# Apache Felix Framework Launching and Embedding

**Contents**



NOTE

This document describes framework launching introduced in Felix Framework 2.0.0 and continuing with the latest releases; it is incompatible with older versions of the Felix framework.

## Introduction

The Apache Felix Framework is intended to be easily launchable and embeddable. For example, the Felix framework implementation avoids the use of system properties for configuration, since these are globals and can cause interference if multiple framework instances are created in the same VM. The framework also tries to multiplex singleton facilities, like the URL stream handler factory. The goal is to make it possible to use the framework in a variety of scenarios; however, this is still just a goal. In other words, this is a work in progress and if any issues arise, it would be greatly appreciated if they are brought to the attention of the Felix community. The next section provides an overview of the standard OSGi launching and embedding API for frameworks, while the remainder of the document is divided into two sections, one focusing on how to launch Felix and one focusing on how to embed Felix into a host application.

## OSGi Launching and Embedding API Overview

The Felix framework is implemented by the `org.apache.felix.framework.Felix` class or just `Felix` for short. As part of the R4.2 OSGi specification, the launching and embedding API of the OSGi framework has been standardized. The approach is to have the framework implement the `org.osgi.framework.launch.Framework` interface, which extends the `org.osgi.framework.Bundle` interface. These interfaces provide the necessary means to launch and manage framework instances. The `Bundle` interface is defined as:

```java
public interface Bundle
{
    BundleContext getBundleContext();
    long getBundleId();
    URL getEntry(String name);
    Enumeration getEntryPaths(String path);
    Enumeration findEntries(String path, String filePattern, boolean recurse);
    Dictionary getHeaders();
    Dictionary getHeaders(String locale);
    long getLastModified();
    String getLocation();
    URL getResource(String name);
    Enumeration getResources(String name) throws IOException;
    ServiceReference[] getRegisteredServices();
    ServiceReference[] getServicesInUse();
    int getState();
    String getSymbolicName();
    Version getVersion();
    boolean hasPermission(Object obj);
    Class loadClass(String name) throws ClassNotFoundException;
    void start() throws BundleException;
    void stop() throws BundleException;
    void uninstall() throws BundleException;
    void update() throws BundleException;
    void update(InputStream is) throws BundleException;
}
```

The `Framework` interface is defined as:

```java
public interface Framework extends Bundle
{
    void init();
    FrameworkEvent waitForStop(long timeout);
}
```

To actually construct a framework instance, the R4.2 specification defines the `FrameworkFactory` interface:

```java
public interface FrameworkFactory
{
    Framework newFramework(Map config);
}
```

The framework factory can be used to create configured framework instances. It is obtained following the standard `META-INF/services` approach.

### Creating and Configuring the Framework Instance

You use the framework factory to construct and configure a framework instance (or by directly instantiating the Felix class). The configuration map may contain any of the framework configuration properties listed in the Apache Felix Framework Configuration Properties document, not the launcher configuration properties. The configuration map is copied and the keys are treated as case insensitive. You are not able to change the framework's configuration after construction. If you need a different configuration, you must create a new framework instance.

WARNING

**WARNING** Felix configuration properties have change considerably starting from `1.4.0`; if you are upgrading from an earlier version, the configuration property document describes the configuration property changes.

### Starting the Framework Instance

The `start()` method is used to start the framework instance. If the `init()` method was not invoked prior to calling `start()`, then it is invoked by `start()`. The two methods result in two different framework state transitions:

- `init()` results in the framework instance in the `Bundle.STARTING` state.
- `start()` results in the framework instance in the `Bundle.ACTIVE` state.

The `init()` method is necessary since the framework does not have a `BundleContext` when it is first created, so a transition to the `Bundle.STARTING` state is required to acquire its context (via `Bundle.getBundleContext()`) for performing various tasks, such as installing bundles. Note that the Felix framework also provides the `felix.systembundle.activators` property that serves a similar purpose, but is not standard. After the `init()` method completes, the follow actions have been performed:

- Event handling is enabled.
- The security manager is installed if it is enabled.
- The framework is set to start level 0.
- All bundles in the bundle caches are reified and their state is set to `Bundle.INSTALLED`.

