OSGi Modularity - Tutorial

segunda-feira, 25 maio 2015, 11:41 AM

OSGi Modularity - Tutorial

Lars Vogel

Version 5.0

18.03.2015

OSGi with Eclipse Equinox

This tutorial gives an overview of OSGi and its modularity layer. For this tutorial Eclipse 4.4 (Luna) is used.

Table of Contents

1. Introduction into software modularity with OSGi

1.1. What is software 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.

If a component uses an API from another component, it has a dependency to the other component, i.e., it requires the other component exists and works correctly.

A component which is used by other components should try to keep its API stable to avoid that a change affects other components. But it should be free to change its internal implementation.

Java, in its current version (Java 8), provides no structured way to describe software component dependencies. Java only supports the usage of access modifiers, but every public class can be called from another software component. What is desired is a way to explicitly define the API of a software component. The OSGi specification fills this gap.

1.2. What is OSGi?

OSGi is a set of specifications which, in 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.

1.3. OSGi implementations

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.

1.4. Plug-in or bundles as software component

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 valid plug-in is always a valid bundle and a valid bundle is always a valid plug-in. In this book the usage of *plug-in* is preferred, to be consistent with the terminology of Eclipse plug-in development.

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.

1.5. Naming convention: simple plug-in

A plug-in can be generated by Eclipse via the *File → New → Other... → Plug-In Development → Plug-In Project* menu entry. The corresponding wizard allows specifying several options. This book calls plug-ins generated with the following options a *simple plug-in* or *simple bundle*.

- No Activator
- No contributions to the user interface
- Not a 3.x rich client application
- Generated without a template

2. OSGi metadata

2.1. The manifest file (MANIFEST.MF)

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

The following listing is an example for a manifest file.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Popup Plug-in
Bundle-SymbolicName: com.example.myosgi; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: com.example.myosgi.Activator
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

The following table gives an explanation of the identifiers in the manifest file. For additional information on the unique identifier see **Section 2.2, “Bundle-SymbolicName and Version”** and for information on the version schema which is typically used in OSGi see **Section 2.3, “Semantic Versioning with OSGi”**.

Table 1. OSGi identifiers in the manifest file

Identifier	Description
Bundle-Name	Short description of the plug-in.
Bundle-SymbolicName	The unique identifier of the plug-in. If this plug-in is using the extension point functionality of Eclipse, it must be marked as Singleton. You do this by adding the following statement after the Bundle-SymbolicName identifier: `; singleton:=true`
Bundle-Version	Defines the plug-in version and must be incremented if a new version of the plug-in is published.
Bundle-Activator	Defines an optional activator class which implements the `BundleActivator` interface. An instance of this class is created when the plug-in gets activated. Its `start()` and `stop()` methods are called whenever the plug-in is started or stopped. An OSGi activator can be used to configure the plug-in during startup. The execution of an activator increases the startup time of the application, therefore this functionality should be used carefully.
Bundle-RequiredExecutionEnvironment (BREE)	Specify which Java version is required to run the plug-in. If this requirement is not fulfilled, then the OSGi runtime does not load the plug-in.
Bundle-ActivationPolicy	Setting this to *`Lazy`* will tell the OSGi runtime that this plug-in should only be activated if one of its components, i.e. classes and interfaces are used by other plug-ins. If not set, the Equinox runtime does not activate the plug-in, i.e., services provided by this plug-in are not available.
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*.

2.2. Bundle-SymbolicName and Version

Each plug-in has a symbolic name which is defined via the `Bundle-SymbolicName` property. The name starts by convention with the reverse domain name of the plug-in author, e.g., if you own the "example.com" domain then the symbolic name would start with "com.example".

Each plug-in has also a version number in the `Bundle-Version` property.

The `Bundle-Version` and the `Bundle-SymbolicName` uniquely identifies a plug-in.

2.3. Semantic Versioning with OSGi

OSGi recommends to use a *`<major>.<minor>.<patch>`* schema for the version number which is defined via the `Bundle-Version` field identifier. If you change your plug-in code you increase the version according to the following rule set.

- *`<patch>`* is increased if all changes are backwards compatible.
- *`<minor>`* is increased if public API has changed but all changes are backwards compatible.
- *`<major>`* is increased if changes are not backwards compatible.

For more information on this version scheme see the **Eclipse Version Numbering Wiki**.

3. Defining the dependencies of the plug-in

3.1. Specifying plug-in dependencies via the manifest file

A plug-in can define dependencies to other software components via its manifest file. OSGi prevents access to classes without a defined dependency and throws a `ClassNotFoundException`. The only exception are packages from the Java runtime environment (based on the Bundle-RequiredExecutionEnvironment definition of the plug-in); these packages are always available to a plug-in without an explicitly defined dependency.

