Eclipse-RCP points:

**Some Facts:**

* OSGi is a set of specifications. Its core specification, defines a component and service model for Java.
* Eclipse Equinox is the reference implementation of the base OSGi specification. It is also the runtime environment on which Eclipse applications are based.
* The OSGi specification defines a bundle as the smallest unit of modularization, i.e., in OSGi a software component is a bundle. The Eclipse programming model typically calls them plug-in but these terms are interchangeable.
* Each plug-in has its own classloader.
* Technically, OSGi plug-ins are .jar files with additional meta information in the META-INF/MANIFEST.MF file. This file is called the manifest file.
* The combination of Bundle-SymbolicName and Bundle-Version uniquely identifies a plug-in.
* Sometimes you need to access information about the bundle, e.g., the bundle version or the bundle name. Access to the bundle and its bundleContext is performed via the Bundle and BundleContext class.

Bundle bundle = FrameworkUtil.getBundle(this.getClass());

BundleContext bundleContext = FrameworkUtil.getBundle(this.getClass()).getBundleContext();

* By default, the Eclipse framework uses a single thread to run all the code instructions UI or Non-UI.
* sdf

----------------------------------------------------------------------------------------------------------------------------------------------------------------

* How to access a resource e.g. image, file inside Eclipse plug-in:

You can access files in your plug-in via two different approaches:

* via the FileLocator class or
* via an Eclipse specific URL.

Both approaches require that your plug-in defines a dependency to the org.eclipse.core.runtime plug-in.

Both approaches return a URL which can be used to get an InputStream.

* org.eclipse.core.runtime.FileLocator

This class contains a collection of helper methods for finding files in bundles.

This class can only be used if the OSGi plugin is available

Bundle bundle = FrameworkUtil.getBundle(this.getClass());

OR

Bundle bundle = Platform.getBundle(PLUGIN\_ID);

Path filePath = new Path(String fileLocation);

URL fileUrl = FileLocator.find(bundle, filePath, null);

InputStream in = fileUrl.openConnection.getInputStream();

* Eclipse specific Url:

URL fileUrl = new URL("platform:/plugin/your\_bundle-symbolicname/path\_to\_file");

InputStream in = fileUrl.openConnection.getInputStream();

----------------------------------------------------------------------------------------------------------------------------------------------------------------

* To use image provided as resource inside Eclipse plug-in:
* Use AbstractUIPlugin

ImageDescriptor imgDesc = AbstractUIPlugin.imageDecriptorFromPlugin(PLUGIN\_ID, imgFilePath);

Image img = imgDesc.createImage();

* Create Image programatically using Eclipse specific Url or FileLocator.

IPath uriPath = new Path("/plugin").append(PLUGIN\_ID).append(imageFilePath);

URI uri = new URI("platform", null, uriPath.toString(), null);

URL url = uri.toURL();

URL fullPathString = FileLocator.find(url);

ImageDescriptor imgDesc = ImageDescriptor.createFromURL(url);

Image img = imgDesc.createImage();

-------------------------------------------------------------------------------------------------------------------------------------------------------------

* To access Image/ImageDescriptor provided/used by Eclipse Workbench
* Eclipse workbench creates and stores images in ImageRegistry through class WorkbenchImages.
* This images can be accessed by calling following methods and providing image path from ISharedImages.

**PlatformUI.getWorkbench().getSharedImages().getImage(ISharedImages.IMG\_ETOOL\_DEF\_PERSPECTIVE);**

* This process returns a NULL if the given image path not available in WorkbenchImages ImageRegistry.
* Use this process to call only Eclipse specific images path provided by ISharedImages constants.

**AbstractUIPlugin.imageDescriptorFromPlugin(PLUGIN\_ID, ISharedImages.IMG\_ETOOL\_DELETE).createImage();**

* This process returns image if available in WorkbenchImages ImageRegistry else try to create image using the PLUGIN\_ID and image path using ImageDescriptor.createFromURL(url).
* If image path not available relative to PLUGIN\_ID then returns NULL.

