# Contents

# Using services and servlets

## Introduction

Adobe Experience Manager (AEM) architecture consists of frameworks such as Open Services Gateway Initiative (OSGi) and Apache Sling. OSGi defines a dynamic component written in Java. The OSGi specifications enable a development model where the dynamic application is comprised of reusable components.

Apache Sling is an open source web application framework that makes it easy to develop content-oriented applications. Apache Sling is a Representational State Transfer (REST)-based web framework. It uses scripts or Java Servlets to process HTTP requests.

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

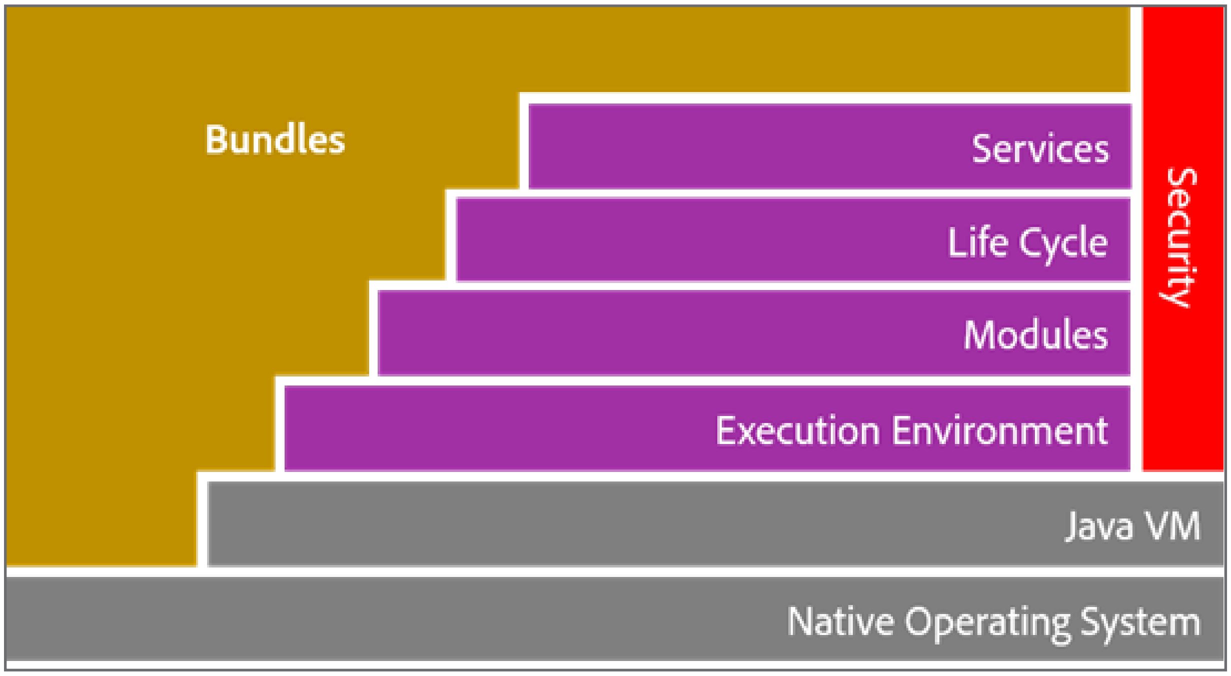
## Objectives

After completing this course, you will be able to:

- Describe OSGi architecture
- Define OSGi annotations
- Implement OSGi configurations
- Describe Apache Sling
- Describe the Sling resolution process
- Create Sling servlets

## OSGi Architecture

The OSGi has a layered model, as shown:



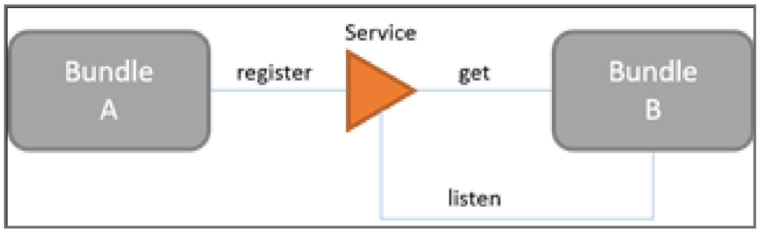
- Bundles: OSGi components created by developers.
- Services: Connect bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects.
- Life Cycle: API that installs, starts, stops, updates, and uninstalls bundles.
- Modules: Define how a bundle can import and export code.
- Security: Handles the security aspects.
- Execution Environment: Defines the methods and classes that are available in a specific platform.

To gain an in-depth understanding of the OSGi architecture, visit:
https://www.osgi.org/developer/architecture/

## 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 the objects that are registered under a specific interface 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 process is shown in the following figure:

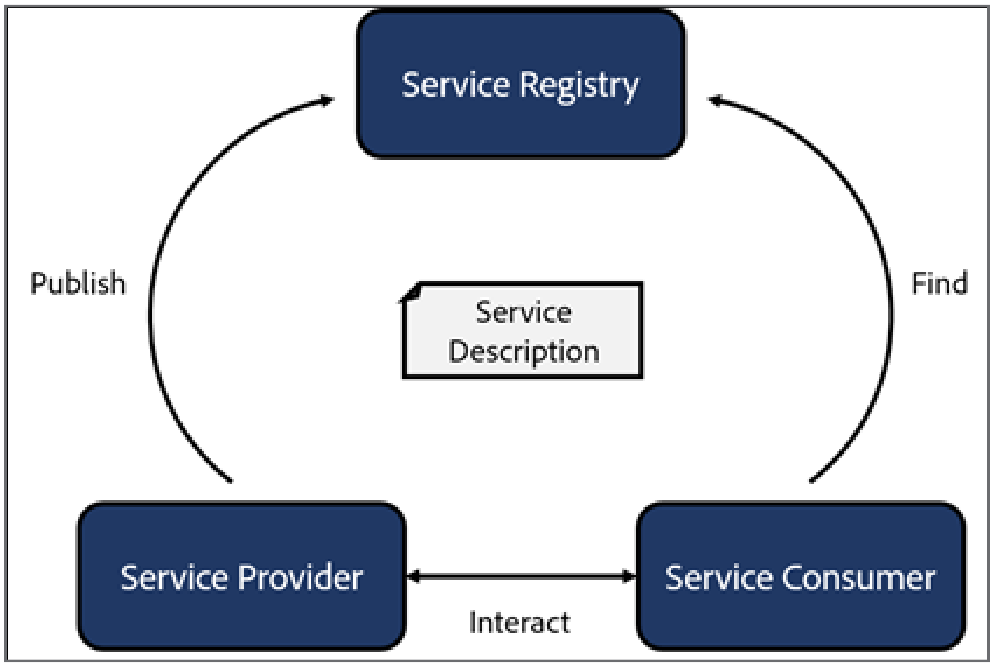


Each service registration has a set of standard and custom properties. An expressive filter language is available to select only the services in which you are interested. You can use properties to find the proper service or they can play other roles at the application level.

Services are dynamic. This means 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.

**Service Registry Model**

OSGi provides a service-oriented component model by using a publish/find/bind mechanism. For example, 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 lookup using the Whiteboard registry pattern, as shown:



## Deployment of Bundles

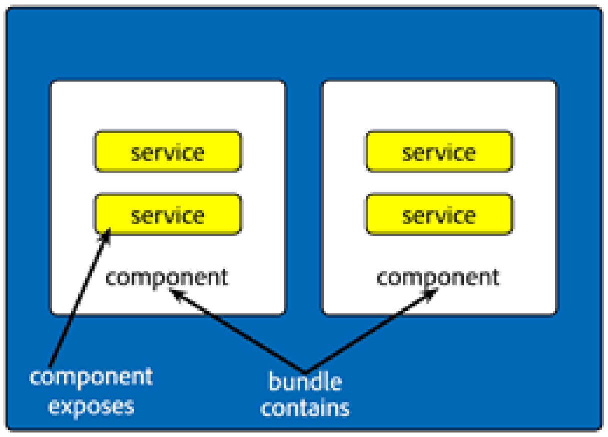
Bundles are deployed on an OSGi framework—the bundle runtime environment. It is a collaborative environment, where the bundles run in the same Virtual Machine (VM) and can actually share code. The framework uses the explicit imports and exports to wire up the bundles, so they do not need to involve themselves 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.

## Components

Components are the main building blocks for OSGi applications.

A component:

- Is provided by a bundle.
- Is a piece of software managed by an OSGi container.
- Is a Java object created and managed by an OSGi container.
- Can provide a service.
- Can implement one or more Java interfaces as services.



A component can publish itself as a service and/or can have dependencies on other components and services. The OSGi container will activate a component only when all the required dependencies are met or available. Each component has an implementation class, and can optionally implement a public interface providing this service.

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

- An XML file, where you describe the service the bundle provides and the dependencies of other services of the OSGi framework.
- A manifest file header entry to declare that the bundle behaves as a component.
- The activate and deactivate methods in the implementation class (or bind and unbind methods).
- Service Component Runtime (SCR). A service of the OSGi framework to manage these components.

