Server and Import to Server options to synchronize content between Eclipse and the AEM server.

Right-click **training** (the parent directory), and go to **Run As** > **Maven build...**

In the Edit Configuration dialog box, enter the Goals as **clean install** and in the Profiles field, enter **autoInstallPackage**. We require to run the **autoInstallPackage** profile so we ensure we have the correct content pages build within the repository-. If we instead tried to run **autoInstallBundle** first, the build would fail to upload the bundle into Apache Felix.



Click Apply, and then click Run. The installation may take a few minutes to complete; you can see the installation activity on the Console tab.

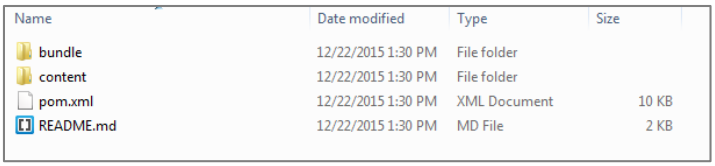
View the package manager in CRXDe Lite. Note that the packages for both the ui.apps and ui.content directories have been installed.

5. Task - Generate project files for Eclipse [optional]

You have configured Eclipse Luna and the plugin to make sure the changes are updated in the Adobe Experience Manager repository. This looks fine if the project was created in Eclipse itself; however, there might be a chance where you have to work on a project that was created using the Maven and command prompt. In this scenario, you must create Eclipse project files for Eclipse using the command prompt. Once that is done, you can import the project in Eclipse and convert it to a Maven project to resolve any dependency. After completing the following steps, you can work on that project in Eclipse without any issues.

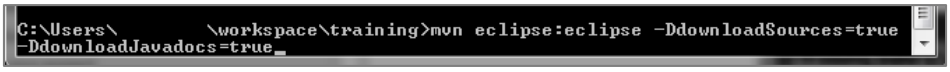
Following these steps will help you to understand the process:

Copy and extract the contents of **training.zip** in a folder under Eclipse workspace as ..\workspace\training



Execute the following command at the project root folder (**workspace\training**).

```
mvn eclipse:eclipse -DdownloadSources=true -DdownloadJavadocs=true
```



This will generate two files with extensions .project, which can be imported into Eclipse:

a. workspace\training\bundle\.project

b. workspace\training\content\.project

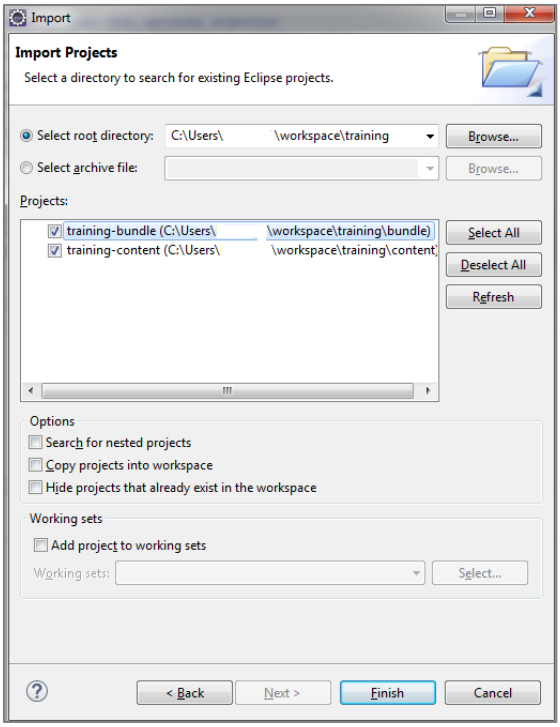
In Eclipse, to import the two project files, we need to do the following:

c. Navigate to **File > Import**.

d. In the **Import** dialog box, under the **General** option, select **Existing Projects into Workspace**.

e. Click **Next**.

f. Click **Browse**, and then navigate to the **training** directory. Eclipse will find the two project files in this directory tree.



Now, we are ready to work on this project using Eclipse.

6. Task - Install and Configure VLT on your system

You can synchronize code (both Java code and JSP code) in Eclipse workspace with the code in the Adobe Experience Manager JCR. For example, assume that you have application logic in Eclipse that represents a JSP component. You can synchronize the code in Eclipse with code in the Adobe Experience Manager JCR using the vault tool. That is, you can check-in code you write in Eclipse into the Adobe Experience Manager JCR.

Likewise, if you make a change in Adobe Experience Manager using CRXDE Lite, you can check-out the code that results in the code in Eclipse workspace (filesystem) being updated. To synchronize code, you need to configure the vault tool by following the steps below:

Locate the VLT tool in the directory: **<AEM installation dir>/crx-quickstart/opt/filevault**

There, you will find two compressed files: **filevault.tgz** and **filevault.zip**. Copy the file that is most convenient to use on your system to a suitable directory (see below), and then unpack it producing the directory **vault-cli-<version>.**

Under **vault-cli-<version>/bin**, you will find the executables **vlt** and **vlt.bat** (the former for UNIX, the latter for Windows).

The directory **vault-cli-<version>** can be left in the current location or moved to another location on your system. In either case, ensure that you include the full path to the **vault-cli-<version>/bin** directory in your system path.

In order to have a persistent path, add the path to the environment variable (as described earlier), through **Control Panel > System > Advanced System Settings > Environment Variables.** Set the path to the bin folder. For example, C:\Users\Administrator\Desktop\Supported Applications\Commons\filevault\vault-cli-3.1.16\bin

If you do not have access to the system settings, you can set the path through command line. However, you must note that this setting would be temporary and you would need to set them each time you start the command prompt.

a. In Windows, you can use the command:

```
SET PATH=%PATH%;<Path to filevault bin folder>
```

b. On a Mac, use:

```
export PATH=<Path to filevault bin folder>:${PATH}
```

You can test whether the tool is correctly installed by typing the following in the command line:

```
vlt --version
```

Keep your command prompt window open.

Additional information about installing and using VLT is available at https://docs.adobe.com/docs/en/crx/2-3/how_to/how_to_use_the_vlttool.html

7. Task - Use VLT to perform content synchronization

You have made many changes in CRX using CRXDE. Now, you want to check-out that change to your local repository (Eclipse workspace) and vice-versa. To accomplish and understand the process, follow these steps:

Verify that VLT is on your $PATH. Using the **cd** command, change the directory to **training\ui.content\src\main\content\jcr_root**

Execute the following command:

```
vlt --credentials admin:admin co --force http://localhost:4502/crx/
```

This performs a VLT check-out of the JCR repository to your local file system.

Warning! This takes several minutes.

You will now have file serializations of the JCR content; for example, in **training\ui.content\src\main\content\jcr_root\.content.xml**.

The JCR paths that are checked out are configured in **training\ui.content\src\main\content\META-INF\vault\filter.xml**. Inspect this file. As an alternative to using this folder, you can specify the filter file in the VLT command line with **--filter filter.xml**, for example:

```
vlt co --filter filter.xml http://localhost:4502/crx/
```

NOTE: The credentials have to be specified only once upon your initial check-out. They will then be stored in your home directory inside ~/.vault/auth.xml.

Navigate to training\ui.apps\src\main\content\jcr_root, and execute the command:

```
vlt --credentials admin:admin co --force http://localhost:4502/crx/
```

NOTE: Ensure you have navigated to this specific directory (above) in order to execute this important command.

In CRXDE Lite, navigate to **/apps/trainingproject/components/content**, and create a node named **my_folder**, of type **sling:Folder**.

Inside the folder, add a node named **my_node** of type **nt:unstructured**.

With **my_node** selected, in the main area of CRXDE Lite, go to the **Properties** tab (which should be open too, by default).

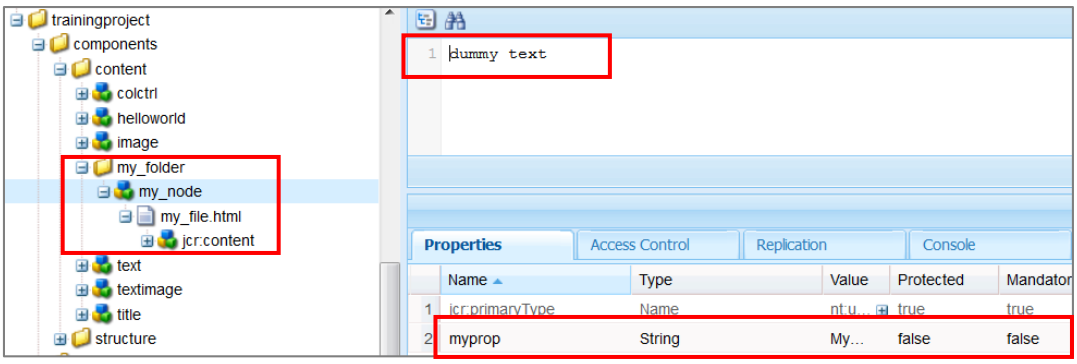
At the bottom of the **Properties** tab, you will see three empty fields at the bottom of the tab. Enter the following properties to the corresponding fields then click **Add.**

Property	Type	Value
myprop	String	Myvalue

Verify a new row (#2) is now on the Properties of the node for my_node. It includes all the information you added in the previous step.

To the same **my_folder** folder, right-click **my_node** and select **Create** > **Create File** and enter the following filename: **my_file.html** then click **OK**. A new tab *my_file.html opens. The "*" indicating you have not made or saved any changes to the file yet.

Double-click the file and in the left panel include the following text: `dummy text`

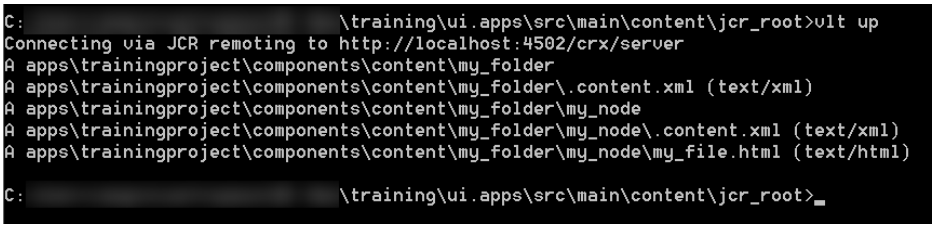


Click **Save All** in the upper-left corner of CRXDE Lite.

On the command line, using the cd command, change the directory to **training\ui.apps\src\main\content\jcr_root**, and execute the following command:

```
vlt up
```

You should see (with paths abbreviated):



In your training directory, change directory to training\ui.apps\src\main\content\jcr_root\apps\trainingproject\components\content\my_folder\my_node.

Open .**content.xml** in a text editor, and add an XML property:

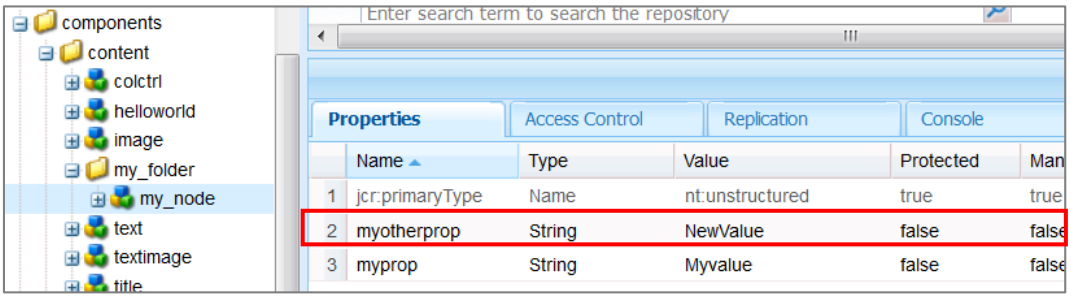
```
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:jcr="http://www.jcp.org/jcr/1.0"
    xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
    jcr:primaryType="nt:unstructured"
    myprop="Myvalue"
    myotherprop="NewValue">
    <my_file.html/>
</jcr:root>
```

Commit to CRX by executing the following command from **training\ui.apps\src\main\content\jcr_root**:

```
vlt ci \apps\trainingproject\components\content\my_folder\my_node\.content.xml
```

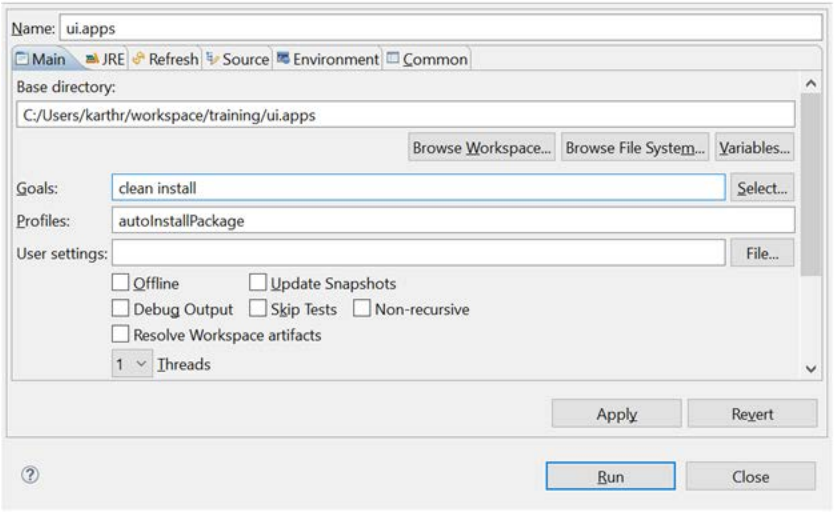
You should see:

Verify the change in CRXDE Lite.



Change the content of **apps/training/components/my_file.html** through a text editor or in eclipse, and commit to CRX not using VLT, but by building the package. To build the package open eclipse and Right-click **training.ui.apps** and click **refresh**.

Right-click **training.ui.apps** and go to **Run As > Maven** build
In the Edit Configuration dialog box, enter the Goals as **clean install** and in the Profiles field, enter **autoInstallPackage** and click Apply and Run.



Verify the change in CRXDE Lite.

Scenario Conclusion

To meet the challenge of developing, and effectively collaborating with other members of the team, you performed the following:

- Installed Eclipse.
- Installed AEM plugin for Eclipse.
- Configured Maven archetype and created a new project.
- Created an Adobe Experience Manager project using Eclipse.
- Set up VLT and tested the checkout process.

CHAPTER FOUR: USING OSGI SERVICES

Overview

This chapter deep dives into the Open Service Gateway initiative (OSGi) architecture. You will see the benefits of using OSGi, the components, the annotations available in OSGi, as well as the configurable services.

Objectives

By the end of this chapter, you will be able to:

- create an OSGi service and use the service.

What is OSGi?

The OSGi Service Platform is a Java-based application server for networked devices. This non-proprietary service platform spans:

- Mobile phones
- Vehicles
- Telematics
- Embedded appliances
- Residential gateways
- Industrial computers
- Desktop PCs
- High-end servers (including mainframes)

Open Services Gateway Initiative (OSGi) is one of the fundamental layers in the Adobe Experience Manager technology stack. It is used to control the composite bundles of Adobe Experience Manager and their configuration. OSGi allows applications to be created from smaller, reusable, and collaborative components, which can then be deployed.

Bundles can be managed as they can be installed, started, or stopped individually, and interdependencies between these modules are automatically handled. Each of the OSGi components is contained in one of the bundles.

OSGi is a set of specifications that create a development model where applications are composed of different reusable components. This also means that the components can hide their implementations from other components, and communicate through services. These services are objects that are shared between components.

To summarize, an OSGi application is a collection of bundles that interact using service interfaces:

- Bundles may be independently developed and deployed.
- Bundles and their associated services may appear or disappear at any time.

There are a number of advantages to using OSGi Service Platform. Some of them are:

- Adding an OSGi Service Platform to a networked device enables you to manage the lifecycle of the software components in the device from anywhere in the network.
- OSGi simplifies the process of development and maintenance of a large number of configurations.
- Multiple java-based components can work efficiently within a single Java Virtual Machine (JVM). This technology provides an extensive security model so that components can run in a shielded environment.

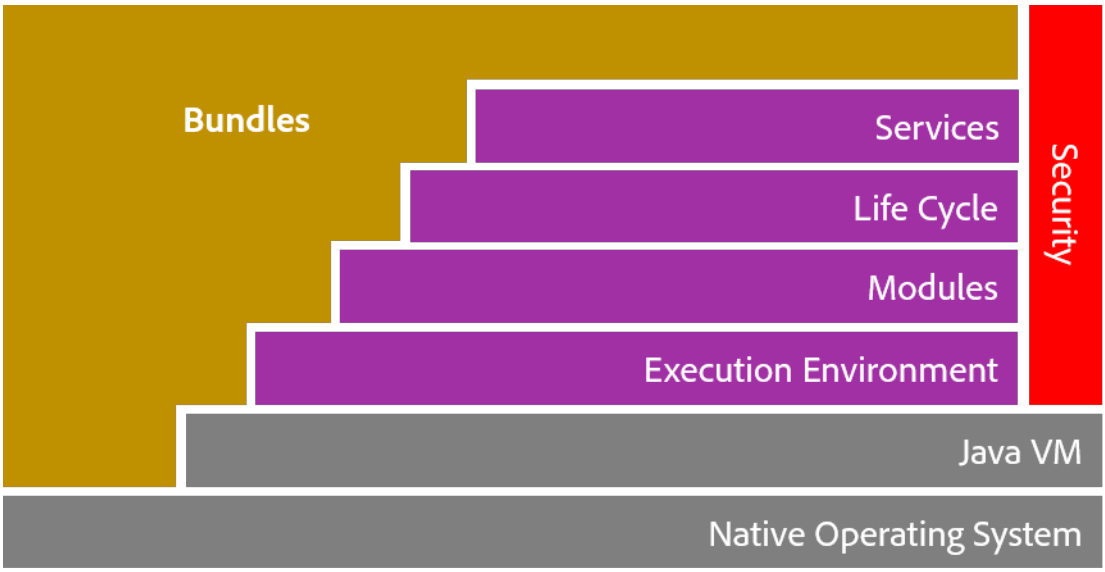
Other advantages:

- Remote component management
- Secure execution environment
- Cooperation between applications

- Commercial Off-the-shelf components
- Simplified Deployment
- Multi-vendor interoperability
- Dynamic nature

OSGi Architecture

The OSGi has a layered model that is depicted in the following figure.



- Bundles: Bundles are the OSGi components made by the developers.
- Services: The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects.
- Life-Cycle: The API to install, start, stop, update, and uninstall bundles.
- Modules: The layer that defines how a bundle can import and export code.
- Security: The layer that handles the security aspects.
- Execution Environment: Defines what methods and classes are available in a specific platform.

The following sections explain a few of the above concepts.

Bundles

Bundles are built on Java's existing standard way of packaging together classes and resources—the JAR file (.jar). An OSGi bundle is just a JAR file, with additional metadata added to the manifest file. The OSGi metadata is provided as header information in the META- INF/MANIFEST.MF file. The additional information consists of:

- Bundle name(s):
    - A "symbolic" name used by OSGi to determine the bundle's unique identity

- An optional, human-readable, descriptive name

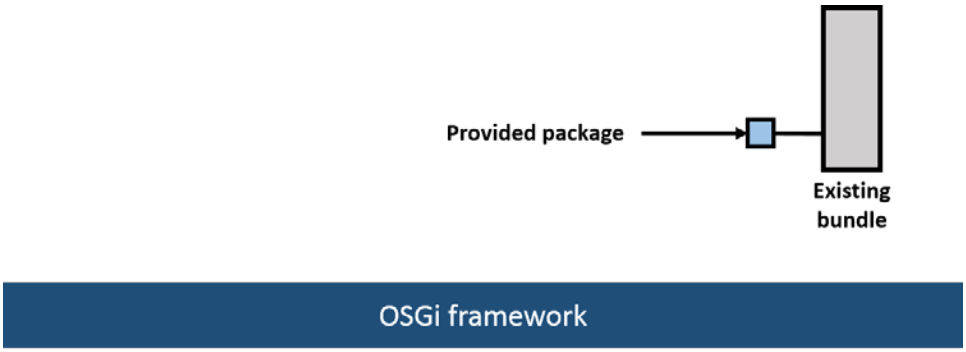
- Bundle version
- The list of services imported and exported by this bundle
- Optional additional information, such as:
  - Minimum Java version that the bundle requires
  - Vendor of the bundle
  - Copyright statement
  - Contact address, and so on

The bundles are loosely coupled.

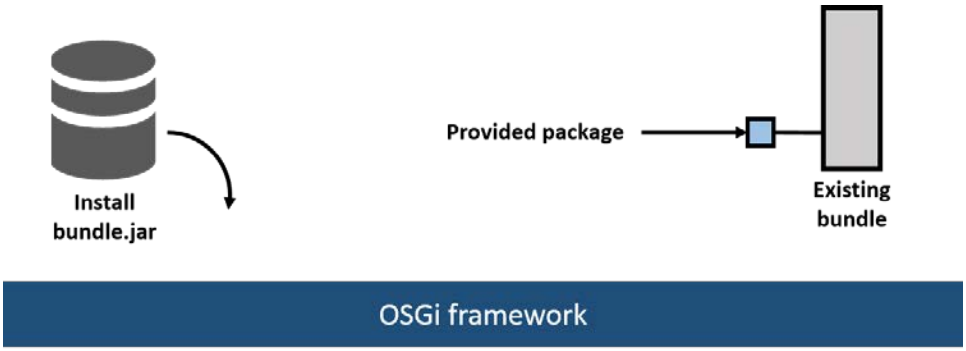
- It includes package imports and exports with versions.
- Dependencies are independent from the bundle organization.
- "'Someone" provides the self-describing package.
- OSGi provides error management for unresolved bundles.
- Modular thinking is required during application design.
- It requires proper metadata and consistent version management.

1..1 Dependency Management Resolution

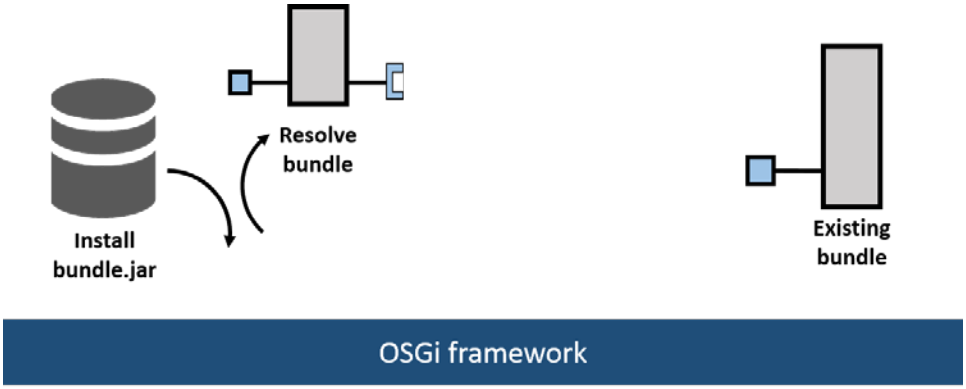
A bundle is present in the container.



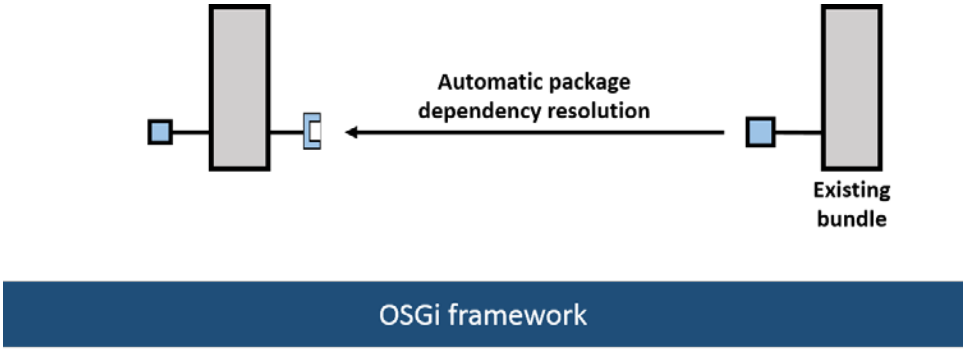
When a new bundle is provided, it is installed into the OSGi container.



The OSGi container resolves the new bundle.



The container provides automatic dependency resolution.



Modules

Modularity is at the core of the OSGi specifications and is embodied in the bundle concept. You should be familiar with the term 'bundle' in Java, which means a JAR file. The difference between a Java bundle and an OSGi bundle is that in Java, the contents of the JAR are completely visible to all other JARs. In OSGi, it "hides" the JAR contents unless they are explicitly exported. If a bundle wants to use another JAR, it must explicitly import the parts that it needs. Therefore, there is no sharing.

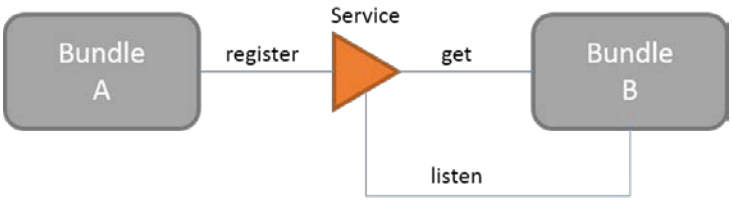
Though the code hiding and explicit sharing provides many benefits (for example, allowing multiple versions of the same library being used in a single VM), the code sharing is only there to support OSGi services model. The services model is about bundles that collaborate.

Services

A bundle can create an object and register it with the OSGi service registry under one or more interfaces. Other bundles can go to the registry and list all objects that are registered under a specific interfaces or class.

A bundle can register a service, get a service, and listen for a service to appear or disappear. Any number of bundles can register the same service type, and any number of bundles can get the same service.

This is shown in the following figure.



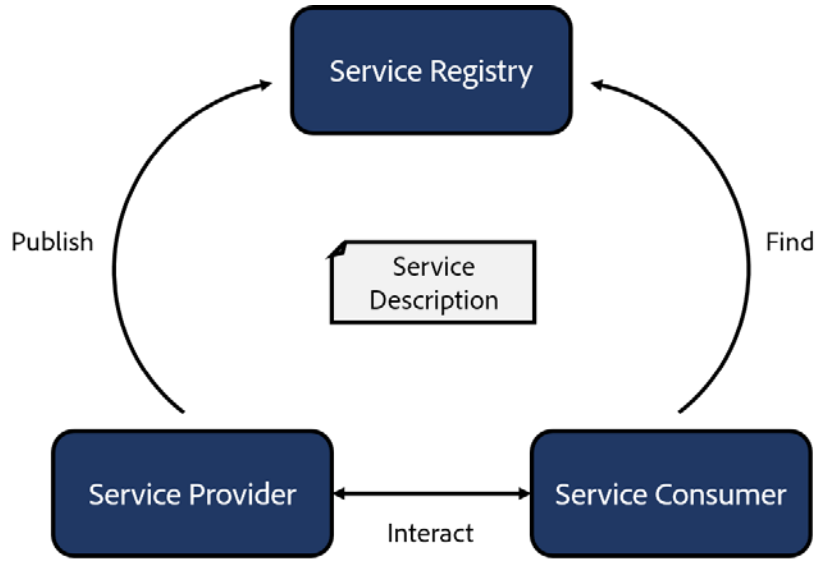
Each service registration has a set of standard and custom properties. An expressive filter language is available to select only the services in which you are interested. You can use properties to find the proper service or they can play other roles at the application level.

Services are dynamic. This means that a bundle can decide to withdraw its service from the registry while other bundles are still using this service. Bundles using such a service must then ensure they no longer use the service object and drop any references. OSGi applications do not require a specific start ordering in their bundles.