* package org.eclipse.ui.internal;

public class SharedImages implements ISharedImages {

Image image = WorkbenchImages.getImage(symbolicName);

ImageDescriptor desc = WorkbenchImages.getImageDescriptor(symbolicName);

}

* Public class WorkbenchImages creates can contain all images in ImageRegistry.
* To access images provided by underlying Operating System
* Images provide by underlying O.S can be accessed by the SWT using following methods:

**Display.getDefault().getSystemImage(SWT.ICON\_INFORMATION);**

**WorkbenchSite, Services**

**org.eclipse.ui.services.IServiceLocator**

A component with which one or more services are registered. The services can be retrieved from this locator using some key -- typically the class representing the interface the service must implement. For example:

IHandlerService service = workbenchWindow.getService(IHandlerService.class);

This interface is not to be implemented or extended by clients.

**org.eclipse.ui.IWorkbenchSite extends IAdaptable, IShellProvider, IServiceLocator**

The common interface between the workbench and its parts, including pages within parts.

The workbench site supports a few services by default. If these services are used to allocate resources, it is important to remember to clean up those resources after you are done with them. Otherwise, the resources will exist until the workbench site is disposed. The supported services are:

ICommandService

IContextService

IHandlerService

IBindingService

public IWorkbenchPage getPage();

public ISelectionProvider getSelectionProvider();

public Shell getShell();

public IWorkbenchWindow getWorkbenchWindow();

public void setSelectionProvider(ISelectionProvider provider);

**org.eclipse.jface.viewers.ISelectionProvider**

Interface common to all objects that provide a selection.

**org.eclipse.jface.viewers.ISelection**

Interface for a selection.

**org.eclipse.jface.viewers.ISelectionChangedListener**

A listener which is notified when a viewer's selection changes.

**org.eclipse.ui.IWorkbenchPartSite extends IWorkbenchSite**

The primary interface between a workbench part and the workbench.

public String getId();

public String getPluginId();

public String getRegisteredName();

public void registerContextMenu(String menuId, MenuManager menuManager,

ISelectionProvider selectionProvider);

public void registerContextMenu(MenuManager menuManager,

ISelectionProvider selectionProvider);

public IWorkbenchPart getPart();

**Abstract class org.eclipse.ui.internal.PartSite implements IWorkbenchPartSite**

PartSite is the general implementation for an IWorkbenchPartSite. A site maintains the context for a part, including the part, its pane, active contributions, selection provider, etc. Together, these components make up the complete behavior for a part as if it was implemented by one person. The PartSite lifecycle is as follows:

1. a site is constructed
2. a part is constructed and stored in the part
3. the site calls part.init()
4. a pane is constructed and stored in the site
5. the action bars for a part are constructed and stored in the site
6. the pane is added to a presentation
7. the SWT widgets for the pane and part are created
8. the site is activated, causing the actions to become visible

**org.eclipse.ui.IEditorSite extends IWorkbenchPartSite**

The primary interface between an editor part and the workbench.

public IEditorActionBarContributor getActionBarContributor();

public IActionBars getActionBars();

public void registerContextMenu(MenuManager menuManager,

ISelectionProvider selectionProvider, boolean includeEditorInput);

public void registerContextMenu(String menuId, MenuManager menuManager,

ISelectionProvider selectionProvider, boolean includeEditorInput);

**org.eclipse.ui.internal.EditorSite extends PartSite implements IEditorSite**

An editor container manages the services for an editor.

**org.eclipse.ui.IViewSite extends IWorkbenchPartSite**

The primary interface between a view part and the workbench.

public IActionBars getActionBars();

public String getSecondaryId();

**org.eclipse.ui.ViewSite extends PartSite implements IViewSite**

A view container manages the services for a view.

**Action, ToolBars and Contribution.**

**Action**

**org.eclipse.jface.action.IAction**

An action represents the non-UI side of a command which can be triggered by the end user.

Actions are typically associated with buttons, menu items, and items in tool bars.

**The controls for a command are built by some container, which furnished the context where these controls appear and configures them with data from properties declared by the action.**

**When the end user triggers the command via its control, the action's run method is invoked to do the real work.**

Actions support a predefined set of properties (and possibly others as well).

