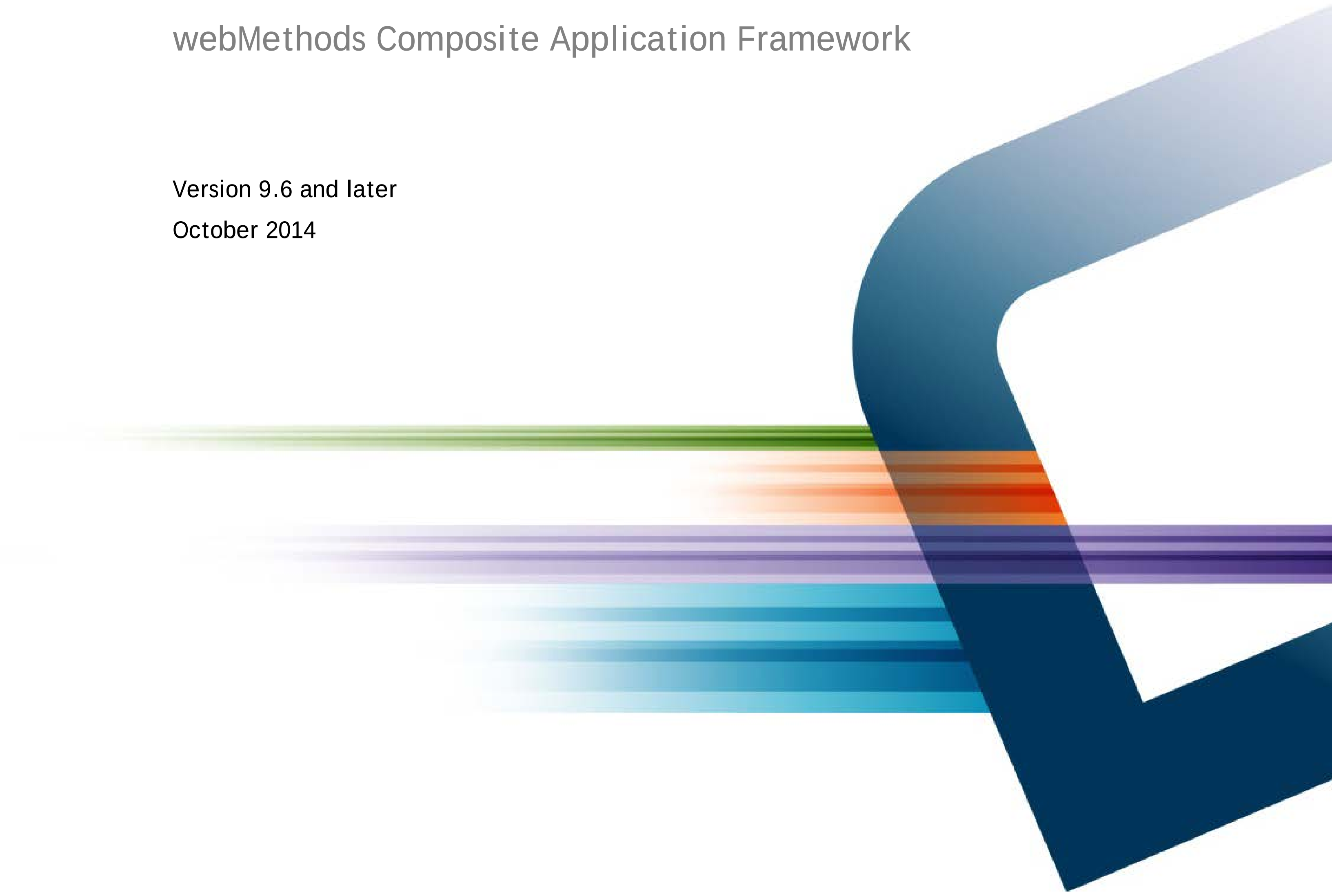
Dynamic Import of Views

For CAF and OpenCAF

webMethods Composite Application Framework

Version 9.6 and later

October 2014



CONTENTS

Rev: 10302014

Overview

This article describes how to create reusable views using CAF and JSF facelets, with examples provided by a sample application and several sample projects. It also provides guidelines on how to import reusable CAF views and use them inside your application.