- The framework gets a valid `BundleContext`.
- All framework-provided services are made available (e.g., PackageAdmin, StartLevel, etc.).
- The framework enters the `Bundle.STARTING` state.

A call to `start()` is necessary to start the framework instance, if the `init()` method is invoked manually. Invoking `init()` or `start()` on an already started framework as no effect.

### Stopping the Framework Instance

To stop the framework instance, invoke the `stop()` method, which will asynchronously stop the framework. To know when the framework has finished its shutdown sequence, use the `waitForStop()` method to wait until it is complete. A stopped framework will be in the `Bundle.RESOLVED` state. It is possible to restart the framework, using the normal combination of `init()` / `start()` methods as previously described.

## Launching a Framework

Launching a framework is fairly simple and involves only four steps:

1. Define some configuration properties.
2. Obtain framework factory.
3. Use factory to create framework with the configuration properties.
4. Invoke the `Framework.start()` method.

In reality, the first step is optional, since all properties will have reasonable defaults, but if you are creating a launcher you will generally want to more than that, such as automatically installing and starting bundles when you start the framework instance. The default Felix launcher defines reusable functionality to automatically install and/or start bundles upon framework startup; see the usage document for more information on configuring the Felix framework and on the various configuration properties.

The remainder of this section describes how the standard Felix launcher works as well as how to create a custom launcher.

### Standard Felix Framework Launcher

The standard Felix framework launcher is very simple and is not intended to solve every possible requirement; it is intended to work for most standard situations. Most special launching requirements should be resolved by creating a custom launcher. This section describes how the standard launcher works. The following code represents the complete `main()` method of the standard launcher, each numbered comment will be described in more detail below:

JAVA

```java
public static void main(String[] args) throws Exception
{
    // (1) Check for command line arguments and verify usage.
    String bundleDir = null;
    String cacheDir = null;
    boolean expectBundleDir = false;
    for (int i = 0; i < args.length; i++)
    {
        if (args[i].equals(BUNDLE_DIR_SWITCH))
        {
            expectBundleDir = true;
        }
        else if (expectBundleDir)
        {
            bundleDir = args[i];
            expectBundleDir = false;
        }
        else
        {
            cacheDir = args[i];
        }
    }

    if ((args.length > 3) || (expectBundleDir && bundleDir == null))
    {
        System.out.println("Usage: [-b <bundle-deploy-dir>] [<bundle-cache-dir>]");
        System.exit(0);
    }

    // (2) Load system properties.
    Main.loadSystemProperties();

    // (3) Read configuration properties.
    Properties configProps = Main.loadConfigProperties();
    if (configProps == null)
    {
        System.err.println("No " + CONFIG_PROPERTIES_FILE_VALUE + " found.");
        configProps = new Properties();
    }

    // (4) Copy framework properties from the system properties.
    Main.copySystemProperties(configProps);

    // (5) Use the specified auto-deploy directory over default.
    if (bundleDir != null)
    {
        configProps.setProperty(AutoProcessor.AUTO_DEPLOY_DIR_PROPERY, bundleDir);
```

```
    }

    // (6) Use the specified bundle cache directory over default.
    if (cacheDir != null)
    {
        configProps.setProperty(Constants.FRAMEWORK_STORAGE, cacheDir);
    }

    // (7) Add a shutdown hook to clean stop the framework.
    String enableHook = configProps.getProperty(SHUTDOWN_HOOK_PROP);
    if ((enableHook == null) || !enableHook.equalsIgnoreCase("false"))
    {
        Runtime.getRuntime().addShutdownHook(new Thread("Felix Shutdown Hook") {
            public void run()
            {
                try
                {
                    if (m_fwk != null)
                    {
                        m_fwk.stop();
                        m_fwk.waitForStop(0);
                    }
                }
                catch (Exception ex)
                {
                    System.err.println("Error stopping framework: " + ex);
                }
            }
        });
    }

    try
    {
        // (8) Create an instance and initialize the framework.
        FrameworkFactory factory = getFrameworkFactory();
        m_fwk = factory.newFramework(configProps);
        m_fwk.init();
        // (9) Use the system bundle context to process the auto-deploy
        // and auto-install/auto-start properties.
        AutoProcessor.process(configProps, m_fwk.getBundleContext());
        // (10) Start the framework.
        m_fwk.start();
        // (11) Wait for framework to stop to exit the VM.
        m_fwk.waitForStop(0);
        System.exit(0);
    }
    catch (Exception ex)
    {
```

```
            System.err.println("Could not create framework: " + ex);
            ex.printStackTrace();
            System.exit(0);
        }
    }
```