Clients of an action may register property change listeners so that they get notified whenever the value of a property changes

**org.eclipse.jface.action.AbstractAction extends EventManager implements IAction**

Some common functionality to share between implementations of IAction. This functionality deals with the property change event mechanism.

**org.eclipse.jface.action.Action extends AbstractAction**

The standard abstract implementation of an action.

Subclasses must implement the **IAction.run** method to carry out the action's semantics.

**Contribution**

**org.eclipse.jface.action.IContributionManager**

A contribution manager organizes contributions to such UI components as menus, toolbars and status lines.

A contribution manager keeps track of a list of contribution items.

The IContributionManager interface provides general protocol for adding, removing, and retrieving contribution items. It also provides convenience methods that make it convenient to contribute actions. This interface should be implemented by all objects that wish to manage contributions.

public void **add**(IContributionItem item);

Adds a contribution item to this manager.

public void **add**(IAction action);

Adds an action as a contribution item to this manager.

Equivalent to add(new ActionContributionItem(action)).

**org.eclipse.jface.action.ContributionManager implements IContributionManager**

Abstract base class for all contribution managers, and standard implementation of IContributionManager. This class provides functionality common across the specific managers defined by this framework.

@Override

public void add(IAction action) {

**add**(new ActionContributionItem(action));

}

@Override

public void **add**(IContributionItem item) {

if (allowItem(item)) {

contributions.add(item);

itemAdded(item);

}

}

**org.eclipse.jface.action.IContributionItem**

A contribution item represents a contribution to a shared UI resource such as a menu or tool bar.

More generally, contribution items are managed by a contribution manager.

For instance, in a tool bar a contribution item is a tool bar button or a separator.

In a menu bar a contribution item is a menu, and in a menu a contribution item is a menu item or separator.

A contribution item can realize itself in different SWT widgets, using the different fill methods. The same type of contribution item can be used with a MenuBarManager, ToolBarManager, CoolBarManager, or a StatusLineManager.

**A ContributionItem** cannot be shared between different **ContributionManagers.**

**org.eclipse.jface.action.ContributionItem implements IContributionItem**

An abstract base implementation for contribution items.

**org.eclipse.jface.action.ActionContributionItem extends ContributionItem**

A contribution item which delegates to an action.

public **ActionContributionItem**(IAction action) {

super(action.getId());

this.action = action;

}

**org.eclipse.jface.action.IToolBarManager extends IContributionManager**

The IToolBarManager interface provides protocol for managing contributions to a tool bar. It extends IContributionManager but does not declare any new members; it exists only to increase the readability of code using tool bars.

**org.eclipse.jface.action.ToolBarManager extends ContributionManager implements IToolBarManager**

A tool bar manager is a contribution manager which realizes itself and its items in a tool bar control.

**org.eclipse.jface.action.ICoolBarManager extends IContributionManager**

The ICoolBarManager interface provides protocol for managing contributions to a cool bar.

A cool bar manager delegates responsibility for creating child controls to its contribution items by calling IContributionItem.fill(CoolBar, int).

**org.eclipse.jface.action.CoolBarManager extends ContributionManager implements ICoolBarManager**

A cool bar manager is a contribution manager which realizes itself and its items in a cool bar control.

**org.eclipse.jface.action.IMenuManager extends IContributionManager, IContributionItem**

The IMenuManager interface provides protocol for managing contributions to a menu bar and its sub menus.

An IMenuManager is also an IContributionItem, allowing sub-menus to be nested in parent menus.

**org.eclipse.jface.action.MenuManager extends ContributionManager implements IMenuManager**

A menu manager is a contribution manager which realizes itself and its items in a menu control; either as a menu bar, a sub-menu, or a context menu.

**org.eclipse.swt.widgets.Widget**

This class is the abstract superclass of all user interface objects.

Widgets are created, disposed and issue notification to listeners when events occur which affect them.

**org.eclipse.swt.widgets.Item extends Widget**

This class is the abstract superclass of all non-windowed user interface objects that occur within specific controls. For example, a tree will contain tree items.