Applicable Versions: 9.6 or later.

Required Fix Levels: MWS 9.6 Fix 3 or later.

The associated sample application and projects demonstrate how to implement a modularized web application in My webMethods Server using technologies such as CAF, OpenCAF with JSF facelets, OSGi, Blueprint, and Bndtools.

The following samples will be referenced in this discussion:

- wm_opencaf_main

  This is the main application that uses and displays all dynamic views.

- opencaf-module-api

  This provides an API for all dynamic modules.

- opencaf_module1

  This is module 1, which uses OpenCAF facelets (*.xhtml files) to implement sharable views. It is an OSGi fragment of wm_opencaf_main.

- module_cafview

  This is module 2 which uses CAF views (.view files) to implement sharable views. It is also an OSGi fragment of wm_opencaf_main.

- shared-module

  This is a simple jar file and bundle that contains facelets under the META-INF/resources folder. The wm_opencaf_main application declares a dependency on shared-module.

Two sample files are provided to accompany this article:

- dynamic_import_samples_binary.zip

  This zip file contains the deployable .war and .jar files for the application and projects described in this article. You can extract these files and deploy them directly to My webMethods Server and observe their behavior.

- dynamic_import_samples_src.zip

  This zip file contains the source files for the application and projects described in this article. You can extract the application and projects and then import them into Software AG Designer, where you can examine their contents and implementation or modify them to test your own solutions.

Deploying the Samples to My webMethods Server

To deploy these samples to My webMethods Server:

1. Start My webMethods Server.
2. Extract the contents of dynamic_import_samples_binary.zip to a temporary folder.
3. Copy the following files from the temporary folder to the *Software AG_directory*\ MWS\server\instanceName\deploy folder of your My webMethods Server instance and wait for each one to complete installation, as shown in the session console:

   opencaf-module-api.jar

   shared-module.jar

   wm_opencaf_main.war

   opencaf_module1.jar

   module_cafview.jar
4. Stop My webMethods Server.
5. Start My webMethods Server.

To verify operation, access these URLs:

Facelets: http://localhost:8585/opencaf.module.app

ImportTemplate view control: http://localhost:8585/opencaf.module.app.cafview

Importing the Project Source Files into Designer

To import the sample application project and other projects into your Designer workspace:

1. Extract the contents of dynamic_import_samples_binary.zip to a temporary folder.
2. In Designer: **File > Import**.
3. In the Import dialog box, click **General** and then **Existing Projects into Workspace**. Then click **Next**.
4. Click **Browse** and navigate to the temporary directory where you extracted the source files.
5. Click **OK** to accept the directory and place it in the **Select root directory** field. The application project and other projects are displayed in the **Projects** list:

   If the projects are not already selected in the **Projects** list, select them.
6. Click **Finish**.

After the import process completes, the projects are available in the Navigator view in the UI Development perspective.

Details About Project Structure and Project Type

With the exception of the main application project, all of the remaining sample projects are utility projects. You can create your own utility projects in Software AG Designer: **File > New Project > Java EE > Utility Project**.