The general steps of the standard launcher are quite straightforward:

1. The launcher supports setting the auto-deploy directory (with the `-b` switch) and setting the bundle cache path with a single argument, so check for this and issue a usage message it there are more than one arguments.
2. Load any system properties specified in the `system.properties` file; this file is typically located in the `conf/` directory of the Felix installation directory, but it can be specified directly using the `felix.system.properties` system property. This file is not needed to launch Felix and is provided merely for convenience when system properties must be specified. The file is a standard Java properties file, but it also supports property substitution using `$\{<property-name>}` syntax. Property substitution can be nested; only system properties will be used for substitution.
3. Load any configuration properties specified in the `config.properties` file; this file is typically located in the `conf/` directory of the Felix installation directory, but it can be specified directly using the `felix.config.properties` system property. This file is used to configure the framework instance created by the launcher. The file is a standard Java properties file, but it also supports property substitution using "`$\{<property-name>}`" syntax. Property substitution can be nested; configuration and system properties will be used for substitution with configuration properties having precedence.
4. For convenience, any configuration properties that are set as system properties are copied into the set of configuration properties. This provide an easy way to add to or override configuration properties specified in the `config.properties` file, since the Felix instance will never look at system properties for configuration.
5. If the `-b` switch was used to specify an auto-deploy directory, then use that to set the value of `felix.auto.deploy.dir`.
6. If a single command-line argument is specified, then use that to set the value of `org.osgi.framework.storage`; relative paths are relative to the current directory unless the `felix.cache.rootdir` property is set.
7. Add a shutdown hook to cleanly stop the framework, unless the hook is disabled.
8. Create a framework instance using the `FrameworkFactory` passing in the configuration properties, then initialize the factory instance; see the custom launcher example below to see how the META-INF/services `FrameworkFactory` is obtained.

| | | |
|---|---|---|
| 9. | Use `org.apache.felix.main.AutoProcessor`, which will automatically deploy any bundles in the auto-deploy directory as well as bundles specified in the `felix.auto.install` and `felix.auto.start` configuration properties during framework startup to automatically install and/or start bundles; see the usage document for more information configuration properties and [bundle auto-deploy | Apache Felix Framework Usage Documentation#auto-deploy]. |

10. Invoke `waitForStop()` to wait for the framework to stop to force the VM to exit; this is necessary because the framework never calls `System.exit()` and some libraries (e.g., Swing) create threads that will not allow the VM to exit.

The framework is not active until the `start()` method is called. If no shell bundles are installed and started or if there is difficulty locating the shell bundles specified in the auto-start property, then it will appear as if the framework is hung, but it is actually running without any way to interact with it since the shell bundles provide the only means of interaction.

