1 aem history2 maven3 archtype4 project creation 5 project integration with eclipse6 components(basic and structure)7 templates(static and editable)8 dialog creation(basic and touch)9 multifield dialogs10 page creation11 sling models12 sling servlets13 sightly14 osgi15 restful service16 database connectiity17 crud operations18 system user19 schdulers20 node creation21 dependencies and embed22 event listner23 content fragments and experience fragments24 msm25 query builder26 work flows and custom work flow

**OSGi Components and Services**

 🡪OSGi Components are different from AEM authoring component

🡪OSGi stands for **Open Service Gateway initiative**. It is a Java framework for developing and deploying **modular software programs and libraries**.

OSGi has two parts —

1. The first part is a specification for modular components called **bundles**, which are commonly referred to as plug-ins. The specification defines an infrastructure for a bundle’s life cycle and determines how bundles will interact.
2. The second part of OSGi is a Java Virtual Machine (JVM)-level **Service Registry ,** that bundles can use to publish, discover and bind to services in a service-oriented architecture (SOA). The components and services can be dynamically installed, activated, deactivated, updated and uninstalled.

 AEM uses Apache Felix implementation. A bundle is the smallest unit of the modularization which means in OSGi, a software component is a bundle.

OSGi Services and Components:

A service is an object that is registered in the OSGi Service Registry and can be looked up using its interface name(s)

A **Component** tends to be an object whose lifecycle is managed, usually by a component framework such as **Declarative Services (DS)**, **Blueprint** or **iPOJO**.

SGi Component — FetchTODO

As we discussed earlier, an OSGi component is the entity whose lifecycle will be managed by the OSGi container. To make a class act as component, we annotate it with @Component. We can start, stop and configure the component using Apache Felix Web Console.

Create a new class org.redquark.aem.tutorials.core.components.FetchTODO and put the following code in it -

package org.redquark.aem.tutorials.core.components;

import org.apache.commons.lang3.StringUtils;

import org.apache.http.HttpEntity;

import org.apache.http.HttpResponse;

import org.apache.http.client.HttpClient;

import org.apache.http.client.methods.HttpGet;

import org.apache.http.impl.client.HttpClientBuilder;

import org.apache.http.util.EntityUtils;

import org.osgi.service.component.annotations.Activate;

import org.osgi.service.component.annotations.Component;

import org.osgi.service.component.annotations.Reference;

import org.redquark.aem.tutorials.core.services.WriteTODOService;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.io.IOException;

import java.time.LocalDateTime;

import static org.redquark.aem.tutorials.core.components.FetchTODO.COMPONENT\_NAME;

import static org.redquark.aem.tutorials.core.constants.AppConstants.TODO\_ENDPOINT;

import static org.redquark.aem.tutorials.core.constants.AppConstants.TODO\_THREAD\_SLEEP\_TIME;

/\*\*

\* @author Anirudh Sharma

\* <p>

\* This component fetches to-do list for different users from external API and store them

\* in the JCR repository

\*/

@Component(immediate = true, name = COMPONENT\_NAME)

public class FetchTODO {

private static final String TAG = FetchTODO.class.getSimpleName();

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

protected static final String COMPONENT\_NAME = "Fetch Todo Component";

@Reference

WriteTODOService writeTODOService;

@Activate

protected void activate() {

LOGGER.info("{}: {} - activated", TAG, COMPONENT\_NAME);

Runnable task = () -> {

try {

Thread.sleep(TODO\_THREAD\_SLEEP\_TIME);

while (!Thread.currentThread().isInterrupted()) {

// Get output from the API

String todoData = fetchData();

// Call the OSGi service which will write data into the repository

writeTODOService.writeData(todoData);

LOGGER.info("{}: Trying to write TODO data in repository at: {}", TAG, LocalDateTime.now());

}

} catch (InterruptedException e) {

LOGGER.error("{}: Exception occurred: {}", TAG, e.getMessage());

}

};

// Create a new Thread with the above Runnable

Thread todoThread = new Thread(task);

// Set the name of the thread

todoThread.setName("TODO Thread");

// Start the thread

todoThread.start();

}

private String fetchData() {

HttpClient httpClient = HttpClientBuilder.create().build();

// Output in String format

String output = StringUtils.EMPTY;

try {

// GET Request

HttpGet request = new HttpGet(TODO\_ENDPOINT);