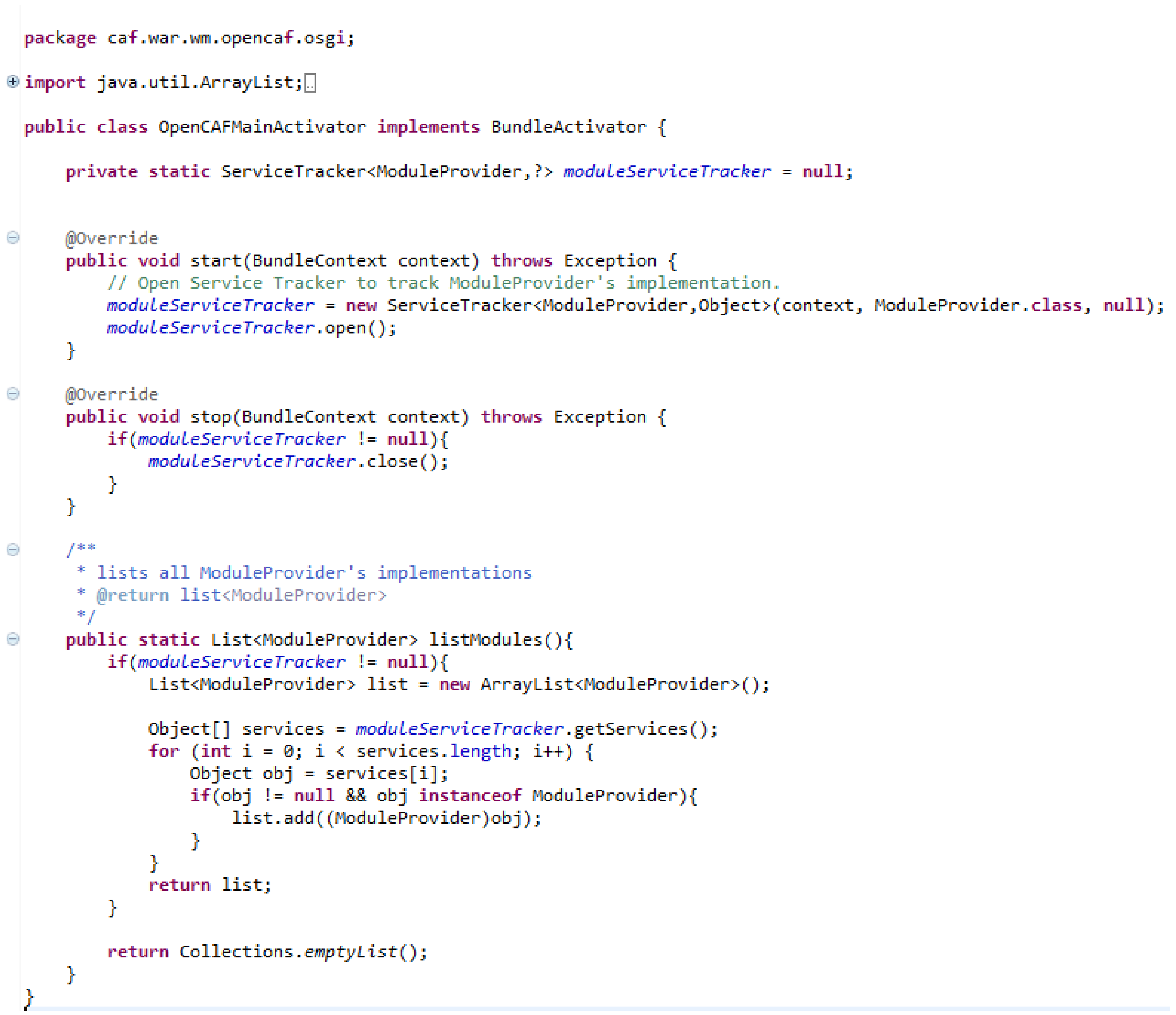
wm_opencaf_main

This is the main application which dynamically imports views from custom modules and also uses static views from shared-modules. This is a portlet application.

It demonstrates two techniques to load views from class path:

- Using facelets for .xhtml files.
- Using an ImportTemplateView control for .view files.

To listen to custom modules, the wm_opencaf_main project has a BundleActivator class (OpenCAFMainActivator) that starts a ServiceTracker for all implementations of the ModuleProvider interface:



```
package caf.war.wm.opencaf.osgi;

import java.util.ArrayList;

public class OpenCAFMainActivator implements BundleActivator {

    private static ServiceTracker<ModuleProvider,?> moduleServiceTracker = null;


    @Override
    public void start(BundleContext context) throws Exception {
        // Open Service Tracker to track ModuleProvider's implementation.
        moduleServiceTracker = new ServiceTracker<ModuleProvider,Object>(context, ModuleProvider.class, null);
        moduleServiceTracker.open();
    }

    @Override
    public void stop(BundleContext context) throws Exception {
        if(moduleServiceTracker != null){
            moduleServiceTracker.close();
        }
    }

    /**
     * lists all ModuleProvider's implementations
     * @return list<ModuleProvider>
     */
    public static List<ModuleProvider> listModules(){
        if(moduleServiceTracker != null){
            List<ModuleProvider> list = new ArrayList<ModuleProvider>();

            Object[] services = moduleServiceTracker.getServices();
            for (int i = 0; i < services.length; i++) {
                Object obj = services[i];
                if(obj != null && obj instanceof ModuleProvider){
                    list.add((ModuleProvider)obj);
                }
            }
            return list;
        }

        return Collections.emptyList();
    }
}
```