## Custom Framework Launcher

This section creates a bare-bones launcher to demonstrate the minimum requirements for creating an interactive launcher for the Felix framework. This example uses the standard Gogo shell bundles for interactivity, but any other bundles could be used instead. This example launcher project has the following directory structure:

```
launcher/
   lib/
      org.apache.felix.main-3.0.0.jar
   bundle/
      org.apache.felix.gogo.command-0.6.0.jar
      org.apache.felix.gogo.runtime-0.6.0.jar
      org.apache.felix.gogo.shell-0.6.0.jar
   src/
      example/
         Main.java
```

The `lib/` directory contains Felix' main JAR file, which also contains the OSGi core interfaces. The main JAR file is used so that we can reuse the default launcher's auto-install/auto-start configuration property handling; if these capabilities are not needed, then it would be possible to use the framework JAR file instead of the main JAR file. The `bundle/` directory contains the shell service and textual shell interface bundles that will be used for interacting with the framework instance. Note: If you do not launch the framework with interactive bundles, it will appear as if the framework instance is hung, but it

is actually just sitting there waiting for someone to tell it to do something. The `src/example/` directory contains the following `Main.java` file, which is a very simplistic framework launcher.

JAVA

```
package example;

import java.io.*;
import org.osgi.framework.launch.*;
import org.apache.felix.main.AutoProcessor;

public class Main
{
    private static Framework m_fwk = null;

    public static void main(String[] argv) throws Exception
    {
        // Print welcome banner.
        System.out.println("\nWelcome to My Launcher");
        System.out.println("======================\n");

        try
        {
            m_fwk = getFrameworkFactory().newFramework(null);
            m_fwk.init();
            AutoProcessor.process(null, m_fwk.getBundleContext());
            m_fwk.start();
            m_fwk.waitForStop(0);
            System.exit(0);
        }
        catch (Exception ex)
        {
            System.err.println("Could not create framework: " + ex);
            ex.printStackTrace();
            System.exit(-1);
        }
    }

    private static FrameworkFactory getFrameworkFactory() throws Exception
    {
        java.net.URL url = Main.class.getClassLoader().getResource(
            "META-INF/services/org.osgi.framework.launch.FrameworkFactory");
        if (url != null)
        {
            BufferedReader br = new BufferedReader(new
InputStreamReader(url.openStream()));
            try
            {
                for (String s = br.readLine(); s != null; s = br.readLine())
```

```
                {
                    s = s.trim();
                    // Try to load first non-empty, non-commented line.
                    if ((s.length() > 0) && (s.charAt(0) != '#'))
                    {
                        return (FrameworkFactory) Class.forName(s).newInstance();
                    }
                }
            }
            finally
            {
                if (br != null) br.close();
            }
        }

        throw new Exception("Could not find framework factory.");
    }
}
```

This launcher relies on the default behavior of `AutoProcessor` to automatically deploy the shell bundles. This simple, generic launcher provides a good starting point if the default Felix launcher is not sufficient. Since very few configuration properties are specified, the default values are used. For the bundle auto-deploy directory, "`bundle`" in the current directory is used, while for the framework bundle cache, "`felix-cache`" in the current directory is used.

By breaking down the above source code into small chunks, it is quite easy to see what is going on.

JAVA
```
            m_fwk = getFrameworkFactory().newFramework(null);
            m_fwk.init()
```

These steps get a the framework factory service and use it to create a framework instance with a default configuration. Once the framework instance is created, it is initialized with `init()`.

JAVA
```
            AutoProcessor.process(null, m_fwk.getBundleContext());
```

The `AutorProcessor` will automatically deploy bundles in the auto-deploy directory and any referenced from the auto-install/start properties. Since we are using an empty configuration, the auto-deploy directory is the `bundle` directory in the current directory and there are no auto properties. Therefore, in this case, the shell bundles will be installed.

JAVA
```
            m_fwk.start();
            m_fwk.waitForStop(0);
```

```
            System.exit(0);
```

These final steps start the framework and cause the launching application thread to wait for the framework to stop and when it does the launching thread calls `System.exit()` to make sure the VM actually exits.

```JAVA
    private static FrameworkFactory getFrameworkFactory() throws Exception
    {
        ...
    }
```

This method retrieves the framework factory service by doing a META-INF/services resource lookup, which it can use to obtain the concrete class name for the factory. If you are using Java 6, then you can use the `ServiceLoader` API in the JRE to further simplify the factory service lookup.