**Subclasses: MenuItem, TabItem, TableItem, ToolItem, TreeItem, CoolItem,**

**TableColumn, TreeColumn, CTabItem**

**org.eclipse.swt.widgets.ToolBar extends Composite**

Instances of this class support the layout of selectable tool bar items.

The item children that may be added to instances of this class must be of type ToolItem.

Note that although this class is a subclass of Composite, it does not make sense to add Control children to it, or set a layout on it.

**org.eclipse.swt.widgets.CoolBar extends Composite**

Instances of this class provide an area for dynamically positioning the items they contain.

The item children that may be added to instances of this class must be of type CoolItem.

Note that although this class is a subclass of Composite, it does not make sense to add Control children to it, or set a layout on it.

**org.eclipse.swt.widgets.Menu extends Widget**

Instances of this class are user interface objects that contain menu items.

**Adding menus, toolbar and popup menus to the Eclipse IDE (Menus and Toolbars)**

**https://www.eclipse.org/articles/Article-RCP-3/tutorial3.html**

Eclipse framework use the commands framework to contribute actions to the user interface.

A command in Eclipse is a declarative description of a component and is independent from the implementation details.

The behavior of a command is defined via a handler.

Eclipse 3.x API uses extension points to define commands. Three extensions points play a role here. The following extension points are relevant to define a menu or toolbar contribution for an Eclipse plug-in.

org.eclipse.ui.command -> Declarative description of the component

org.eclipse.ui.handlers -> Defines the behavior, e.g., the Java class which should be called

org.eclipse.ui.menu -> where and how should the command be included in the user interface,

Ex., menu, pop-up-menu, toolbar, etc.

The org.eclipse.ui.menu extension point is used to define where the command is visible in the application For example, you could define that the command is visible in the **application menu** or in a **view menu**.

|  |  |  |
| --- | --- | --- |
| Contribution to | Description | Uri |
| Application menu | Displays the command in the menu of the application | menu:org.eclipse.ui.main.menu |
| Application toolbar | displays the command in the toolbar of the application | toolbar:org.eclipse.ui.main.toolbar |
| View toolbar | displays the command in the toolbar of the view | toolbar:viewId - For example to display a menu to view with the Id "View1" use "toolbar:View1". |
| Context menu / pop-up | Command is displayed in a context menu, e.g., right mouse click on an object | popup:ID |

In Eclipse jargon, "action bar" is a catch-all term for menus, toolbars, and status bars. The ActionBar Advisor handles creating Actions within these locations. A plug-in can also contribute actions dynamically with its plugin.xml file.

Letting people configure all the menus and toolbars was one of the first, most basic requirement of the Rich Client Platform. There are two ways to add menus and toolbars in an RCP application:

* Extend the ActionBarAdvisor class in Java code, or
* Use the org.eclipse.ui.actionSets extension in the plug-in manifest

Extending ActionBarAdvisor is the only way to reference built-in Workbench actions. If you're trying to use the same code in a stand-alone RCP application and a plug-in for the Eclipse IDE, the best practice is to use org.eclipse.ui.actionSets as much as possible.

**Eclipse Background Processing**

By default, the Eclipse framework uses a single thread to run all the code instructions.

This thread runs the event loop for the application. \It is the only thread that is allowed to interact with the user interface (UI).

It is called the *main thread*. Sometimes it is also called the *UI thread*, but this is a misnomer as it handles all events not only the ui events.

If another thread tries to update the UI, the Eclipse framework throws an SWTException exception.

All events in the user interface are executed one after another.

Blocking the user interaction is considered a bad practice. Therefore it is important to perform all long running operations in a separate thread.

As only the main thread is allowed to modify the user interface, the Eclipse framework provides ways for a thread to synchronize itself with the main thread. It also provides the Eclipse Jobs framework which allows you to run operations in the background and providing feedback of the job status to the Eclipse platform.