If you add a dependency to your manifest file, the Eclipse IDE automatically adds the corresponding *`JAR`* file to your project classpath.

You can define dependencies either as plug-in dependencies or package dependencies. If you define a plug-in dependency your plug-in can access all exported packages of this plug-in. If you specify a package dependency you can access this package. Using package dependencies allows you to exchange the plug-in which provides this package at a later point in time. If you require this flexibility prefer the usage of package dependencies.

A plug-in can define that it depends on a certain version (or a range) of another bundle, e.g., plug-in A can define that it depends on plug-in C in version 2.0, while plug-in B defines that it depends on version 1.0 of plug-in C.

The OSGi runtime ensures that all dependencies are present before it starts a plug-in. OSGi reads the manifest file of a plug-in during its installation. It ensures that all dependent plug-ins are also resolved and, if necessary, activates them before the plug-in starts.

3.2. Life cycle of plug-ins in OSGi

With the installation of a plug-in in the OSGi runtime the plug-in is persisted in a local bundle cache. The OSGi runtime then tries to resolve its dependencies.

If all required dependencies are resolved, the plug-in is in the *RESOLVED* status otherwise it stays in the *INSTALLED* status.

In case several plug-ins exist which can satisfy the dependency, the plug-in with the highest valid version is used.

If the versions are the same, the plug-in with the lowest unique identifier (ID) is used. Every plug-in gets this ID assigned by the framework during the installation.

When the plug-in starts, its status is *STARTING*. After a successful start, it becomes *ACTIVE*.

This life cycle is depicted in the following graphic.

3.3. Dynamic imports of packages

For legacy reasons OSGi supports a dynamic import of packages. See **OSGi Wiki for dynamic imports** for details. You should not use this feature, it is a symptom of a non-modular design.

4. Defining the API of a plug-in

4.1. Specifying the API of a plug-in

In the *`MANIFEST.MF`* file a plug-in also defines its API via the Export-Package Identifier. 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.

Unfortunately OSGi can not prevent you from using Java reflection to access these classes. This is because OSGi is based on the Java runtime which does not yet support a modularity layer.

4.2. Provisional API

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.

The following screenshot shows how to set a package as `x-internal` in the manifest editor.

This is how the corresponding manifest file looks like.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Provider
Bundle-SymbolicName: de.vogella.osgi.xinternal.provider
Bundle-Version: 1.0.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Export-Package: de.vogella.osgi.xinternal.provider;x-internal:=true
```

You can configure how the Eclipse Java editor shows the usage of provisional API. Such an access can be configured to be displayed as, error, warning or if such access should be result in no additional message.

The default is to display a warning message. You can adjust this in the Eclipse preferences via the *Window → Preferences → Java → Compiler → Errors/Warnings* preference setting.

4.3. Provisional API with exceptions via x-friends

You can define that a set of plug-ins can access provisional API without a warning or error message. This can be done via the `x-friends` directive. This flag is added if you add a plug-in to the *Package Visibility* section on the *Runtime* tab of the manifest editor.

```
Manifest-Version: 1.0
```

```
Bundle-ManifestVersion: 2
Bundle-Name: Provider
Bundle-SymbolicName: de.vogella.osgi.xinternal.provider
Bundle-Version: 1.0.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Export-Package: de.vogella.osgi.xinternal.provider;x-friends:="another.bundle"
```

The `x-friends` setting has the same effect as `x-internal` but all plug-ins mentioned in the x-friends setting can access the package without receiving an error or warning message.

5. Find the plug-in for a certain class

You frequently have to find the plug-in for a given class. The Eclipse IDE makes it easy to find the plug-in for a class. After enabling the *Include all plug-ins from target into Java Search* setting in the Eclipse IDE preferences you can use the *Open Type* dialog (**Ctrl**+**Shift**+**T**) to find the plug-in for a class. The *JAR* file is shown in this dialog and the prefix of the JAR file is typically the plug-in which contains this class.

6. Using the OSGi console

6.1. The OSGi console

The OSGi console is like a command-line shell. In this console you can type a command to perform an OSGi action. This can be useful to analyze problems on the OSGi layer of your application.

Use, for example, the command `ss` to get an overview of all bundles, their status and bundle-id. The following table is a reference of the most important OSGi commands.

Table 2. OSGi commands

Command	Description
help	Lists the available commands.
ss	Lists the installed bundles and their status.
ss *vogella*	Lists bundles and their status that have *vogella* within their name.
start *<bundle-id>*	Starts the bundle with the *<bundle-id>* ID.
stop *<bundle-id>*	Stops the bundle with the *<bundle-id>* ID.
diag *<bundle-id>*	Diagnoses a particular bundle. It lists all missing dependencies.
install *URL*	Installs a bundle from a URL.
uninstall *<bundle-*	Uninstalls the bundle with the *<bundle-id>* ID.

id>	
bundle *<bundle-id>*	Shows information about the bundle with the *<bundle-id>* ID, including the registered and used services.
headers *<bundle-id>*	Shows the MANIFST.MF information for a bundle.
services *filter*	Shows all available services and their consumer. Filter is an optional LDAP filter, e.g., to see all services which provide a ManagedService implementation use the "services (objectclass=*ManagedService)" command.

6.2. Required bundles

You have to add the following bundles to your runtime configuration to use the OSGi console.

- org.eclipse.equinox.console
- org.apache.felix.gogo.command
- org.apache.felix.gogo.runtime
- org.apache.felix.gogo.shell

6.3. Telnet

If you specify the *-console* parameter in your launch configuration Eclipse will allow you to interact with the OSGi console. An OSGi launch configuration created with the Eclipse IDE contains this parameter by default. Via the following parameter you can open a port to which you can connect via the telnet protocol.