1..2 Service Registry Model

OSGi provides a service-oriented component model using a publish/find/bind mechanism. Using the OSGi Declarative Services, the consuming bundle says 'I need X' and 'X' is injected.

Because you cannot depend on any particular listener or a consumer being present in the container at any particular time, OSGi provides dynamic service look up using the Whiteboard registry pattern. Briefly defined, the Whiteboard registry pattern works as follows:

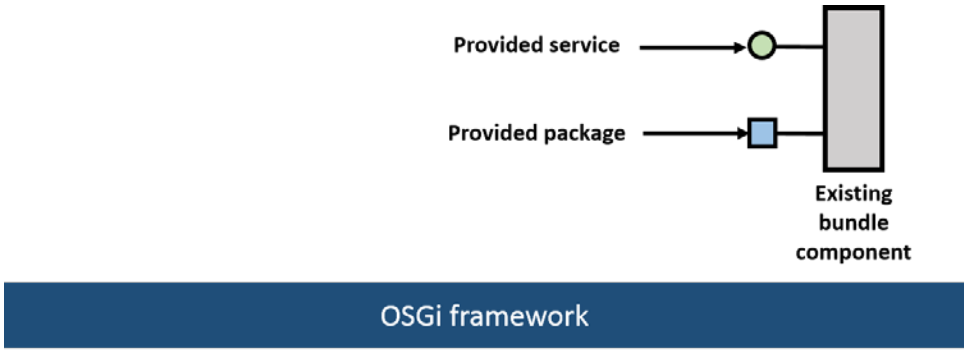


Instead of registering a listener/consumer object with the service provider, the consumer creates an object that implements the OSGi listener interface, providing a method that should be called when the event of interest occurs. When the desired event occurs, the service provider requests a list of all the services of the same object type and then calls the action method for each of those services. The burden of maintaining the relationships between service providers and service consumers is shifted to the OSGi framework. The advantage to doing this is that the OSGi framework is aware of bundle status and life cycle and will unregister a bundle's services when the bundle stops.

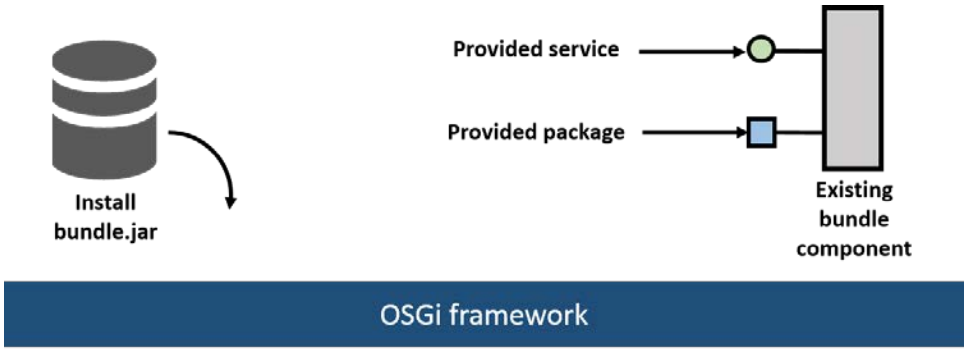
1..3 Dynamic Service Lookup

The following steps show how the lookup is managed for dynamic services.

Existing service

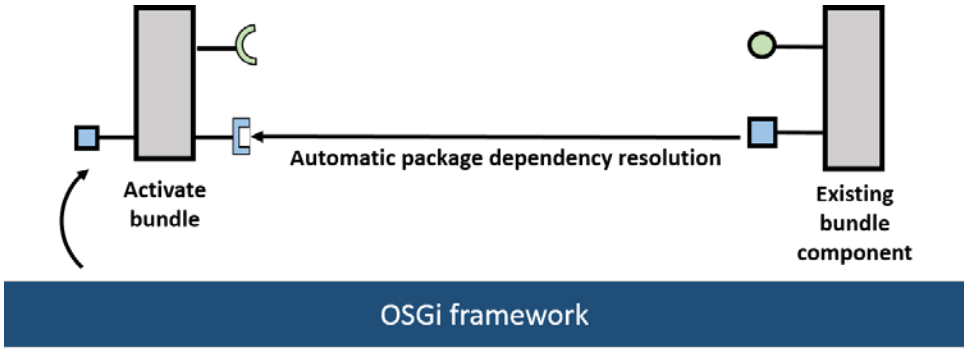


Install new bundle.

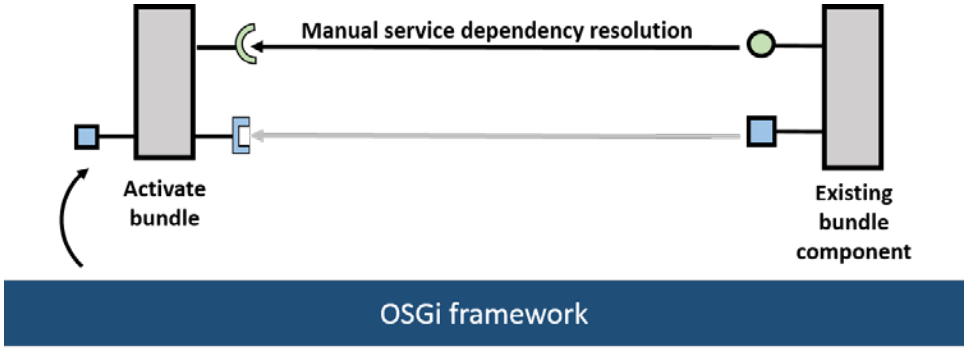


Activate new bundle.

Automatic resolution of package dependency.



Manual service dependency resolution.



1..4 OSGi Service Advantages

In OSGi-based systems, functionality is mainly provided through services. Services implement one or more interfaces, which define the type of service provided. It is the life cycle of the bundle that defines the life cycle of the service. A service object may be instantiated when the bundle is started and is automatically removed when the bundle is stopped.

The advantages of OSGi services are as follows:

- Lightweight services
- Lookup is based on the interface name
- Direct method invocation
- Good design practice
- Separates the interface from implementation
- Enables reuse, substitutability, loose coupling, and late binding

1..5 Declarative Services

Declarative Services are a part of the OSGi container, and simplifies the creation of components that publish and/or reference OSGi Services.

Features:

- No need to write explicit code to publish or consume services.
- Components that publish services are delayed, meaning that the service implementation class is not loaded or instantiated until the service is actually requested by a client.
- Components have their own lifecycle, bounded by the lifecycle of the bundle in which they are defined.
- Components can automatically receive configuration data from Configuration Admin.

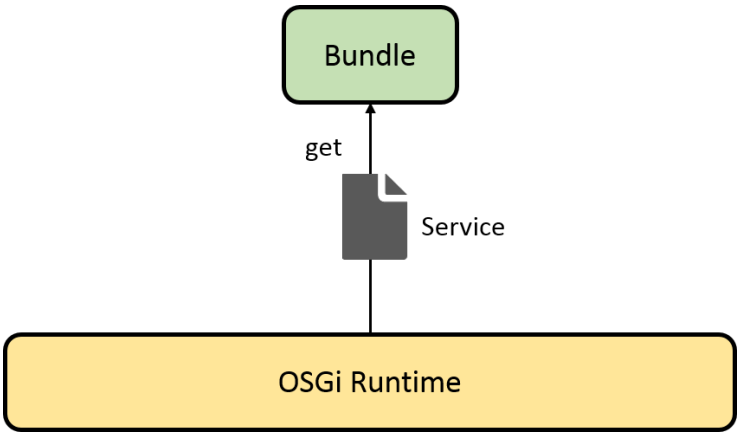
Components are declared using XML configuration files contained in the respective bundle and listed in the Service-Component bundle manifest header. Declarative Services resolve the bundle through the XML configuration files. You may handwrite and register these configuration files.

Declarative Services are a good alternative to:

- Writing an Activator
- Registering the bundle in the framework
- Using the service tracker

The OSGi Service Component (Declarative Services) reads the descriptions from started bundles. The descriptions are in the form of XML files, which define the set of components for a bundle. It is through the XML configuration definition information that the container:

- registers the bundle's services
- keeps track of dependencies among the bundles
- starts and stops services
- invokes the optional activation and deactivation method
- provides access to bundle configuration



Deployment

Bundles are deployed on an OSGi framework—the bundle runtime environment. It is a collaborative environment, where the bundles run in the same VM and can actually share code. The framework uses the explicit imports and exports to wire up the bundles so they do not have to concern themselves with class loading. A simple API allows bundles to install, start, stop, and update other bundles, as well as enumerate the bundles and their service usage.

Benefits of OSGi

- **Reduced Complexity:** Bundles are the components of OSGi, and these bundles are modules. Due to the lack of interdependencies between these modules, developers have more freedom to change the modules at a later stage, which helps to reduce the number of bugs and simplify development.
- **Reuse:** The OSGi component model makes it very easy to use many third-party components in an application. An increasing number of open source projects provide their JARs readymade for OSGi. Commercial libraries are also available as readymade bundles.
- **Real World:** The OSGi framework is dynamic. As the real world is highly dynamic, having dynamic services that come and go make the services a perfect match for many real world scenarios. Applications can therefore reuse the powerful primitives of the service registry in their own domain. This not only saves writing code, it also provides global visibility, debugging tools, and more functionality implemented for a dedicated solution.
- **Easy Deployment:** The OSGi technology is not just a standard for components. It also specifies how components are installed and managed. The standardized management API makes it very easy to integrate OSGi technology in existing and future systems.
- **Dynamic Updates:** The OSGi component model is a dynamic model. You can install, start, stop update, and uninstall bundles without bringing down the whole system.
- **Adaptive: T**he dynamic service model of OSGi allows bundles to find out what capabilities are available on the system and adapt the functionality they can provide. This makes code more flexible and resilient to changes.
- **Transparency:** The management API provides access to the internal state of a bundle as well as how it is connected to other bundles. For example, most frameworks provide a command shell that shows this internal state. You can stop parts of the applications to debug a certain problem or bring in diagnostic bundles. OSGi applications can often be debugged with a live command shell.
- **Versioning:** In the OSGi environment, all bundles are carefully versioned and only bundles that can collaborate are wired together in the same class space. This allows various bundles to function with their own library.
- **Simple:** A number of easy-to-use annotations tell the runtime how a particular class wants to use the dynamics, configuration, and dependencies on other services. The defaults completely hide the dynamics and OSGi. This simple model allows the gradual use of more advanced features.
- **Small:** The OSGi Release 4 Framework can be implemented in about a 300KB JAR file. This is a small overhead for the amount of functionality that is added to an application by including OSGi. Therefore, OSGi runs on a large range of devices—from very small to small, to mainframes. It only asks for a minimal Java VM to run and adds very little on top of it.
- **Fast:** One of the primary responsibilities of the OSGi framework is loading the classes from bundles. OSGi pre-wires bundles and knows for each bundle exactly which bundle provides the class. This lack of searching is a significant speed up factor at startup.
- **Secure: T**he OSGi security model leverages the fine grained security model but improves the usability by having the bundle developer specify the requested security details in an easily audited form while the operator of the environment remains fully in charge.

- **Non-Intrusive:** Applications (bundles) in an OSGi environment are left on their own. They can use virtually any facility of the VM without the OSGi restricting them. Best practice in OSGi is to write Plain Old Java Objects and for this reason, there is no special interface required for OSGi services—even a Java String object can act as an OSGi service. This strategy makes application code easier to port to another environment.
- **Runs Everywhere:** Two issues are taken care of by the OSGi specification. First, the OSGi APIs does not use classes that are not available on all environments. Second, a bundle does not start if it contains code that is not available in the execution environment.

Components and Annotations in OSGi

The following sections describe the purpose and use of components in OSGi, as well as the annotations used for service components and Java compilers.

Components

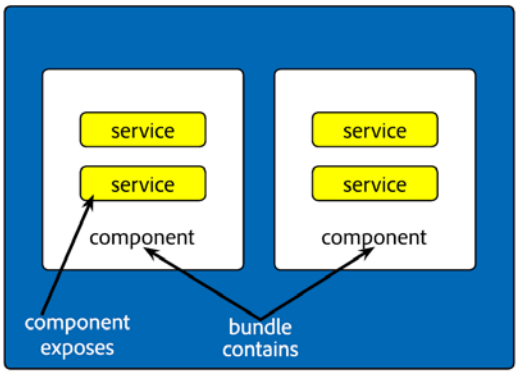
Components are the main building blocks for OSGi applications. Components in OSGi are, by definition, provided by a bundle. A bundle will contain and provide one or more components.

Therefore, a component is a:

- piece of software managed by an OSGi container.
- Java object created and managed by a container.

The OSGi container manages component configuration and the list of consumed services.

- A component can provide a service.
- A component can implement one or more Java interfaces as services.



A component is similar to a runtime service. They can publish themselves as a service and/or can have dependencies on other components and services. These dependencies will influence their life cycle because the component will only be activated by the OSGi container when all required dependencies are met/available. Each component has an implementation class, and can optionally implement a public interface, effectively providing this "service."

A service can be consumed/used by components and other services. Basically, a bundle needs the following to become a component:

XML file, where you describe the service the bundle provides and the dependencies of other services of the OSGi framework

- Manifest file header entry to declare that the bundle behaves as a component
- Activate and Deactivate methods in the implementation class (or bind and unbind methods)
- SCR. A service of the OSGi framework to manage the components: Declarative service of Equinox or Apache Felix SCR

NOTE: By default, a device is only started if someone else uses it. The immediate flag forces a service start. The OSGi Runtime calls the bundle - not the other way around. Do not confuse OSGi components with Adobe Experience Manager components!

As a best practice, always upload the bundle using the JCR. That way, the release engineers and system administrators have one common mechanism for managing bundles and configurations.

Bundles

You can upload bundles into the Felix container through the Web Console or through the placement of the bundle into a folder named 'install' in the JCR. Use of the JCR allows easy maintenance of the bundle throughout its life cycle. For example, deletion of the bundle from the JCR causes a deletion of the Felix bundle by Sling. Subsequent replacement of the JAR file with a new version in the JCR will create the new version of the bundle in the Felix container.

Annotations

You can annotate OSGi components using the annotations provided by the org..apache.felix.dependencymanager.annotation bundle.

The following annotations are supported:

- Component
- Activate
- Deactivate
- Modified
- Service
- Reference
- Property

1..6 @Component

To register your components without annotations, you would have to implement Activators, which extend the `DependencyActivatorBase` class. The `@Component` annotation allows the OSGi Declarative Services to register your component for you.

The `@Component` annotation is the only required annotation.

If this annotation is not declared for a Java class, the class is not declared as a component. This annotation is used to declare the `<component>` element of the component declaration. The required `<implementation>` element is automatically generated with the fully qualified name of the class containing the component annotation.

```
package com.adobe.osgitraining.impl;
import org.apache.felix.scr.annotations.Component;
@Component
public class MyComponent {
}
```

1..6.1 @Component Modifiers

The following are some of the attributes that you can use with the `@Component` annotation.

- **metatype:** Indicates whether the Metatype Service data is generated or not. If this parameter is set to true, then the Metatype Service is generated in the metatype.xml file for this component. The properties defined in @Property are configurable in the Web Console or sling:OsgiConfig nodes.
- **immediate:** Indicates whether the component is immediately activated.
- **enabled:** Indicates whether the component is enabled when the bundle starts.
- **label:** Serves as a title for the object described by the meta type. You can localize this name by prepending a % sign to the name.
- **description:** Provides a description for the object described by the meta type. You can localize this name by prepending a % sign to the name.

Example:

```
@Component (metatype=true, immediate=true, label="Hello Bundle",
description="This is a bundle")
```

1..7 @Activate, @Deactivate, and @Modified

The OSGi Declarative Service allows you to specify the name for the `activate`, `deactivate`, and `modified` methods. In the following code, note the `@Activate` and `@Deactivate` annotations. These actions specify what happens when the component is activated and deactivated.

```
package com.adobe.osgitraining.impl;
import org.apache.felix.scr.annotaions.Activate;
import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Deactivate;
@Component
public class MyComponent {
@Activate
protected void activate() {
// do something
}
@Deactivate
protected void deactivate() { // do something
}
}
```

1..8 @Service

The `@Service` annotation defines whether and which service interfaces are provided by the component. This is a class annotation.

```
package com.adobe.osgitraining.impl;
import org.apache.felix.scr.annotaions.Component;
import org.apache.felix.scr.annotaions.Service;
import org.osgi.service.event.EventHandler
@Component
@Service(value=EventHandler.class)
public class MyComponent implements EventHandler {
```

1..9 @Reference

The `@Reference` annotation defines references to other services. These other services (consumed services) are made available to the component by the SCR. This annotation declares the `<reference>` elements of the component declaration. The `@Reference` annotation may be declared on a Class level or on any Java field to which it might apply. Depending on where the annotation is declared, the parameters may have different default values.

1..10 @Property

The `@Property` annotation defines properties that are made available to the component through the `ComponentContext.getProperties()` method. These tags are not strictly required but may be used by components to define initial configuration. This tag declares `<property>` elements of the component declaration. This tag may also be defined in the Java class comment of the component or in a comment to a field defining a constant with the name of the property.

```
@Component
@Service(value-EventHandler.class)
@Properties({
@Property(name=”event.topics”, value=”*”,
propertyPrivate=true),
@Property(name=”event.filter”, value=”(event.
distribute=*)”,
propertyPrivate-true)
})
public class DistributeEventHandler
implements EventHandler {
protected static final int DEFAULT CLEANUP PERIOD = 15
@Property(intValue=DEFAULT CLEANUP PERIOD)
private static final String PROPERTY_CLEANUP_
PERIOD=”cleanup.period”;
@Reference
protected ThreadPool threadPool;
@Activate
protected void activate (final Map<String, Object>props){
this.cleanupPeriod = toInt(props.get(PROP_CLEANUP_PERIOD));
}
```

Additionally, properties may be set using this tag, to identify the component if it is registered as a service; for example, the `service.description` and `service.vendor` properties.

1..11 Java Compiler Annotations

Note that the following annotations are defined by the Java compiler.

1..11.1 @Override

The `@Override` annotation informs the compiler the element is meant to override an element declared in a superclass.

1..11.2 @SuppressWarnings

The `@SuppressWarnings` annotation informs the compiler to suppress specific warnings that it would otherwise generate.

Configurable Services

The OSGi Configuration Admin Service provides for configuration storage and for the delivery of the configuration automatically or on demand to clients. Configuration objects are identified by Persistent Identifiers (PID) and are bound to bundles when used. For Declarative Services, the name of the component is used as the PID to retrieve the configuration from the Configuration Admin Service.

The Configuration Admin Service not only allows components to get or retrieve configuration, it also provides the entry point for Management Agents to retrieve and update configuration data. The combination of the configuration properties and Meta type description for a given PID is used to build the UI to configure the service and/or component.

Perform Task – Create an OSGi service and consume it in JSP

Chapter 4 Lab Activity - I

Scenario

You need to display a customized message on your webpage.

Overview

In the process of creating the service you need to:

- Create an OSGi service (interface and implementation).
- Log the (de)activation of the service.
- Retrieve a reference to the repository and log the repository name.
- Consume the service from a JSP.

Challenge

Ensure the service that you are creating is active, and provides the required properties.

Pre-requisites

You need to have the project—trainingproject available, which can be used to create the service. You also need a site, which can be used to consume the newly created service and to display the output on the page.

Steps

1. Task - Create and consume an OSGI service

   To Activate code inside our bundle, we need to implement the **org.osgi.framework.BundleActivator** interface. Open Eclipse Luna.

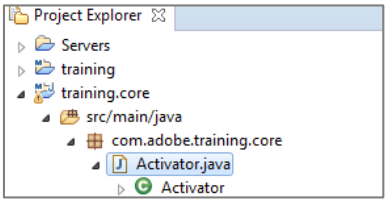
   Right-click the **trainingproject** and create a new class named **Activator** under **training.core** > **src/main/java** > **com.adobe.training.core**.
   NOTE: When you click **Finish**, the .java extension is added to the file name.

   Copy and paste the following code from the exercise files provided with this course.

NOTE: Do not copy and paste the code from the student guide PDF. Code files are provided along with the course materials. Open the code file in an editor such as Notepad++, copy the entire code content from there and then paste into Eclipse.

```
package com.adobe.training.core;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Activator implements BundleActivator {
    private static final Logger LOGGER = LoggerFactory.getLogger(Activator.class);
    /*
     * (non-Javadoc)
     * @see
    org.osgi.framework.BundleActivator#start(org.osgi.framework.BundleContext)
     */
    public void start(BundleContext context) throws Exception {
        LOGGER.info("bundle started");
    }
    /*
     * (non-Javadoc)
     * @see
    org.osgi.framework.BundleActivator#stop(org.osgi.framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception {
        LOGGER.info("bundle stopped");
    }
}
```

Activator.java is using the **start()** and **stop()** methods to create log entries based on the bundle status.



Let us add the bundle to POM file, open training.core in eclipse and edit pom.xml. Add the following code <Bundle-Activator>com.adobe.training.core.Activator</Bundle-Activator>

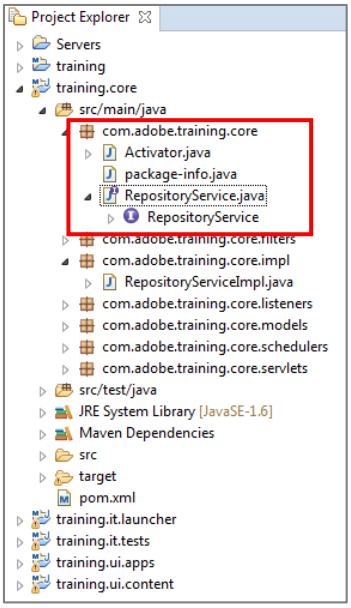
```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <!--
            <Embed-Dependency>
                artifactId1,
                artifactId2;inline=true
            </Embed-Dependency>
            -->
            <Sling-Model-Packages>com.adobe.training.core</Sling-Model-Packages>
            <Import-Package>javax.inject;version=0.0.0,*</Import-Package>
            <Bundle-Activator>com.adobe.training.core.Activator</Bundle-Activator>
            <Private-Package>com.adobe.training.core.impl</Private-Package>
        </instructions>
    </configuration>
```

Create a java interface called **RepositoryService.java** under same location as the **Activator.java** class with following code:

```
package com.adobe.training.core;

public interface RepositoryService {
    public String getRepositoryName();
}
```

To implement the RepositoryService, under **training.core** > **src/main/java**, create a new package called **com.adobe.training.core.impl** and create a java class underneath this new package named **RepositoryServiceImpl.java** and write the following code:

```
package com.adobe.training.core.impl;

import com.adobe.training.core.RepositoryService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Activate;
import org.apache.felix.scr.annotations.Deactivate;
import org.apache.felix.scr.annotations.Service;
import org.apache.felix.scr.annotations.Reference;
import javax.jcr.Repository;

@Component

@Service(value = RepositoryService.class)
public class RepositoryServiceImpl implements RepositoryService {

    private static final Logger LOGGER =
    LoggerFactory.getLogger(RepositoryServiceImpl.class);

    @Reference
    private Repository repository;

    public String getRepositoryName() {
        return repository.getDescriptor(Repository.REP_NAME_DESC);
    }

    @Activate
    protected void activate() {
        LOGGER.info("service activated" );
    }

    @Deactivate
    protected void deactivate() {
        LOGGER.info ("service deactivated");
    }
}
```



To deploy the project, right-click the maven folder called **training.core** and click **Run As** > **Maven install**.

Go to web console at http://localhost:4502/system/console/bundles, and check that the **TrainingProject-core** bundle is installed.

Note that the RepositoryService has been added in the bundle.

Service ID 4358	Types: com.adobe.training.core.RepositoryService Service PID: com.adobe.training.core.impl.RepositoryServiceImpl Component Name: com.adobe.training.core.impl.RepositoryServiceImpl Component ID: 2528 Vendor: Adobe

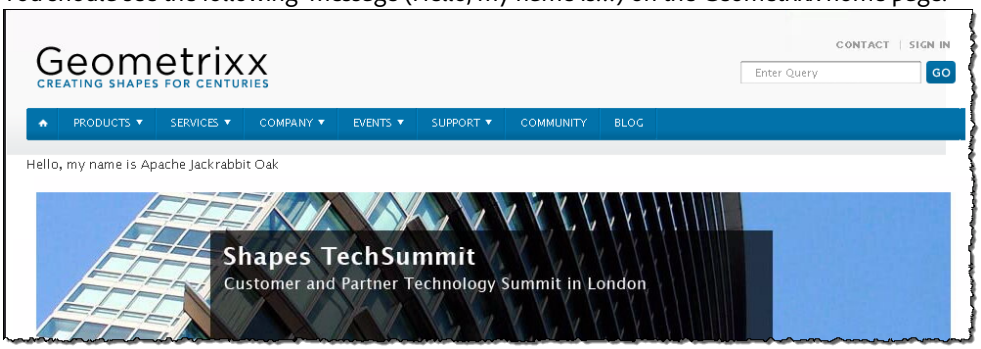
Under the web console, go to the **Components** tab (OSGI > components) http://localhost:4502/system/console/components and see the component.

2528	com.adobe.training.core.impl.RepositoryServiceImpl	
	Bundle	com.adobe.training.core (448)
	Implementation Class	com.adobe.training.core.impl.RepositoryServiceImpl
	Default State	enabled
	Activation	delayed
	Configuration Policy	optional
	Service Type	singleton
	Services	com.adobe.training.core.RepositoryService
	Reference repository	["Satisfied","Service Name: javax.jcr.Repository","Cardinality: 1..1","Policy: static","Policy Option: reluctant"]
	Properties	component.id = 2528 component.name = com.adobe.training.core.impl.RepositoryServiceImpl service.pid = com.adobe.training.core.impl.RepositoryServiceImpl service.vendor = Adobe

Navigate to CRXDE Lite http://localhost:4502/crx/de and go to the Geometrixx homepage component **content.jsp** script.
(Navigate to **/apps/geometrixx/components/homepage/content.jsp**) and replace the following code:

```
<%@page session="false"%><%--
  Copyright 1997-2009 Day Management AG
  Barfuesserplatz 6, 4001 Basel, Switzerland
  All Rights Reserved.

  This software is the confidential and proprietary information of
  Day Management AG, ("Confidential Information"). You shall not
  disclose such Confidential Information and shall use it only in
  accordance with the terms of the license agreement you entered into
  with Day.

  ==============================================================================

  Default content script.

  Draws the HTML content.

  ==============================================================================

--%>
<%@include file="/libs/foundation/global.jsp" %>
<div id="main">
    <div class="container_16">
        <% com.adobe.training.core.RepositoryService repositoryService =
    sling.getService(com.adobe.training.core.RepositoryService.class); %>
        Hello, my name is <%= repositoryService.getRepositoryName()%>
        <div class="grid_16">
            <cq:include path="carousel"
    resourceType="foundation/components/carousel"/>
        </div>
        <div class="grid_12 body_container">
            <cq:include path="lead" resourceType="geometrixx/components/lead"/>
            <cq:include path="par" resourceType="foundation/components/parsys"/>
        </div>
        <div class="grid_4 right_container">
            <cq:include path="rightpar" resourceType="foundation/components/parsys"/>
        </div>
        <div class="clear"></div>
    </div>
</div>
```

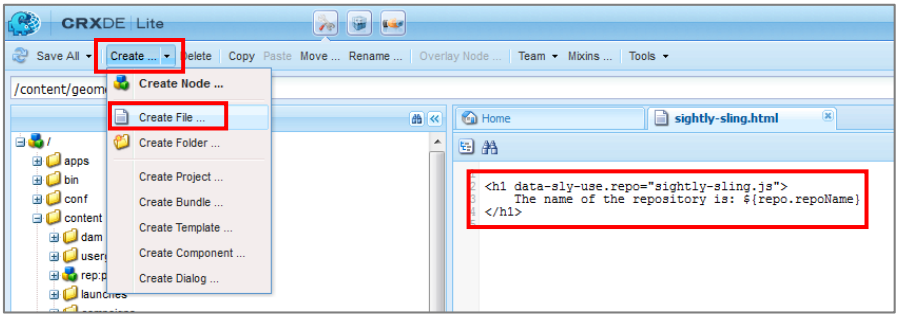
To see the service in action, go to the Geometrixx page using the following URL:
http://localhost:4502/content/geometrixx/en.html
You should see the following message (Hello, my name is...) on the Geometrixx home page:



This is how we can call a service in a page using JSP.