The following command compiles the launcher when run from the root directory of the launcher project:

```SH
javac -d . -classpath lib/org.apache.felix.main-3.0.0.jar src/example/Main.java
```

After executing this command, an `example/` directory is created in the current directory, which contains the generated class file. The following command executes the simple launcher when run from the root directory of the launcher project:

```SH
java -cp .:lib/org.apache.felix.main-3.0.0.jar example.Main
```

After executing this command, a "`felix-cache/`" directory is created that contains the cached bundles, which were installed from the `bundle/` directory.

## Embedding the Felix Framework

Embedding the Felix framework into a host application is a simple way to provide a sophisticated extensibility mechanism (i.e., a plugin system) to the host application. Embedding the Felix framework is very similar to launching it as described above, the main difference is that the host application typically wants to interact with the framework instance and/or installed bundles/services from the outside. This is fairly easy to achieve, but there are some subtle issues to understand. This section presents the mechanisms for embedding Felix into a host application and the issues in doing so.

### Host/Felix Interaction

In the section on launching the framework above, the `Felix` class accepts a configuration property called `felix.systembundle.activators`, which is a list of bundle activator instances. These bundle

activator instances provide a convenient way for host applications to interact with the Felix framework.

WARNING

**WARNING** The `felix.systembundle.activators` configuration property is specific to the Felix framework implementation. If you want your code to work with other framework implementations, you should call `init()` on the framework instance and use `getBundleContext()` directly. Otherwise, the approach would be very similar.

Each activator instance passed into the constructor effectively becomes part of the system bundle. This means that the `start()` / `stop()` methods of each activator instance in the list gets invoked when the system bundle's activator `start()` / `stop()` methods gets invoked, respectively. Each activator instance will be given the system bundle's `BundleContext` object so that they can interact with the framework. Consider following snippet of a bundle activator:

JAVA

```
public class HostActivator implements BundleActivator
{
    private BundleContext m_context = null;

    public void start(BundleContext context)
    {
        m_context = context;
    }

    public void stop(BundleContext context)
    {
        m_context = null;
    }

    public Bundle[] getBundles()
    {
        if (m_context != null)
        {
            return m_context.getBundles();
        }
        return null;
    }
}
```

Given the above bundle activator, it is now possible to embed the Felix framework into a host application and interact with it as the following snippet illustrates:

JAVA

```
public class HostApplication
{
    private HostActivator m_activator = null;
    private Felix m_felix = null;
```

```
    public HostApplication()
    {
        // Create a configuration property map.
        Map config = new HashMap();
        // Create host activator;
        m_activator = new HostActivator();
        List list = new ArrayList();
        list.add(m_activator);
        configMap.put(FelixConstants.SYSTEMBUNDLE_ACTIVATORS_PROP, list);

        try
        {
            // Now create an instance of the framework with
            // our configuration properties.
            m_felix = new Felix(config);
            // Now start Felix instance.
            m_felix.start();
        }
        catch (Exception ex)
        {
            System.err.println("Could not create framework: " + ex);
            ex.printStackTrace();
        }
    }

    public Bundle[] getInstalledBundles()
    {
        // Use the system bundle activator to gain external
        // access to the set of installed bundles.
        return m_activator.getBundles();
    }

    public void shutdownApplication()
    {
        // Shut down the felix framework when stopping the
        // host application.
        m_felix.stop();
        m_felix.waitForStop(0);
    }
}
```

Notice how the `HostApplication.getInstalledBundles()` method uses its activator instance to get access to the system bundle's context in order to interact with the embedded Felix framework instance. This approach provides the foundation for all interaction between the host application and the embedded framework instance.

## Providing Host Application Services

Providing services from the host application to bundles inside the embedded Felix framework instance follows the basic approach laid out in above. The main complication for providing a host application service to bundles is the fact that both the host application and the bundles must be using the same class definitions for the service interface classes. Since the host application cannot import classes from a bundle, this means that the service interface classes _must be accessible on the class path, typically as part of the host application itself. The host application then must export the service interface package via the system bundle so that bundles installed into the embedded framework instance can import it. This is achieved using the `org.osgi.framework.system.packages.extra` configuration property previously presented.