```
-console 5555
```

If you open a telnet session to the OSGi console, you can use tab completion and a history of the commands similar to the *Bash* shell under Linux.

6.4. Access to the Eclipse OSGi console

You can also access the OSGi console of your running Eclipse IDE. In the *Console View* you find a menu entry with the tooltip *Open Console*. If you select *Host OSGi Console*, you will have access to your running OSGi instance.

Please note that interfering with your running Eclipse IDE via the OSGi console, may put the Eclipse IDE into a bad state.

7. Download the Eclipse SDK

If you plan to build additional functionality upon the Eclipse platform, you should download the latest Eclipse release. Official releases are stable and a good foundation for building your additional functionality on top of it.

The Eclipse IDE is provided in different flavors. While you can install the necessary tools in any Eclipse package, it is typically easier to download the Eclipse Standard distribution which contains all necessary tools for plug-in development. Other packages adds more tools which are not required for Eclipse plug-in development.

Browse to the **Eclipse download site** and download the *Eclipse Standard* package.

8. Exercise: Data model plug-in

8.1. Target of the exercise

In this exercise you create a plug-in for the definition of your data model. You also make this data model available to other plug-ins.

8.2. Create the plug-in for the data model

Create a simple plug-in project (see **Section 1.5, “Naming convention: simple plug-in”**) called *com.example.e4.rcp.todo.model*.

The following screenshot depicts the second page of the plug-in project wizard and its corresponding settings. Press the *Finish* button on this page to avoid the usage of templates.

8.3. Create the base class

Create the `com.example.e4.rcp.todo.model` package and the following model class.

```
package com.example.e4.rcp.todo.model;

import java.util.Date;

public class Todo {

  private final long id;
  private String summary = "";
  private String description = "";
  private boolean done = false;
  private Date dueDate;

}
```

Note: You see an error for your final id field. This error is solved in the **Section 8.4, “Generate constructors”** section.

8.4. Generate constructors

Select *Source → Generate Constructor using Fields...* to generate a constructor using all fields. Use the same approach to create another constructor using only the `id` field.

Warning: Ensure that you have created both constructors, because they are required in the following exercises.

8.5. Generate getter and setter methods

Use the *Source → Generate Getter and Setter...* menu to create getters and setters for your fields.

> **Note:**The *id* is final and therefore Eclipse creates only a getter. This correct and desired.

8.6. Adjust the generated getter and setter methods

Adjust the generated getter and setter for the `dueDate()` field to make defensive copies. The `Date` class is not immutable and we want to avoid that an instance of `Todo` can be changed from outside without the corresponding setter.

```
public Date getDueDate() {
  return new Date(dueDate.getTime());
}

public void setDueDate(Date dueDate) {
  this.dueDate = new Date(dueDate.getTime());
}
```

The resulting class should look like the following listing.

```
package com.example.e4.rcp.todo.model;

import java.util.Date;

public class Todo {

  private final long id;
  private String summary = "";
  private String description = "";
  private boolean done = false;
  private Date dueDate;

  public Todo(long id) {
    this.id = id;
  }

  public Todo(long id, String summary, String description, boolean done,
      Date dueDate) {
    this.id = id;
    this.summary = summary;
    this.description = description;
    this.done = done;
    this.dueDate = dueDate;
  }

  public long getId() {
    return id;
  }

  public String getSummary() {
    return summary;
  }

  public void setSummary(String summary) {
    this.summary = summary;
  }

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }

  public boolean isDone() {
    return done;
  }

  public void setDone(boolean done) {
    this.done = done;
  }

  public Date getDueDate() {
    return new Date(dueDate.getTime());
  }

  public void setDueDate(Date dueDate) {
    this.dueDate = new Date(dueDate.getTime());
  }

}
```

Note: Why is the *id* field marked as final? We will use this field to generate the `equals` and `hashCode()` methods therefore it should not be mutable. Changing a field which is used in the `equals` and `hashCode()` methods can create bugs which are hard to identify, i.e., an object is contained in a `HashMap` but not found.

8.7. Generate toString(), hashCode() and equals() methods

Use Eclipse to generate a `toString()` method for the `Todo` class based on the *id* and *summary* field. This can be done via the Eclipse menu *Source → Generate toString()...*.

Also use Eclipse to generate a `hashCode()` and `equals()` method based on the *id* field. This can be done via the Eclipse menu

Source → Generate hashCode() and equals()....

8.8. Write a copy() method

Add the following `copy()` method to the class.