You can also consume the same service using Sightly code. You have to create a new file with the "html" extension as sightly-sling.html.

```
<h1 data-sly-use.repo="sightly-sling.js">
    The name of the repository is: ${repo.repoName}
</h1>
```



Create sightly-sling.js for the page with following code:

```
use(function () {
    var Repository = Packages.com.adobe.training.core.RepositoryService;
    return {
        repoName: sling.getService(Repository).getRepositoryName()
    }
});
```

Also, in the content.jsp file, add the following line:

```
<cq:include script="sightly-sling.html"/>
```

To see the corresponding output on the page, go to the URL:
http://localhost:4502/content/geometrixx/en.sightly-sling.html

Scenario Conclusion

We have created an OSGi service to display the customized message on the existing Geometrixx page.

CHAPTER FIVE: DEEP DIVE INTO SLING

Overview

This chapter deep dives into the Sling architecture, with focus on Sling Models, Servlets, Job scheduling, and event modeling. The Lab Activity at the end of the module has hands-on exercises for writing servlets, event handlers, and job scheduling.

Objectives

By the end of this chapter, you will:

- have in-depth knowledge of Sling architecture
- be able to write servlets, and job schedulers
- use Sling Models in your project

The Sling Architecture

Apache Sling is an open source web application framework that makes it easy to develop content-oriented applications. Sling can be described as:

- Representational State Transfer (REST) based web framework
- Content-driven, using a JCR content repository
- Powered by OSGi
- Using Scripts or Java Servlets to process HTTP requests

Sling applications are a set of OSGi bundles that use the OSGi core and compendium services. Apache Felix OSGi framework and console provide a dynamic runtime environment in which code and content bundles can be loaded, unloaded, and reconfigured at runtime.

Sling is resource-oriented, and maps into JCR nodes. In Sling, a request URL is first resolved to a resource, and then based on the resource, it selects the actual Servlet or script to handle the request.

RESTful Architecture

REST is a style of software architecture for distributed hypermedia systems such as the World Wide Web. Systems using REST architecture communicate through Hyper Text Transfer Protocol (HTTP), using GET and POST methods. The REST interface involves a collection of resources with identifiers. For example, `/training/english`.

REST is an architectural style:

- For network-based applications
- To induce the same architectural properties as the World Wide Web

Addressable resources present a uniform interface that allows transfers of state; for example, reading and updating of the resource's state. The best example of a RESTful architecture is the web, where resources have URIs and the uniform interface is HTTP.

This architectural style has the following properties:

- Performance and network efficiency
- Scalability
- Simplicity of interfaces
- Modifiability of components
- Visibility of communication
- Portability of components
- Reliability in resistance to internal failure

For most cases, a framework like HTTP (addressable resources plus "verbs" plus standard ways to transmit metadata) is all you need to build a distributed application. HTTP is a rich application protocol that gives you

capabilities like content negotiation and distributed caching. RESTful web applications try to leverage HTTP in its entirety using specific architectural principles.

Resource-oriented: Any piece of information (content object)—news entry, product description, or photo is a resource.

Addressable resources: With REST over HTTP, every object will have its own specific URI. From the URI, you know how to communicate with the object, find its location in the network, and identify it on the server it resides.

A uniform, constrained interface: When using REST over HTTP, stick to the methods provided by the protocol. This means following the meaning of GET, POST, PUT, and DELETE as defined with no additional methods or services.

Representation oriented: You interact with services using representations of that service. An object referenced by one URI can have different formats or renderings available. Different clients need different formats—Ajax may need JSON, a Java application may need XML, a browser may need HTML. You may also need a print-friendly rendering.

Communicate statelessly: REST, as implemented in HTTP, tends to be stateless; for example, it does not use cookies, and clients need to re-authenticate on every request. Client session data is not stored on the server. Session-specific information is held and maintained by the client and transferred to the server on each request, as needed. Stateless applications are easier to scale.

Advantages of REST

The following are known advantages of REST architecture.

- Uses well-documented, well-established, well-used technology and methodology
- Resource-centric rather than method-centric
- URI accessible by anyone
- No multiple protocols
- No specific format for response payload
- Uses the inherent HTTP security model
- Easy restriction of methods to URIs by firewall configuration, unlike other XML over HTTP messaging formats

Working with Sling Servlets

Servlets can be registered as OSGi services. Sling applications use scripts or Java servlets, selected based on simple name conventions, to process HTTP requests in a RESTful way. Being a REST framework, Sling is oriented around resources, which usually map into JCR nodes. With Sling, a request URL is first resolved to a resource, and then based on the resource, it selects the actual servlet or script to handle the request.

The GET method has default behaviors, and therefore requires no additional parameters. By decomposing the URL, Sling can determine the resource URL and other information that can be used to process that resource.

The POST method is handled by the Sling PostServlet. The PostServlet enables writes to a data store (usually the JCR) directly from HTML forms or from the command line, using cURL.

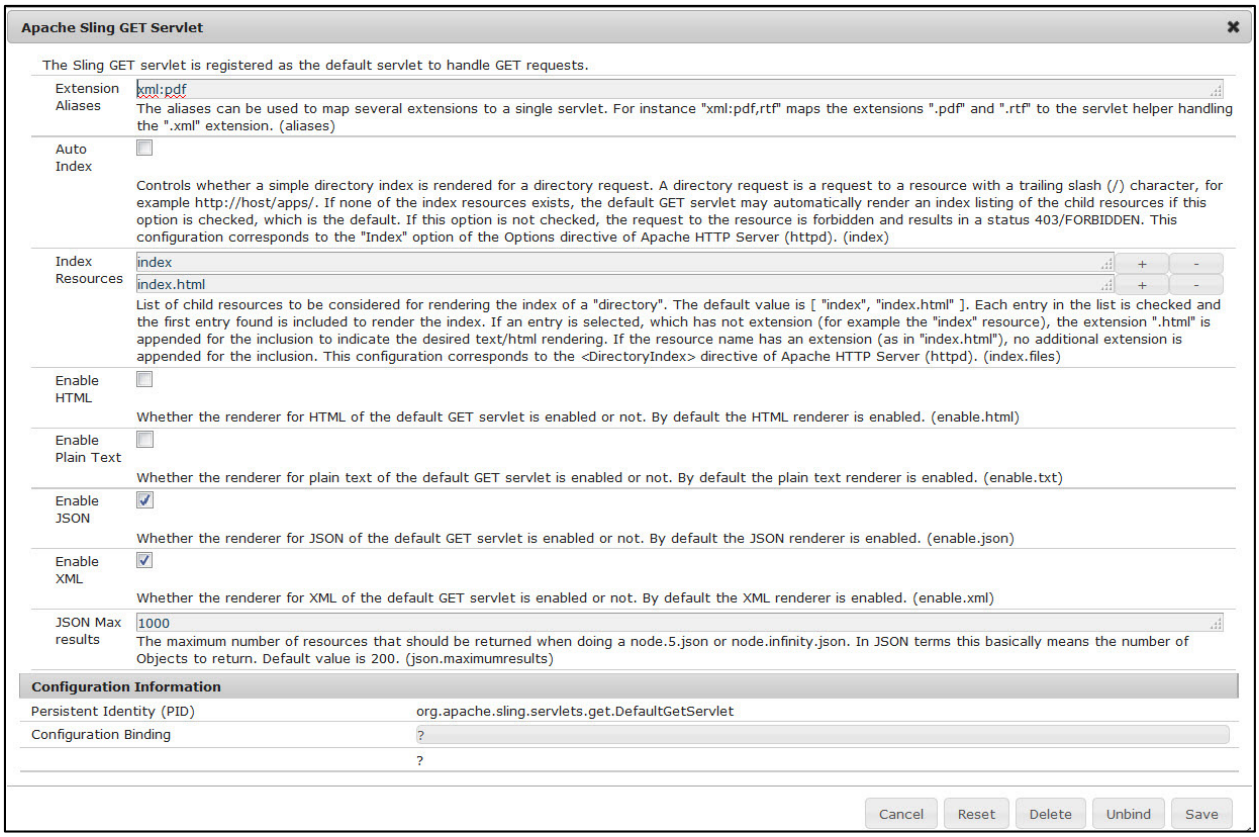
You can view the existing servlets through the Web Console, or you can click here:
http://localhost:4502/system/console/configMgr

Configuring the Default Sling GET Servlet

Sling provides default renderers for the following mime types of the default GET servlet:

- txt
- JSON
- docview.xml
- sysview.xml
- html

You can use the Configuration tab of the Web Console to configure the GET servlet.



Configuring the Sling POST Servlet

The Sling POST Servlet provides a RESTful interface that lets you manipulate data content stored in the Adobe Experience Manager JCR. This servlet maps nodes in the JCR repository to URIs and allows applications to modify JCR content. A Sling POST Servlet is performed on the client using JavaScript.

To create content in the JCR, send an http POST request with the path of the node where you want to store the content and the actual content, sent as request parameters. For example, consider the following script:

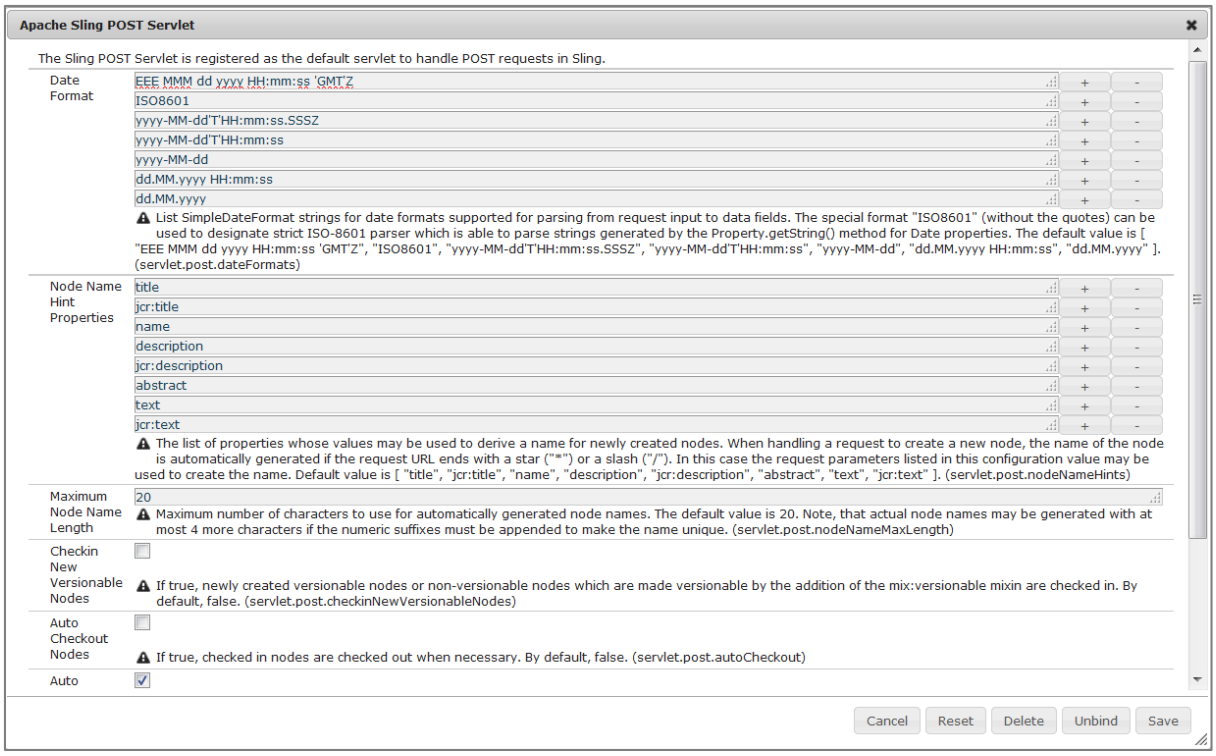
```
<form method="POST" action="http://host/mycontent"enctype="multipart-form/data">
<input type="text" name="title" value="" />
<input type="text" name="text" value="" />
</form>
```

The above lines will set the title and text properties on a node in the location `/mycontent`. If this node does not exist, it will be created; otherwise, the existing content at that URL would be modified.

You can perform a similar operation using the following curl commands:

```
curl –u admin:admin -F"jcr:primaryType=nt:unstructured" -Ftitle="some title text" -Ftext="some
    body text content" http://host/some/new/content
curl –u admin:admin -F":operation=delete" http://host/content/sample
```

You can use the Configuration tab of the Web Console to configure the Sling POST servlet.



Perform Task – Writing a Servlet, from the Lab Activity section.

Creating System Users

> Prior to AEM 6.1, system users were not required to access OSGi services in AEM. After deprecating ***session = repository.loginAdministrative(null);*** in AEM 6.0, Adobe has removed support for it altogether in AEM 6.1 and now requires service users to be created and configured for all sessions tied to OSGi services. Here is what the problem and solution are as identified in the Apache Sling Documentation.

Problem

To access the data storage in the Resource Tree and/or the JCR Repository authentication is required to properly setup access control and guard sensitive data from unauthorized access. For regular request processing this authentication step is handled by the Sling Authentication subsystem.

On the other hand there are also some background tasks to be executed with access to the resources. Such tasks cannot in general be configured with user names and passwords: Neither hard coding the passwords in the code nor having the passwords in – more or less – plain text in some configuration is considered good practice.

To solve this problem for services to identify themselves and authenticate with special users properly configured to support those services.

The solution presented here serves the following goals:

- Prevent over-use and abuse of administrative ResourceResolvers and/ro JCR Sessions
- Allow services access to ResourceResolvers and/or JCR Sessions without requiring to hard-code or configure passwords
- Allow services to use *service users* which have been specially configured for service level access (as is usually done on unixish systems)
- Allow administrators to configure the assignment of service users to services

Concept

A *Service* is a piece or collection of functionality. Examples of services are the Sling queuing system, Tenant Administration, or some Message Transfer System. Each service is identified by a unique *Service Name*. Since a service will be implemented in an OSGi bundle (or a collection of OSGi bundles), services are named by the bundles providing them.

A Service may be comprised of multiple parts, so each part of the service may be further identified by a *Subservice Name*. This Subservice Name is optional, though. Examples of

Subservice Name are names for subsystems in a Message Transfer System such as accepting messages, queueing messages, delivering messages.

Ultimately, the combination of the *Service Name* and *Subservice Name* defines the *Service ID*. It is the *Service ID* which is finally mapped to a Resource Resolver and/or JCR Repository user ID for authentication.

Thus the actual service identification (service ID) is defined as:

```
1 service-id = service-name [ ":" subservice-name ] .
```

The `service-name` is the symbolic name of the bundle providing the service.

Implementation

The implementation in Sling of the *Service Authentication* concept described above consists of three parts:

i. `ServiceUserMapper`

The first part is a new OSGi Service ServiceUserMapper. The ServiceUserMapper service allows for mapping *Service IDs* comprised of the *Service Names* defined by the providing bundles and optional *Subservice Name* to ResourceResolver and/or JCR Repository user IDs. This mapping is configurable such that system administrators are in full control of assigning users to services.

The `ServiceUserMapper` defines the following API:

```
1 String getServiceUserID(Bundle bundle, String subServiceName);
```

ii. `ResourceResolverFactory`

The second part is support for service access to the Resource Tree. To this avail, the `ResourceResolverFactory` service is enhanced with a new factory method

```
1  ResourceResolver getServiceResourceResolver(Map<String, Object>
2  authenticationInfo)     throws LoginException;
```

This method allows for access to the resource tree for services where the service bundle is the bundle actually using the `ResourceResolverFactory` service. The optional Subservice Name may be provided as an entry in the `authenticationInfo` map.

In addition to having new API on the `ResourceResolverFactory` service to be used by services, the `ResourceProviderFactory` service is updated with support for Service Authentication: Now

new API is required, though but additional properties are defined to convey the service to authenticate for.

iii. `SlingRepository`

The third part is an extension to the `SlingRepository`service interface to support JCR Repository access for services:

```
1  Session loginService(String subServiceName, String workspace)     throws
2  LoginException, RepositoryException;
```

This method allows for access to the JCR Repository for services where the service bundle is the bundle actually using the `SlingRepository` service. The additional Subservice Name may be provided with the `subServiceName` parameter.

Deprecation of administrative authentication

Originally the `ResourceResolverFactory.getAdministrativeResourceResolver` and `SlingRepository.loginAdministrative` methods have been defined to provide access to the resource tree and JCR Repository. These methods proved to be inappropriate because they allow for much too broad access.

Consequently these methods are being deprecated and will be removed in future releases of the service implementations.

The following methods are deprecated:

- `ResourceResolverFactory.getAdministrativeResourceResolver`
- `ResourceProviderFactory.getAdministrativeResourceProvider`
- `SlingRepository.loginAdministrative`

The implementations we have in Sling's bundle will remain implemented in the near future. But there will be a configuration switch to disable support for these methods: If the method is disabled, a `LoginException` is always thrown from these methods. The JavaDoc of the methods is extended with this information.

Perform Task – Creating a System User, from the Lab Activity section.

Understanding the Sling Resolution Process

Sling is resource-oriented, and maintains all resources in the form of a virtual tree. A resource is usually mapped to a JCR node, but can also be mapped to a file system or database. Some of the common properties that a resource can have are:

- Path – Includes the names of all resources beginning from the root, each separated by a slash (/). It is similar to a URL path or file system path.
- Name – This is the last name in the resource path.
- Resource Type - Each resource has a resource type that is used by the Servlet and Script resolver to find the appropriate Servlet or Script to handle the request for the Resource.

When using Sling, the type of content to be rendered is not the first processing consideration. Instead, the main consideration is whether the URL resolves to a content object for which a script can then be found to "perform the rendering." This provides excellent support for web content authors to build pages, which are easily customized to their requirements. The advantages of this flexibility are apparent in applications.

Resource First Request Processing

When a request URL comes in, it is first resolved to a resource. Then, based on the resource, it selects the servlet or script to handle the request.

Traditional Web Application Framework	Sling Framework

Basic Steps of Processing Requests

Each content item in the JCR repository is exposed as an HTTP resource, so the request URL addresses the data to be processed, not the procedure that does the processing. After the content is determined, the script or servlet to be used to handle the request is determined in cascading resolution that checks, in order of priority:

- Properties of the content item itself
- The HTTP method used to make the request
- A simple naming convention within the URL that provides secondary information

For every URL request, the following steps are performed to get it resolved:

Decompose the URL

Search for a file indicated by the URL

Resolve the Resource

Resolve the rendering script/servlet

Create rendering chain

Invoke rendering chain

iv. Decomposing the URL

Processing is done based on the URL requests submitted by the user. The elements of the URL are extracted from the request to locate and access the appropriate script.

Example

URL: `http://myhost/tools/spy.printable.a4.html/a/b?x=12`

The above URL can be decomposed into the following components:

protocol	Host	Content path	Selector(s)	Extension		Suffix		Param(s)
http://	myhost	tools/spy	.printable.a4	html	/	a/b	?	x=12

The following table describes the components.

Protocol	Hypertext transfer protocol
Host	Name of the website
Content path	Path specifying the content to be rendered. It is used in combination with the extension.
Selector(s)	Used for alternative methods of rendering the content
Extension	Content format; also specifies the script to be used for rendering.
Suffix	Can be used to specify additional information
Param(s)	Any parameters required for dynamic content

v. Mapping Request to Resources

Once the URL is decomposed, the content node is located from the content path. This node is identified as the resource, and performs the following steps to map to the request.

Consider the URL request: `http://myhost/tools/spy.html`

Sling checks whether a node exists at the location specified in the request. In the above example, it searches for the node `spy.html`

If no node is found at that location, the extension is dropped and the search is repeated. For example, it searches for the node `spy`.

If no node is found, then Sling will return the http code 404 (Not Found).

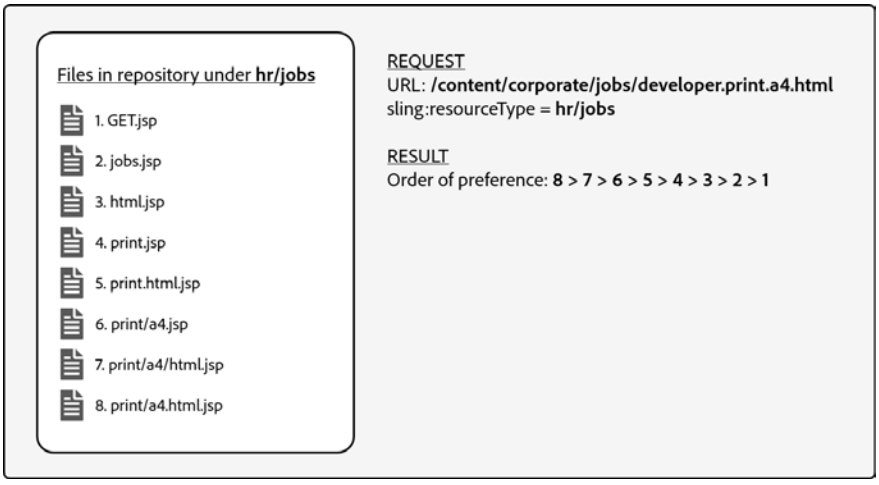
If a node is found, then the sling resource type for that node is extracted, and used to locate the script to be used for rendering the content.

vi. Locating and Rendering Scripts

All scripts are stored in either the /apps or /libs folder, and are searched in the same order. If no matching script is found in either of the folders, then the default script is rendered. When the resource is identified from the URL, its resource type property is located and the value extracted. This value is either an absolute or a relative path that points to the location of the script to be used for rendering the content.

For multiple matches of the script, the script with the best match is selected. The more selector matches, the better.

Example



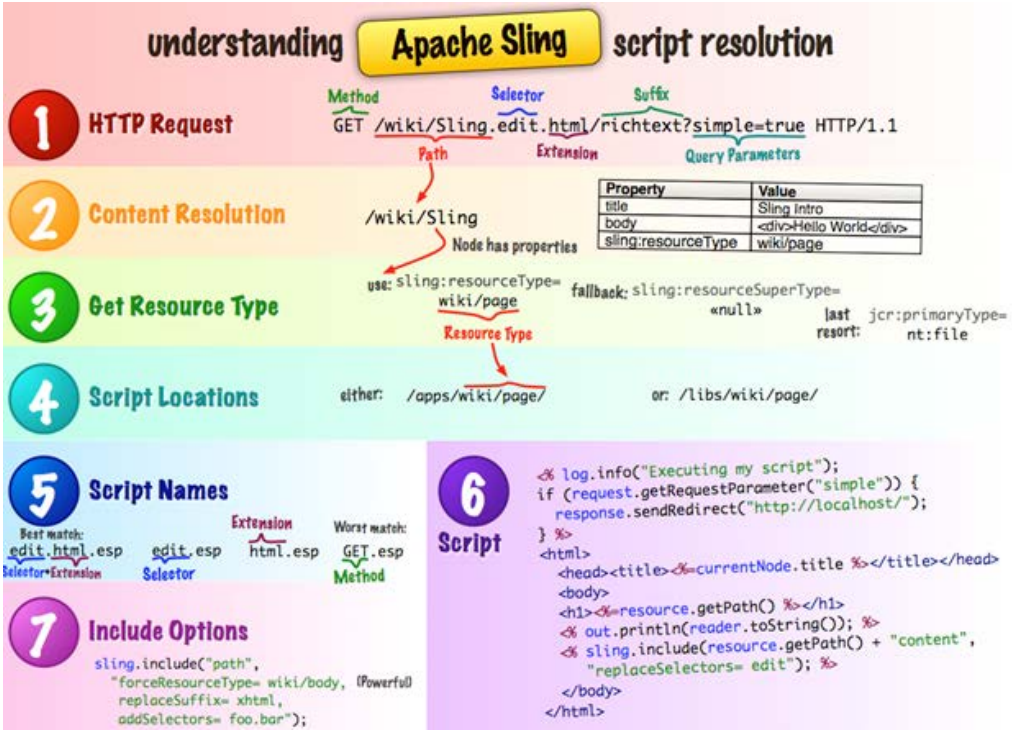
Super types are also taken into consideration when trying to locate a script. The advantage of resource super types is that they may form a hierarchy of resources where the default resource type `sling/servlet/default` (used by the default servlets) is effectively the root.

The resource super type of a resource may be defined in two ways:

`sling:resourceSuperType` property of the resource.

`sling:resourceSuperType` property of the node to which the `sling:resourceType` points.

To summarize how a script is resolved:



The Resource Resolver

The Resource Resolver is a part of Sling that resolves incoming requests to actual or virtual resources. For example, a request for `/training/english` will be resolved to a corresponding JCR node. However, if you do not want to expose the internal JCR structure, or if it cannot be resolved to a JCR node, you can define a set of resolver rules through the Web Console. You can also use the standard OSGi configuration mechanisms in the CRX to define a set of resolver rules.

The Resource Resolver abstracts:

- The path resolution
- Access to the persistence layer(s)

Resource mapping is used to define redirects, vanity URLs, and virtual hosts. You can access the Resolver Map entries through the Resource Resolver tab of the Web Console. You can access the console with this link: http://localhost:4502/system/console/jcrresolver

vii. Mappings for Resource Resolution

The following node types help with the definition of redirects and aliases.

Node Types	Description
sling:ResourceAlias	Mixin node type defines the sling:alias property, and may be attached to any node, which does not allow the setting of a property named sling:alias
sling:MappingSpec	Mixin node type defines the sling:match, sling:redirect, sling:status, and sling:internalRedirect properties.
sling:Mapping	The primary node type used to construct entries in /etc/map.

The following properties are important when dealing with new resources.

Properties	Description
sling:match	Defines a partial regular expression used on a node's name to match the incoming request.
sling:redirect	Causes a redirect response to be sent to the client.
sling:status	Defines the HTTP status code sent to the client.
sling:internalredirect	Causes the current path to be modified internally to continue with resource resolution.
sling:alias	Indicates an alias name for the resource.

Each entry in the mapping table is a regular expression, which is constructed from the resource path below `/etc/map`. The following rules apply:

- If any resource along the path has a `sling:match` property, the respective value is used in the corresponding segment instead of the resource name.
- Only resources having a `sling:redirect` or `sling:internalRedirect` property are used as table entries. Other resources in the tree are just used to build the mapping structure.