---------------------------------------------------------\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*--------------------------------------------------------------

* Usage of Job API to run long non-UI operations in background with Progress of operation displays in UI

Jobs are units of runnable work that can be scheduled to be run with the job manager. Once a job has completed, it can be scheduled to run again (jobs are reusable).

org.eclipse.core.runtime.jobs.Job //Abstract class

org.eclipse.core.runtime.jobs.IJobManager

org.eclipse.core.runtime.IProgressMonitor

org.eclipse.core.runtime.IStatus

final Job exportJob = new Job(“Export Result”) {

@Override

**protected** IStatus run(IProgressMonitor monitor) {

//Do Some Long Operation

monitor.worked(10); //Status of progress monitor on UI

**//To update the UI**

//Display.*getDefault*().asyncExec()

Display.*getDefault*().syncExec(new Runnable(){

@Override

**public** void run(){

//Operation on User Interface

// Update Text Field

}

};

return Status.OK\_STATUS;

}

};

exportJob.setUser(true);

exportJob.schedule();

Job creates a non-UI thread to run long running task.

To update some widget on UI from non-UI thread we have to use syncExec() and asyncExce().

-------------------------------------------------\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-------------------------------------------------------------------

* The Display class which provides the syncExec() and asyncExec() methods to update the user interface from another non-UI thread.

**void org.eclipse.swt.widgets.Display.syncExec(Runnable runnable)**

Causes the run() method of the runnable to be invoked by the user-interface thread at the next reasonable opportunity. The thread which calls this method is suspended until the runnable completes.

**void org.eclipse.swt.widgets.Display.asyncExec(Runnable runnable)**

Causes the run() method of the runnable to be invoked by the user-interface thread at the next reasonable opportunity. The caller of this method continues to run in parallel, and is not notified when the runnable has completed.

--------------------------------------------------- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*--------------------------------------------------------------

* The UIJob is a Job that runs within the UI Thread via an asyncExec.

org.eclipse.ui.progress.UIJob extends Job //(Abstract Class)

final UIJob uiJob = new UIJob(Display.*getDefault*(), "Refresh Job"){

@Override

**public** IStatus runInUIThread(IProgressMonitor monitor){

//Some long running non-UI and UI activity.

**return** Status.***OK\_STATUS***;

}

}

uiJob.schedule(long delay);

* Schedule() will call run() which internally call runInUIThread() in asyncExec() of provided Display.
* User can also user == > new UI(“Refresh Job”)

In this case workbench Display will be used as PlatformUI.getWorkbench.getDisplay().

-------------------------------------------------------\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*---------------------------------------------------------

* JFace also provides mechanism to process long running task in background non-UI thread and display progress in UI using **ProgressMonitorDialog**.

org.eclipse.jface.dialog.ProgressMonitorDialog

org.eclipse.jface.operation.IRunnableWithProgress

org.eclipse.core.runtime.ProgressMonitor

ProgressMonitorDialog pmDialog = **new** ProgressMonitorDialog(Display.*getDefault*().getActiveShell());

IRunnableWithProgress runnableWithProgress = **new** IRunnableWithProgress() {

@Override

**public** **void** run(**final** IProgressMonitor monitor) **throws** InvocationTargetException,

InterruptedException {

//Some long running non-UI task

Monitor.worked(10);

**//To update the UI**

//Display.*getDefault*().asyncExec( () -> {

//Update Text box

});

monitor.done();

}

};

pmDialog.run(**true**, **true**, runnableWithProgress);

**Eclipse Modeling Framework (EMF)**

A data model, sometimes also called domain model, represents the data you want to work with.

For example, if you develop an online flight booking application, you might model your domain model with objects like Person, Flight and Booking etc.

A good practice is to model the data model of an application independently of the application logic or user interface. This approach leads to classes with almost no logic and a lot of properties, e.g., a Person class could have the firstName, lastName, Address properties, etc.

With EMF you define your domain model explicitly. This helps to provide clear visibility of the model.

EMF generates interfaces and a factory to create your objects; therefore, it helps you to keep your application clean from the individual implementation classes.

The Eclipse Modeling Framework (EMF) is a set of Eclipse plug-ins which can be used to model a data model and to generated code or other output based on this model.