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

## Annotations in OSGi

With Declarative Services, OSGi components are developed using annotations to generate bundle descriptors as well as metatype description for their configuration. Since AEM 6.2, the official OSGi R6 Declarative Services Annotations are supported and Adobe recommends using those annotations as they are defined as a standard and allow for simpler and cleaner code.

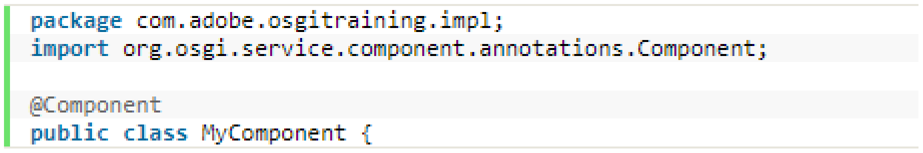
Projects based on the previously recommended Felix SCR annotations (now in maintenance mode) can be easily migrated and both annotations styles can coexist within a bundle during the migration phase.

The following annotations are supported:

- Component
- Activate
- Deactivate
- Modified
- Reference

**@Component**

The @Component annotation enables the OSGi Declarative Services to register your component. This is the only required annotation for an OSGi 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.



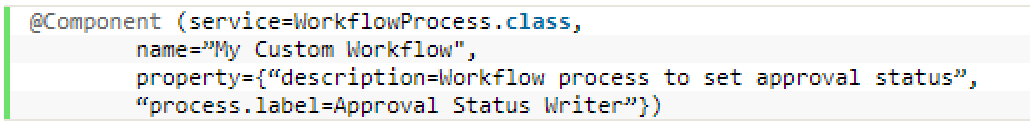
```
package com.adobe.osgitraining.impl;
import org.osgi.service.component.annotations.Component;

@Component
public class MyComponent {
```

## Component Modifiers

With OSGi DS annotations, type and property definitions are not done using annotations, as they are with Felix SCR annotations, but through component modifiers (or attributes).
In fact, what we are doing when creating a component is to register the type of service (by implementing one or several interfaces) and to declare the properties made available by metatype generation, all this with the same annotation.
Some of the attributes you can use with the @Component annotation are:

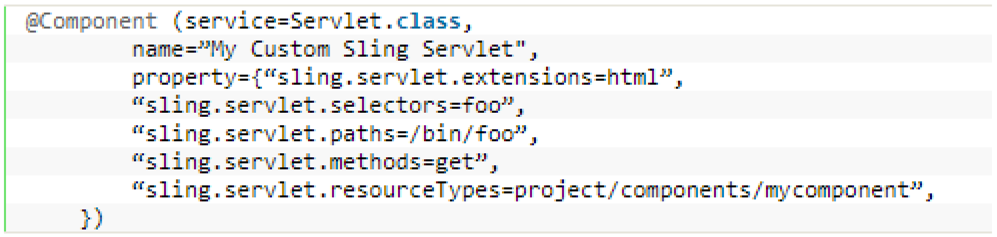
- service: Types under which to register this component as a service
- properties: Property entries for this component
- name: Name of the component
- immediate: Indicates whether the component is immediately activated on bundle start

Example:



```
@Component (service=WorkflowProcess.class,
        name="My Custom Workflow",
        property={"description=Workflow process to set approval status",
        "process.label=Approval Status Writer"})
```

When creating a Sling servlet, the servlet properties are defined in the property component modifier like in this example:



```
@Component (service=Servlet.class,
        name="My Custom Sling Servlet",
        property={"sling.servlet.extensions=html",
        "sling.servlet.selectors=foo",
        "sling.servlet.paths=/bin/foo",
        "sling.servlet.methods=get",
        "sling.servlet.resourceTypes=project/components/mycomponent",
    })
```

### @Activate, @Deactivate, and @Modified

Methods annotated with these annotations specify what happens when the component is activated, deactivated, or its configuration is modified.

```
import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Deactivate;
import org.osgi.service.component.annotations.Modified;
@Component
public class MyComponent{
@Activate
    protected void activate() {
        // do something
    }
@Deactivate
    protected void deactivate() {
        // do something
    }
@Modified
    protected void readConfig() {
        // get values
    }
}
```

## Configurable Services

The OSGi Configuration Admin Service enables components to get or retrieve a configuration, and provides the entry point for management agents to retrieve and update configuration data. 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.

Unlike SCR annotations, for which properties need to be read by overriding the activate() method, OSGi DS annotations provide a mechanism for metatype declaration through an interface, avoiding reading and converting each property. This interface is annotated with @ObjectClassDefinition and defines each property with its type, using the annotations @AttributeDefinition and @AttributeType.

Here is an example:

```
import org.osgi.service.metatype.annotations.ObjectClassDefinition;
import org.osgi.service.metatype.annotations.AttributeDefinition;
import org.osgi.service.metatype.annotations.AttributeType;
@ObjectClassDefinition(name = My Configuration Service)
public @interface MyConfigInterface{
   @AttributeDefinition(
name="Enter a String",
description="Your description",
type=AttributeType.STRING
)
    String myconfig_property() default "";
}
```

This interface is then defined in the component using the annotation @Designate. A modifier named factory can be added to define the configuration as a factory.

```
import org.osgi.service.component.annontations.Component;
import org.osgi.service.metatype.annotations.Designate;
@Component
@Designate(ocd=MyConfigInterface.class, factory=true)
public class MyComponent {
   private myString
  @Activate
   protected void activate(MyConfigInterface config) {
           myString = config.myconfig_property()
   }
}
```

# Exercise 1: Create and use a custom service

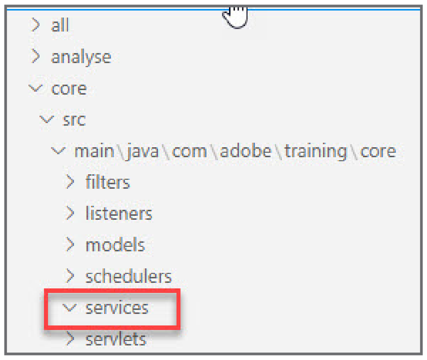
In this exercise, you will create and use a custom service.

This exercise includes three tasks:

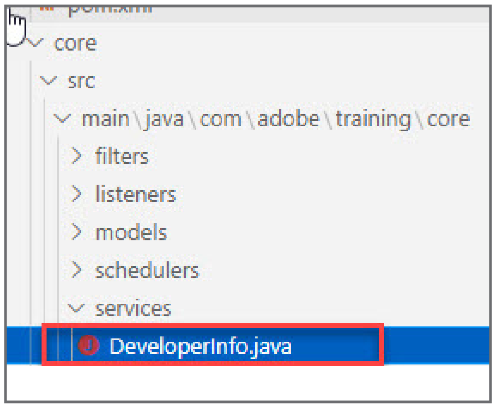
1. Create a service and an implementation
2. Deploy the bundle and expose the service in HTL
3. Test the service

Task 1: Create a service and an implementation

1. Open the IDE containing your Maven project for this course if not already opened.
2. In the IDE navigation on the left, find the **core** module.
3. Navigate to **core** > **src** > **main/java/com/adobe/training/core.**
4. Right-click the **core** folder and choose **New Folder.**
5. Name the folder as **services**, as shown:



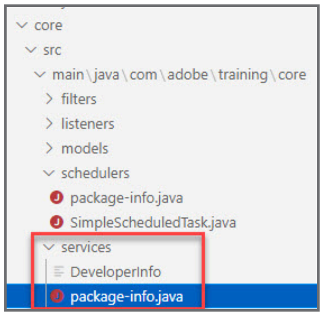
6. Right-click the **services** folder and choose **New File**.
7. Name the file as **DeveloperInfo.java**, as shown:



8. To add code to your **DeveloperInfo.java**, open up the **Exercise_Files-EC** folder for this course and navigate to **/core/src/main/java/com/adobe/training/core/services/**.
9. Open **DeveloperInfo.java** and copy the contents.
10. In your IDE, paste the contents into your Maven Project > **DeveloperInfo.java**.
11. Examine the service **DeveloperInfo.java** interface, as shown:

```
package com.adobe.training.core.services;
/**
 * Service interface to get the information about the bundle Developer
 *
 * Example code can be inserted into the Helloworld HTL component:
 * /apps/training/components/helloworld/helloworld.html
 *
 * Example HTL:
 * <div data-sly-use.devInfo=»com.adobe.training.core.DeveloperInfo»>Developer Info: ${devInfo.DeveloperInfo}</div>
 *
 *
 */
public interface DeveloperInfo {
   public String getDeveloperInfo();

}
```

12. Click **Ctrl+S** or click **File** > **Save** to save your changes.
13. Navigate to **core** > **src** > **main/java/com/adobe/training/core/schedulers** and copy the file **Package-info.java**.
14. Paste the file into the **services** folder, as shown:

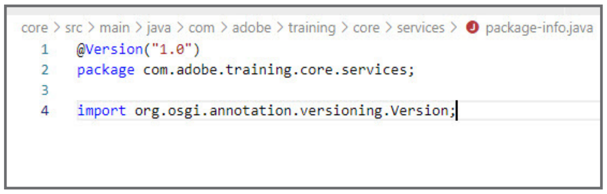


**Note**: The **package-info.java** allows for this package to be visible in OSGI.

15. Copy the contents of the file **package-info.java** from **Exercise_Files-EC** under **/core/src/main/java/com/adobe/training/core/services/package-info.java**.
16. Open the newly created **package-info.java**. Since the file was copied from the schedulers package, the package statement (line 2) needs to be updated. Update this statement to reflect the services package:

```
package com.adobe.training.core.services
```

17. Verify the completed file, as shown:



```
core > src > main > java > com > adobe > training > core > services > package-info.java
@Version("1.0")
package com.adobe.training.core.services;

import org.osgi.annotation.versioning.Version;
```

18. Click **Ctrl+S** or click **File** > **Save** to save your changes to **package-info.java** in the IDE editor.
19. Right-click the **services** folder and select **New Folder.**
20. Type the name as **impl**.
21. Right-click the **impl** folder and select **New File.**
22. Name the file as **DeveloperInfoImpl.java**.
23. Copy the contents of the file **DeveloperInfoImpl.simple.txt** from **Exercise_Files-EC** under **\core\src\main\java\com\adobe\training\core\services\impl\**.
24. In your IDE, if not already open, double-click **DeveloperInfoImpl.java** to open it and replace the contents of the file with the copied content.

25. Examine the implementation of the Simple component of the **DeveloperInfoImpl.java** class, as shown:

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

import java.util.Map;

import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Deactivate;
import org.osgi.service.component.annotations.Modified;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.adobe.training.core.services.DeveloperInfo;

/**
 * Component implementation of the DeveloperInfo Service.
 */

@Component(service = DeveloperInfo.class,
           immediate = true)
public class DeveloperInfoImpl implements DeveloperInfo {
    private final Logger logger = LoggerFactory.getLogger(getClass());

    @Activate
    @Modified
    protected void activate(Map<String, Object> config) {
        logger.info("#############Component config saved");
    }

    @Deactivate
    protected void deactivate() {
        logger.info("#############Component (Deactivated) Good-bye ");
    }


    // Method used to show a simple OSGi service/component relationship

    public String getDeveloperInfo()
{ return "Hello! I do not know who my developer is. I am a product of random development!!!";
    }


}
```

26. Examine the method **getDeveloperInfo()** in the **DeveloperInfoImpl.java** class.

27. Save (**Ctrl+S** in Windows) your changes.

## Task 2: Deploy the bundle and expose the service in HTL

1. Open a command prompt to the location of your Maven project. For example: **C:/adobe/< myproject >**

---

**Note**: If you are using an IDE with an integrated terminal such as Visual Studio Code, you can run your Maven commands there instead since it is already open to your project.

---

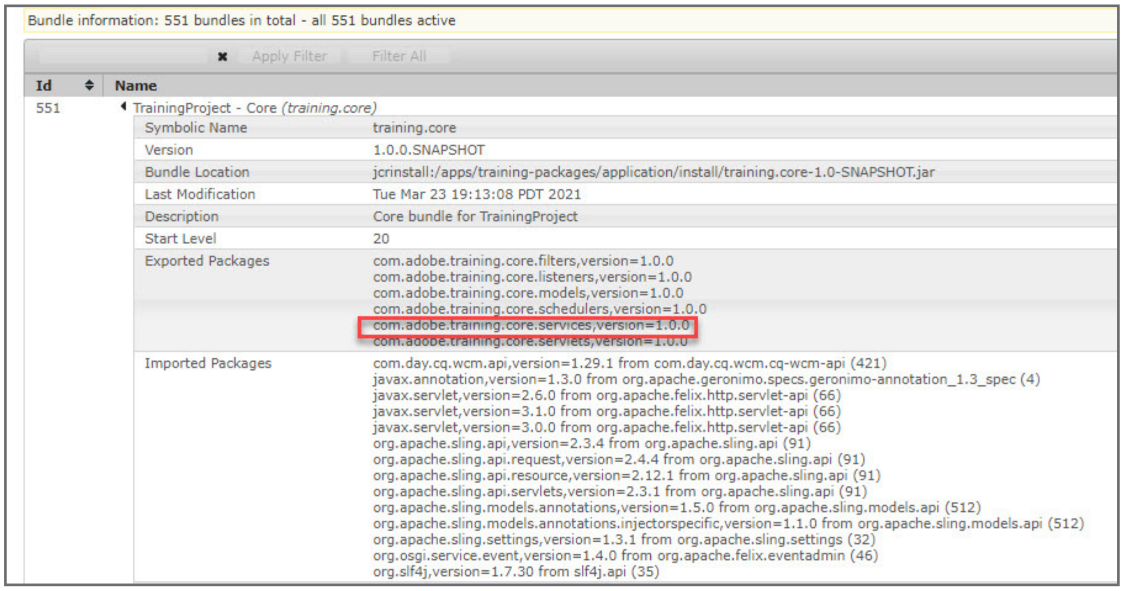
2. In the command prompt run the command:

```
$ mvn clean install -Padobe-public -PautoInstallSinglePackage
```

   Your project has now successfully installed into your local AEM Server

3. Open a browser and go to the AEM Web Console, http://localhost:4502/system/console/. The **Adobe Experience Manager Web Console Bundles** page opens.

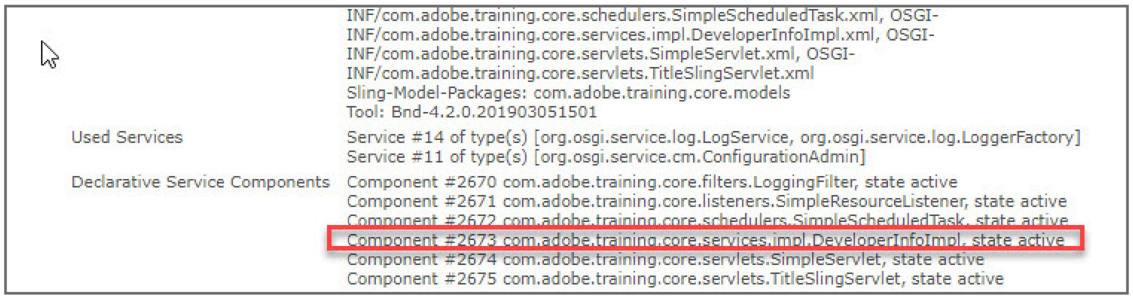
4. Search for and click **TrainingProject – Core** (**training.core**) to expand it. Notice your core services package has been exported, as shown:



Bundle information: 551 bundles in total - all 551 bundles active

| Id | Name | |
|---|---|---|
| 551 | TrainingProject - Core *(training.core)* | |
| | Symbolic Name | training.core |
| | Version | 1.0.0.SNAPSHOT |
| | Bundle Location | jcrinstall:/apps/training-packages/application/install/training.core-1.0-SNAPSHOT.jar |
| | Last Modification | Tue Mar 23 19:13:08 PDT 2021 |
| | Description | Core bundle for TrainingProject |
| | Start Level | 20 |
| | Exported Packages | com.adobe.training.core.filters,version=1.0.0<br>com.adobe.training.core.listeners,version=1.0.0<br>com.adobe.training.core.models,version=1.0.0<br>com.adobe.training.core.schedulers,version=1.0.0<br>com.adobe.training.core.services,version=1.0.0<br>com.adobe.training.core.servlets,version=1.0.0 |
| | Imported Packages | com.day.cq.wcm.api,version=1.29.1 from com.day.cq.wcm.cq-wcm-api (421)<br>javax.annotation,version=1.3.0 from org.apache.geronimo.specs.geronimo-annotation_1.3_spec (4)<br>javax.servlet,version=2.6.0 from org.apache.felix.http.servlet-api (66)<br>javax.servlet,version=3.1.0 from org.apache.felix.http.servlet-api (66)<br>javax.servlet,version=3.0.0 from org.apache.felix.http.servlet-api (66)<br>org.apache.sling.api,version=2.3.4 from org.apache.sling.api (91)<br>org.apache.sling.api.request,version=2.4.4 from org.apache.sling.api (91)<br>org.apache.sling.api.resource,version=2.12.1 from org.apache.sling.api (91)<br>org.apache.sling.api.servlets,version=2.3.1 from org.apache.sling.api (91)<br>org.apache.sling.models.annotations,version=1.5.0 from org.apache.sling.models.api (512)<br>org.apache.sling.models.annotations.injectorspecific,version=1.1.0 from org.apache.sling.models.api (512)<br>org.apache.sling.settings,version=1.3.1 from org.apache.sling.settings (32)<br>org.osgi.service.event,version=1.4.0 from org.apache.felix.eventadmin (46)<br>org.slf4j,version=1.7.30 from slf4j.api (35) |

5. Scroll down and look for the Service, **com.adobe.training.core.services.impl.DeveloperInfoImpl**.
6. Verify that the service exists in the bundles, as shown:



```
                                INF/com.adobe.training.core.schedulers.SimpleScheduledTask.xml, OSGI-
                                INF/com.adobe.training.core.services.impl.DeveloperInfoImpl.xml, OSGI-
                                INF/com.adobe.training.core.servlets.SimpleServlet.xml, OSGI-
                                INF/com.adobe.training.core.servlets.TitleSlingServlet.xml
                                Sling-Model-Packages: com.adobe.training.core.models
                                Tool: Bnd-4.2.0.201903051501
Used Services                   Service #14 of type(s) [org.osgi.service.log.LogService, org.osgi.service.log.LoggerFactory]
                                Service #11 of type(s) [org.osgi.service.cm.ConfigurationAdmin]
Declarative Service Components  Component #2670 com.adobe.training.core.filters.LoggingFilter, state active
                                Component #2671 com.adobe.training.core.listeners.SimpleResourceListener, state active
                                Component #2672 com.adobe.training.core.schedulers.SimpleScheduledTask, state active
                                Component #2673 com.adobe.training.core.services.impl.DeveloperInfoImpl, state active
                                Component #2674 com.adobe.training.core.servlets.SimpleServlet, state active
                                Component #2675 com.adobe.training.core.servlets.TitleSlingServlet, state active
```

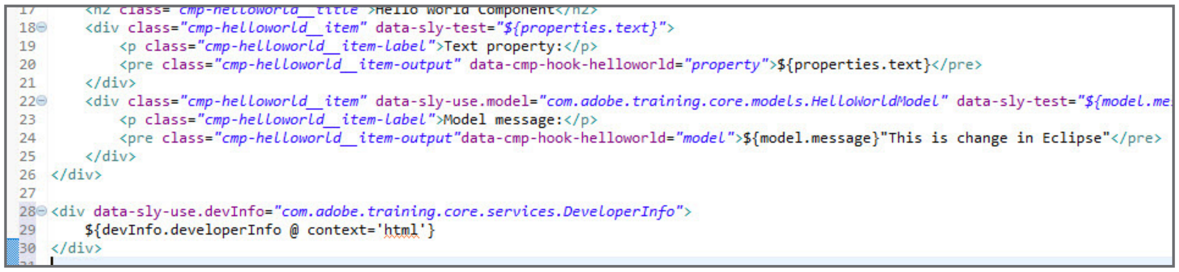
This indicates that the class was successfully installed in the OSGI.

7. In your IDE, under Project Explorer, navigate to **training.ui.apps** > **src/main/content/jcr_root** > **apps** > **training** > **components** > **helloworld**.
8. Expand **helloworld.html** and double-click **helloworld.html**.

With HTL, you can consume an OSGi service by calling it with data-sly-use. In this step, you will call the **DeveloperInfo** service and use the implemented method in the **DeveloperInfoImpl** component.

9. Add the following element to the **helloworld.html** file:

```
<div data-sly-use.devInfo="com.adobe.training.core.services.
DeveloperInfo">
   ${devInfo.developerInfo @ context='html'}
</div>
```



```
    <h2 class="cmp-helloworld__title">Hello World Component</h2>
    <div class="cmp-helloworld__item" data-sly-test="${properties.text}">
        <p class="cmp-helloworld__item-label">Text property:</p>
        <pre class="cmp-helloworld__item-output" data-cmp-hook-helloworld="property">${properties.text}</pre>
    </div>
    <div class="cmp-helloworld__item" data-sly-use.model="com.adobe.training.core.models.HelloWorldModel" data-sly-test="${model.me
        <p class="cmp-helloworld__item-label">Model message:</p>
        <pre class="cmp-helloworld__item-output"data-cmp-hook-helloworld="model">${model.message}"This is change in Eclipse"</pre>
    </div>
</div>

<div data-sly-use.devInfo="com.adobe.training.core.services.DeveloperInfo">
    ${devInfo.developerInfo @ context='html'}
</div>
```

**Note**: This code is also available in the file under /**Exercise_Files-EC/ui.apps/src/main/content/jcr_root/apps/training/components/helloworld/helloworld .html**.

10. Save the changes to the file. If you are using a sync extension with your IDE, the saved file should automatically sync to AEM.

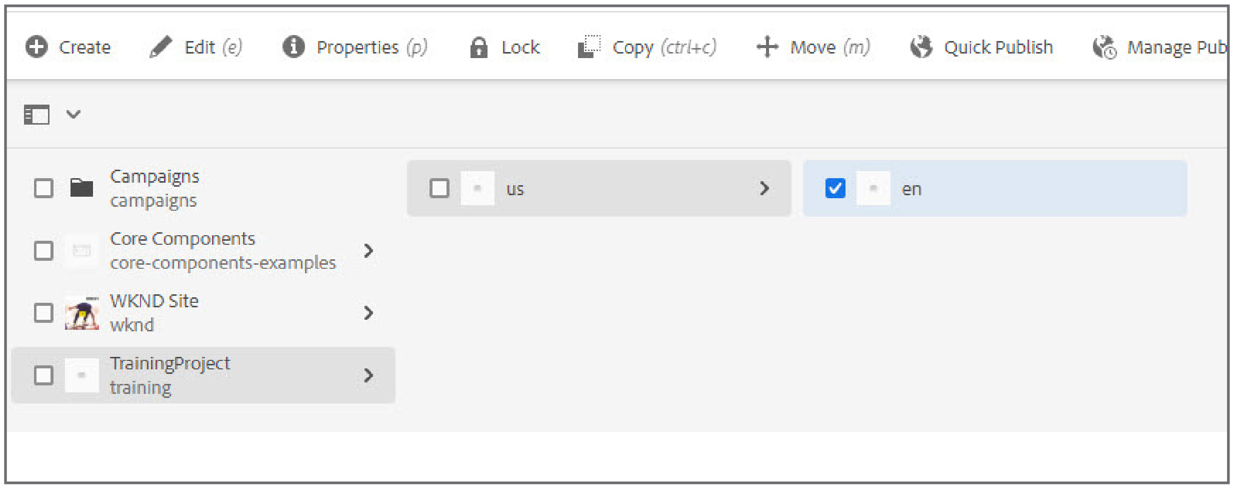
**Note**: If you are not using a sync extension you will need to rerun the maven install command:

```
mvn clean install -Padobe-public -PautoInstallSinglePackage
```

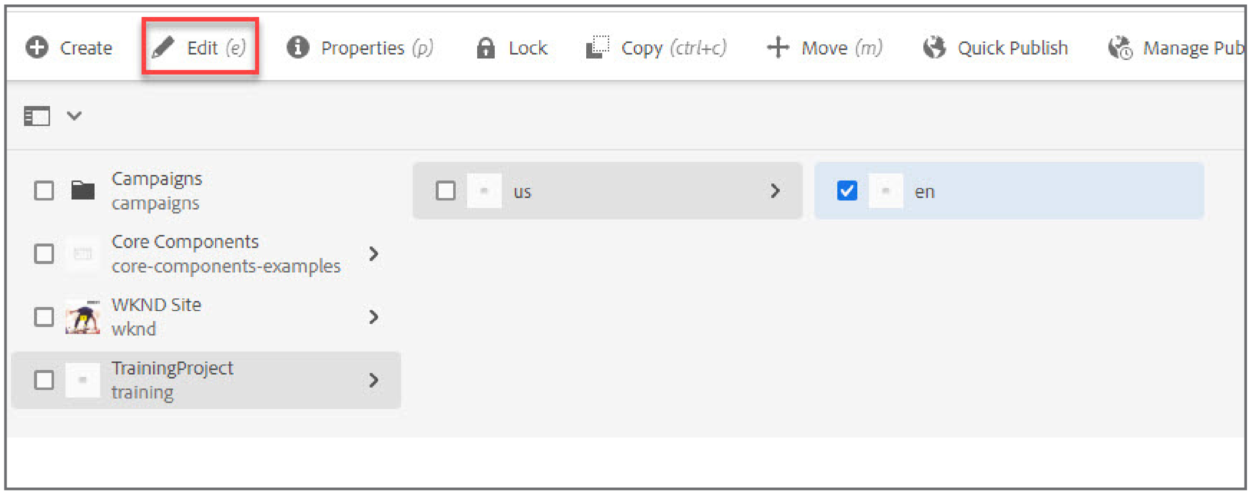
11. To verify whether your file successfully saved to the JCR, open **CRXDE Lite** to **/apps/training/components/helloworld/helloworld.html** and see if that file contains your changes.
12. If your changes are missing, you can force an update with a full redeploy to the JCR by going back to your IDE and doing a Maven install**.**
13. Verify the bundle is installed successfully.

Task 3: Test the service

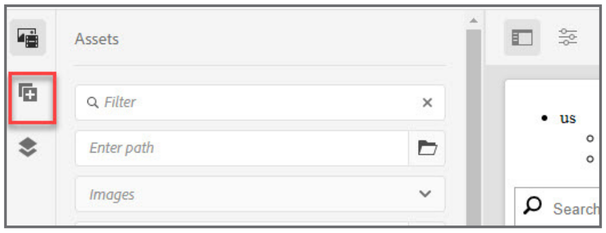
1. In your web browser, log in to AEM and click **Adobe Experience Manager** in the upper left.
2. On the Sites console, select **TrainingProject** > **us** > **en**, as shown:



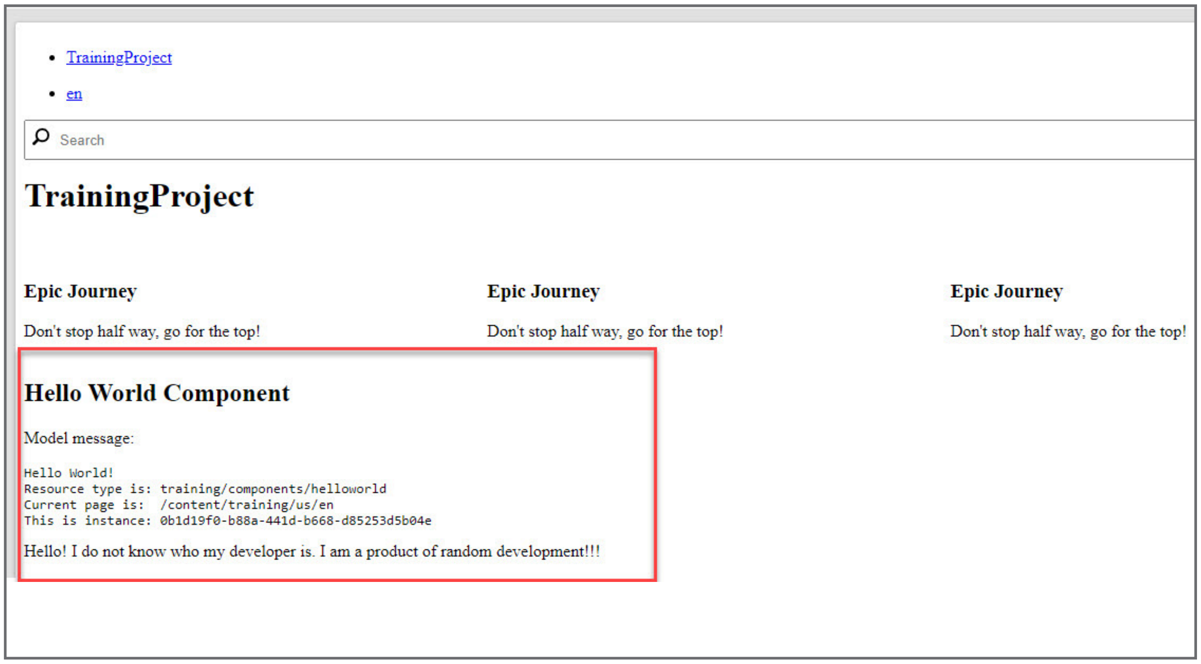
3. Click **Edit (*e*)**, as shown:



4. If the **Hello World component** is not on the page, perform the following steps.
5. Click on the Components icon on the page.



6. Scroll down to find and then drag the **HelloWorld Component** to the **Drag components here** area on your page.
7. Verify you see the following message, as shown:



TrainingProject

Epic Journey

Don't stop half way, go for the top!

Epic Journey

Don't stop half way, go for the top!

Epic Journey

Don't stop half way, go for the top!

Hello World Component

Model message:

```
Hello World!
Resource type is: training/components/helloworld
Current page is:  /content/training/us/en
This is instance: 0b1d19f0-b88a-441d-b668-d85253d5b04e
```

Hello! I do not know who my developer is. I am a product of random development!!!

# Exercise 2: Code OSGi configurations

In this exercise, you will code the custom OSGi configurations for an OSGi component. This will allow an administrator to configure the component as we did earlier. These configurations can then be further targeted by using run modes.

This exercise includes the following tasks:

1. Create a service with configurations
2. Create a JSON configuration file
3. Install via Command Line
4. Test the service

Task 1: Create a service with configurations

1. In your IDE, navigate to **core** > **src** > **main/java/com/adobe/training/core/services/impl**.
2. Double-click **DeveloperInfoImpl.java** to edit the file.
3. Copy the contents from **Exercise_Files-EC** under **/core/src/main/java/com/adobe/training/core/services/impl/DeveloperInfoImpl.java** to **DeveloperInfoimpl.java** in your IDE, as shown:

```java
package com.adobe.training.core.services.impl;
import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Deactivate;
import org.osgi.service.component.annotations.Modified;
import org.osgi.service.metatype.annotations.Designate;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.util.Arrays;
import com.adobe.training.core.DeveloperInfo;
 /**component implementation
 of the DeveloperInfo Service. This gets the developer info from the OSGi Configuration
 * There are 4 OSGi Configuration Examples:
 * -Boolean
 * -String
 * -String Array
 * -Dropdown
 */

@Component(service = DeveloperInfo.class, immediate = true)

@Designate(ocd = DeveloperInfoConfiguration.class)
public class DeveloperInfoImpl implements DeveloperInfo {
    private final Logger logger = LoggerFactory.getLogger(getClass());

    //local variables to hold OSGi config values
    private boolean showDeveloper;
    private String developerName;
    private String[] developerHobbiesList;
    private String langPreference;

    @Activate
    @Modified
    //http://blogs.adobe.com/experiencedelivers/experience-management/osgi_activate_
deactivatesignatures/
    protected void activate(DeveloperInfoConfiguration config) {

        showDeveloper = config.developerinfo_showinfo();
        developerName = config.developerinfo_name();
        developerHobbiesList = config.developerinfo_hobbies();
        langPreference = config.developerinfo_language();
        logger.info("############Component config saved");
    }

    @Deactivate
    protected void deactivate() {
        logger.info("############Component (Deactivated) Good-bye " + developerName);
    }

    /**
     * Method used to show how OSGi configurations can be brought into a OSGi component
    **/
    public String getDeveloperInfo(){

        String developerHobbies = Arrays.toString(developerHobbiesList);

        if(showDeveloper)

            return "Created by " + developerName
                + ". <br>Hobbies include: " + developerHobbies
              + ". <br>Preferred programming language in AEM is " + langPreference;
        return "";
    }

    /*
     * Method used to show a simple OSGi service/component relationship
    public String getDeveloperInfo(){ return
 «Hello! I do not know who my developer is. I am a product of random development!!!»;
    }
    */

      }
```

**Note**: Ignore errors that mention Developer Info Configuration. We will fix these errors in a few steps.

4. Examine the modified code.
5. Examine the various methods used in the class such as **activate()**, **deactivate()** and **getDeveloperInfo()** to understand the logic behind them.

Notice the activate method takes a special class called **DeveloperInfoConfiguration**. This is a custom configuration class where you set what properties should be exposed to OSGi through the Web Console

6. In your IDE, navigate to **core** > **src** > **main/java/com/adobe/training/core/services/impl**.
7. Right-click on **impl** and select **New File**.
8. Type the following value for the name:
   › Name: **DeveloperInfoConfiguration.java**
9. Copy the contents of the file **DeveloperInfoConfiguration .java** from **\core\src\main\java\com\adobe\training\core\services\impl** to **DeveloperInfoConfiguration.java** in your IDE.

**Note**: The code is provided as part of the **Exercise_Files-EC** under **\core\src\main\java\com\adobe\training\core\services\**. Copy and paste the file from the exercise files referenced to **DeveloperInfoConfiguration.java** to Eclipse. DO NOT copy the code from this exercise book. The code given here is for illustrative purposes only.

10. Examine **DeveloperInfoConfiguration.java**, as shown:

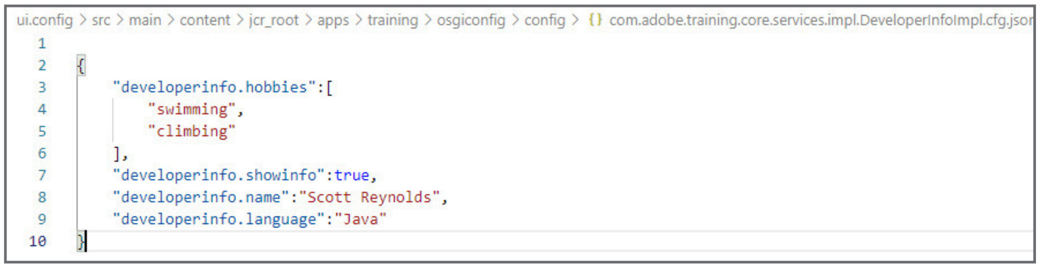
```
package com.adobe.training.core.services.impl;

import org.osgi.service.metatype.annotations.AttributeDefinition;
import org.osgi.service.metatype.annotations.ObjectClassDefinition;
import org.osgi.service.metatype.annotations.AttributeType;
import org.osgi.service.metatype.annotations.Option;