Example of resource mapping

Consider the following content:

```
/etc/map
      +-- http
           +-- example.com.80
           |    +-- sling:redirect = "http://www.example.com/"
           +-- www.example.com.80
           |    +-- sling:internalRedirect = "/example"
           +-- any_example.com.80
           |    +-- sling:match = ".+\.example\.com\.80"
           |    +-- sling:redirect = "http://www.example.com/"
           +-- localhost_any
           |    +-- sling:match = "localhost\.\d*"
           |    +-- sling:internalRedirect = "/content"
           |    +-- cgi-bin
           |    |    +-- sling:internalRedirect = "/scripts"
           |    +-- gateway
           |    |    +-- sling:internalRedirect = "http://gbiv.com"
           |    +-- (stories)
           |         +-- sling:internalRedirect = "/anecdotes/$1"
           +-- regexmap
                +-- sling:match = "$1.example.com/$2"
                +-- sling:internalRedirect = "/content/([^/]+)/(.*)"
```

Based on the above definition, we get the following mappings:

Regular Expression	Redirect	Internal	Description
http/example.com.80	http://www.example.com	no	Redirect all requests to the Second Level Domain to www
http/www.example.com.80	/example	yes	Prefix the URI paths of the requests sent to this domain with the string /example
http/.+.example.com.80	http://www.example.com	no	Redirect all requests to sub domains to www. The actual regular expression for the host.port segment is taken from the sling:match property.
http/localhost.\d*	/content	yes	Prefix the URI paths with /content for requests to localhost, regardless of actual port the request was received on. This entry only applies if the URI path does not start with /cgi-bin, gateway or stories because there are longer match entries. The actual regular expression for the host.port segment is taken from the sling:match property.
http/localhost.\d*/cgi-bin	/scripts	yes	Replace the /cgi-bin prefix in the URI path

			with /scripts for requests to localhost, regardless of actual port the request was received on.
http/localhost.\d*/gateway	http://gbiv.com	yes	Replace the /gateway prefix in the URI path with http://gbiv.com for requests to localhost, regardless of actual port the request was received on.
http/localhost.\d*/(stories)	/anecdotes/stories	yes	Prepend the URI paths starting with /stories with /anecdotes for requests to localhost, regardless of actual port the request was received on.

viii. Adapting Resources

An adapter design pattern is used when you want two different classes with incompatible interfaces to work together. Sling offers an adapter pattern to conveniently translate objects that implement the Adaptable interface. This interface provides a generic `adaptTo()` method that will translate the object to the class type being passed as the argument.

The Resource and Resource Resolver interfaces are defined with a method `adaptTo()`, which adapts the object to other classes. The `adaptTo` pattern is used to adapt two different interfaces (for example, resource to node).

```
Node node = resource.adaptTo(Node.class)
```

Using this mechanism, the JCR session of the Resource Resolver calls the `adaptTo` method with the `javax.jcr.Session` class object. Likewise, the JCR node on which a resource is based can be retrieved by calling the `Resource.adaptTo` method with the `javax.jcr.Node` class object.

`Adaptable.adaptTo()` can be implemented in multiple ways:

- By the object itself, implementing the method itself and mapping to certain objects
- By an AdapterFactory which can map arbitrary objects
- A combination of both

Use Cases for adaptTo()

- Get implementation-specific objects
- Shortcut creation of objects that require internal context objects to be passed
- Shortcut to services

Understanding Sling Events

Events are used to trigger jobs or workflows. Sling enables you to manage these events within your application. Apache Sling provides support for eventing, handling jobs, and scheduling. Sling's event mechanism leverages the OSGi Event Admin Specification. The OSGi API for events is simple and lightweight. Sending an event involves generating the event object and calling the event admin. Receiving an event requires implementation of

a single interface and a declaration, through properties, on the interested topics. Each event is associated with an event topic, and event topics are hierarchically organized.

The OSGi specification for event handling uses a publish/subscribe mechanism based on these hierarchical topics. There is a loose coupling of the services, based on the whiteboard pattern. When the publisher has an event object to deliver, it calls all event listeners in the registry.

Various types of events can be handled:

- Predefined events include Framework events; for example, a Bundle event, which indicates a change in a bundle's life cycle
- Service events, which indicate a service life cycle change
- Configuration events, which indicate that a configuration has been updated or deleted.
- JCR observation events
- Application-generated events
- Events from messaging systems (~JMS)
- External events

You can find details regarding to the events in the Web Console, under the Sling and OSGi tabs.

- http://localhost:4502/system/console/events
- http://localhost:4502/system/console/slingevent

Listening to OSGi Events

The event mechanism is a mechanism that must:

- Implement the `EventHandler` interface.
- Subscribe by service registration using the Whiteboard pattern, that is, polling for events.
- Select the event with service properties, based on the event topic and a filter.

Events are received by the event mechanism and distributed to registered listeners. Concepts like durable listeners, guarantee of processing, and so on are not part of the event mechanism itself.

To listen to OSGi events in Sling, you need to register an `org.osgi.service.event.EventHandler` service with an `event.topics` property that describes which event topics the handler is interested in.

For example:

```
@scr.property name="event.topics"
valueRef="ReplicationAction.EVENT _ TOPIC"
```

or

```
@scr.property name="event.topics"
valueRef="org.apache.sling.api.SlingConstants.TOPIC _ RESOURCE _ ADDED"
```

The annotation `@Service` is set to `value = EventHandler.class` to indicate this is an event handler service. The code outline looks like this:

```
@Component
@Property(name = “event.topics”, value =
ReplicationAction.EVENT_TOPIC)
@Service(value = EventHandler.class)
public class MyEventListener implements JobConsumer,EventHandler {
public void handleEvent(Event event) {
if (EventUtil.isLocal(event)) {
JobUtil.processJob(event, this);
}
}
```

Publishing Events

To publish an event, you need to:

Get the EventAdmin service.

Create the event object (with a topic and properties).

Send the event (synchronously or asynchronously).

Sending Job Events

A job event is an event with the topic `/org/apache/sling/event/job` and the topic of the event is stored in the `event.job.topic` property. Job events are always processed, and are used in situations such as sending notification messages, or post-processing images. These events must be handled only once.

To send a job event, the service needs to implement:

- the `org.osgi.service.event.EventHandler` interface
- the `org.apache.sling.event.JobConsumer` interface

Implementing Event Handling

Event handling can be done at any of the following levels:

- JCR level, with observation
- Sling level, with event handlers and jobs
- Adobe Experience Manager level, with workflows and launchers

Perform Task – Write an Event Handler, from the Lab Activity section.

Working with Sling Schedules

The Sling Scheduler enables you to easily schedule jobs within your application. Jobs can be executed at a specific time, regularly at a given period or at the time given by a cron expression by leveraging the Sling scheduler service. It makes use of the open source Quartz library.

OSGi Service Fired by Quartz

To make use of the Quartz library using the scheduler's whiteboard pattern, the following outline code is needed:

```
package <package name>;
@Component
@Service (interface=”java.lang.Runnable”)
@Property (name=”scheduler.expression” value=”0 0/10 * * * ?”, type=”String”)
public class MyScheduledTask implements Runnable {
public void run() {
//place events to run here.
}
}
```

- The `@Component` annotation is used to register the component. This should include immediate=true to activate the component immediately.
- The `@Service(interface=”java.lang.Runnable”)` annotation makes it possible to schedule this service.
- `@Property(name=”scheduler.expression”, value=”0 0/10 * * * ?”, type=”String”)` is where we write the Quartz expression for the scheduled time interval.
- The class must also implement Runnable and contain a `run()` method.

ix. Quartz Trigger Syntax

The cron trigger expression comprises a series of parameters separated by white space, arranged as follows:

`<Seconds> <Minutes> <hours> <Day of Month> <Month> <Day of Week> <Year>`

It can include special characters such as:

- '*' – used to specify all values
- '?' - allowed for the day-of-month and day-of-week fields. It is used to specify 'no specific value'. This is useful when you need to specify something in one of the two fields, but not the other.
- '-' – used to specify ranges. For example '10-12' in the hour field means 'the hours 10, 11, and 12'.
- ',' – used to specify additional values. For example 'MON,WED,FRI' in the day-of-week field means 'the days Monday, Wednesday, and Friday'.
- '/' - used to specify increments. For example '0/15' in the seconds field means 'the seconds 0, 15, 30, and 45.

For example, '`0 0 12 * * ?`' means 'Fire at 12 pm (noon) every day'.

Scheduling Jobs

The following sections describe various methods of scheduling jobs.

x. Scheduling at periodic times

Instead of specifying a Quartz expression for the time interval, in simpler cases, we can just specify a time period, and the job will run repeatedly at this interval in seconds.

For example, for a 10-second interval:

```
@Property(name="scheduler.period", value="10", type="Long")
```

xi. Preventing concurrent execution

If you do not want concurrent execution of the job, you can prevent it using the `scheduler.concurrent` parameter:

```
@Property(name="scheduler.concurrent",value="false",type="Boolean",
private="true")
```

xii. Scheduling Jobs programmatically

To programmatically schedule a job, you need a Runnable class and can then add schedule times using appropriate methods:

```
@Reference
private Scheduler scheduler;
this.scheduler.addJob("myJob", job, null, "0 15 10 ? * MON FRI", true);
// periodic:
this.scheduler.addPeriodicJob("myJob", job, null, 3*60, true);
// one time
this.scheduler.fireJobAt("myJob", job, null, fireDate);
```

Perform Task – Schedule a job, from the Lab Activity section.

Perform Task – Change job scheduler configuration settings using repository, from the Lab Activity section.

Working with Sling Models

It is considered a best practice to use Sling Modeling for developing code. Sling Models let you map Java objects to Sling resources. When developing an Adobe Experience Manager project, you can define a model object (a Java object) and map that object to Sling resources. They have the following objectives:

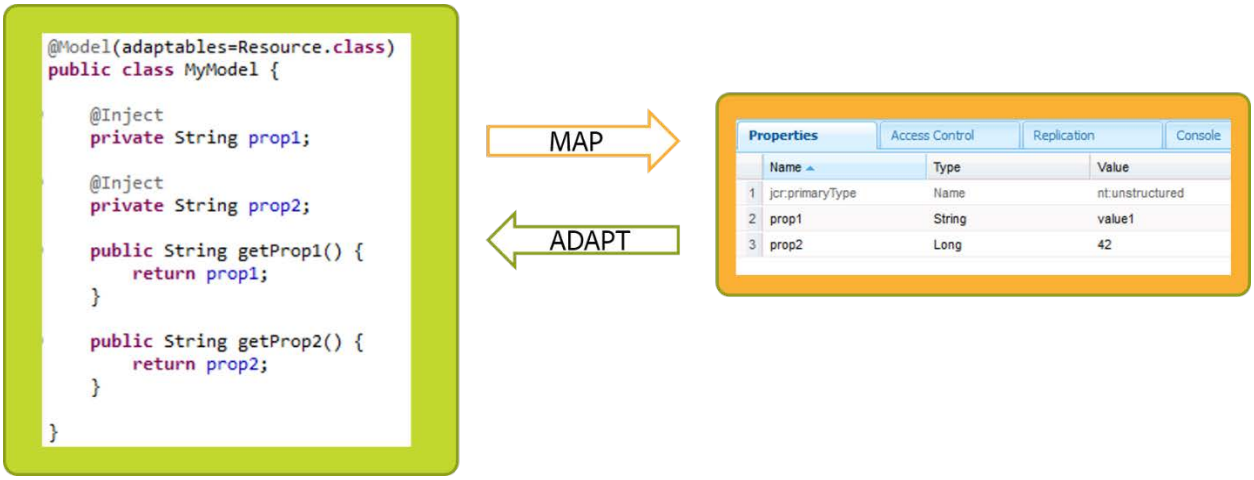
- Entirely annotation-driven.
- Use standard annotations where possible.
- Pluggable
- OOTB, support resource properties, SlingBindings, OSGi services, request attributes

- Adapt multiple objects
- Support both classes and interfaces.
- Work with existing Sling infrastructure

When to use Sling Models?

Basically, you would use Sling Models when you have one of the following situations:

- You have a model object (as a Java class or interface)
- You need to use it with Adobe Experience Manager without creating your own adapters
- You need to map your Plain Old Java Object (POJO) into a Sling Resource



Benefits of Using Sling Model

- Saves time from creating own adapters (avoiding boilerplate code)
- Support both classes and interfaces
- Support OOTB (out-of-the-box) resource properties (via ValueMap), SlingBindings, OSGi services, request and attributes
- Allows you to adapt multiple objects - minimal required Resource and SlingHttpServletRequest
- Client doesn't know/care that these objects are different than any other adapter factory
- Works with existing Sling infrastructure (i.e. not require changes to other bundles)
- Provides the ability to mock dependencies with tools like Mockito *@InjectMocks*

Understanding the Sling Model

Sling Models allow developers to inject method return values (in an interface) and fields (in a class) based on Sling resources and OSGi services. Out-of-the-box, they support resource properties (via ValueMap), SlingBindings, OSGi services, and request attributes. Most injections are available when adapting either a `Resource` or `SlingHttpServletRequest` object.

Sling models are entirely annotation-driven. A Sling Model is implemented as an OSGi bundle; a Java class located in the OSGi bundle is annotated with @Model and the Adaptable class (for example, `@Model(adaptables = Resource.class)`). The data members (fields) use @Inject annotations.

```
import javax.inject.Inject;

import org.apache.sling.api.resource.Resource;
import org.apache.sling.models.annotations.Model;

@Model(adaptables=Resource.class)
public class MyModel {

    @Inject
    private String prop1;

    @Inject
    private String prop2;

    public String getProp1() {
        return prop1;
    }

    public String getProp2() {
        return prop2;
    }
}
```

Sling Models let you to access Sling content without creating your own adapters, thus avoiding a large amount of boilerplate code.

In most cases, using a Sling Model Interface will require less code to accomplish the same task. In the case of Sling Model classes, the most common case where you would want to use them is if you need to do any filtering or manipulation of the values being returned. When using a class for your model, you can inject the values into the fields and generate a formatted return value.

In the simplest case, the class is annotated with @Model and the adaptable class. Fields that need to be injected are annotated with @Inject:

```
@Model(adaptables=Resource.class)
public class MyModel {

    @Inject
    private String propertyName;
```

In this case, a property named propertyName will be looked up from the Resource (after first adapting it to a ValueMap) and it is injected.

For an interface, it is similar:

```
@Model(adaptables=Resource.class)
public interface MyModel {

    @Inject
    String getPropertyName();
}
```

Client code doesn't need to be aware that Sling Models is being used. It just uses the Sling Adapter framework:

```
MyModel model = resource.adaptTo(MyModel.class)
```

If the field or method name doesn't exactly match the property name, you can use @Named:

```
@Model(adaptables=Resource.class)
public class MyModel {

    @Inject @Named("secondPropertyName")
    private String otherName;
}
```

xiii. Annotation Usage

Sling Model Annotation	Code Snippet	Injector
@Model	@Model(adaptables = Resource.class)	Class is annotated with @Model and the adaptable class
@Inject	@Inject private String propertyName; (class) @Inject String getPropertyName(); (interface)	A property named "propertyName" will be looked up from the Resource (after first adapting it to a ValueMap) and it is injected, else will return null if property not found.
@Default	@Inject @Default(values="AEM") private String technology;	A default value (for Strings, Arrays, primitives etc.)
@Optional	@Inject @Optional private String otherName	@Injected fields/methods are assumed to be required. To mark them as optional, use @Optional for example, resource or adaptable may or may not have property.
@Named	@Inject @Named("title") private String page Title;	Inject a property whose name does not match the Model field name.
@Via	@Model(adaptables=SlingHttpServletRequest.class) Public interface SlingModelDemo { @Inject @Via("resource") String getPropertyName(); }	Use a JavaBean property of the adaptable as the source of the injection //Code snippet will return request.getResource().adaptTo(ValueMap.class).get("propertyName", String.class)

@Source	@Model(adaptables=SlingHttpServletRequest.class) @Inject @Source("script-bindings") Resource getResource();	Explicitly tie an injected field or method to a particular injector (by name). Can also be on other annotations. //Code snippet will ensure that "resource" is retrieved from the bindings, not a request attribute.
@PostConstruct	@PostConstruct protected void sayHello() { logger.info("hello"); }	Methods to call upon model option creation (only for model classes).

xiv. Injector Specific Annotations

To create a custom injector, simply implement the `org.apache.sling.models.spi.Injector` interface and register your implementation with the OSGi service registry. Here's a list of the available injectors:

https://sling.apache.org/documentation/bundles/models.html#available-injectors

Sometimes it is necessary to use customized annotations that aggregate the standard annotations described above. This generally has the following advantages over using the standard annotations:

- Less code to write (only one annotation is necessary in most of the cases)
- More robust (in case of name collisions among the different injectors, you make sure that the right injector is used)
- Better IDE support (because the annotations provide elements for each configuration which is available for that specific injector; for example, filter only for OSGi services)

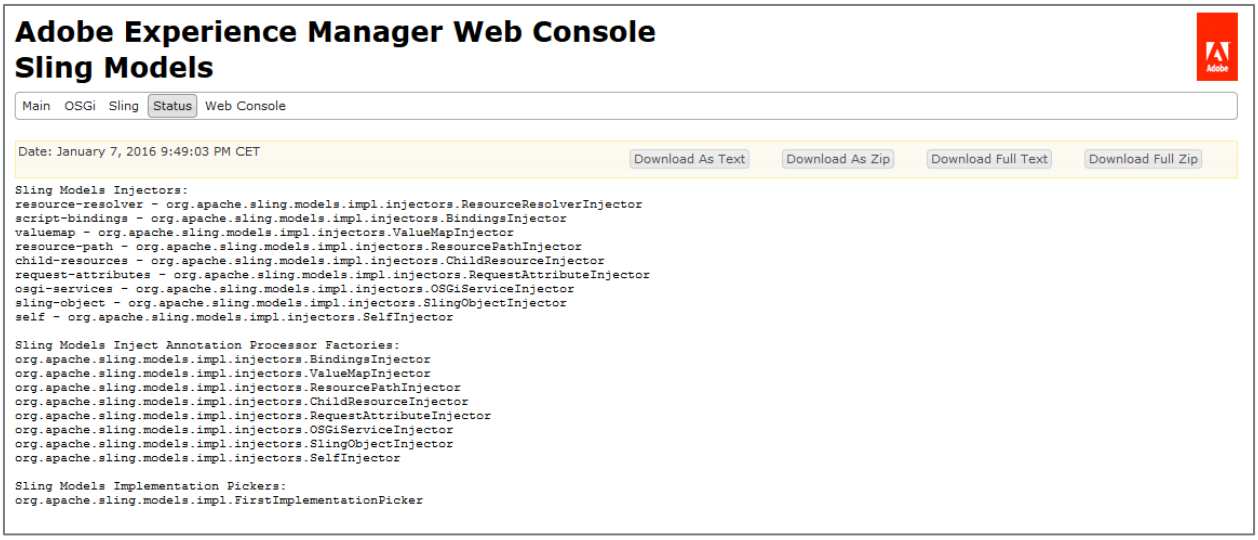
The following annotations are provided, which are tied to specific injectors:

Annotation	Supported Optional Elements	Injector
@ScriptVariable	optional and name	script-bindings
@ValueMapValue	optional, name and via	valuemap
@ChildResource	optional, name and via	child-resources
@RequestAttribute	optional, name and via	request-attributes
@ResourcePath	optional, path, and name	resource-path
@OSGiService	optional, filter	osgi-services
@Self	optional	self
@SlingObject	optional	sling-object

xv. Injectors Lookup in Web Console

List of available injectors in the Felix console:

http://localhost:4502/system/console/status-slingmodels



Debugging

In order to see logging specific to Sling Models, add a logger for the package org.apache.sling.models, set the log level to TRACE and dump into appropriate log file.

Perform Task – Implementing Sling Model, from the Lab Activity section.

Chapter 5 Lab Activity - I

Scenario

You need to create servlets that can display the title of the page, as well as the properties of the nodes using Sling servlet capabilities.

When a page is replicated to the publish server, the action needs to be logged in the log file.

Monitor your repository for unwanted resources created by Adobe Experience Manager processes, and remove them accordingly.

You need to manage custom configurations using Adobe Experience Manager repository.

Display the name of a user stored in a different node in the repository.

Challenge

You need to use best practices while creating servlets, event handlers, and job schedules.

Overview

In your implementation, you need to write a Servlet so that you can consume it on your website pages. The servlet will serve the **DOGET request** to provide the **JCR repository properties** on a page. So wherever needed, this servlet will be invoked and the servlet will return the JCR properties on the page.

Write a class to perform a task on an event occurrence in the repository. Event can be anything related to JCR repository update/change. You will **write an event handler** to **create audit log** whenever a **replication event** occurs.

In your instance, you need to complete a set of task periodically; such tasks can be defined in a job and you can schedule that job to complete the tasks. As an example, you have to **delete temporary nodes** in your repository. This configuration shall be stored in a configuration node, which can be seen and edited in the Web console.

Modify the configuration node setting using JCR.

Using Sling Model best practices, create a class that uses the values provided by another class.

Prerequisites

You need to have the project—trainingproject provided in the USB contents. You also need:

Adobe Experience Manager Author instance

Adobe Experience Manager Publish instance

Eclipse, Maven, and the plugins for Eclipse installed

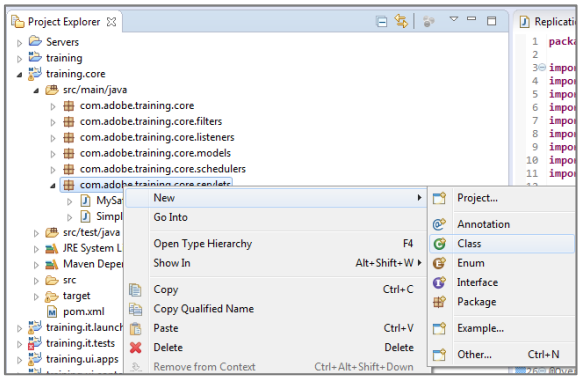
Steps

1. Task - Writing a Servlet

   In this task, you will write a servlet that serves only DOGET requests, and expose it at a static URI. The response shall contain the JCR repository properties as JSON.

   You have to use the project—trainingproject provided in the USB contents. Here, you can write your servlet to serve the DOGET request. Follow the steps to complete this task.

1. Open Eclipse and navigate to **training.core** > **src/main/java** and right-click on **com.adobe.training.core.servlets** and select **New** > **Class**.



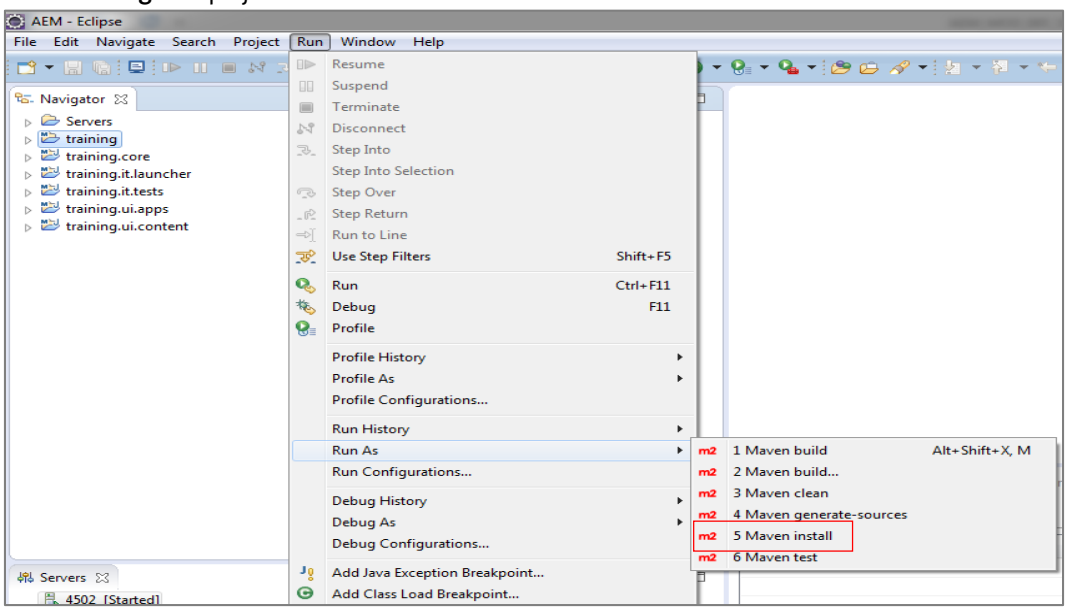
2. Create a java class named **TitleSlingServlet** and click **Finish**.

3. Paste the following code in the file and save the changes by selecting **File** > **Save.**

```
package com.adobe.training.core.servlets;

import java.io.IOException;
import javax.servlet.ServletException;
import org.apache.felix.scr.annotations.sling.SlingServlet;
import org.apache.sling.api.SlingHttpServletRequest;
import org.apache.sling.api.SlingHttpServletResponse;
import org.apache.sling.api.servlets.SlingSafeMethodsServlet;

@SlingServlet(paths="/bin/company/titleservlet")

public class TitleSlingServlet extends SlingSafeMethodsServlet {
    private static final long serialVersionUID = -3960692666512058118L;
    @Override
    protected void doGet(SlingHttpServletRequest request,
    SlingHttpServletResponse response) throws ServletException, IOException {
        response.getWriter().print("<h1>Sling Servlet injected this title</h1>");
    }
}
```

Please note that the servlet **Path** mentioned in the code is **/bin/company/titleservlet**, so the request can be served on this path.

4. Select **training.core** project and click **Run As** > **Maven install**.



Eclipse will build the project and install it on the AEM server.

5. As a test and to verify, you can call the title servlet via http://localhost:4502/bin/company/titleservlet

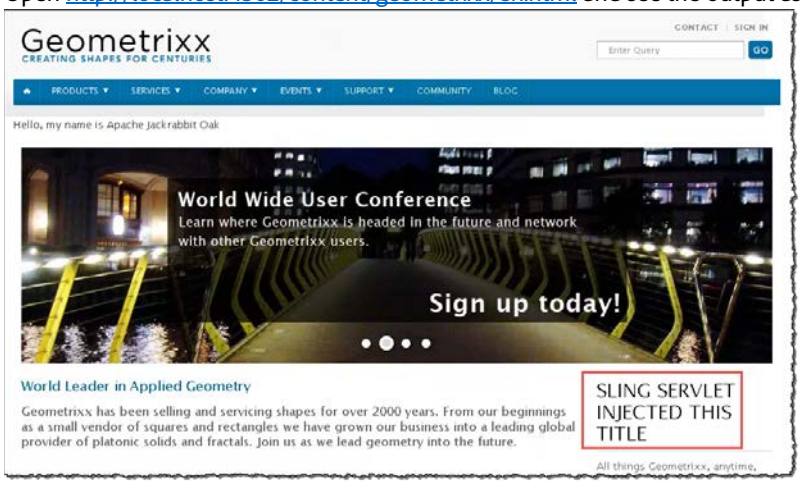
6. In this step, you will change the servlet implementation by changing the code in **TitleSlingServlet.java** as follows:

```
import java.io.IOException;
import javax.servlet.ServletException;
import org.apache.felix.scr.annotations.sling.SlingServlet;
import org.apache.sling.api.SlingHttpServletRequest;
import org.apache.sling.api.SlingHttpServletResponse;
import org.apache.sling.api.servlets.SlingSafeMethodsServlet;

@SlingServlet(paths="/bin/company/titleservlet",
    resourceTypes="/apps/geometrixx/components/title", extensions="html")

public class TitleSlingServlet extends SlingSafeMethodsServlet {
    private static final long serialVersionUID = -3960692666512058118L;
    @Override
    protected void doGet(SlingHttpServletRequest request, SlingHttpServletResponse
    response) throws ServletException, IOException {
        response.getWriter().print("<h1>Sling Servlet injected this title</h1>");
    }
}
```

You can call the servlet as a resource type. Note that we changed the **@SlingServlet** annotation using **Path** and **extension**. This is the suggested way to create a servlet. The request is served on the path "/bin/company/titleservlet".