For an example of the facelets method of importing views, examine the Main portlet, available at:

wm_opencaf_main\WebContent\Main\default.xhtml

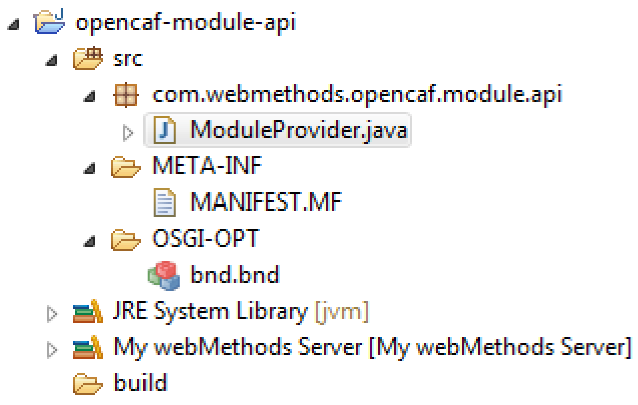
This portlet uses the <ui:include> tag for importing a view, and also demonstrates how to use a template for defining a layout of your application using the <ui:composition> tag.

You can also use a ImportTemplateView control to import CAF views (.view files) inside your application. For an example of this method, examine the Cafview portlet, available at:

wm_opencaf_main\WebContent\Cafview\default.view

opencaf-module-api

This is a utility project with the following project structure:



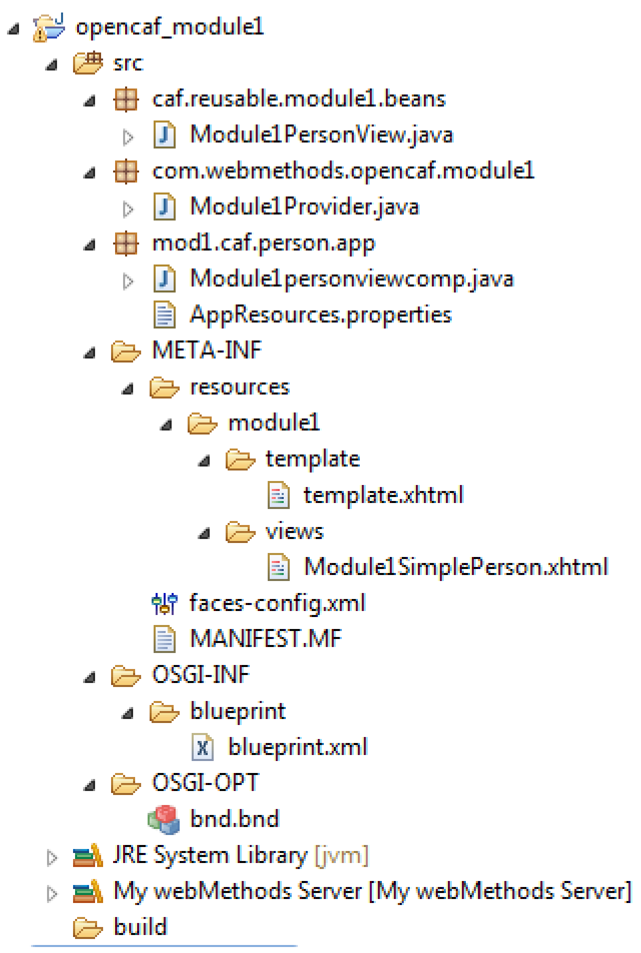
```
opencaf-module-api
  src
    com.webmethods.opencaf.module.api
      ModuleProvider.java
    META-INF
      MANIFEST.MF
    OSGI-OPT
      bnd.bnd
  JRE System Library [jvm]
  My webMethods Server [My webMethods Server]
  build
```

The project has only one interface, ModuleProvider, which defines all the methods that each module has to implement.