@ObjectClassDefinition(name = "Training DeveloperInfo Config»)
public @interface DeveloperInfoConfiguration {

  @AttributeDefinition(
      name = "Show Info",
      description = "Should the Developer information be shown?",
      type = AttributeType.BOOLEAN
  )
  boolean developerinfo_showinfo() default false;

  @AttributeDefinition(
      name = "Name",
      description = "Name of the Developer",
      type = AttributeType.STRING
  )
  String developerinfo_name() default "";

  @AttributeDefinition(
      name = "Hobbies",
      description = "List your favorite Hobbies",
      type = AttributeType.STRING
  )
  String[] developerinfo_hobbies() default {"swimming", "climbing"};

  @AttributeDefinition(
      name = "Language",
      description = "Favorite Language Preference",
      options = {
          @Option(label = "HTL", value = "HTL"),
          @Option(label = "Java", value = "Java"),
          @Option(label = "JSP", value = "JSP"),
          @Option(label = "HTML", value = "HTML"),
          @Option(label = "JavaScript", value = "JavaScript")
      }
  )
  String developerinfo_language() default "";
}
```

11. Save the changes.

**Note**: You get the configurations in the **activate()** method by using the methods defined in the **DeveloperInfoConfiguration** class.

## Task 2: Create a JSON configuration file

1. In your IDE, navigate to **ui.config** > **src/main/content/jcr_root** > **apps** > **training** > **osgiconfig**.
2. Right-click the folder **config** and choose **New file**.
3. Enter the name as **com.adobe.training.core.services.impl.DeveloperInfoImpl.cfg.json**.
4. To add the json properties to your file, open up the **Exercise_Files-EC** folder for this course and navigate to **ui.config/src/main/content/jcr_root/apps/training/osgiconfig/config.**
5. Open the file **com.adobe.training.core.services.impl.DeveloperInfoImpl.cfg.json** using a text editor. Copy the contents and paste it to the file **com.adobe.training.core.services.impl. DeveloperInfoImpl.cfg.json** in your IDE, as shown:



ui.config > src > main > content > jcr_root > apps > training > osgiconfig > config > {} com.adobe.training.core.services.impl.DeveloperInfoImpl.cfg.json

```

{
    "developerinfo.hobbies":[
        "swimming",
        "climbing"
    ],
    "developerinfo.showinfo":true,
    "developerinfo.name":"Scott Reynolds",
    "developerinfo.language":"Java"
}
```

6. Select **File** > **Save** from the menu to save the file.

## Task 3: Install via command line

1. Open a command prompt to the location of your Maven project. For example: **C:/adobe/< myproject >**

**Note**: If you are using an IDE with an integrated terminal such as Visual Studio Code, you can run your Maven commands there instead since it is already open to your project.

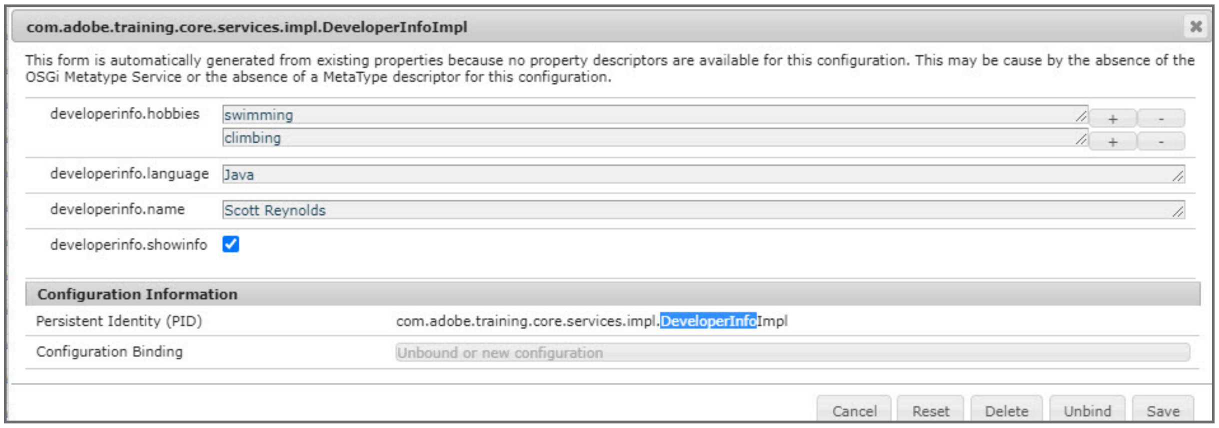
2. In the command prompt run the command:

```
$ mvn clean install -Padobe-public -PautoInstallSinglePackage
```

Your project has now been successfully installed on your local AEM Server.

## Task 4: Test the service

1. Navigate to the Web Console in AEM (http://localhost:4502/system/console/configMgr). The **Adobe Experience Manager Web Console** page opens.
2. Under **OSGi** > **Configuration**, search for **DeveloperInfo**.
3. Click **Training DeveloperInfo Config**. The **Training DeveloperInfImpl Config** opens, as shown:



**com.adobe.training.core.services.impl.DeveloperInfoImpl**

This form is automatically generated from existing properties because no property descriptors are available for this configuration. This may be cause by the absence of the OSGi Metatype Service or the absence of a MetaType descriptor for this configuration.

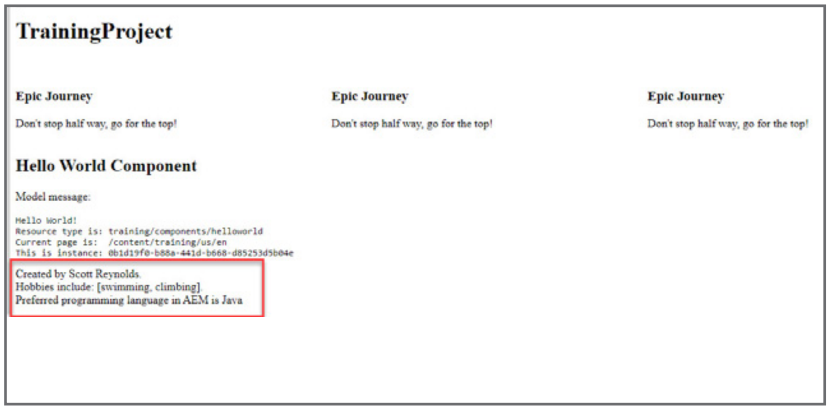
| | |
|---|---|
| developerinfo.hobbies | swimming<br>climbing |
| developerinfo.language | Java |
| developerinfo.name | Scott Reynolds |
| developerinfo.showinfo | ✓ |

**Configuration Information**

| | |
|---|---|
| Persistent Identity (PID) | com.adobe.training.core.services.impl.DeveloperInfoImpl |
| Configuration Binding | Unbound or new configuration |

   a. Verify the properties are the same as the JSON config you created in your project.

4. Click **Cancel** to close the window.
5. Go back to your AEM author instance and navigate to **Sites** (http://localhost:4502/sites.html/content). The **Sites** console opens.
6. Select **TrainingProject** > **us** > **en** > **Edit (e)** to open the page in a new tab in your browser.
7. Verify the following message, as shown:



**TrainingProject**

**Epic Journey**

Don't stop half way, go for the top!

**Epic Journey**

Don't stop half way, go for the top!

**Epic Journey**

Don't stop half way, go for the top!

**Hello World Component**

Model message:

```
Hello World!
Resource type is: training/components/helloworld
Current page is:  /content/training/us/en
This is instance: 0b1d19f0-b88a-441d-b668-d85253d5b04e
```

Created by Scott Reynolds.
Hobbies include: [swimming, climbing].
Preferred programming language in AEM is Java

## Sling Servlets

Apache Sling is resource-oriented and maintains all the 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.

The following are some of the common properties that a resource can have:

- 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. 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 type, it selects the servlet or script to handle the request.

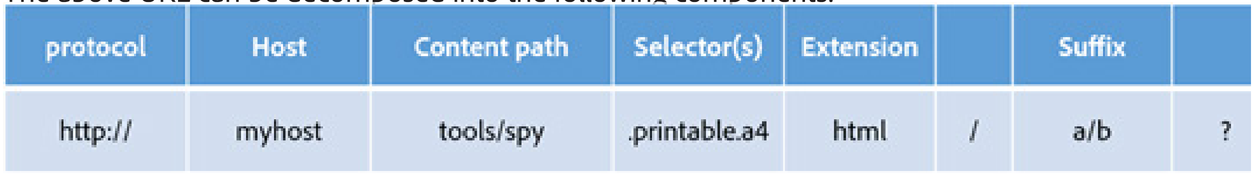
### Basic Steps of Processing Requests

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.

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 |  |
|---|---|---|---|---|---|---|---|
| http:// | myhost | tools/spy | .printable.a4 | html | / | a/b | ? |

The following table describes the components.

| **Component** | **Description** |
|---|---|
| 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 |

## Working with Sling Servlets

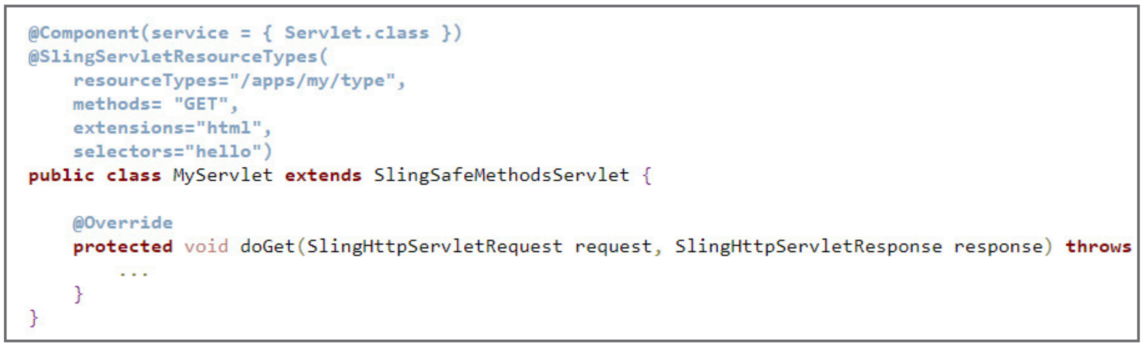
Servlets can be registered as OSGi services. Apache Sling applications use scripts or Java servlets, selected based on simple name conventions, to process HTTP requests in a RESTful way. Being a REST framework, Apache Sling is oriented around resources, which usually map to JCR nodes. With Apache Sling, 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, Apache Sling can determine the resource URL and other information that can be used to process that resource.
For a Servlet registered as an OSGi service to be used by the Sling Servlet Resolver, the sling.servlet.resourceTypes service reference property must be set. If it is not set, the Servlet service is ignored.

Common properties for Sling Servlets

| Name | Description |
|---|---|
| sling.servlet.resourceTypes | The resource type(s) supported by the servlet. The property value must either be a single String, an array of Strings or a Vector of Strings. Either this property or the sling.servlet.paths property must be set, or the servlet is ignored. If both are set, the servlet is registered using both ways. |
| sling.servlet.resourceSuperType | The resource super type, indicating which previously registered servlet could intercept the request if the request matches the resource super type better. The property value must be a single String. This property is only considered for the registration with sling.servlet.resourceTypes. (since version 2.3.0 of the org.apache.sling.api.servlets API, version 2.5.2 of the org.apache.sling.servlets.resolver bundle) |
| sling.servlet.selectors | The request URL selectors supported by the servlet. The selectors must be configured as they would be specified in the URL that is as a list of dot-separated strings such as print.a4. In case this is not empty the first selector(s) (i.e. the one(s) on the left) in the request URL must match, otherwise the servlet is not executed. After that may follow arbitrarily many non-registered selectors. The property value must either be a single String, an array of Strings or a Vector of Strings. This property is only considered for the registration with sling.servlet.resourceTypes. |
| sling.servlet.extensions | The request URL extensions supported by the servlet for requests. The property value must either be a single String, an array of Strings or a Vector of Strings. This property is only considered for the registration with sling.servlet.resourceTypes. |

| Name | Description |
|---|---|
| sling.servlet.methods | The request methods supported by the servlet. The property value must either be a single String, an array of Strings or a Vector of Strings. This property is only considered for the registration with sling.servlet.resourceTypes. If this property is missing, the value defaults to GET and HEAD, regardless of which methods are actually implemented/handled by the servlet. A value of * leads to a servlet being bound to all methods.. |
| sling.servlet.paths | A list of absolute paths under which the servlet is accessible as a Resource. The property value must either be a single String, an array of Strings or a Vector of Strings.<br>A servlet using this property might be ignored unless its path is included in the Execution Paths (servletresolver.paths) configuration setting of the SlingServletResolver service. Either this property or the sling.servlet.resourceTypes property must be set, or the servlet is ignored. If both are set, the servlet is registered using both ways. |
| sling.servlet.paths.strict | When set to true, this enables strict selection mode for servlets bound by path. In this mode, the above .extensions, .selectors and .methods service properties are taken into account to select such servlets.<br>If this property is not set to true the behavior is unchanged from previous versions and only the .paths property is considered when selecting such servlets.<br>The special value .EMPTY. can be used for the .selectors and .extensions properties to require the corresponding request values to be empty for the servlet to be selected. |
| sling.servlet.prefix | The prefix or numeric index to make relative paths absolute. If the value of this property is a number (int), it defines the index of the search path entries from the resource resolver to be used as the prefix. The defined search path is used as a prefix to mount this servlet. The number can be -1 which always points to the last search entry. If the specified value is higher than the highest index of the search paths, the last entry is used. The index starts with 0. If the value of this property is a string and parseable as a number, the value is treated as if it would be a number. If the value of this property is a string starting with "/", this value is applied as a prefix, regardless of the configured search paths! If the value is anything else, it is ignored. If this property is not specified, it defaults to the default configuration of the sling servlet resolver |
| sling.core.servletName | The name with which the servlet should be registered. This registration property is optional. If one is not explicitly set, the servlet's name will be determined from either the property component.name, service.pid or service.id (in that order). This means that the name is always set (as at least the last property is always ensured by OSGi) |

Example Code:



```
@Component(service = { Servlet.class })
@SlingServletResourceTypes(
    resourceTypes="/apps/my/type",
    methods= "GET",
    extensions="html",
    selectors="hello")
public class MyServlet extends SlingSafeMethodsServlet {

    @Override
    protected void doGet(SlingHttpServletRequest request, SlingHttpServletResponse response) throws
        ...
    }
}
```

# Exercise 3: Create a Sling servlet

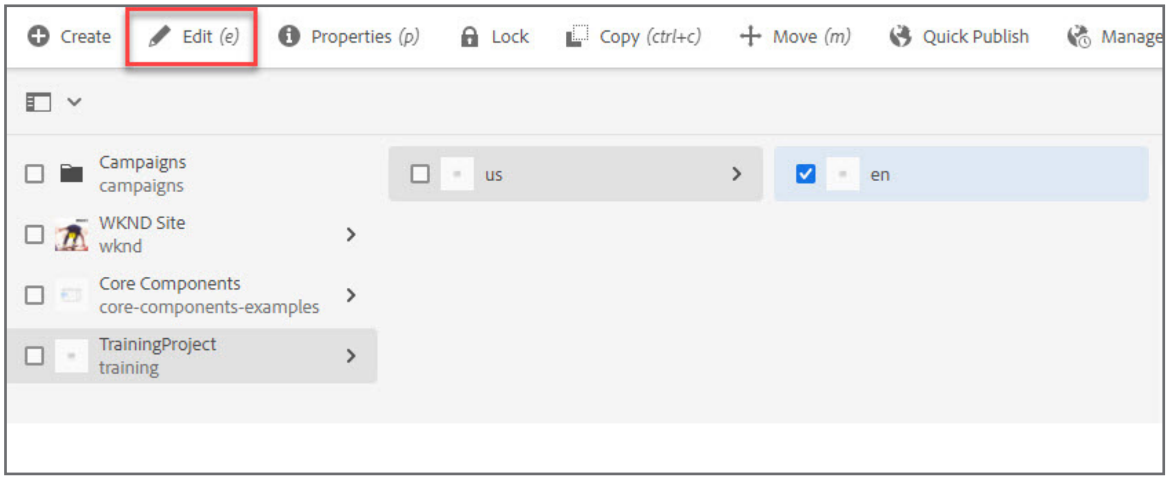
In this exercise, you will write a servlet that serves only doGet requests with a specific selector or resource type. The response shall contain the JCR repository properties as JSON.

This exercise includes the following tasks:

1. Create a servlet with a resource type
2. Use a selector to trigger the servlet

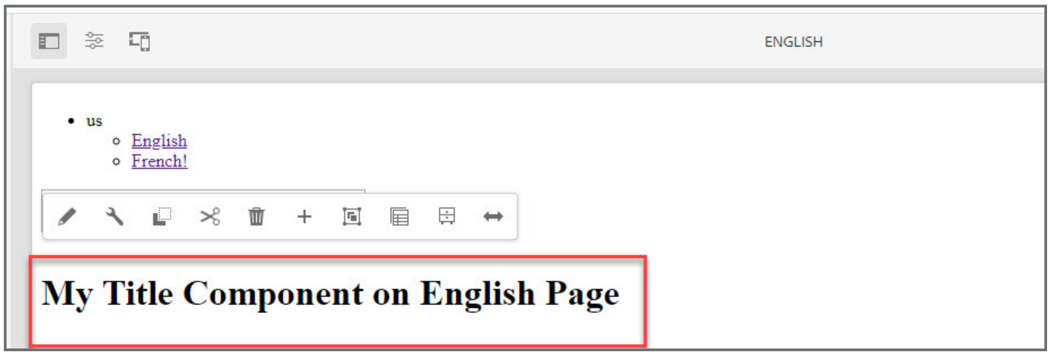
Task 1: Create a servlet with a resource type

1. In AEM, navigate to **Sites** > **TrainingProject** >**us** and select the **en** page.
2. Click **Edit**, as shown: The English page opens in edit mode.



3. Click the Components icon.

4. Drag the **Title** component to the **en** Page and add some dummy text **My Title Component on English Page** ,as shown:



My Title Component on English Page

5. Click **Done** to close the **Title** window.

Next, you will create a Sling servlet that intercepts the resourceType for the title component.

6. In your IDE, navigate to **core** > **src** > **main/java/com/adobe/training/core/servlets**.
7. Right-click the **servlets** folder and select **New File**.
8. Name the file as **TitleSlingServlet.java**.
9. Copy the contents from **Exercise_Files-EC** under **/core/src/main/java/com/adobe/training/core/ servlets/TitleSlingServlet.java** to **TitleSlingServlet.java** in your IDE.
10. Observe the code, as shown:

```
:
package com.adobe.training.core.servlets;
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import org.apache.sling.api.SlingHttpServletRequest;
import org.apache.sling.api.SlingHttpServletResponse;
import org.apache.sling.api.servlets.SlingSafeMethodsServlet;
import org.apache.sling.servlets.annotations.SlingServletResourceTypes;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;
import com.day.cq.wcm.api.Page;
import com.day.cq.wcm.api.PageManager;
import com.day.cq.wcm.api.PageManagerFactory;
/**
 * TitleSlingServlet can use resourceType, selector, method, and extension to bind to URLs
 *
 * Example URL: http://localhost:4502/content/training/us/en.html
 * Example URL with selector: http://localhost:4502/content/training/us/en.foobar.html
 *
 */
@Component(service = { Servlet.class })
@SlingServletResourceTypes(
       resourceTypes=TitleSlingServlet.RESOURCE_TYPE,
      selectors="foobar",
       extensions="html")
public class TitleSlingServlet extends SlingSafeMethodsServlet {
                    private static final long serialVersionUID = 1L;
                    protected static final String RESOURCE_TYPE = "training/components/title";
                    //Get a PageManager instance from the factory Service
          @Reference private PageManagerFactory pageManagerFactory;

          @Override
  protected void doGet(SlingHttpServletRequest request, SlingHttpServletResponse response) throws ServletException, IOException {
                    //Use pageManagerFactory to get page manager from the request.resourceResolver
                    PageManager pm = pageManagerFactory.getPageManager(request.getResourceResolver());
                    //Use the PageManager to find the containing page of the resource (component)
          Page curPage = pm.getContainingPage(request.getResource());

                    //Verify the page exists and it is a site page and not an XF
          if(curPage != null && !curPage.getName().equals("master")) {
                              String responseStr = "";
                              response.setHeader("Content-Type", "text/html");
h1>";                         responseStr = "<h1>Sling Servlet injected this title on the " + curPage.getName() + " page.</
                              response.getWriter().print(responseStr);
                              response.getWriter().close();
                    }
  }
}


  }
}
```

11. Save the changes.
12. Open a command prompt to the location of your Maven project. For example: **C:/adobe/< myproject >**

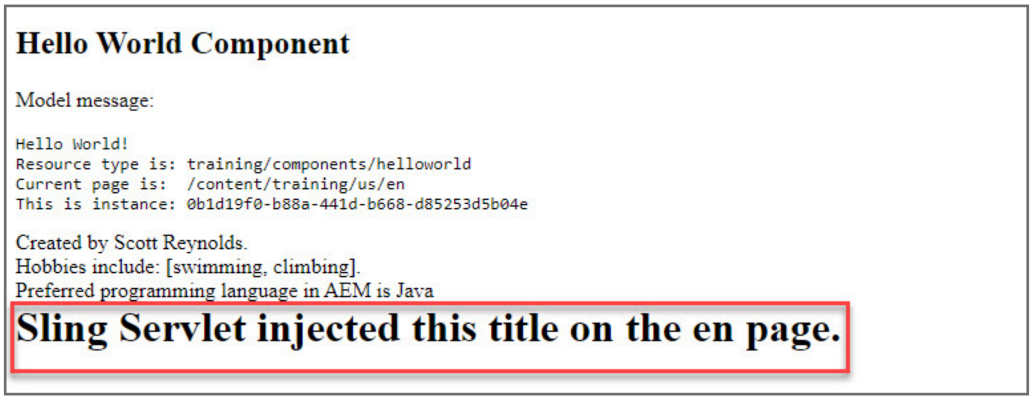
**Note**: If you are using an IDE with an integrated terminal such as Visual Studio Code, you can run your Maven commands there instead since it is already open to your project.

13. In the command prompt run the command:

```
$ mvn clean install -Padobe-public -PautoInstallSinglePackage
```

Your project has now successfully installed into your local AEM Server

14. Open http://localhost:4502/content/training/us/en.html and see the output, as shown:



**Hello World Component**

Model message:

```
Hello World!
Resource type is: training/components/helloworld
Current page is:  /content/training/us/en
This is instance: 0b1d19f0-b88a-441d-b668-d85253d5b04e
```

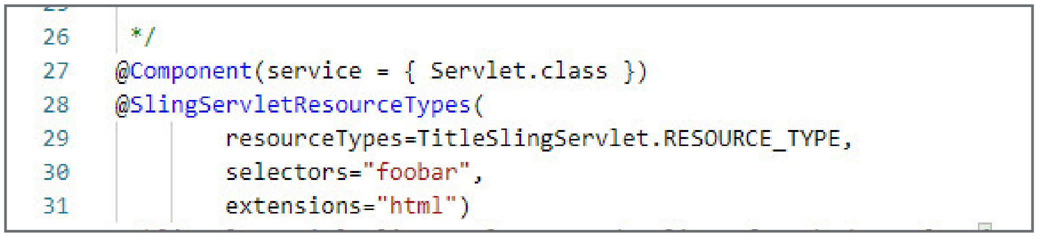
Created by Scott Reynolds.
Hobbies include: [swimming, climbing].
Preferred programming language in AEM is Java

**Sling Servlet injected this title on the en page.**

## Task 2: Use a selector to trigger the servlet

In this task you will add a selector to the servlet so that the servlet is only injected to the title component when the selector is also added.

1. In your IDE**,** edit **TitleSlingServlet**(**core** > **src** > **main/java/com/adobe/training/core/servlets**) to change the servlet implementation by uncommenting line 30 (by deleting //), as shown:

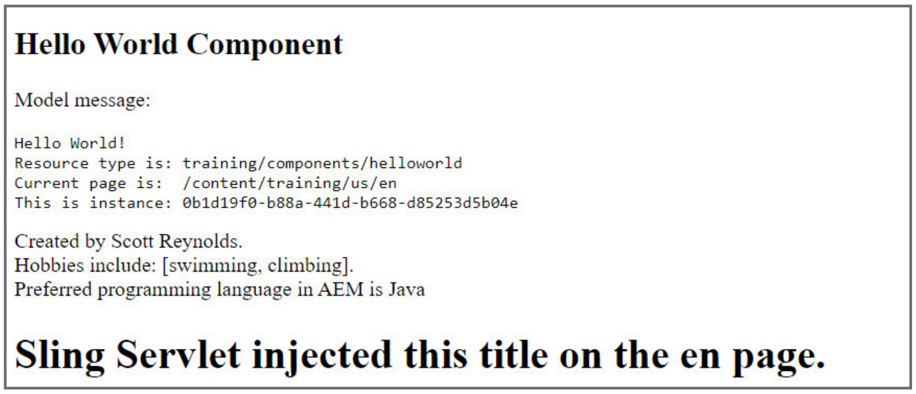


```
26   */
27  @Component(service = { Servlet.class })
28  @SlingServletResourceTypes(
29          resourceTypes=TitleSlingServlet.RESOURCE_TYPE,
30          selectors="foobar",
31          extensions="html")
```

2. Save the changes.
3. In the command prompt run the command:

```
$ mvn clean install -Padobe-public -PautoInstallSinglePackage
```

4. Go to http://localhost:4502/content/training/us/en.html and notice how the servlet is no longer being injected.
5. To see the servlet on the page, go to http://localhost:4502/content/training/us/en.foobar.html which includes the foobar selector which will now show the servlet injected::



**Hello World Component**

Model message:

```
Hello World!
Resource type is: training/components/helloworld
Current page is:  /content/training/us/en
This is instance: 0b1d19f0-b88a-441d-b668-d85253d5b04e
```

Created by Scott Reynolds.
Hobbies include: [swimming, climbing].
Preferred programming language in AEM is Java

**Sling Servlet injected this title on the en page.**

## References

For more information on OSGi, refer:

https://www.osgi.org/developer/architecture/

http://felix.apache.org/documentation/subprojects/apache-felix-service-component-runtime.html

OSGi Components – Simply Simple – Part I

https://blog.osoco.de/2015/08/osgi-components-simply-simple-part-i/