7. Right-click **training.core**, and perform **Run As** > **Maven install** to build the project.

8. Open http://localhost:4502/content/geometrixx/en.html and see the output as follows:



9. Create another class LeadSlingServlet.java under training.core > src/main/java > com.adobe.training.core.servlets with the following code:

```
package com.adobe.training.core.servlets;
import java.io.IOException;
import javax.jcr.Repository;
import javax.servlet.ServletException;
import org.apache.felix.scr.annotations.Reference;
import org.apache.felix.scr.annotations.sling.SlingServlet;
import org.apache.sling.api.SlingHttpServletRequest;
import org.apache.sling.api.SlingHttpServletResponse;
import org.apache.sling.api.servlets.SlingSafeMethodsServlet;
@SlingServlet(resourceTypes="/apps/geometrixx/components/lead", extensions="html",
    selectors="lead-sling-servlet")
public class LeadSlingServlet extends SlingSafeMethodsServlet {
    private static final long serialVersionUID = 6165211752185245787L;
    @Reference
    private Repository repository;
    @Override
    protected void doGet(SlingHttpServletRequest request, SlingHttpServletResponse
    response) throws ServletException, IOException {
        response.setHeader("Content-Type", "text/html");
        String[] keys = repository.getDescriptorKeys();
        response.getWriter().print("<br><table border='1'><th>Repository
    Descriptor</th><th>Value</th>");
        for (int i = 0; i< keys.length; i++) {
            try{
                response.getWriter().print("<tr><td>" + keys[i] + "</td><td>" +
    repository.getDescriptor(keys[i]) + "</td></tr>");
            }
            finally {
            }
        }
        response.getWriter().print("</table>");
    }
}
```

This servlet shows how to access a JCR repository via a servlet using the OSGi framework. It also demonstrates the selector attribute.

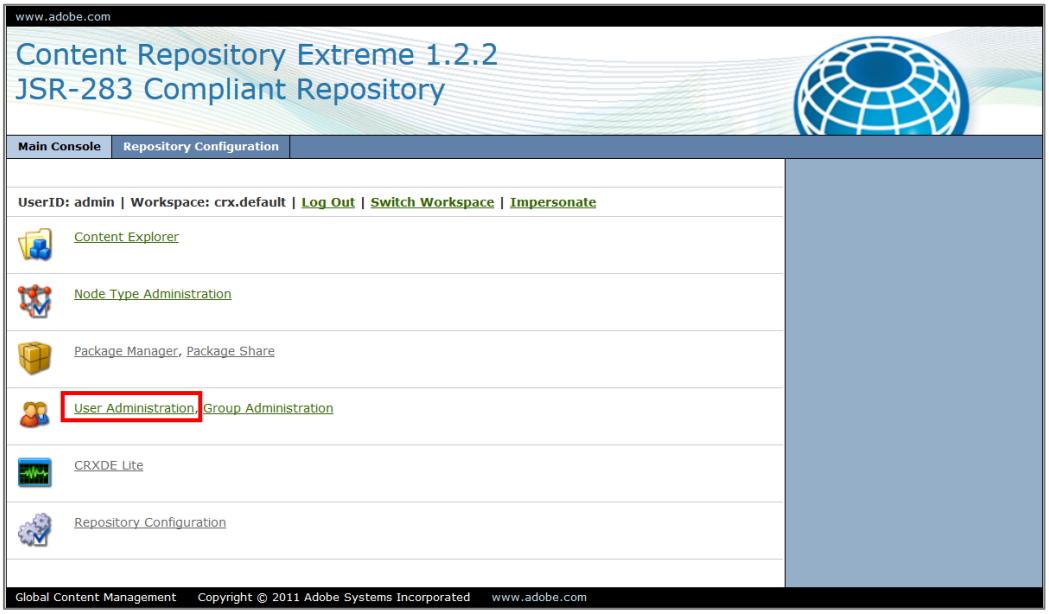
10. Build the project by right-click **training.core**, and selecting **Run As** > **Maven install**.

11. Open http://localhost:4502/content/geometrixx/en.lead-sling-servlet.html and see the table output in the page as:

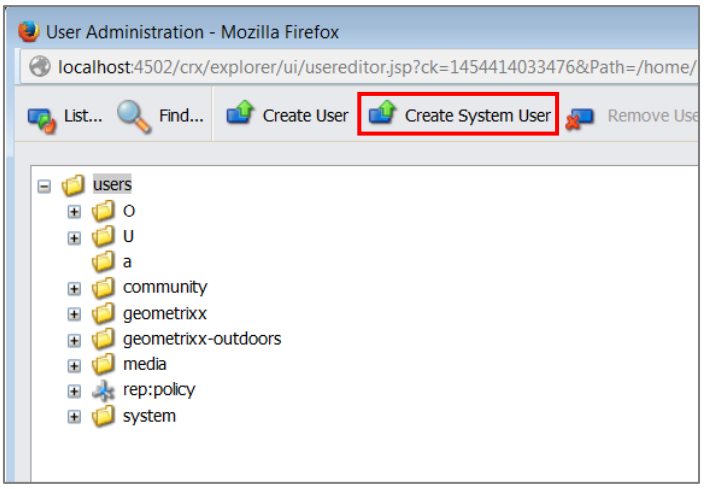
2. Task – Create a service user

Navigate to http://localhost:4502/crx/explorer

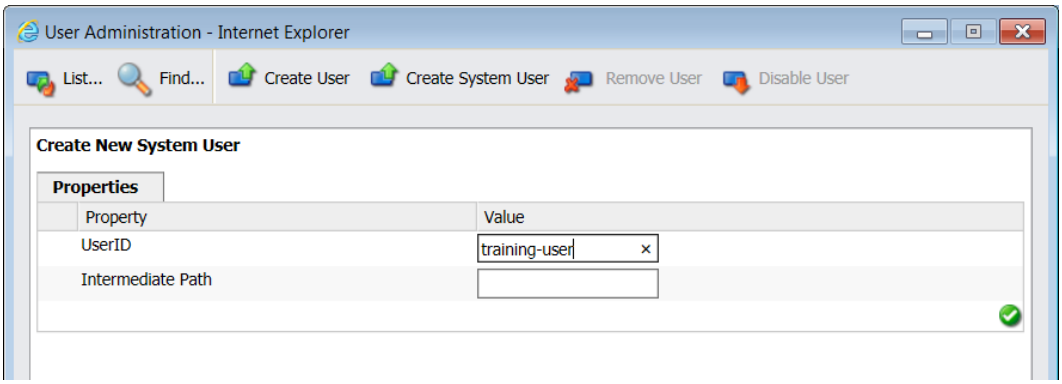
Click User Administration.



In the popup, click **Create System User**.



Enter the UserID: **training-user**



Click the green checkmark on the lower right corner of the dialog box.

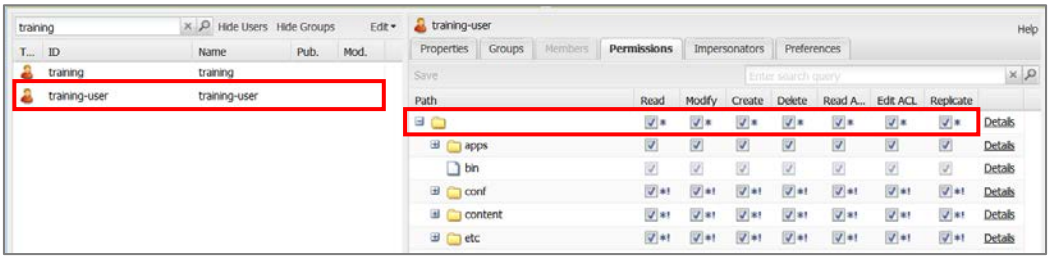
Click **Close**.

Navigate to http://localhost:4502/useradmin

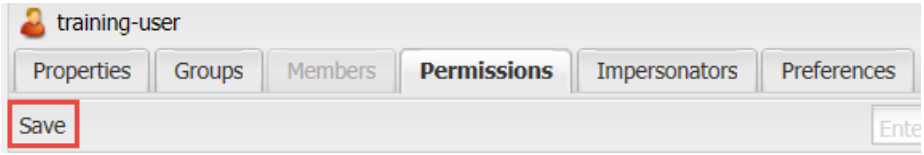
Search for "training" in the search box. You will find the training-user.

Double-click **training-user**. In the right panel, click the **Permissions** tab.

For the root folder, select all: **Read**, **Modify**, **Create**, **Delete**, **Read ACL**, **Edit ACL**, and **Replicate**.



Click the **Save** button:

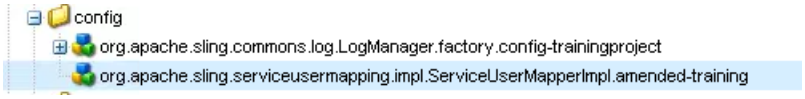


NOTE: For training purposes, we've given full rights access to the training-user. Typically, when creating a service user, it should only have permissions associated with the tasks it will be executing.

Configure the Service User Mapping service:

Navigate to **CRXDE Lite:** http://localhost:4502/crx/de

Under /apps/trainingproject/config, create a node named: org.apache.sling.serviceusermapping.impl.ServiceUserMapperImpl.amended-training, of type sling:OsgiConfig:



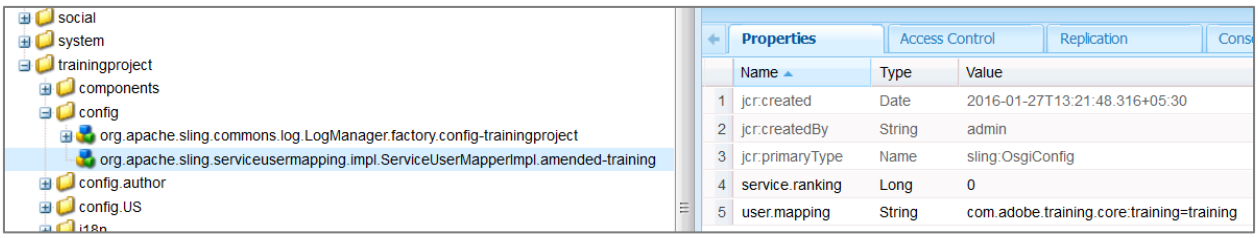
Add the following two properties to the node:

a. **Name**=service.ranking, **type**=Long, **value**=0

b. **Name**=user.mapping, **type**=String, **value**=com.adobe.training.core:training=training-user

The user.mapping allows us to map a bundle with a subservice name to a service user as shown in the screenshot.

Bundle: **com.adobe.training.core**
Subservice name: **training**
Service User: **training-user**



Save your changes.

NOTE: If different permissions are needed for different components in a bundle for security reasons, you will need to create new config nodes with new Subservices and different Service Users.

3. Task - Write an event handler

You will demonstrate the implementation of an event handler; you are going to write an audit log for cq:Page replication events, which will listen for ReplicationAction.EVENT_TOPIC events and log the page's title.

Here are the steps to configure the event handler:

Create an event listener that listens to page replications and starts the job. Navigate to **training.core** > **src/main/java** > **com.adobe.training.core.listeners, and** create a new class named **ReplicationListener**.

Add the following code to the newly created class from the USB code.

```
package com.adobe.training.core.listeners;

import java.util.HashMap;

import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Property;
import org.apache.felix.scr.annotations.Reference;
import org.apache.felix.scr.annotations.Service;

import org.osgi.service.event.Event;
import org.osgi.service.event.EventHandler;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import com.day.cq.replication.ReplicationAction;
import com.day.cq.replication.ReplicationActionType;

import org.apache.sling.event.jobs.JobManager;

@Component(immediate = true)
@Service(value = EventHandler.class)
@Property(name = "event.topics", value = ReplicationAction.EVENT_TOPIC)

public class ReplicationListener implements EventHandler {
        private static final Logger LOGGER = LoggerFactory.getLogger(ReplicationListener.class);

        private static final String TOPIC = "com/adobe/training/core/replicationjob";

        @Reference
        private JobManager jobManager;

        @Override
        public void handleEvent(final Event event) {

                ReplicationAction action = ReplicationAction.fromEvent(event);

                if (action.getType().equals(ReplicationActionType.ACTIVATE)) {

                        if (action.getPath() != null)
                        {
                                try {
                                        // Create a properties map that contains things we want to
pass through the job
                                        HashMap<String, Object> jobprops = new HashMap<String,
Object>();
                                        jobprops.put("PAGE_PATH", action.getPath());

                                        // Add the job
                                        jobManager.addJob(TOPIC, jobprops);

                                } catch (Exception e) {
                                        LOGGER.error("******************************** ERROR CREATING
JOB : NO PAYLOAD WAS DEFINED");
                                        e.printStackTrace();
                                }
                        }
                }
        }
}
```

This service implements the EventHandler interface that will listen to replication actions and start a job with the method addJob(topic, properties) from the org.apache.sling.event.jobs.JobManager interface.
When creating a job, two arguments can be handled:

- A topic (a String that defines our job). This parameter is required. Convention is to use a name based on the package symbolic name with "/" separators between names.
- A property map (optional) to pass serializable properties to the job consuming process.

Create a job consumer service to execute the job

In Eclipse, navigate to **training.core** > **src/main/java** and right click on **com.adobe.training.core** and select **New** > **Class**. Create a class named **ReplicationLogger.**

Paste the following code (from the USB):

```
package com.adobe.training.core;

import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Property;
import org.apache.felix.scr.annotations.Reference;
import org.apache.felix.scr.annotations.Service;
import org.apache.sling.api.resource.LoginException;
import org.apache.sling.api.resource.ResourceResolver;
import org.apache.sling.api.resource.ResourceResolverFactory;
import org.apache.sling.event.jobs.Job;

import org.apache.sling.event.jobs.consumer.JobConsumer;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import com.day.cq.wcm.api.Page;
import com.day.cq.wcm.api.PageManager;

import java.util.HashMap;
import java.util.Map;

@Component(immediate = true)
@Service(value={JobConsumer.class})
@Property(name=JobConsumer.PROPERTY_TOPICS, value =
"com/adobe/training/core/replicationjob")

public class ReplicationLogger implements JobConsumer {
```

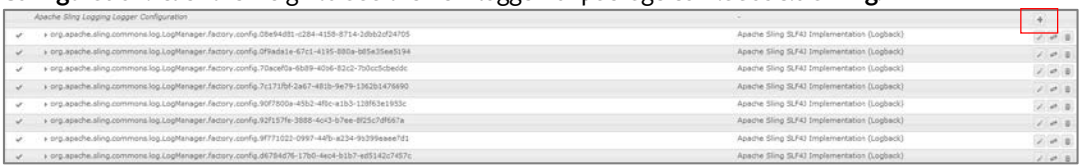
```
      private static final Logger LOGGER =
LoggerFactory.getLogger(ReplicationLogger.class);

      public static final String TOPIC =
"com/adobe/training/core/replicationjob";


    static final String myTopic = JobConsumer.PROPERTY_TOPICS;


      @Reference

      private ResourceResolverFactory resourceResolverFactory;



      @Override

      public JobResult process(final Job job) {


            final String pagePath =
job.getProperty("PAGE_PATH").toString();


            ResourceResolver resourceResolver = null;

            try {



      Map<String, Object> serviceParams = new HashMap<String, Object>();

        serviceParams.put(ResourceResolverFactory.SUBSERVICE, "training");

        resourceResolver =
resourceResolverFactory.getServiceResourceResolver(serviceParams);



            } catch (LoginException e) {

                  e.printStackTrace();

            }


            // Create a Page object to log its title

            final PageManager pm =
resourceResolver.adaptTo(PageManager.class);

            final Page page = pm.getContainingPage(pagePath);
```

```
            if(page != null) {
                  LOGGER.info("****************** ACTIVATION OF PAGE :
{}", page.getTitle());
            }



            return JobConsumer.JobResult.OK;
      }
}
```

Save all your files, and perform **Run As** > **Maven install** on **training.core**.

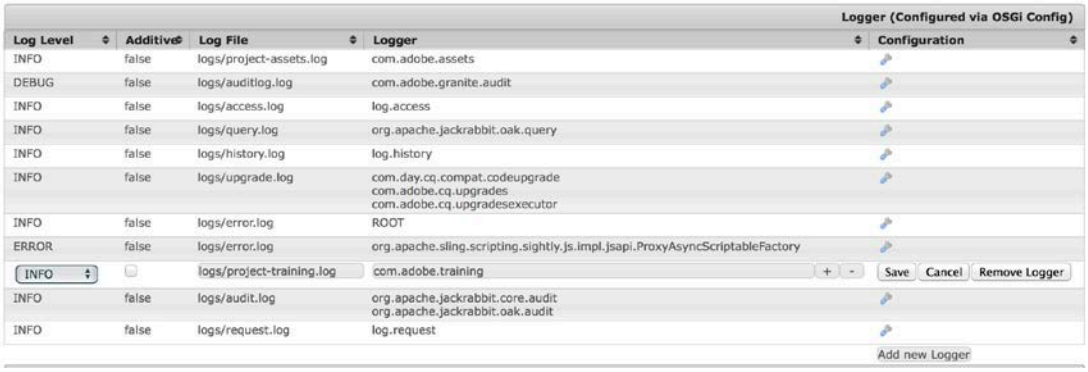
You have created the class file, which will handle the **Activation** Event, but Adobe Experience Manager doesn't know where to write this event debug messages. You have to configure the logger in Adobe Experience Manager so that **com.adobe.training** package is allowed to write the logs.

Open http://localhost:4502/system/console/configMgr and search for **Apache Sling Logging Logger Configuration.** Click the + sign to add the new logger for package **com.adobe.training**.



Navigate to http://localhost:4502/system/console/slinglog. Here you can see a list the loggers that are currently configured in AEM. Currently the Eclipse project built by the Maven archetype has a logger created already: **project-training.log**. The full **com.adobe.training** package can currently log to this file, however we will change this behavior so that the package will instead log to the **error.log** file. Select the wrench icon to the right of this logger configuration.

Select Remove Logger to delete this configuration.



Select the following values for the fields in the prompt window:

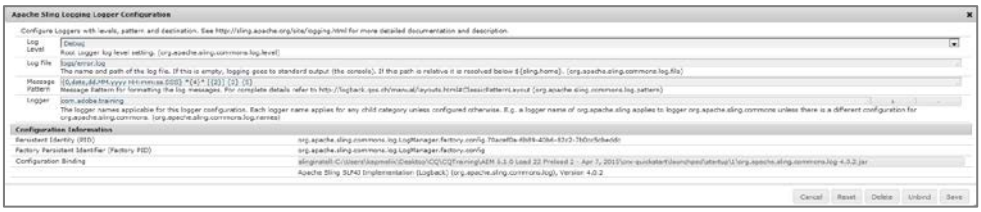
a. **Log Level**: Debug

b. **Logger**: com.adobe.training

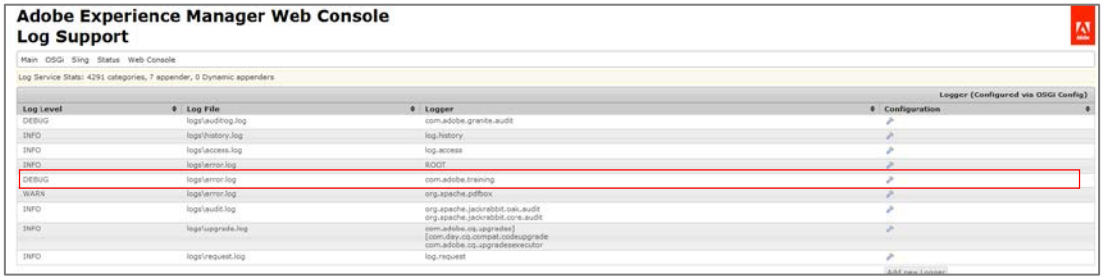
Do not change (but verify) the following values as these value will come by default after clicking on the "**+**" sign:

c. **Log File**: logs/error.log

d. **Message Pattern**: {0,date,dd.MM.yyyy HH:mm:ss.SSS} *{4}* [{2}] {3} {5}



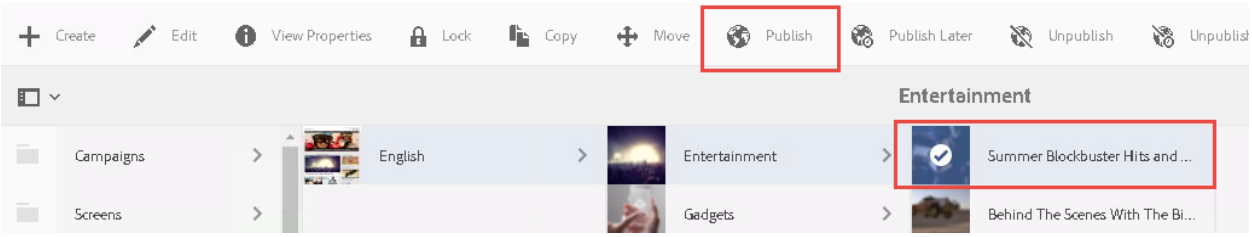
Click **Save** and open http://localhost:4502/system/console/slinglog to see that the logger is created successfully.
NOTE: You can make changes to the config here as well by clicking on the wrench icon to the right.



Go back to Eclipse, and deploy the project by selecting the **training.core** project, and click **Run As** > **Maven install.**

Notice the **ReplicationAction.EVENT_TOPIC** property, which is being listened for, and the **@Component(immediate = true)** statement, which is needed to ensure that the component **com.adobe.training.core.ReplicationLogger** is active immediately. You can verify that the component is available in Adobe Experience Manager at: http://localhost:4502/system/console/components; search the component by its name:

NOTE: To activate (publish) a page, select the page, and select **Publish**.



Activate a page in the Site Admin, and inspect the **error.log** file. You should see logged messages giving the titles of pages as they are activated.

```
30.03.2016 11:32:49.359 *DEBUG* [pool-6-thread-17] com.adobe.training.core.listeners.SimpleResourceListener Reso
org/apache/sling/api/resource/Resource/CHANGED at:
/var/eventing/jobs/assigned/41a31a46-ea64-49f3-b154-abe29b047904/com.adobe.training.core.replicationjob/2016/3/3
a64-49f3-b154-abe29b047904_84
30.03.2016 11:32:49.451 *DEBUG* [0:0:0:0:0:0:0:1 [1459317769450] GET /content/trainingproject/en.pages.json HTTP
com.adobe.training.core.filters.LoggingFilter request for /content/trainingproject/en, with selector pages
30.03.2016 11:32:49.534 *INFO* [pool-12-thread-14-<main queue>(com/adobe/training/core/replicationjob)]
com.adobe.training.core.ReplicationLogger ********************* ACTIVATION OF PAGE : TestPage
30.03.2016 11:32:49.581 *DEBUG* [pool-6-thread-8] com.adobe.training.core.listeners.SimpleResourceListener Resou
org/apache/sling/api/resource/Resource/REMOVED at:
/var/eventing/jobs/assigned/41a31a46-ea64-49f3-b154-abe29b047904/com.adobe.training.core.replicationjob/2016/3/3
a64-49f3-b154-abe29b047904_84
30.03.2016 11:32:50.000 *INFO* [pool-7-thread-3] com.adobe.training.core.CleanupServiceImpl running now
30.03.2016 11:32:52.012 *DEBUG* [0:0:0:0:0:0:0:1 [1459317772012] GET /libs/granite/csrf/token.json HTTP/1.1]
com.adobe.training.core.filters.LoggingFilter request for /libs/granite/csrf/token, with selector null
30.03.2016 11:32:55.224 *INFO* [pool-7-thread-3] com.adobe.training.core.CleanupServiceImpl running now
```

The Sling Jobs page in the Felix console shows statistics about our job, that can be found by topic:

http://localhost:4502/system/console/slingevent

Topic Statistics: com/adobe/training/core/replicationjob	
Last Activated	11:44:19:028 2016-Oct-18
Last Finished	11:44:19:036 2016-Oct-18
Finished Jobs	7
Failed Jobs	0
Cancelled Jobs	0
Processed Jobs	7
Average Processing Time	4 ms
Average Waiting Time	145 ms

4. Task - Schedule a job

In this task, you will write a job that periodically deletes temporary nodes in the repository. The configuration will be stored in a configuration node, which you can see and edit in the Web Console.

Write a new class named CleanupServiceImpl.java under training.core > src/main/java >com.adobe.training.core package.

Write the following code in **CleanupServiceImpl.java**:

```java
package com.adobe.training.core;

import java.util.Dictionary;
import javax.jcr.RepositoryException;
import javax.jcr.Session;
import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Property;
import org.apache.felix.scr.annotations.Reference;
import org.apache.felix.scr.annotations.Service;
import org.apache.sling.commons.osgi.PropertiesUtil;
import org.apache.sling.jcr.api.SlingRepository;
import org.osgi.service.component.ComponentContext;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@SuppressWarnings("deprecation")
@Component(immediate = true, metatype = true, label = "Cleanup Service")
@Service(value = Runnable.class)
@Property(name = "scheduler.expression", value = "*/5 * * * * ?") // Every 5

        // seconds
public class CleanupServiceImpl implements Runnable {
    private static final Logger LOGGER = LoggerFactory.getLogger(CleanupServiceImpl.class);
    @Reference
    private SlingRepository repository;
    @Property(label = "Path", description = "Delete this path", value = "/mypathtraining")
    public static final String CLEANUP_PATH = "cleanupPath";
    private String cleanupPath;

    protected void activate(ComponentContext componentContext) {
        configure(componentContext.getProperties());
    }

    protected void configure(Dictionary<?, ?> properties) {
        this.cleanupPath = PropertiesUtil.toString(properties.get(CLEANUP_PATH), null);
        LOGGER.info("configure: cleanupPath='{}'", this.cleanupPath);
    }

    @Override
    public void run() {
        LOGGER.info("running now");
        Session session = null;
        try {
            session = repository.loginService("training-user", null);
            if (session.itemExists(cleanupPath)) {
                session.removeItem(cleanupPath);
                LOGGER.info("node deleted");
                session.save();
```

```
}
        } catch (RepositoryException e) {
            LOGGER.error("exception during cleanup", e);
        } finally {
            if (session != null) {
                session.logout();
            }
        }
    }
}
```

NOTE:

- metatype = true enables configuration in the Web Console.
- Configure will be called when the configuration changes. This is used to obtain the new value for the cleanup path when it is changed.

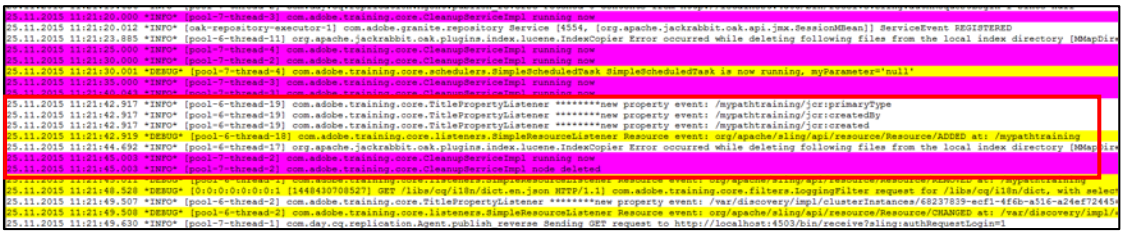
Build the project by selecting the **training.core** project and clicking **Run As** > **Maven install.**

Refresh your browser (Reload the page) where you are working with your Adobe Experience Manager instance (http://localhost:4502).