It also has an OSGI-OPT/bnd.bnd file which is used to change the manifest instruction of the deployed bundle inside My webMethods Server.

opencaf-module1 and module_cafview

These two projects are also utility projects like opencaf-module-api, with the only difference being that they contain a faces-config.xml file and other web resources under the META-INF folder, as shown below:



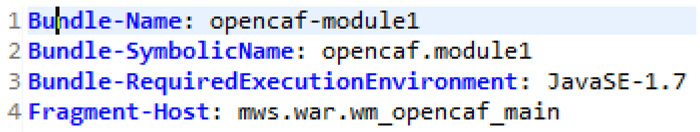
```
opencaf_module1
  src
    caf.reusable.module1.beans
      Module1PersonView.java
    com.webmethods.opencaf.module1
      Module1Provider.java
    mod1.caf.person.app
      Module1personviewcomp.java
      AppResources.properties
    META-INF
      resources
        module1
          template
            template.xhtml
          views
            Module1SimplePerson.xhtml
      faces-config.xml
      MANIFEST.MF
    OSGI-INF
      blueprint
        blueprint.xml
    OSGI-OPT
      bnd.bnd
  JRE System Library [jvm]
  My webMethods Server [My webMethods Server]
  build
```

These applications also provide an implementation for the ModuleProvider interface from the opencaf-module-api project:

com.webmethods.opencaf.module1.Module1Provider and com.webmethods.opencaf.module1.Module1Provider class.

These projects are OSGi fragments for the wm_opencaf_main application. There are some limitations with the OSGi fragment lifecycle. For example:

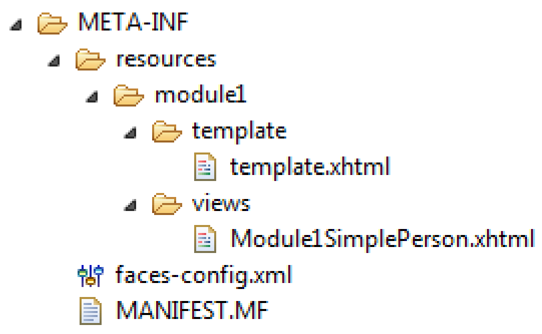
- A fragment cannot have a Bundle Activator, as fragments cannot be started. The fragment uses a blueprint.xml file to register and expose the OSGi service.
- A fragment uses an OSGi-OPT/bnd.bnd file and registers itself as a fragment of the wm_opencaf_main application:



```
Bundle-Name: opencaf-module1
Bundle-SymbolicName: opencaf.module1
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Fragment-Host: mws.war.wm_opencaf_main
```

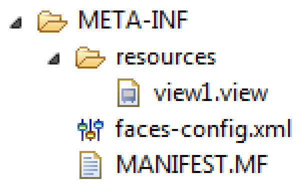
All the JSF-related resources are put under the META-INF/resources folder. You can specify all the managed beans in the META-INF/faces-config.xml file, and you can also put your .view files under the same folder structure.

The following image shows the facelets:



```
META-INF
  resources
    module1
      template
        template.xhtml
      views
        Module1SimplePerson.xhtml
  faces-config.xml
  MANIFEST.MF
```

Here is an example of a .view file in the META-INF/resources folder:



```
META-INF
  resources
    view1.view
  faces-config.xml
  MANIFEST.MF
```

shared-module

This project is also a utility project which has JSF resources in the META-INF/resources/views folder, but it does not register itself as an OSGi fragment of the wm_opencaf_main project. Instead, the wm_opencaf_main project declares a dependency on shared-module to reuse views from shared-module.

The only difference here is the inclusion of an OSGI-OPT/bnd.bnd file.