EMF has a distinction between the meta-model and the actual model. The meta-model describes the structure of the model. A model is a concrete instance of this meta-model.

EMF allows the developer to create the meta-model via different means, e.g., XMI, Java annotations, UML or an XML scheme. It also allows to persist the model data; the default implementation uses a data format called XML Metadata Interchange.

The EMF meta-model consists of two parts; the **ecore** and the **genmodel** description files.

The ecore file contains the information about the defined classes.

The genmodel file contains additional information for the code generation, e.g., the path and file information. The genmodel file also contains the control parameter how the code should be generated.

The **ecore file** allows to define the following elements.

* EClass: represents a class, with zero or more attributes and zero or more references.
* EAttribute: represents an attribute which has a name and a type.
* EReference: represents one end of an association between two classes. It has flags to indicate if it represents a

containment and a reference class to which it points.

* EDataType: represents the type of an attribute, e.g., int, float or java.util.Date

The Ecore model shows a root object representing the whole model. This model has children which represent the packages, whose children represent the classes, while the children of the classes represent the attributes of these classes.

**OSGi Modularity**

An application consists of different parts, these are typically called *software components* or *software modules*.

These components interact with each other via an Application Programming Interface (API).

The API is defined as a set of classes and methods which can be used from other components. A component also has a set of classes and methods which are considered as internal to the software component.

Java 9 provides its own module system to describe software component dependencies.

The OSGi specification defines its own module system, which is a bit more flexible than the Java module system.

OSGi is a set of specifications. Its core specification, defines a component and service model for Java.

A practical advantage of OSGi is that every software component can define its API via a set of exported Java packages and that every component can specify its required dependencies.

The components and services can be dynamically installed, activated, de-activated, updated and de-installed.

The OSGi specification has several implementations, for example Eclipse Equinox, Knopflerfish OSGi or Apache Felix.

Eclipse Equinox is the reference implementation of the base OSGi specification. It is also the runtime environment on which Eclipse applications are based.

The OSGi specification defines a bundle as the smallest unit of modularization, i.e., in OSGi a software component is a bundle. The Eclipse programming model typically calls them *plug-in* but these terms are interchangeable.

A plug-in is a cohesive, self-contained unit, which explicitly defines its dependencies to other components and services. It also defines its API via Java packages.

Technically, OSGi plug-ins are *.jar* files with additional meta information in the *META-INF/MANIFEST.MF* file. This file is called the *manifest* file. It is part of the standard Java specification. The OSGi specification defines additional metadata for this file. According to the Java specification, any Java runtime must ignore unknown metadata. Therefore, OSGi plug-ins can be used without restrictions in other Java environments.

Bundle-RequiredExecutionEnvironment (BREE) -- Specify which Java version is required to run the plug-in. If this requirement is not fulfilled, then the runtime does not load the plug-in.

Bundle-ClassPath -- The Bundle-ClassPath specifies where to load classes from the bundle. The default is '.' which allows classes to be loaded from the root of the bundle. You can also add JAR files to it, these are called *nested JAR files*.

The combination of Bundle-SymbolicName and Bundle-Version uniquely identifies a plug-in.

OSGi recommends to use the following schema for versions in the Bundle-Version field identifier.

*<major>.<minor>.<patch/service>*

**Sometimes you need to access information about the bundle, e.g., the bundle version or the bundle name. Access to the bundle and its bundleContext is performed via the Bundle and BundleContext class.**

You can use the FrameworkUtil class from the OSGi framework to access the BundleContext for a class.

**Bundle bundle = FrameworkUtil.getBundle(this.getClass());**

**BundleContext bundleContext = FrameworkUtil.getBundle(this.getClass()).getBundleContext();**

In the *MANIFEST.MF* file a plug-in also defines its API via the Export-Package identifier. This allows you to define internal API, provisional API and API.

All packages which are not explicitly exported are not visible to other plug-ins.

All these restrictions are enforced via a specific OSGi classloader. Each plug-in has its own classloader. Access to restricted classes is not possible without using reflection.

Via the x-internal flag the OSGi runtime can mark an exported package as provisional. This allows other plug-ins to consume the corresponding classes, but indicates that these classes are not considered as official API.