Inspect the log output (project-trainingproject.log):

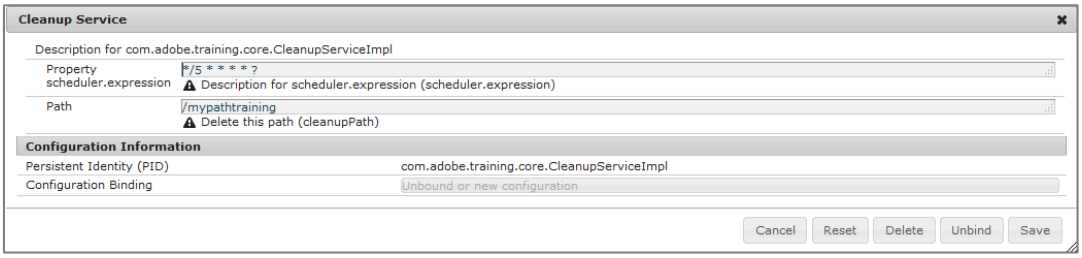
```
25.11.2015 10:54:31.205 *INFO* [qtp1977343400-303] org.apache.sling.auth.core.impl.SlingAuthenticator getA
25.11.2015 10:54:31.209 *INFO* [qtp1977343400-281] org.apache.sling.auth.core.impl.SlingAuthenticator getA
25.11.2015 10:54:35.000 *INFO* [pool-7-thread-1] com.adobe.training.core.CleanupServiceImpl running now
25.11.2015 10:54:40.000 *INFO* [pool-7-thread-2] com.adobe.training.core.CleanupServiceImpl running now
25.11.2015 10:54:45.000 *INFO* [pool-7-thread-2] com.adobe.training.core.CleanupServiceImpl running now
25.11.2015 10:54:49.485 *INFO* [pool-6-thread-13] com.adobe.training.core.TitlePropertyListener ********ne
```

In CRXDE Lite, create a node at the repository root called **/mypathtraining**, save the changes, and then see it being deleted by the cleanup service. This will run every five seconds. You need to refresh the web browser window for CRXDE Lite to see the node disappear. You can verify this in the error.log



Refresh the Web Console Components tab (http://localhost:4502/system/console/components). Click the wrench icon to the right of the **com.adobe.training.core.CleanupServiceImpl** entry to open the Configuration pop-up window.

Configure a different path, for example "**/mynewpathtraining**", and click **Save**..



Now, create a node with the new path name.

Refresh the browser, and confirm that it is deleted.

You can change the configuration settings using the Web Console as you have seen earlier. Another way of changing the settings is through the repository. This allows you to easily configure the settings for different runmodes. These configurations are made by creating sling:OsgiConfig nodes in the repository for the system to reference. These nodes mirror the OSGi configurations and form a UI to them. You can create folders to differentiate various runmodes:

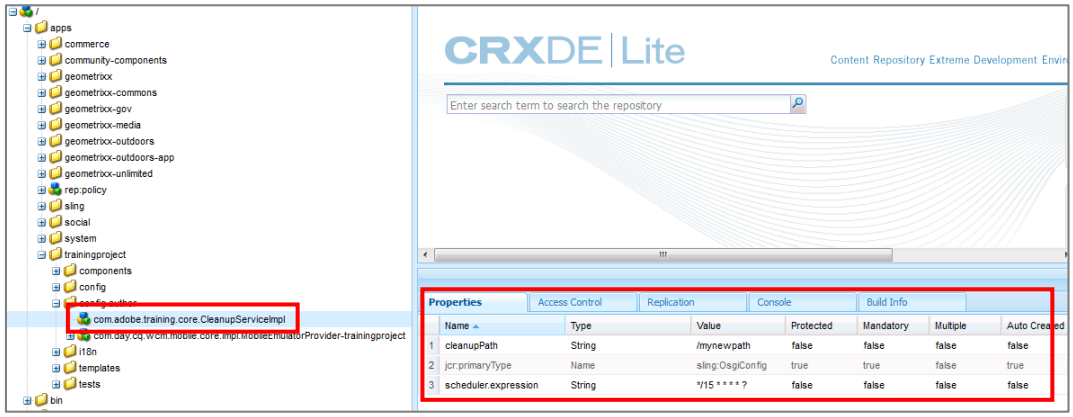
- **config**: For all runmodes
- **config.author**: For the author instance
- **config.publish**: For the publish instance
- **config.<run-mode>**: As appropriate

5. Task - Change job scheduler configuration settings using repository

It is assumed that the project (trainingproject) you have created in Eclipse earlier is synched with the Adobe Experience Manager server.

In the **config.author** folder, create a node of type **sling:OsgiConfig** with the name **com.adobe.training.core.CleanupServiceImpl**, with the following properties:

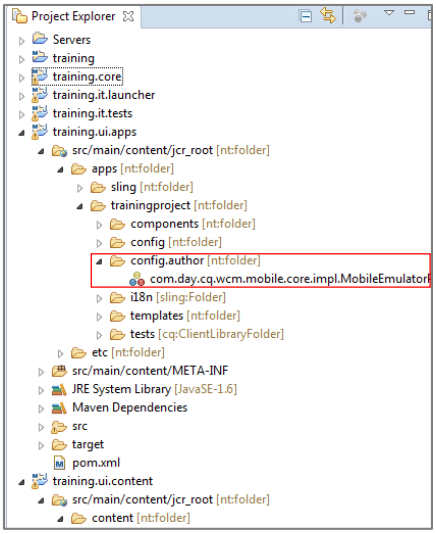
- **Name**: cleanupPath, **Type**: String, **Value**: /mynewpath
- **Name**: scheduler.expression, **Type**: String, **Value**: */5 * * * * ?



Click Save All.

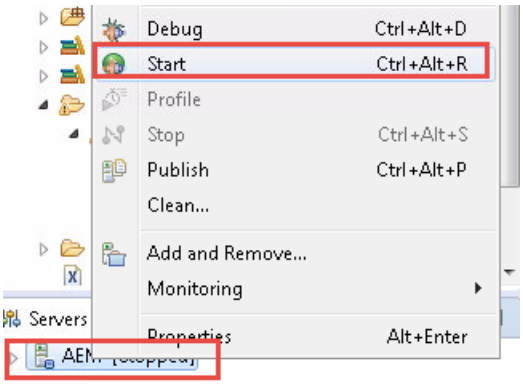
Open web console and confirm the updated configuration on
http://localhost:4502/system/console/components/com.adobe.training.core.CleanupServiceImpl

To sync the **config.author** configuration to your workspace, you need to import from the server. Before checking out, Eclipse looks as follows:

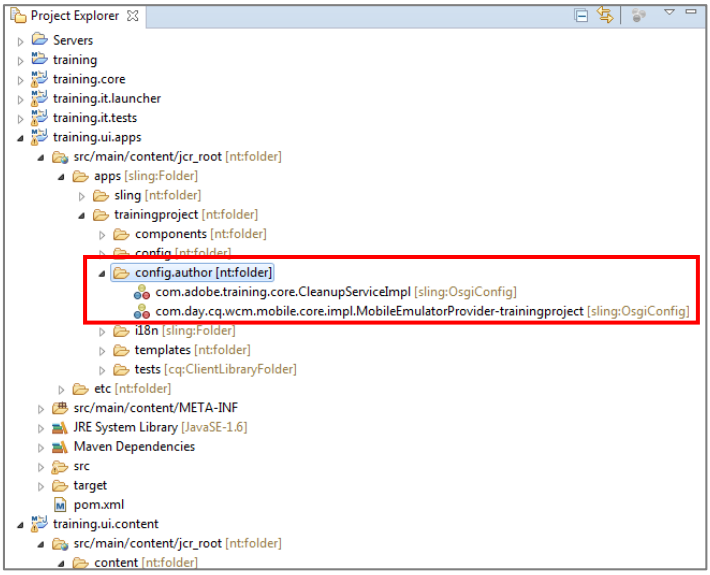


Right-click **training.ui.apps**, select **AEM**, and then select **Import from server...**

NOTE: You may need to ensure your server in Eclipse has started. If it is not, start it:



Click **Finish**. You will see that the project is updated with CRXDE changes done above.



6. Task- Implementing Sling Model

It is assumed the project (trainingproject) you created in Eclipse earlier is synched with the AEM server. In this task, we will create a Sling Model as a Java class and map it into a resource that will consist of a set of properties in a JCR node in the repository. The Sling servlet performs the mapping, which is used to return the values of the properties. In fact, we'll use this class Model as a wrapper to format the class fields, and output them in the browser.

In the sub-package **com.adobe.training.core.models**, add a new class **MyModel.java** with the following code:

```
package com.adobe.training.core.models;
import javax.inject.Inject;
import javax.inject.Named;
import org.apache.sling.api.resource.Resource;
import org.apache.sling.models.annotations.Default;
import org.apache.sling.models.annotations.Model;
@Model(adaptables=Resource.class)
public class MyModel {

    @Inject
    private String firstName;

    @Inject @Named("lastName") @Default(values="No name defined")
    protected String name;

    public String message() {
        return firstName + " " + name;
    }
}
```

Notice the annotation @Model used to make a Sling Resource adaptable to our model, and the annotations @Inject used to inject resource properties into the data members of our class. See also the @Named annotation to inject a property with a different name than the Model field name, and the way we can set a default value with @Default.

In the sub-package **training.core.servlets**, add the class **SlingModelServlet.java** with following code:

```
package com.adobe.training.core.servlets;

import java.io.IOException;

import javax.servlet.ServletException;

import org.apache.felix.scr.annotations.sling.SlingServlet;
import org.apache.sling.api.SlingHttpServletRequest;
import org.apache.sling.api.SlingHttpServletResponse;
import org.apache.sling.api.resource.Resource;
import org.apache.sling.api.resource.ResourceResolver;
import org.apache.sling.api.resource.ValueMap;
import org.apache.sling.api.servlets.SlingAllMethodsServlet;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.adobe.training.core.models.MyModel;


@SlingServlet(resourceTypes = "geometrixx/components/homepage", methods="GET",
    selectors="model")
public class SlingModelServlet extends SlingAllMethodsServlet{

    private static final long serialVersionUID = 1L;
    Logger logger = LoggerFactory.getLogger(this.getClass());


    ResourceResolver resourceResolver;

    public void doGet(SlingHttpServletRequest request, SlingHttpServletResponse
    response)throws ServletException,IOException{
        logger.info("inside sling model test servlet");
        response.setContentType("text/html");

        try {
            // Get the resource (a node in the JCR) using ResourceResolver from the
    request
            resourceResolver = request.getResourceResolver();
```

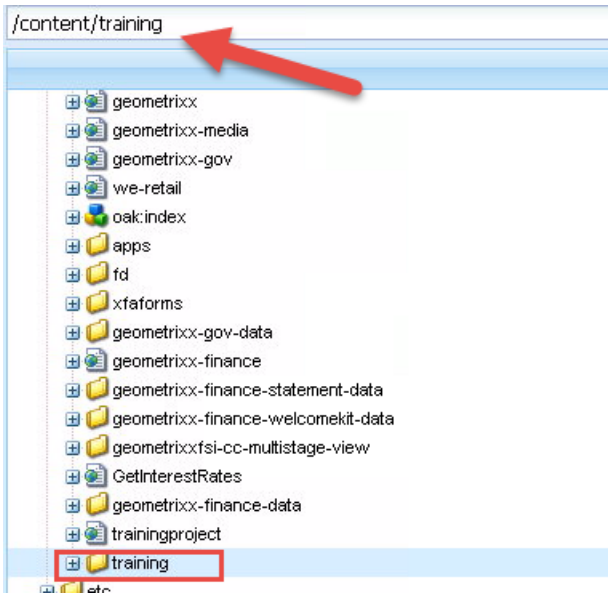
```
            String nodePath = "/content/training/slingmodel";
            Resource resource = resourceResolver.getResource(nodePath);

            // Adapt resource properties to variables using ValueMap, and log their values
            ValueMap valueMap=resource.adaptTo(ValueMap.class);
            logger.info("Output from ValueMap is : "+valueMap.get("firstName").toString() + " " +
    valueMap.get("lastName").toString());

            //Adapt the resource to our model
            MyModel mymodel = resource.adaptTo(MyModel.class);
            logger.info("Output message from MyModel is : "+mymodel.message());

            // Print the response in the browser
            response.getWriter().print("My name is : " + mymodel.message());

        } catch (Exception e) {
            logger.error(e.getMessage());
        }

    }
}
```

This class extends the Java class named org.apache.sling.api.servlets.SlingAllMethodsServlet, to define an AEM Sling Servlet. We'll use this servlet to inject our resource properties into our model.

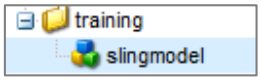
In this servlet, we get the node resource using the ResourceResolver.getResource() method. ResourceResolver is created from the request with the getResourceResolver() method. You can see that the values of the node properties are adapted to a ValueMap to be manipulated within the servlet. This step is optional; it only demonstrates how we can extract each injected field in the servlet. The resource is adapted to our model with the resource.adaptTo(MyModel.class) call.

Build the bundle (Run the training.core – Run > Maven install command).

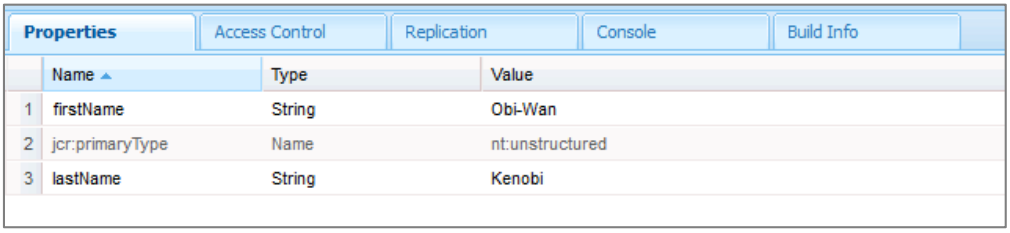
With CRXDE lite, create a node structure that matches the path defined by the nodePath property. First, create a node below **/content** with the name **training** and the type **sling:Folder:**



Next, create a child node under **training** named **slingmodel** of type **nt:unstructured**.



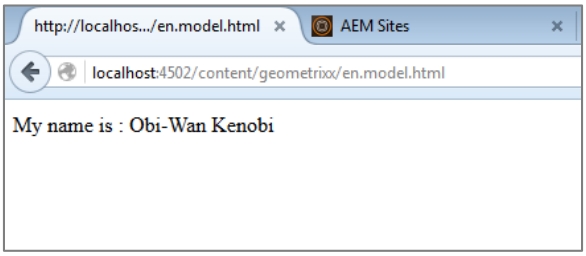
In the newly created node, create two properties of type String: one named **firstName** and the second **lastName**. For the values, use your own name.



Click Save All.

Finally, open an Adobe Experience Manager page in your browser whose template has the **sling:resourceType** referenced by our servlet (for example, the Geometrixx English homepage), and insert a selector in it named model to execute the servlet: http://localhost:4502/content/geometrixx/en.model.html

A message, returned by the model, is displayed in the browser.



Congratulations! You've adapted a simple resource to use in your code, but you can imagine adapting many kinds of things. It's up to you now!

Scenario Conclusion

You used best practices and successfully created the following:

A servlet that displays the title and the JCR properties of the page.

An event handler that monitors changes to the jcr:title property.

A job scheduler that monitors the repository for unwanted nodes.

A class that monitors and deletes a node structure.

A class using Sling Models that uses the injected values from another resource.

CHAPTER SIX: CONFIGURING RUNMODES AND CONFIG NODES

Overview

This chapter explains the various types of run modes available with Adobe Experience Manager, as well as provides detailed descriptions on their configurations. The module also covers instructions on how to create configuration nodes.

Objectives

By the end of this chapter, you will be able to:

- create different run modes.
- customize run modes.
- create config nodes in repository.
- explain how config nodes are resolved.

Using Custom Run Modes

You can run your Adobe Experience Manager instance for specific purposes by using specific run modes. For example, author, publish, test, development, and so on. For run modes, you can:

- define collections of configuration parameters for each run mode. A basic set of configuration parameters is applied to all run modes; you can then configure additional sets to meet your specific environment. These are applied as required.
- define additional bundles to be installed for a specific mode.

Adobe Experience Manager has built-in author and publish run modes (do not remove these). If required, you can add additional custom run modes. For example, you can configure run modes for:

- **Environment:** local, development, test, and production
- **Location:** Berlin, Basel, Timbuktu
- **Company:** acme, partner, customer
- **Special system type:** importer

While run modes can be changed after installation, several specific run modes, called installation run modes, cannot be altered once an Adobe Experience Manager instance is installed. These include author / publish and samplecontent / nosamplecontent. These two pairs of run modes are mutually exclusive, i.e., AEM can be defined as author or publish, but not both. Author can be combined with samplecontent and nosamplecontent, but not both at the same time.

Customized run modes can differentiate instances by purpose, stage of development, or location. Some examples of complex run modes (based on different locations and facilities) are:

- author, development
- publish, test
- author, intranet, us

All settings and definitions are stored in the repository, and activated by setting the appropriate run mode.

Setting Run Modes

It is possible to define specific run mode(s) that a specific instance should run on. By default, an Author instance runs on run-mode Author and a Publish instance runs on run-mode Publish. It is possible to define several run modes for one instance; for example, author, foo, and dev. These run-modes have to be set as VM options. For example, from the command line:

```
java -jar cq-quickstart.jar -r author,foo,dev
```

Alternatively, in the start script:

```
::* runmode(s)
set CQ_RUMMODE=author,foo,dev
```

Or, by adding entries to the crx-quickstart/conf/sling.properties file. For example:

```
sling.run.modes=author,dev,berlin
```

Configurations per Run Mode

To create separate configuration settings per run mode, create folder names in the form: `config.<runmode>` they are used. For example:`"config.publish": /apps/myapp/config.publish`

For systems with run-modes publish and berlin: `/apps/myapp/config.publish.berlin`

Configurations for Different Run Modes

Some examples of configuration settings that may be needed for different run modes:

Different mail server configurations per location:

```
config.basel/com.day.cq.mailer.DefaultMailService
config.berlin/com.day.cq.mailer.DefaultMailService
```

Enabling or disabling debugging per environment:

```
config.prod/com.day.cq.wcm.core.impl.WCMDebugFilter
config.dev/com.day.cq.wcm.core.impl.WCMDebugFilter
```

Additional Information on Run Modes

When using different configurations for separate run modes, the following apply:

- Partial configurations are not supported.
- The configuration with maximum matching run modes wins.

To avoid unexpected results:

- Always set all properties to avoid confusion.
- Use a type indicator (for example, {Boolean}, {String}) in every property.

Using Custom Run Modes with Configurations

In this section, you will examine how custom run modes affect bundle configuration. In this next exercise, you will cause the training log file to have a different name on the Publish Instance as compared to the author instance.

Perform Task – Creating custom run mode, from the Lab Activity section.

Creating Configuration Nodes

You can use two methods to create configurations:

- Web Console
- Config node in the Repository

The following sections dive into the process of creating configuration nodes in the repository.

Creating Configurations in the Web Console

The Web Console is an out-of-the-box interface for OSGi configuration. Its UI enables you to edit properties by selecting from a predefined list. Any configurations made with this console are applied immediately, requires no restart, and applicable to the current instance, regardless of the current run mode, or any subsequent changes to the run mode.

You can configure all bundles through the Configuration tab, which means that you can use it to configure Adobe Experience Manager system parameters.

Creating Configurations in the Repository

You can define configurations for each run mode by creating specific nodes in the repository. These nodes are of type `sling:OsgiConfig`. Using this method, it is possible to share configuration among several instances.

If you modify the configuration data in the repository, the changes are immediately applied to the relevant OSGi configuration as if the changes were made using the Web console, with the appropriate validation and consistency checks. This also applies to the action of copying a configuration from `/libs` to `/apps`.

You should consider these factors:

- **Persistent Identity** (PID) of the service
- **Run modes**—Check whether a specific run mode is required. If so, create the folder specific to that run mode. For example, `config` for all run modes, `config.author` for the author environment, `config.publish` for the publish environment.
- Requirements of **configurations** or **factory configurations**
- **Parameters**—the individual parameters to be configured; including any existing parameter definitions that will need to be recreated.
- **Existing configurations**—customize existing configurations by copying the required configurations from the `/libs` folder to the `/apps` folder, and then modifying it in `/apps`. You can search for the required configurations using the Query tool available in CRXDE Lite.

1.1.1 Configuration Persistence

It is important to note a few details on how changes to the configurations are persisted.

- By default, you can find any change made to a configuration in `/apps/system/config`.
- Settings that are changed by `admin` are saved in `*.config` files under `/crx-quickstart/launchpad/config`. You must never edit the folders or files under this folder.

- It is a good practice to keep a backup of the following configurations:
    - Apache Felix OSGi Management Console
    - CRX Sling Client Repository

1.1.2 Packaging Best Practices

After you create application packages, you need a process for deploying these to other environments. On a basic level, this can be moving the package zip file to a new environment, uploading it to the repository, and importing its contents. Alternatively, you can activate the package using the tools section of the Admin Console. However, you should give some consideration to managing this process to ensure your applications can be deployed efficiently and without any problems.

The following points are what you need to consider while creating packages:

- Verify your application code is separate from the configuration because you may need to use different configurations with the same application for different environments. Creating separate packages makes it easy to deploy the appropriate configuration for each environment.
- Ensure your application code does not depend on specific content because this may not exist in all environments.
- Code packages will be created in the development environment and will then travel from development to test to production. Content will generally travel from production to test to development to be used as test content so separate packages and processes may be needed.

1.1.3 Adding a New Configuration to the Repository

While creating config nodes, you must ensure that their names are equal to the Persistent Identity (PID) of the configuration. For example, you can navigate to http://localhost:4502/system/console/configMgr, and see these names listed as service.pid properties. These configuration nodes have to be child-nodes of node type `sling:folder` with a name starting with `config` followed with a dot. All the run-modes that the config applies to are also separated with a dot.

Examples: `config.author`, `config.publish`, config.author.dev, config.author.foo.dev, and so on.

To create a config node, you need to know the following:

- The Persistent Identity (PID) of the service. You can do this through the **Configurations** field in the Web console. The name is shown in brackets after the bundle name (or in the **Configuration Information** towards the bottom of the page).
- Whether a specific run mode is required. Create the folder:
    - Config—for all run modes
    - config.author—for the author environment
    - config.publish—for the publish environment
    - config.<*run-mode*>—as appropriate
- Whether a Configuration or Factory Configuration is necessary.
- The individual parameters to be configured; including any existing parameter definitions that will need to be recreated. Reference the individual parameter field in the Web console. The name is shown in brackets for each parameter.
- Does a configuration already exist in `/libs`? To list all configurations in your instance, use the Query tool in CRXDE Lite to submit the following SQL query:

```
select * from sling:OsgiConfig
```

If so, this configuration can be copied to `/apps/<yourProject>/`, then customized in the new location.

You can create a config node in the repository using CRXDE Lite as follows:

Open AEM then click **Tools** > **CRXDE Lite**. CRXDE Lite opens.

Navigate to apps and create a config folder called: `sling:Folder`. Under this folder, create a node with the following properties:

a. Name: **sling**

b. Type: **sling:OSGiConfig**

Click **OK**.

For each parameter that you want to configure, create a property on this node.

Changes are applied as soon as the node is updated by restarting the service (as with changes made in the Web console).

Perform Task – Creating configuration node, from the Lab Activity section.

Resolution of Config Nodes

The following sections show how to resolve the config nodes.

1.1.4 Resolution Order at Startup

The following order of precedence is used:

Repository nodes under `/apps/*/config`, either with type `sling:OsgiConfig` or property files.

Repository nodes with type `sling:OsgiConfig` under `/libs/*/config`.

Any config files from `<cq-installation-dir>/crx-quickstart/launchpad/config/` on the local file system.

This means that project specific configuration under `/apps` takes precedence over `/libs`.

1.1.5 Resolution Order at Runtime

Configuration changes made while the system is running triggers a reload with the modified configuration. The following order of precedence applies:

Modifications made in the Web Console.

Modifications made in `/apps`.

Modifications made in `/libs`.

1.1.6 Resolution of Multiple Run Modes

For run mode-specific configurations, you can combine multiple run modes. For example, you can create configuration folders in the following style: `/apps/*/config.<runmode1>.<runmode2>/`

Configurations in these folders will be applied if all run modes match a run mode defined at startup. For example, if an instance was started with the run modes `author,dev,emea`, configuration nodes in `/apps/*/config.emea`, `/apps/*/config.author.dev/` and `/apps/*/config.author.emea.dev/` will be applied, while configuration nodes in `/apps/*/config.author.asean/` and `/config/author.dev.emea.noldap/` will not be applied.

If multiple configurations for the same PID are applicable, the configuration with the highest number of matching run modes is applied. For example, if an instance was started with the run modes `author,dev,emea`, and both `/apps/*/config.author/` and `/apps/*/config.emea.author/` define a configuration for `com.day.cq.wcm.core.impl.VersionManagerImpl`, the configuration in `/apps/*/config.emea.author/` will be applied.

Chapter 6 Lab Activity

Scenario

Your company has decided to expand its website coverage to at least three other geographical locations—US, Singapore, and India. For this, you would need to set up additional configurations to run the instance on each of these locations.

Overview

You can start a server with a custom run mode in following ways. For example, with a location based server in the United States (US):

Using Sling Properties file.
Example: sling.run.modes=author, US

Using Java Virtual Machine(JVM) arguments
Example: java -Xmx512m -jar aem-author-4502.jar -Dsling.run.modes=author,US

Using AEM arguments
Example: java -Xmx512m -jar aem-author-4502.jar -r author,US

In the following tasks, you will be creating these run modes using the first method, which is editing the Sling Properties file.

Challenge

Setup a server to run in the United States by using the sling properties file and a configuration node.

Pre-requisites

Existing Adobe Experience Manager Author server running at your machine.