// Set the API media type in http accept header

request.setHeader("accept", "application/json");

// Send the request, It will immediately return the response in HttpResponse object

HttpResponse response = httpClient.execute(request);

// Get the status code

int statusCode = response.getStatusLine().getStatusCode();

if (statusCode != 200) {

throw new RuntimeException(TAG + ": Failed with error code: " + statusCode);

}

// Pull back the response object

HttpEntity httpEntity = response.getEntity();

// Getting output in String

output = EntityUtils.toString(httpEntity);

} catch (IOException e) {

LOGGER.error("{}: Exception occurred: {}", TAG, e.getMessage());

}

return output;

}

}

In the above code, we are creating a new OSGi component by annotating the class with @Component with two properties —

immediate — This signifies that as soon as the bundle is activated, the component will be activated.

name — User specified name of the component

**@Reference** — This identifies the annotated member or parameter as a reference of a Service Component. in simpler words, it will inject the reference of the field (much like @Autowired in Spring). Here, we are getting reference of WriteTODOService (which we will create later in this tutorial).

**@Activate** Method annotated with this annotation executes as soon as the Component becomes active. In this method, we are creating a java.lang.Runnable task which is getting the data from the API which is mentioned in the method fetchData(). The method fetchData() is a simple method which is calling an endpoint and receiving data in response.

Once we get data, we are calling WriteTODOService which is responsible for storing the API data in JCR.

What is HTL?

HTL stands for HTML Template Language. As the name suggests, it is a template language that lets us perform minor logical or computational operations in the HTML code itself. HTL uses an expression language to insert content in the rendered output. It also uses HTML5 data attributes to define conditions and iterations in the markup.

### Advantages of HTL

<h1>HTL Code Snippets</h1>

<h2><u>Example 1:</u></h2>

<h3>Title of the page is: ${currentPage.title}</h3>

<h2><u>Example 2:</u></h2>

<h3>Sling Resource Type property of the component node: ${properties.sling:resourceType}</h3>

<h2><u>Example 3:</u></h2>

<h3>Array join: ${['Batman', 'Superman', 'Flash'] @ join='-'}</h3>

<h2><u>Example 4:</u></h2>

<sly data-sly-test="${wcmmode.edit}">

<h3>We are in the edit mode</h3>

</sly>

<sly data-sly-test="${wcmmode.preview}">

<h3>We are in the preview mode</h3>

</sly>

<h2><u>Example 5:</u></h2>

<sly data-sly-list="${currentPage.parent.listChildren}">

<ul>

<li>

<h3>${item.title}</h3>

</li>

</ul>

</sly>

<h2><u>Example 6:</u></h2>

<h3 data-sly-text="${currentPage.title}">This text will be replaced.</h3>

<h2><u>Example 7:</u></h2>

<h3 data-sly-attribute.title=${currentPage.title}>Attributes are set for this element</h3>

<h2><u>Example 8:</u></h2>

<h3 data-sly-repeat="${currentPage.parent.listChildren}">${item.name}</h3>

<h2><u>Example 9:</u></h2>

<div data-sly-resource="${'text' @ resourceType='aemtutorials/components/content/text'}"></div>

<h2><u>Example 10:</u></h2>

<sly data-sly-include="index.html" />

<h2><u>Example 11:</u></h2>

<sly data-sly-template.object>

This can be called by using 'call'

</sly>

<h3 data-sly-call="${object}"></h3>

# Sling Models

## Sling Models

if we have a direct way to map your JCR node properties to our model class?

Sling Models are **pure** POJOs which map Sling objects (resources, request objects etc.).

Since Sling Models are annotation-driven Plain Old Java Objects (POJOs), annotations are used a lot. They allow us to map resource properties, assign default values, inject OSGi services and much more.

The Resource is one of the central parts of Sling. Extending from JCR's Everything is Content, Sling assumes Everthing is a Resource.

// Defines which type of adaptable we are dealing with. Generally, there are two -

// 1. Resource

// 2. SlingHttpServletRequest

adaptables

// Defines under which type(s) the model implementation should be registered in the Models Adapter Factory.

adapters

// With which resource this model class is associated with. In our case, it is associated with the "User" component

resourceType

// Specifies attributes that are required to be injected by default

defaultInjectionStrategy

 