Consider the follow simple property lookup service:

```java
package host.service.lookup;

public interface Lookup
{
    public Object lookup(String name);
}
```

This package is simply part of the host application, which is potentially packaged into a JAR file and started with the "`java -jar`" command. Now consider the following host application bundle activator, which will be used to register/unregister the property lookup service when the embedded framework instance starts/stops:

```java
package host.core;

import java.util.Map;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import host.service.lookup;

public class HostActivator implements BundleActivator
{
    private Map m_lookupMap = null;
    private BundleContext m_context = null;
    private ServiceRegistration m_registration = null;

    public HostActivator(Map lookupMap)
    {
        // Save a reference to the service's backing store.
        m_lookupMap = lookupMap;
```

```
    }

    public void start(BundleContext context)
    {
        // Save a reference to the bundle context.
        m_context = context;
        // Create a property lookup service implementation.
        Lookup lookup = new Lookup() {
            public Object lookup(String name)
            {
                return m_lookupMap.get(name);
            }
        };
        // Register the property lookup service and save
        // the service registration.
        m_registration = m_context.registerService(
            Lookup.class.getName(), lookup, null);
    }

    public void stop(BundleContext context)
    {
        // Unregister the property lookup service.
        m_registration.unregister();
        m_context = null;
    }
}
```

Given the above host application bundle activator, the following code snippet shows how the host application could create an embedded version of the Felix framework and provide the property lookup service to installed bundles:

JAVA

```
package host.core;

import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;
import host.service.lookup.Lookup;
import org.apache.felix.framework.Felix;
import org.apache.felix.framework.util.FelixConstants;
import org.osgi.framework.Constants;

public class HostApplication
{
    private HostActivator m_activator = null;
    private Felix m_felix = null;
```

```
    private Map m_lookupMap = new HashMap();

    public HostApplication()
    {
        // Initialize the map for the property lookup service.
        m_lookupMap.put("name1", "value1");

        m_lookupMap.put("name2", "value2");
        m_lookupMap.put("name3", "value3");
        m_lookupMap.put("name4", "value4");

        // Create a configuration property map.
        Map configMap = new HashMap();
        // Export the host provided service interface package.
        configMap.put(Constants.FRAMEWORK_SYSTEMPACKAGES_EXTRA,
            "host.service.lookup; version=1.0.0");
        // Create host activator;
        m_activator = new HostActivator(m_lookupMap);
        List list = new ArrayList();
        list.add(m_activator);
        configMap.put(FelixConstants.SYSTEMBUNDLE_ACTIVATORS_PROP, list);

        try
        {
            // Now create an instance of the framework with
            // our configuration properties.
            m_felix = new Felix(configMap);
            // Now start Felix instance.
            m_felix.start();
        }
        catch (Exception ex)
        {
            System.err.println("Could not create framework: " + ex);
            ex.printStackTrace();
        }
    }

    public void shutdownApplication()
    {
        // Shut down the felix framework when stopping the
        // host application.
        m_felix.stop();
        m_felix.waitForStop(0);
    }
}
```

Rather than having the host application bundle activator register the service, it is also possible for the the host application to simply get the bundle context from the bundle activator and register the service directly, but the presented approach is perhaps a little cleaner since it allows the host application to register/unregister the service when the system bundle starts/stops.

### Using Services Provided by Bundles

Using services provided by bundles follows the same general approach of using a host application bundle activator. The main complication for the host application using a service from a bundle is the fact that both the host application and the bundle must be using the same class definitions for the service interface classes. Since the host application cannot import classes from a bundle, this means that the service interface classes *must* be accessible on the class path, typically as part of the host application itself. The host application then must export the service interface package via the system bundle so that bundles installed into the embedded framework instance can import it. This is achieved using the `org.osgi.framework.system.packages.extra` configuration property previously presented.