An OSGi runtime allows the developer to mark Java packages as public, provisional or internal APIs. The internal API is private, therefore not visible. The provisional API are to test non-finalized APIs, therefore are visible but non-stable. The public API, or simply API, are the visible and stable API, that can be reused by other components.

**How To provide OSGi Service**

Step-1 : Create a plug-in.

Step-2 : Create an interface RegisterService (This interface acts as service provider).

Step-3 : Create an implementation class RegisterServiceImpl of RegisterService interface.

Step-4 : Annotate RegisterServiceImpl class with @Component annotation.

[org](eclipse-javadoc:%E2%98%82=com.airbus.eventdemo/C:%5C/Users%5C/ng61feb%5C/Desktop%5C/Softwares%5C/eclipse-oxygen%5C/eclipse%5C/configuration%5C/org.eclipse.osgi%5C/403%5C/0%5C/.cp%5C/lib%5C/annotations.jar%3Corg).[osgi](eclipse-javadoc:%E2%98%82=com.airbus.eventdemo/C:%5C/Users%5C/ng61feb%5C/Desktop%5C/Softwares%5C/eclipse-oxygen%5C/eclipse%5C/configuration%5C/org.eclipse.osgi%5C/403%5C/0%5C/.cp%5C/lib%5C/annotations.jar%3Corg.osgi).[service](eclipse-javadoc:%E2%98%82=com.airbus.eventdemo/C:%5C/Users%5C/ng61feb%5C/Desktop%5C/Softwares%5C/eclipse-oxygen%5C/eclipse%5C/configuration%5C/org.eclipse.osgi%5C/403%5C/0%5C/.cp%5C/lib%5C/annotations.jar%3Corg.osgi.service).[component](eclipse-javadoc:%E2%98%82=com.airbus.eventdemo/C:%5C/Users%5C/ng61feb%5C/Desktop%5C/Softwares%5C/eclipse-oxygen%5C/eclipse%5C/configuration%5C/org.eclipse.osgi%5C/403%5C/0%5C/.cp%5C/lib%5C/annotations.jar%3Corg.osgi.service.component).[annotations](eclipse-javadoc:%E2%98%82=com.airbus.eventdemo/C:%5C/Users%5C/ng61feb%5C/Desktop%5C/Softwares%5C/eclipse-oxygen%5C/eclipse%5C/configuration%5C/org.eclipse.osgi%5C/403%5C/0%5C/.cp%5C/lib%5C/annotations.jar%3Corg.osgi.service.component.annotations).Component

Identify the annotated class as a Service Component. The annotated class is the implementation class of the Component. This annotation is not processed at runtime by Service Component Runtime. It must be processed by tools and used to add a Component Description to the bundle.

As soon as @Component annotation added to RegisterServiceImpl class compiler generated an OSGI-INF folder with a .xml file containing Component information.

For each Component i.e. implementation class of Service interface a .xml file created in OSGI-INF folder.

All the @Component classes gets register as service to frameworks OSGi service registry.

Step-5 : Now to use the RegisterService first retrieve all the implementations of RegisterService.

**final** BundleContext bundleContext = FrameworkUtil.*getBundle*(RegisterService.class).getBundleContext();

**final** ServiceTracker<T, T> tracker =

**new** ServiceTracker<>(bundleContext, clazz, **null**);

**try**{

tracker.open();

**return** tracker.getService();

}**catch** (Exception e) {

**throw** **new** RuntimeException(e);

}**finally** {

tracker.close();

}

**final** ServiceTracker<T, T> tracker = **new** ServiceTracker<>(bundleContext, clazz, **null**);

**try**{

tracker.open();

**final** T[] services = tracker.getServices((T[])Array.*newInstance*(RegisterService.class, 1));

**if**(services.length == 1 && services[0] == **null**) {

**return** **new** ArrayList<>();

}**else** {

**return** Arrays.*asList*(services);

}

}**catch** (Exception e) {

**throw** **new** RuntimeException(e);

}**finally** {

tracker.close();

}

**EventHandler**