Steps

1. Task - Create custom run mode

Locate the current AEM install folder on your machine and navigate to **AEM** > **Author** > **crx-quickstart** > **conf**

Open **sling.properties** file with a text editor (by right-clicking on the file and selecting **Edit** or **Edit with <Text File Editor>** and search for **sling.run.mode.install.options.**

```
31  sling.installer.dir=${sling.launchpad}/installer
32  org.apache.sling.commons.log.level=INFO
33  ee-1.7=JavaSE-1.7,JavaSE-1.6,J2SE-1.5,J2SE-1.4,J2SE-1.3, J2SE-1.2,JRE-1.1,JRE-1.0,OSGi/Minimum-1.2,OSGi/Minimum-1.1, OSGi/Minimum-1.0
34  sling.ignoreSystemProperties=true
35  granite.product=Adobe Experience Manager
36  gosh.args=--noshutdown --nointeractive
37  org.osgi.framework.executionenvironment=${ee-${java.specification.version}}
38  felix.webconsole.work.context=system
39  eecap-1.7=osgi.ee; osgi.ee\="OSGi/Minimum"; version\:List<Version>\="1.0,1.1,1.2", osgi.ee; osgi.ee\="JavaSE"; version\:List<Version>\="1
40  sling.run.mode.install.options=author,publish|crx3|crx3tar,crx3mongo,crx3rdb,crx3h2,crx3segment|samplecontent,nosamplecontent
41  felix.cm.dir=${sling.launchpad}/config
42  sling.context.default=default
43  sling.bootdelegation.ibm=com.ibm.xml.*
44  sling.fileinstall.writeback=false
45  osgi-core-packages=org.osgi.dto; version\=1.0.0, org.osgi.framework; version\=1.8.0, org.osgi.framework.dto; version\=1.8.0, org.osgi.fra
    version\=1.0.0, org.osgi.framework.hooks.service; version\=1.1.0, org.osgi.framework.hooks.weaving; version\=1.1.0, org.osgi.framework.la
    org.osgi.framework.startlevel; version\=1.0.0, org.osgi.framework.startlevel.dto; version\=1.0.0, org.osgi.framework.wiring; version\=1.1
    org.osgi.resource.dto; version\=1.0.0, org.osgi.service.packageadmin; version\=1.2.0, org.osgi.service.startlevel; version\=1.1.0, org.os
46  felix.service.urlhandlers=true
```

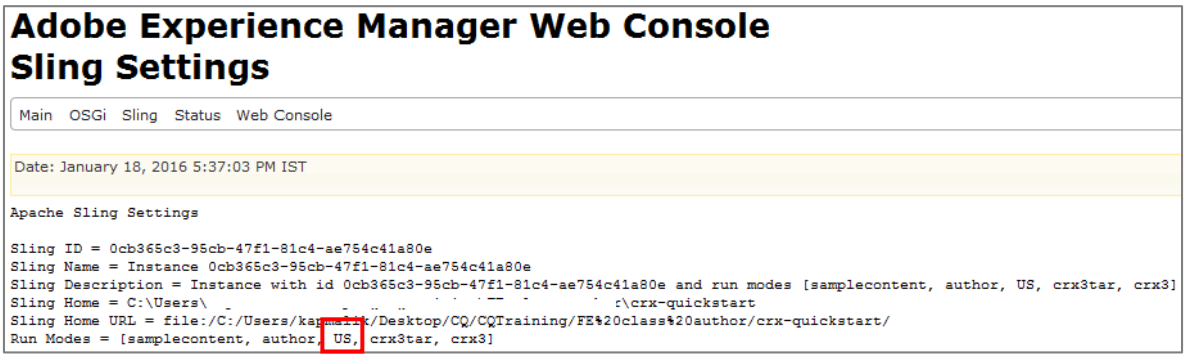
You should see that there are already existing run modes as author, publish|crx3|crx3tar ,crx3mongo, crx3rdb, crx3h2, crx3segment|samplecontent, and nosamplecontent.

Add the following code just below the highlighted line in the existing **sling.properties** file:

```
sling.run.modes=US
```

Save the **sling.properties** file and restart the AEM instance to make the changes effective.

After restarting the instance, open web console and navigate to http://localhost:4502/system/console/status-slingsettings to know the run mode.



Note that if you set the sling.properties file, it will always override any other settings for runmodes, including -r.

2. Task - Create a configuration node

You can start a server with a custom run mode as explained in the previous task. Now you have to create a custom configuration for one of the run modes that you created. When you begin an author instance, it always loads the **projects.html** page by default. Select US run mode and create the configuration to open **en.html** of your training site as the default page.

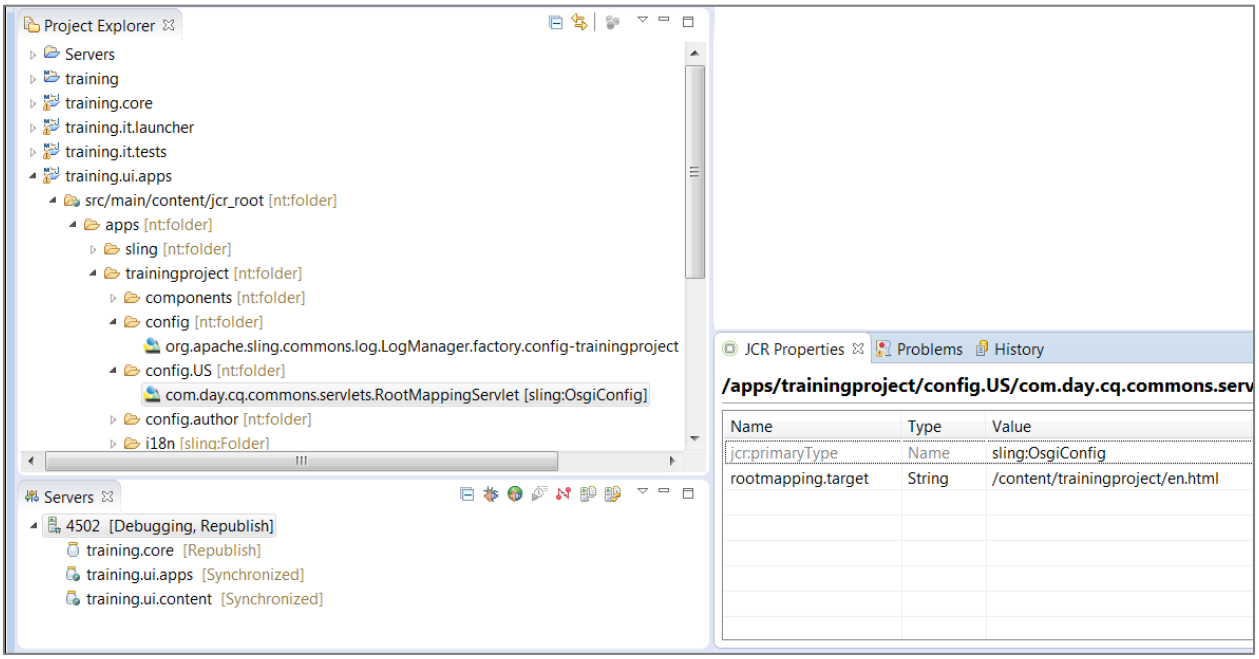
It is assumed that US run mode created in the last task exists in the **crx-quickstart/conf/sling.properties** file.

Open Eclipse and navigate to training.ui.apps > src/main/content/jcr_root > apps > trainingproject and create a folder named config.US.

Select config.US, right-click to select New > Node.. and create a node of type sling:OsgiConfig and name it: com.day.cq.commons.servlets.RootMappingServlet

Click **OK**.

With the com.day.cq.commons.servlets.RootMappingServlet node selected, navigate to the JCR Properties tab and create a property named rootmapping.target, Type=String, and with the value=/content/trainingproject/en.html.



Click **Save All** to save the configuration.

Open **sling.properties** file with a text editor and search for **sling.run.mode.install.options** Verify that the following line was added in previous task:

```
sling.run.modes=US
```

Restart the server using the following command:

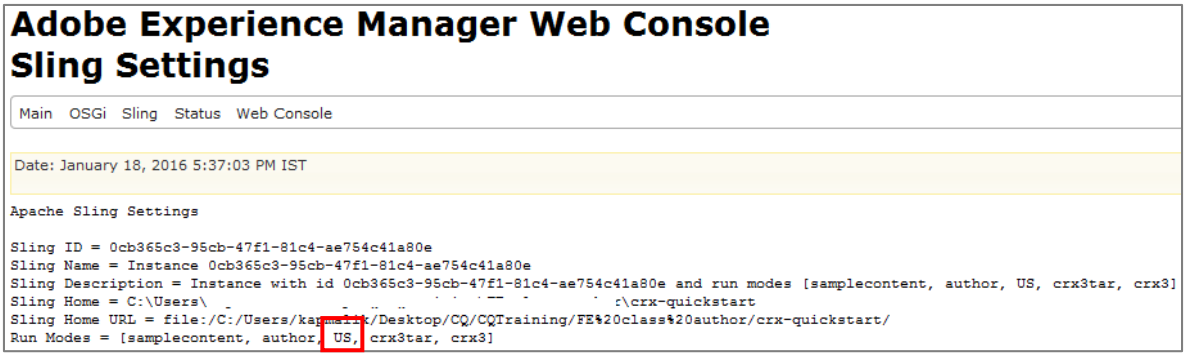
```
java -Xmx512m -jar aem-author-4502.jar -gui
```

Notice the run mode is now set to US.

```
*******************************************************************************
The JVM MBean:PS Perm Gen reports a maximum size of 256 MB, meets our expectation of 256 MB +/- 20
Available memory below specified limits and low-memory action set to fork, will fork to get enough memory
Not forking JVM as -nofork option is set
Setting properties from filename 'C:/Users/mogra/Desktop/AEM Loads/BE Load/aem-author-4502.jar'
Option '-quickstart.server.port' set to '4502' from filename aem-author-4502.jar
Setting 'sling.run.modes' to 'US' from sling.properties.
Ignoring value '' from command line.
Verbose option not active, closing stdin and redirecting stdout and stderr
Redirecting stdout to C:\Users\mogra\Desktop\AEM Loads\BE Load\crx-quickstart\logs\stdout.log
Redirecting stderr to C:\Users\mogra\Desktop\AEM Loads\BE Load\crx-quickstart\logs\stderr.log
Press CTRL-C to shutdown the Quickstart server...
```

The instance starts with **/content/trainingproject/en.html** page.

Open the web console and navigate to http://localhost:4502/system/console/status-slingsettings to check the run mode.



3. Task – Create a configuration package

In this task, you will create a config package, activate it, and check the results. You have already created the config.author folder in the previous task.

In CRXDE Lite, navigate to **apps** > **trainingproject**, and create a new folder named "**config.publish**".

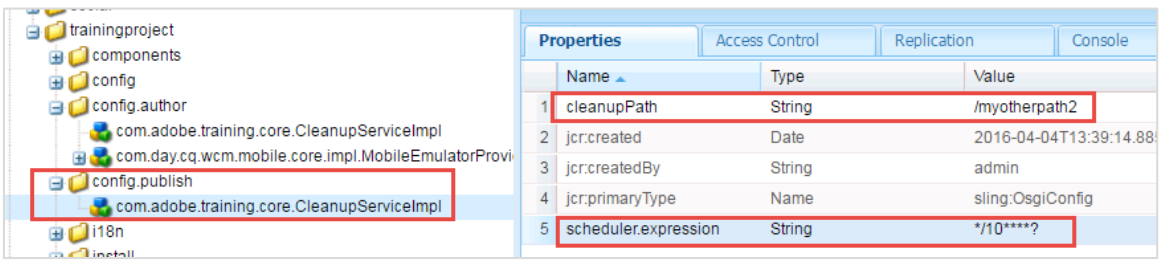
Click **Save All**; otherwise, you won't be able to perform the copy and past operation in the next step.

Copy the config file com.adobe.training.core.CleanupServiceImpl from the config.author, and paste it under config.publish.

Modify the following **values** of the config file:

a. cleanupPath—change Value to **/myotherpath2**

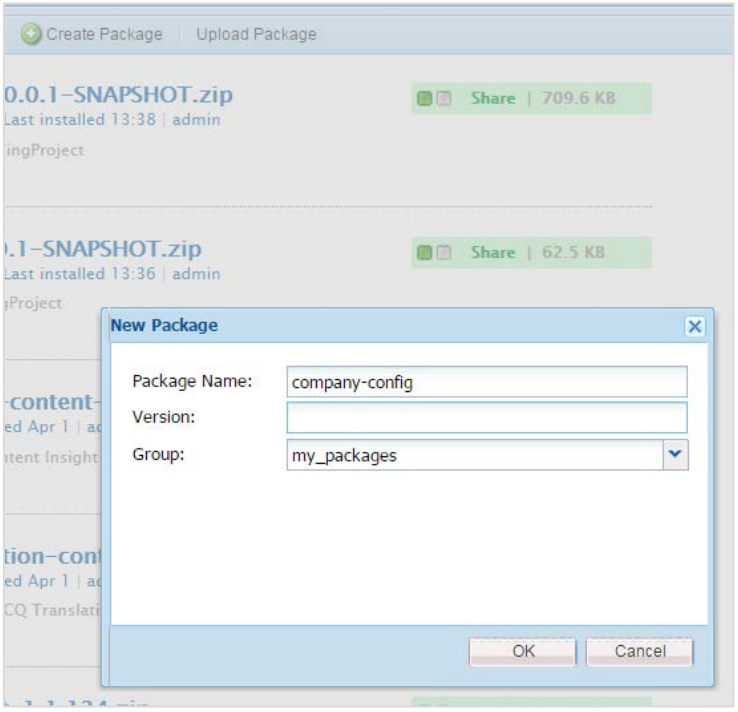
b. scheduler.expression—change to ***/10****?**



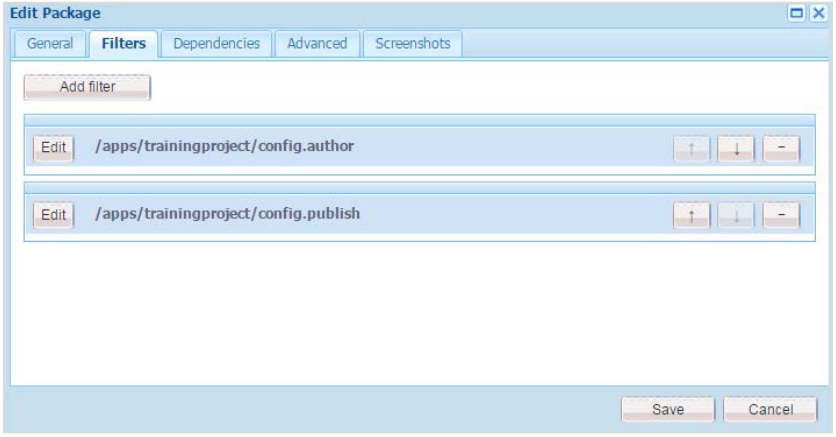
Save your changes.

Create a package containing the config.author and config.publish folders. Go to the Package Manager (http://localhost:4502/crx/packmgr/index.jsp), and click **Create Package**.

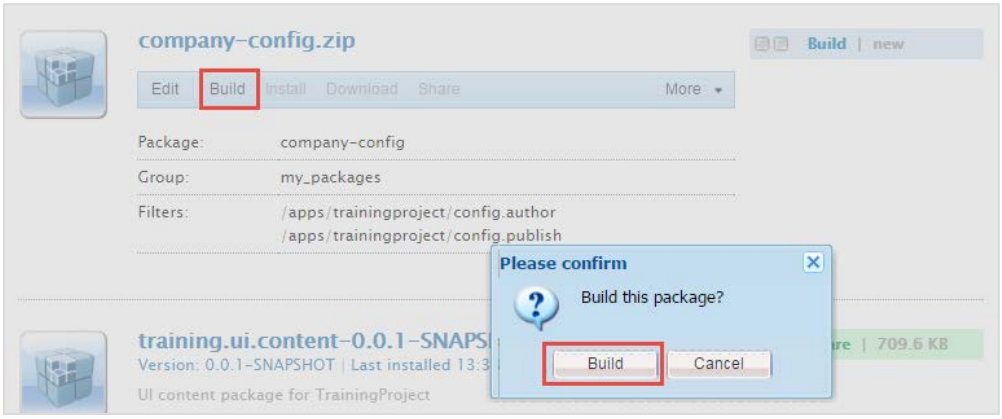
Add the package name **company-config**, group name **my-packages**, and then click **OK**.



Open the package, click Edit, and add the two filters /apps/trainingproject/config.author, and /apps/trainingproject/config.publish. Click Save.



Build the package.

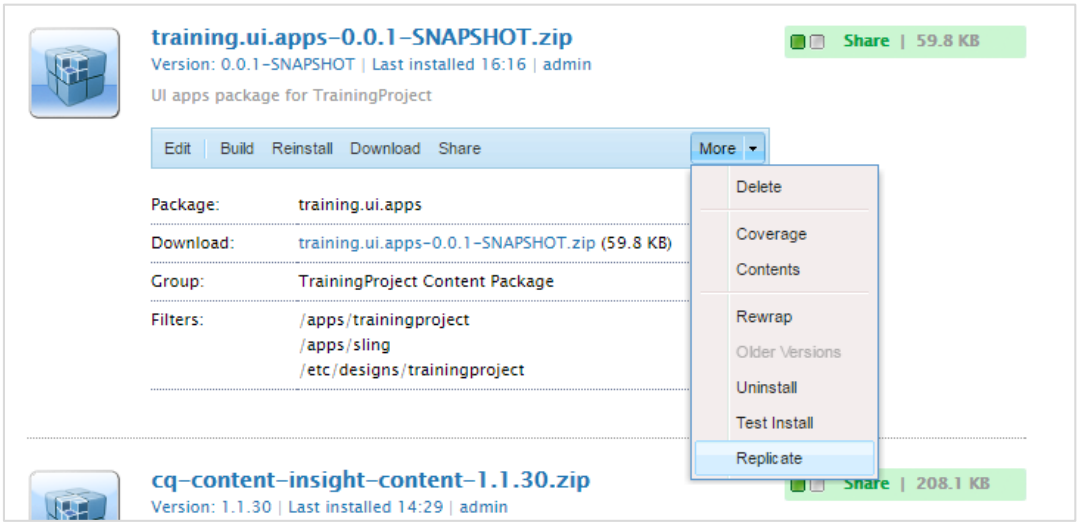


In your local file system workspace, navigate to **\training\ui.apps\src\main\content\jcr_root.**

Execute the command `vlt up` to copy the new configuration information to the file system.

Start the publish server.

In the author instance, you will find a package named **training.ui.apps-0.0.1-SNAPSHOT.zip**. Select this package, click **More**, and then select **Replicate**.



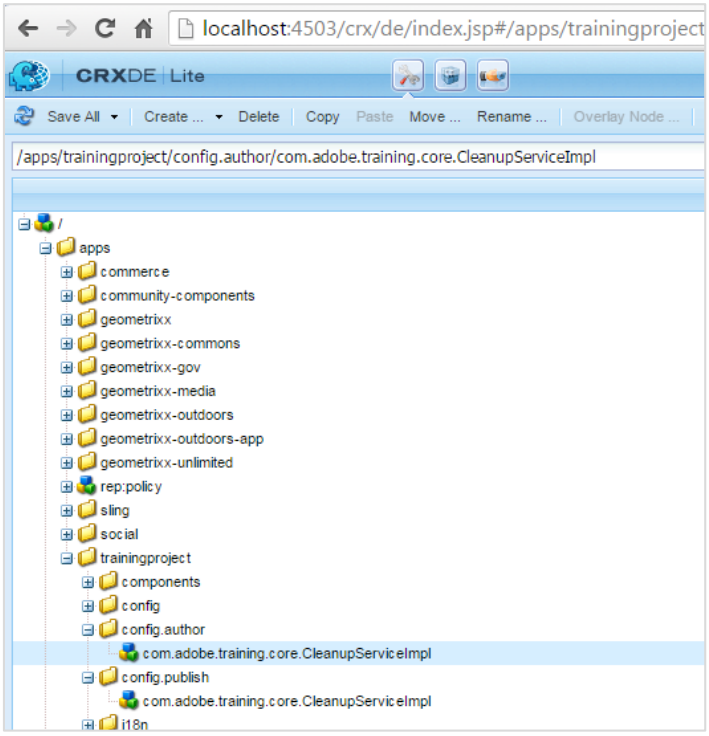
Click **Replicate** in the confirmation window. This will replicate the package to the publish instance.

Next, select the newly created **company-config.zip** package, click **More**, and then select **Replicate**.

Click **Replicate** in the confirmation window. This will replicate the package to the publish instance.

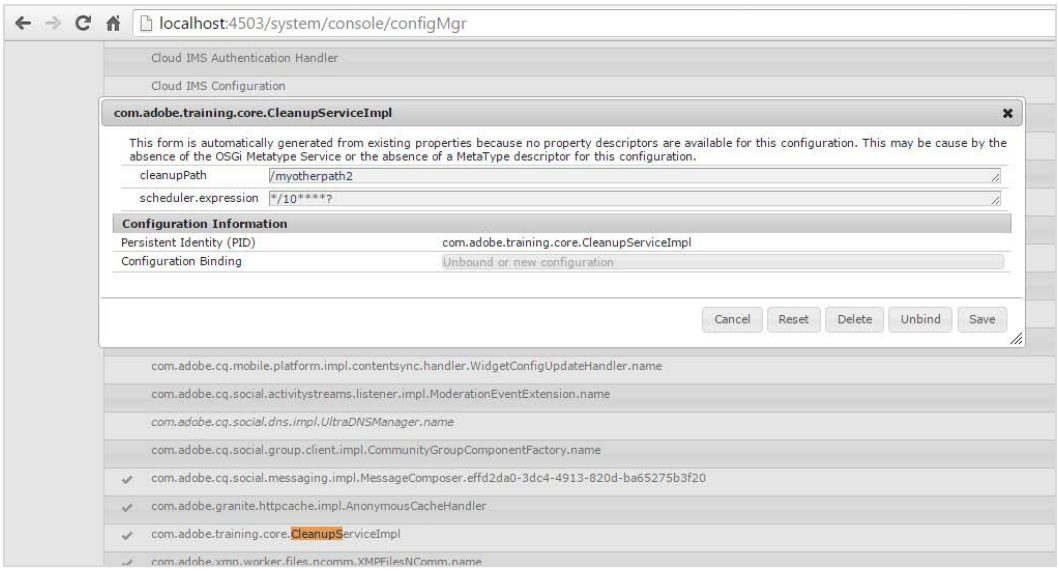
Navigate to http://localhost:4503/crx/de/index.jsp#. This is the CRXDE Lite of the publish instance.

Go to **/apps/trainingproject**, and check that config.author and config.publish are replicated.



Navigate to http://localhost:4503/system/console/configMgr and search for the **CleanupServiceImpl** config.

Click on the **CleanupServiceImpl** config, and you will see the properties and values of the config created under **config.publish**.



Scenario Conclusion

To meet the challenge of configuring the instances to run in three geographical locations, you have set up the custom run modes, and configured them accordingly.

CHAPTER SEVEN: JCR DEEP DIVE

Overview

This chapter deep dives into the Java Content Repository (JCR) model, with emphasis on David's model. David's model is a methodology for modeling data in a content repository. It is based on the principle that developers have been modeling data for a long time, but have not been modeling data in a content repository space for as long. You will learn how data is stored in Adobe Experience Manager, as well as the structure of the JCR.

Objectives

By the end of this chapter, you will be able to:

- describe the JCR Model
- identify best practices for data modeling in the JCR
- describe the JCR Observation

Explaining the JCR Model

The JCR model is represented as a tree of nodes and properties. Nodes are addressed as a path that is similar to the file system and are used to organize the content in hierarchical form; properties store the actual data, either as simple types (such as string, Boolean, etc.) or as binary streams for storing files of arbitrary size.

How is JCR different from RDBMS?

A JCR repository:

- is hierarchical—you can organize your content according to your requirements, with related information stored together. This helps achieve better navigability.
- is flexible—the content can adapt and evolve to be either schema-less or completely restrictive.
- uses a standard Java API.
- abstracts where the information is really stored.
- supports queries and full-text search.

JCR Features

- Query (XPath, SQL, JQL)
- Export/import (XML)
- Referential integrity
- Authentication
- Access control
- Versioning
- Observation
- Locking and transactions (JTA)

JSR-283 Implementation for Adobe Experience Manager

JSR-283 specifies a content repository that aims at providing a unified and widely accepted way to manage data, independent of the hardware or operating system. The JCR describes a content repository that is based on JSR-283. It represents the same technology as JSR-283, but the name is not tied to the Java specification process

The JCR specification defines an abstract model and a Java API for data storage and related services commonly used by content-oriented applications. The repository model enables efficient access to both large binary objects and finely structured hierarchical data through a simple, generic API, and a robust and extensible object typing system.

The Content Repository eXtreme (CRX) is a JCR-compliant repository that allows you to store, manage, and access data using a standardized Java interface. Adobe **CRX** is the commercial content repository component used in Adobe Experience Manager, which uses some elements of Jackrabbit. CRX provides additional features such as development tools & clustering capabilities and has its own storage mechanism, which differs from the Jackrabbit implementation.

Understanding David's Model

David's model has the following set of guidelines on how to model content in a repository.

Data First. Structure Later. Maybe.

Drive the content hierarchy, don't let it happen.

References considered harmful.

Files are Files are Files.

IDs are evil.

Let us look at what each of these mean.

Data First. Structure Later. Maybe

The basic idea of this concept is that while planning your repository, you need not use a structured approach to storing your data. Get used to storing data as nodes and properties, especially nodes of type `nt:unstructured`. Also, structure is expensive and (in many cases) unnecessary to explicitly declare structure to the underlying storage.

For example, if you have a site with a time stamp on it, then this data can be stored in the form of an `nt:unstructured` node as you don't have to do anything with that data. Your app will automatically know how to extract that information and there is no need to declare it explicitly. Applications would implicitly know the structure that you use.

Data constraints such as mandatory or type and value constraints should only be applied where required for data integrity reasons.

Drive the content hierarchy, don't let it happen

The content hierarchy is a valuable asset, which is why you need to have good human-readable names for nodes, and not arbitrary numbers. Don't directly convert your existing relational model into a hierarchical one, plan it accordingly.

The content hierarchy drives:

- ACLs
- URLs
- caching
- search and content organization

For example, if you were to model a blog, then you would keep the blog posts separate from the comments due to access controls applied. That way, you can ensure that anonymous users can only create comments, and not have access to the rest of the workspace except on a read-only basis.

References considered harmful

Reference implies referential integrity. References are costly from:

- the repository point of view because it must manage the references.
- the content point of view because it affects flexibility.

For example, if you have a page (a) that has a reference to another page (b), and if this reference is at the repository level, then you cannot export or import this page individually, as the target reference would be missing. Therefore, it is better to model them as either "weak references," or use a path to refer to another object.

Files are Files are Files

If a content model has something that even remotely resembles a file or a folder, then use the `nt:file`, `nt:folder`, or `nt:resource`.

Many generic applications allow interaction with `nt:folder` and `nt:files` implicitly and know how to handle and display those events if they are enriched with additional meta-information.

- If you need to store the filename and the mime-type, then use: `nt:file/nt:resource`
- If you could have multiple files, then use an `nt:folder`.
- If you need to add meta information to your resource, such as an "author" or "description" property, extend `nt:resource` not the `nt:file`. You would rarely extend `nt:file` and frequently extend `nt:resource`.

IDs are evil

In relational databases, IDs are used to express relationships between various fields (or data) – this is not the case in content modeling. If your model is full of IDs, then it means you are not leveraging the hierarchy properly. The `mix:referenceable` property available in the repository is sufficient for node identification.

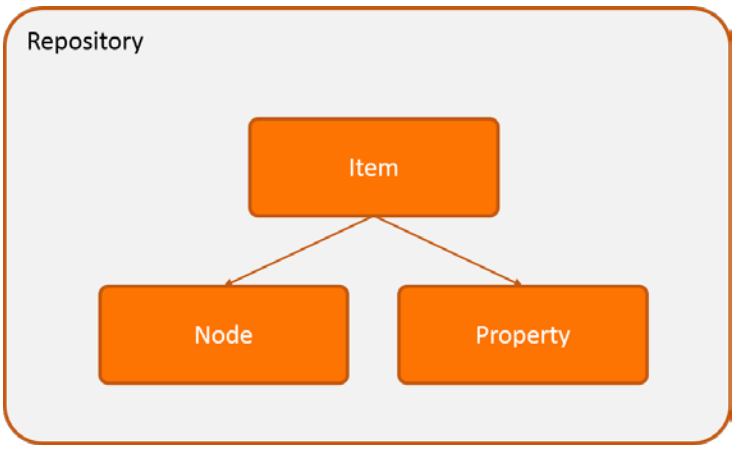
Content Services of JCR

JCR provides the following services, which makes Adobe Experience Manager a robust environment.

- Author-based versioning
- Full-text searching
- Fine-grained access control
- Content categorization
- Content event monitoring

Structure of the JCR

- A JCR repository is composed of a directed acyclic graph of items where the edges represent the parent-child relation.
- An item is either a node or a property. A node can have zero or more child items. A property cannot have child items but can hold zero or more values.
- The nodes form the structure of the stored data while the actual content is stored in the values of the properties.
- An additional advantage of the JCR is the support for namespaces. Namespaces prevent naming collisions among items and node types that come from different sources and application domains. JCR namespaces are defined with a prefix, delimited by a single colon (:) character (for example - `jcr:title`).