Consider the following simple command service interface for which bundles provide implementations, such as might be used to create an extensible interactive shell:

```java
package host.service.command;

public class Command
{
    public String getName();
    public String getDescription();
    public boolean execute(String commandline);
}
```

This package is simply part of the host application, which is potentially packaged into a JAR file and started with the "`java -jar`" command. Now consider the previously introduced host application bundle activator below, which simply provides access to the system bundle context:

```java
package host.core;

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

public class HostActivator implements BundleActivator
{
    private BundleContext m_context = null;

    public void start(BundleContext context)
    {
```

```
        m_context = context;
    }

    public void stop(BundleContext context)
    {
        m_context = null;
    }

    public BundleContext getContext()
    {
        return m_context;
    }
}
```

With this bundle activator, the host application can use command services provided by bundles installed inside its embedded Felix framework instance. The following code snippet illustrates one possible approach:

JAVA

```
package host.core;

import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import host.service.command.Command;
import org.apache.felix.framework.Felix;
import org.apache.felix.framework.util.FelixConstants;
import org.apache.felix.framework.cache.BundleCache;
import org.osgi.framework.Constants;
import org.osgi.util.tracker.ServiceTracker;

public class HostApplication
{
    private HostActivator m_activator = null;
    private Felix m_felix = null;
    private ServiceTracker m_tracker = null;

    public HostApplication()
    {
        // Create a configuration property map.
        Map configMap = new HashMap();
        // Export the host provided service interface package.
        configMap.put(Constants.FRAMEWORK_SYSTEMPACKAGES_EXTRA,
            "host.service.command; version=1.0.0");
        // Create host activator;
        m_activator = new HostActivator();
        List list = new ArrayList();
```

```
        list.add(m_activator);
        configMap.put(FelixConstants.SYSTEMBUNDLE_ACTIVATORS_PROP, list);

        try
        {
            // Now create an instance of the framework with
            // our configuration properties.
            m_felix = new Felix(configMap);
            // Now start Felix instance.
            m_felix.start();
        }
        catch (Exception ex)
        {
            System.err.println("Could not create framework: " + ex);
            ex.printStackTrace();
        }

        m_tracker = new ServiceTracker(
            m_activator.getContext(), Command.class.getName(), null);
        m_tracker.open();
    }

    public boolean execute(String name, String commandline)
    {
        // See if any of the currently tracked command services
        // match the specified command name, if so then execute it.
        Object[] services = m_tracker.getServices();
        for (int i = 0; (services != null) && (i < services.length); i++)
        {
            try
            {
                if (((Command) services[i]).getName().equals(name))
                {
                    return ((Command) services[i]).execute(commandline);
                }
            }
            catch (Exception ex)
            {
                // Since the services returned by the tracker could become
                // invalid at any moment, we will catch all exceptions, log
                // a message, and then ignore faulty services.
                System.err.println(ex);
            }
        }
        return false;
    }

    public void shutdownApplication()
```

```
    {
        // Shut down the felix framework when stopping the
        // host application.
        m_felix.stop();
        m_felix.waitForStop(0);
    }
}
```

The above example is overly simplistic with respect to concurrency issues and error conditions, but it demonstrates the overall approach for using bundle-provided services from the host application.

Using Bundle Services via Reflection

It possible for the host application to use services provided by bundles without having access to the service interface classes and thus not needing to put the service interface classes on the class path. To do this, the host application uses the same general approach to acquire the system bundle context object, which it can use to look up service objects. Using either an LDAP filter or the service interface class name, the host application can retrieve the service object and then use standard Java reflection to invoke methods on the service object.

Other Approaches

The Transloader project is another attempt at dealing with issues of classes loaded from different class loaders and may be of interest.

## Caveat

The code in this document has not been thoroughly tested nor even compiled and may be out of date with respect to the current Felix source code. If you find errors please report them so the that they can be corrected.

### Feedback

| | |
|---|---|
| Subscribe to the Felix users mailing list by sending a message to users-subscribe@felix.apache.org; after subscribing, email questions or feedback to [users@felix.apache.org | mailto:users@felix.apache.org]. |

Contents