The data in a JCR consists of a tree of nodes with associated properties.

Nodes

- Nodes may optionally have one or more types associated with them, which dictate the kinds of properties, number and type of child nodes, and certain behavioral characteristics of the nodes.
- Nodes may point to other nodes via a special reference type property.
- Nodes may also be of the version-able type. This makes the repository track a document's history and store copies of each version of the document.

Properties

- Properties are either single or multi-valued.
- Each value has one of the 12 possible types. These types include familiar data storage types such as strings, numbers, Booleans, binaries and dates, as well as types that hold pointers to other nodes.

Node Types

Node types are used to enforce structural restrictions on the nodes and properties in a workspace by defining for each node its required and permitted child nodes and properties.

- Primary node types are typically used to define the core characteristics of a node (this is indicated by the property `jcr:primaryType`).
- Mixin node types are used to add additional characteristics often related to specific repository functions or to metadata.

Node Type Definitions

- Node types are stored in the form of node type definitions.
- They are reflected in the API as `javax.jcr.nodetype.NodeType`.
- Call `javax.jcr.Node.getPrimaryNodeType()` to get the node type definition of a node.
- A node type definition consists of a set of mandatory attributes.

Node Type Inheritance

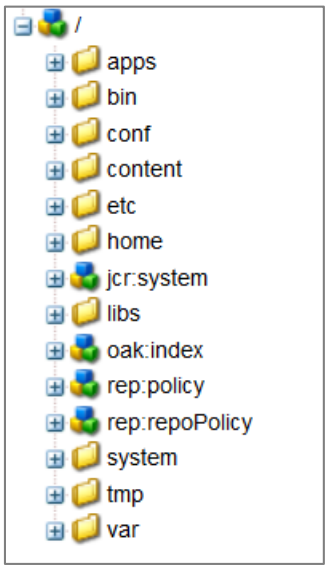
A node may have one or more super types. Seen from the super type, the inheriting node type is a subtype. Super types are discoverable using the JCR API.

A subtype inherits:

- property definitions
- child node definitions
- other attributes, such as 'isMixin'

Storage of Data in Adobe Experience Manager

Adobe Experience Manager has the following folder structure that you can access in CRXDE Lite.



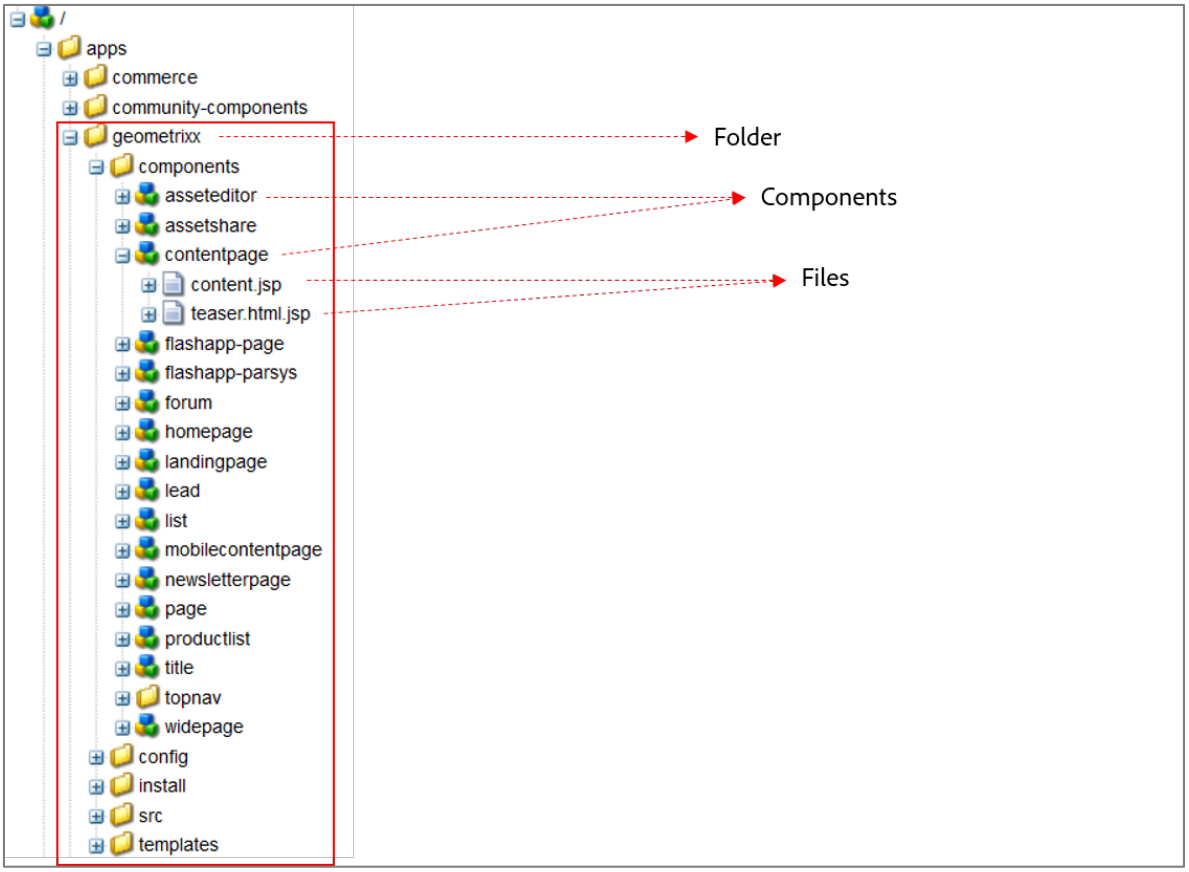
The following sections explain some of the common folders.

/apps

All custom templates, components, and any other definitions related to your site are stored here. When you inherit foundation components, it is done from the libs folder. Always copy the components from libs to apps, and then modify it in apps. By doing this, no code will be lost when Adobe Experience Manager is upgraded. You will only need to make the updates to the apps folder.

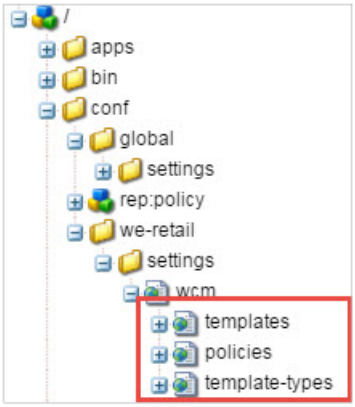
However, a best practice is to use a feature known as Sling Resource Merger, where you need to create the required folder structure (empty folders) under /apps. Modify the nodes or properties wherever required.

Typically, a site is divided into the following folders: components, config, install, src, and templates. However, you can create any folder structure that you need within apps. Consider the following example, the Geometrixx site. You can see the structure created for this site. The components folder has page components, title components, navigation component, and so on. Each of these components would have script files that define the functionality of that component.



/conf

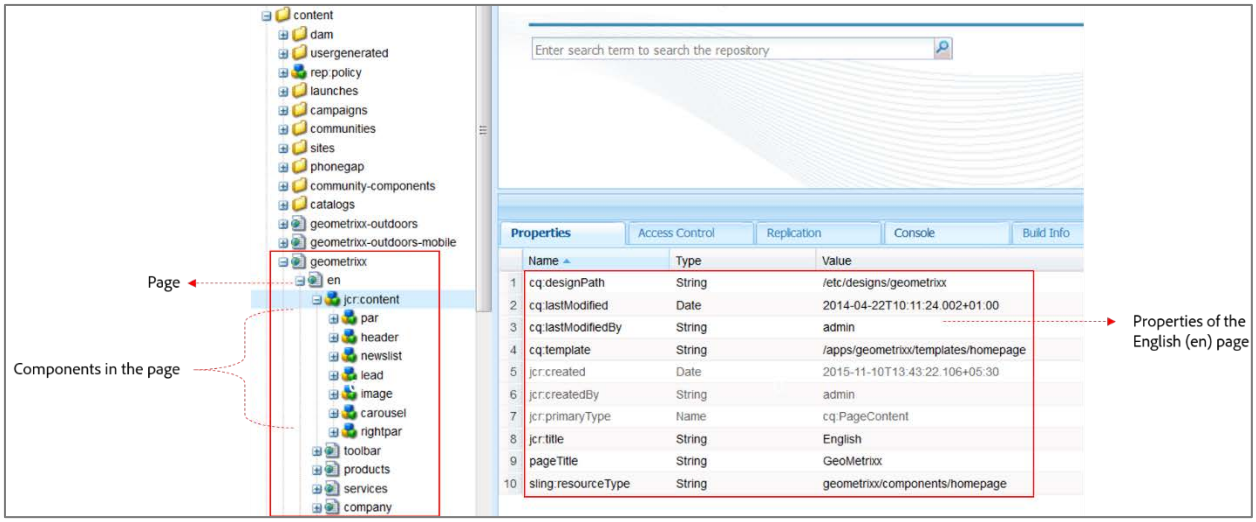
All configuration files appropriate to your site are stored in this folder. More specifically, from 6.2, this folder is used to store the dynamic templates and content policies of your site.



/content

All content of your website is stored in this folder. In the following Geometrixx site example, each page is stored as a node under the `/content` folder. The components/content that an author places within the page is stored under the `jcr:content` node of the specific page. You can define specific properties for each of these nodes in the Properties tab.

The nodes in this folder are automatically created when the page and its content are created.



/libs

All libraries and definitions that belong to Adobe Experience Manager code are stored here. It includes out-of-the-box components, templates, and other Adobe Experience Manager features such as search or replication. It is also referred to as the foundation components. Avoid making modifications to any of the components in libs.

/etc

This folder contains all resources related to utilities and tools. It also includes the designs of your project.

/oak:index

This node contains Jackrabbit Oak index definitions. Each node specifies the details of one index. Standard indexes for the AEM application are visible and additional custom indexes can be created.

/home

This folder contains all the information related to users and groups.

/var

This folder contains files that change and are updated by the system, such as audit logs, statistics, and event handling.

/tmp

This is a temporary working area.

For example, if you were to create a new site, you would store the basic structure, components, scripts, and templates in the `/apps` folder, designs for the site in the `/etc/designs` folder, and contents of the site/page in the `/content` folder. To use existing foundation components, you would refer to the `/libs` folder.

As already discussed, in Adobe Experience Manager, data is stored in the form of nodes and properties. Based on the type of data being stored, the node type may vary. The following are a list of the node types that are most commonly used throughout this course.

Node Type	Description
cq:ClientLibraryFolder	Defines the JavaScript and CSS libraries, and make them available to HTML pages. This node contains one or more source files that, at runtime, are merged into a single JS or CSS file.

cq:Dialog	Indicates a dialog box component for the touch-optimized UI. Alternatively, when you create a dialog box, it is jcr:primaryType is set to cq:Dialog.
cq:EditConfig	Allows the configuration of many important features of the component's editing experience such as dialog box display behavior, listeners, in-place editing, and so on.
cq:InplaceEditingConfig	Defines an in-place editing configuration for the component.
cq:Page	Indicates a page component. Every page created has its jcr:primaryType set to cq:Page.
cq:Template	Indicates a template. Every template created has its jcr:primaryType set to cq:Template.
nt:file	Stores scripts. Alternatively, this node also lets you create a new file, and sets its jcr:primaryType set to nt:file.
nt:folder	Indicates a folder. Alternatively, you can create a new folder, which will set its jcr:primaryType set to nt:folder.
sling:OSGiConfig	This is an OSGi configuration node.
nt:unstructured	• Stores unstructured content. • Allows any number of child nodes or properties with any names. • Allows multiple nodes having the same name as well as both multi-value and single-value properties with any names. • Supports client-orderable child nodes.

Understanding Java Content Repository Observation

An observation is a JCR mechanism that enables code to listen for events generated during repository operations and accordingly performs certain operations. A repository may support an observation, which enables the application to receive notification of persistent changes to a workspace.

There are two types of observations:

- Journaled observation – used to query the repository for a log of past events
- Asynchronous observation – allows applications to react to events in real time

You can check whether any or both of the observations are supported, by querying the repository descriptor table with the following keys:

- Repository.OPTION_OBSERVATION_SUPPORTED
- Repository.OPTION_JOURNALED_OBSERVATION_SUPPORTED

If the queries return a true value, then it is supported.

Event Modelling

A persisted change to a workspace is represented by a set of one or more events. Each event reports a single change to the structure of the persistent workspace in terms of an item added, changed, moved or removed.

The scope of event reporting is implementation-dependent. An implementation should make a best-effort attempt to report all events, but may exclude events if reporting them would be impractical given

implementation or resource limitations. For example, on an import, move, or remove of a subgraph containing a large number of items, an implementation may choose to report only events associated with the root node of the affected graph and not those for every sub-item in the structure.

1.1.7 The Event Object

Each event generated by the repository is represented by an Event object.

1.1.7.1 Types of Events

There are various types of events. The type of event is identified and retrieved through the method `int Event.getType()`.

Types of events are:

- Node added
- Node moved
- Node removed
- Property added
- Property removed
- Property changed
- Persist

1.1.7.2 Event Information

Each event is associated with the following information:

- Event path – retrieved through `String Event.getPath()`
- Identifier – retrieved through `StringEvent.getIdentifier()`
- Information map – retrieved through `java.util.Map Event.getInfo()`

If the event is `NODE_ADDED` or `NODE_REMOVED`:

- `Event.getPath()` returns the absolute path of the node that was added or removed.
- `Event.getIdentifier()` returns the identifier of the node that was added or removed.
- `Event.getInfo()` returns an empty Map object.

If the event is `NODE_MOVED`:

- `Event.getPath()` returns the absolute path of the destination of the move.
- `Event.getIdentifier()` returns the identifier of the node that was moved.
- `Event.getInfo()` returns a Map containing parameters information from the method that caused the event.

If the event is `PROPERTY_ADDED`, `PROPERTY_CHANGED`, or `PROPERTY_REMOVED`:

- `Event.getPath()` returns the absolute path of the property that was added, changed, or removed.
- `Event.getIdentifier()` returns the identifier of the parent node of the property that was added, changed, or removed.

- `Event.getInfo()` returns an empty Map object.

If the event is PERSIST,

- `Event.getPath()` returns null.
- `Event.getIdentifier()` returns null.
- `Event.getInfo()` returns an empty Map.

1.1.7.3 User ID

An event records the identity of the session that caused it.

- `String Event.getUserID()` returns the user ID of the session, which is the same value that is returned by `Session.getUserID()`

1.1.7.4 User Data

An event may contain arbitrary string data specific to the session that caused the event. You can access this information by `String Event.getUserData()`. Information retrieved is used to provide additional context for the event, beyond that provided by the identity of the causing session alone.

1.1.7.5 Event Date

An event records the time of the change that caused it. This is acquired through `long Event.getDate().`

The date is represented as a millisecond value that is an offset from the epoch January 1, 1970 00:00:00.000 GMT (Gregorian). The granularity of the returned value is implementation-dependent.

1.1.8 Bundling of Events

Some repositories support bundling of events. In such a repository, each event bundle corresponds to a single atomic change to a persistence workspace and contains only events caused by that change. By grouping events, additional contextual information is provided, simplifying the interpretation of the event stream.

In both asynchronous and journaled observation, the order of events within a bundle and the order of event bundles is not guaranteed to correspond to the order of the operations that produced them.

Introduction to Asynchronous Observation

An application connects with the asynchronous observation mechanism by registering an event listener with the workspace. An event listener is an application-specific class implementing the `EventListener` interface that responds to the stream of events to which it has been subscribed. In this type of observation, execution continues normally on the thread that performed the operation.

1.1.9 Understanding Asynchronous Observation

The following sections help you understand the functionalities of asynchronous observations.

1.1.9.1 Observation Manager

Registration of event listeners is done through the `ObservationManager` object acquired from the workspace through `ObservationManager Workspace.getObservationManager().`

1.1.9.2 Adding an Event Listener

An event listener is added to a workspace with the following code:

```
void ObservationManager.addEventListener(EventListener listener,
int eventTypes,
String absPath,
boolean isDeep,
String[] uuid,
String[] nodeTypeName,
boolean noLocal)
```

The `EventListener` object passed is provided by the application. As defined by the `EventListener` interface, this class must provide an implementation of the `onEvent` method:

```
void EventListener.onEvent(EventIterator events)
```

When an event occurs that falls within the scope of the listener, the repository calls the `onEvent` method invoking the application-specific logic that processes the event.

1.1.9.3 Event Filtering

In the upcoming JSR-333 specification, event filtering is done through an `EventFilter` object passed to the `ObservationManager` at registration time. The `EventFilterObject` provides the `eventTypes`, `absPath`, `isDeep`, `Identifiers`, `NodeTypes`, and `Nolocal` values as parameters of the class.

In JSR-283, you have to deal 'manually' with those objects and not through the `EventFilter`. A listener receives events based on the following:

- Access Privileges
- Event Types
- Local and Nonlocal
- Node Characteristics

1.1.9.4 Re-registration of Event Listeners

The filters of an already-registered `EventListener` can be changed at runtime by re-registering the same `EventListener` Java object with a new set of filter arguments. The implementation must ensure that no events are lost during the changeover.

1.1.9.5 Event Iterator

In asynchronous observation the `EventIterator` holds an event bundle or a single event, if bundles are not supported. `EventIterator` inherits the methods of `RangeIterator` and adds an Event-specific next method: `Event EventIterator.nextEvent().`

1.1.9.6 Listing Event Listeners

A set of EventListener objects are retrieved using the `EventListenerIterator`.
The `EventListenerIterator` class inherits the methods of `RangeIterator` and adds an `EventListener`-specific next method:

```
EventListener EventListenerIterator.nextEventListener()
```

1.1.9.7 Removing Event Listeners

You can remove the event listener using:

```
void ObservationManager.removeEventListener(EventListener listener)
```

1.1.9.8 User Data

You can set the user data using:

```
void ObservationManager.setUserData(String userData)
```

Introduction to Journaled Observation

Journaled observation allows an application to periodically connect to the repository and receive a report of changes that have occurred since some specified point in the past (for example, since the last connection).

1.1.10 Understanding Journaled Observation

The following sections help you understand the functionalities of journaled observations.

1.1.10.1 Event Journals

Events reported by this `EventJournal` instance are filtered according to the current session's access rights, any additional restrictions specified through implementation-specific configuration and, in the case of the second signature, by the parameters of the method. These parameters are interpreted in the same way as in the method `addEventListener`.

An `EventJournal` is an extension of `EventIterator` that provides the additional method `skipTo(Calendar date).`

1.1.10.2 Journaling Configuration

An implementation is free to limit the scope of journaling both in terms of coverage (that is, which parts of a workspace may be observed and which events are reported) and in terms of time and storage space. For example, a repository can limit the size of a journal log by stopping recording after it has reached a certain size, or by recording only the tail of the log (deleting the earliest event when a new one arrives). Any such mechanisms are assumed to be within the scope of implementation configuration.

1.1.10.3 Bundling of Events

In journaled observation dispatching is done by the implementation writing to the event journal.

If event bundling is supported, a `PERSIST` event is dispatched when a persistent change is made to workspace bracketing the set of events associated with that change. This exposes event bundle boundaries in the event journal.

A `PERSIST` event will never appear within an `EventIterator` since, in asynchronous observation, the iterator itself serves to define the event bundle. In repositories that do not support event bundling, `PERSIST` events do not appear in the event journal.

Perform Task – Create an Observation Listener, from the Lab Activity section.

Chapter 7 Lab Activity

Scenario

In your project, you need to keep a track of some changes/updates made to the repository, without requiring any task to be performed. You need to only display a notification of the change on your page. For example, if there is a change made to the title of the page, then the notification about that change would be visible on the page.

Challenge

You need to identify which properties need to be tracked for changes. When a change is made to the same property by multiple users, it is difficult to track any change beyond the first one.

Overview

Create an observation listener that checks for new or modified property, named jcr:title, of a page. The title of a page is stored in jcr:title under the jcr:content node. You have to write a listener so that a change in jcr:title property can be highlighted with an exclamation mark (!) at the end of its value.

Pre-requisites

You need to have the project—trainingproject, running Author instance, Eclipse.

Steps

1. Task - Create an Observation Listener

   Open Eclipse.

   Under training.core > src/main/java > com.adobe.training.core, create a new class TitlePropertyListener.java, and paste the following code:

```
package com.adobe.training.core;
import javax.jcr.Property;
import javax.jcr.RepositoryException;
import javax.jcr.Session;
import javax.jcr.observation.Event;
import javax.jcr.observation.EventIterator;
import javax.jcr.observation.EventListener;
import javax.jcr.observation.ObservationManager;
import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Reference;
import org.apache.sling.jcr.api.SlingRepository;
import org.osgi.service.component.ComponentContext;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
@Component
public class TitlePropertyListener implements EventListener {
    private final Logger LOGGER = LoggerFactory.getLogger(TitlePropertyListener.class);
    @Reference
    private SlingRepository repository;
    private Session session;
    private ObservationManager observationManager;
    protected void activate(ComponentContext context) throws Exception {
        session = repository.loginService(null,null);
        observationManager = session.getWorkspace().getObservationManager();
        observationManager.addEventListener(this, Event.PROPERTY_ADDED | Event.PROPERTY_CHANGED,
    "/", true, null,
                null, true);
        LOGGER.info("********added JCR event listener");
    }
    protected void deactivate(ComponentContext componentContext) {
        try {
            if (observationManager != null) {
                observationManager.removeEventListener(this);
                LOGGER.info("********removed JCR event listener");
            }
        }
        catch (RepositoryException re) {
            LOGGER.error("********error removing the JCR event listener", re);
        }
```

```
finally {
            if (session != null) {
                session.logout();
                session = null;
            }
        }
    }
    public void onEvent(EventIterator it) {
        while (it.hasNext()) {
            Event event = it.nextEvent();
            try {
                LOGGER.info("********new property event: {}", event.getPath());
                Property changedProperty = session.getProperty(event.getPath());
                if (changedProperty.getName().equalsIgnoreCase("jcr:title")
                        && !changedProperty.getString().endsWith("!")) {
                    changedProperty.setValue(changedProperty.getString() + "!");
                    session.save();
                }
            }
            catch (Exception e) {
                LOGGER.error(e.getMessage(), e);
            }
        }
    }
}
```

NOTE: The method call addEventListener: Event types are bitwise OR'ed. The last parameter (noLocal) prevents changes made by this session to be sent to the listener, which would result in an endless loop in this case. You can also register for events sent by specific Node types or at specific paths only. (See Event class on the JCR API.)

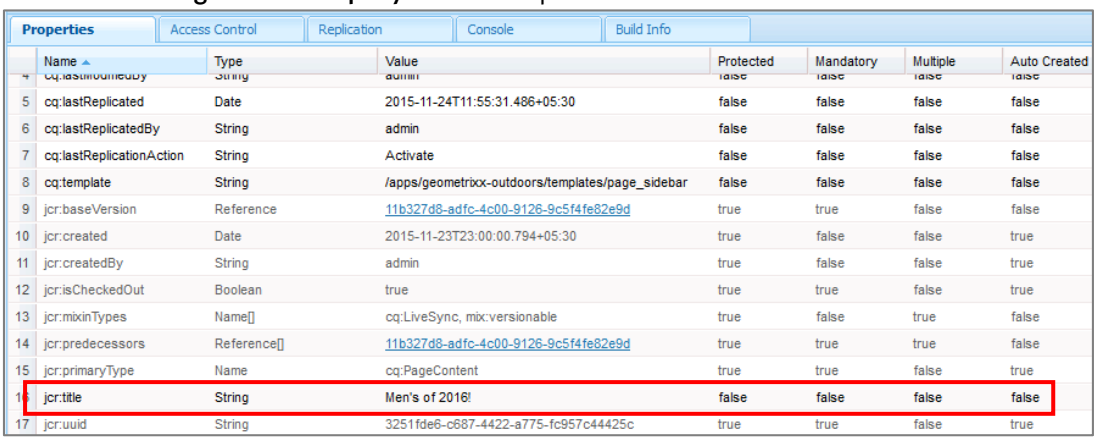
In the **onEvent** method of the **EventListener** interface in our **TitlePropertyListener.java** code check the **jcr:title** value, if the value is added or updated then the value will have"!" at the end of it.

Building this will install the **TitlePropertyListener** component. The component will register itself as the listener on activation. Make sure it deregisters on deactivation so that you do not get a memory leak.

Build the project by selecting **training.core** and click **Run** > **Run As** > **Maven install**.

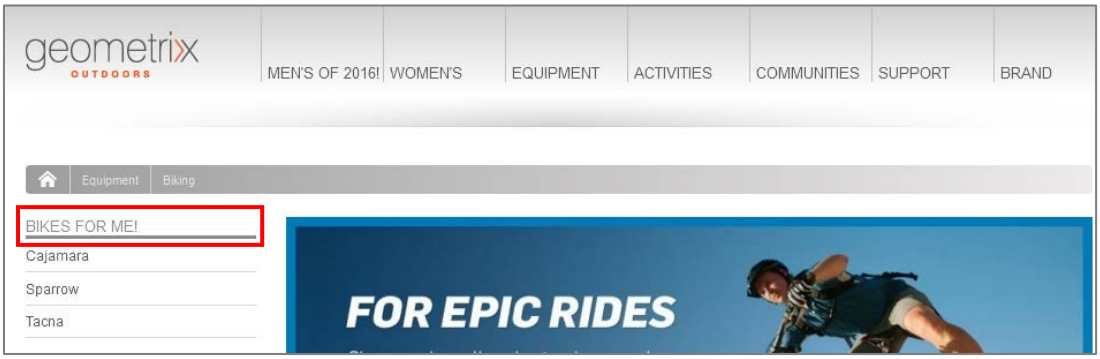
Check that the component is installed at: http://localhost:4502/system/console/components

Change property **jcr:title** in CRXDE Lite. Refresh CRXDE Lite to see that '!' is added by **com.adobe.training.core.TitlePropertyListener** component.



	Name	Type	Value	Protected	Mandatory	Multiple	Auto Created
5	cq:lastReplicated	Date	2015-11-24T11:55:31.486+05:30	false	false	false	false
6	cq:lastReplicatedBy	String	admin	false	false	false	false
7	cq:lastReplicationAction	String	Activate	false	false	false	false
8	cq:template	String	/apps/geometrixx-outdoors/templates/page_sidebar	false	false	false	false
9	jcr:baseVersion	Reference	11b327d8-adfc-4c00-9126-9c5f4fe82e9d	true	true	false	false
10	jcr:created	Date	2015-11-23T23:00:00.794+05:30	true	false	false	true
11	jcr:createdBy	String	admin	true	false	false	true
12	jcr:isCheckedOut	Boolean	true	true	true	false	true
13	jcr:mixinTypes	Name[]	cq:LiveSync, mix:versionable	true	false	true	false
14	jcr:predecessors	Reference[]	11b327d8-adfc-4c00-9126-9c5f4fe82e9d	true	true	true	false
15	jcr:primaryType	Name	cq:PageContent	true	true	false	true
16	jcr:title	String	Men's of 2016!	false	false	false	false
17	jcr:uuid	String	3251fde6-c687-4422-a775-fc957c44425c	true	true	false	true

You can also change any page title in the Page Editor, and see that the page title is added with an '!'.



Scenario Conclusion

You have created an observation listener that looks for any changes made to the jcr:title property of a page, and displayed the notification accordingly.