```
public Todo copy() {
  return new Todo(this.id, this.summary,
      this.description, this.done,
      this.dueDate);
}
```

8.9. Create the interface for the todo service

Create the following `ITodoService` interface.

```
package com.example.e4.rcp.todo.model;

import java.util.List;

public interface ITodoService {

  Todo getTodo(long id);

  boolean saveTodo(Todo todo);

  boolean deleteTodo(long id);

  List<Todo> getTodos();
}
```

8.10. Define the API of the model plug-in

Define that the `com.example.e4.rcp.todo.model` package is released as API. This requires that you export this package.

For this open the `MANIFEST.MF` file and select the *Runtime* tab. Add `com.example.e4.rcp.todo.model` to the exported packages.

9. Exercise: Service bundle

9.1. Target of the exercise

In this exercise you create a plug-in for a service implementation which provides access to the data.

This service implementation uses transient data storage, i.e., the data is not persisted between application restarts. To persist the data you could extend this class to store the data for example in a database or the file system. As this storage is not special for Eclipse RCP applications, it is not covered in this book.

9.2. Create a data model provider plug-in (service plug-in)

Create a new simple plug-in (see **Section 1.5, "Naming convention: simple plug-in"**) project called *com.example.e4.rcp.todo.services*. This plug-in is called *todo service* plug-in in the following description.

> **Tip:**The MacOS operating system treads folders ending with *`.service`* special, therefore we use the *`.services`* ending.

9.3. Define the dependencies in the service plug-in

Add the `com.example.e4.rcp.todo.model` plug-in as dependency to your service plug-in. To achieve this open the *`MANIFEST.MF`* file and select the *Dependencies* tab and add the `com.example.e4.rcp.todo.model` package to the *Imported Packages*.

9.4. Provide an implementation of the ITodoService interface

Create the `com.example.e4.rcp.todo.services.internal` package in your service plug-in and create the following class.

```
package com.example.e4.rcp.todo.services.internal;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.model.Todo;
```

/** * example service impl, data is not persisted * between application restarts * */

```
public class MyTodoServiceImpl implements ITodoService {

  private static int current = 1;
  private List<Todo> todos;

  public MyTodoServiceImpl() {
    todos = createInitialModel();
  }

  // always return a new copy of the data
  @Override
  public List<Todo> getTodos() {
    List<Todo> list = new ArrayList<Todo>();
    for (Todo todo : todos) {
      list.add(todo.copy());
    }
    return list;
  }
```

```java
// create a new or update an existing Todo object
@Override
public synchronized boolean saveTodo(Todo newTodo) {
  Todo updateTodo = findById(newTodo.getId());
  if (updateTodo == null) {
    updateTodo= new Todo(current++);
    todos.add(updateTodo);
  }
  updateTodo.setSummary(newTodo.getSummary());
  updateTodo.setDescription(newTodo.getDescription());
  updateTodo.setDone(newTodo.isDone());
  updateTodo.setDueDate(newTodo.getDueDate());

  return true;
}

@Override
public Todo getTodo(long id) {
  Todo todo = findById(id);

  if (todo != null) {
    return todo.copy();
  }
  return null;
}

@Override
public boolean deleteTodo(long id) {
  Todo deleteTodo = findById(id);

  if (deleteTodo != null) {
    todos.remove(deleteTodo);
    return true;
  }
  return false;
}

// example data
private List<Todo> createInitialModel() {
  List<Todo> list = new ArrayList<Todo>();
  list.add(createTodo("Application model", "Flexible and extensible"));
  list.add(createTodo("DI", "@Inject as programming mode"));
  list.add(createTodo("OSGi", "Services"));
  list.add(createTodo("SWT", "Widgets"));
  list.add(createTodo("JFace", "Especially Viewers!"));
  list.add(createTodo("CSS Styling","Style your application"));
  list.add(createTodo("Eclipse services","Selection, model, Part"));
  list.add(createTodo("Renderer","Different UI toolkit"));
  list.add(createTodo("Compatibility Layer", "Run Eclipse 3.x"));
  return list;
}
```

```
  private Todo createTodo(String summary, String description) {
    return new Todo(current++, summary, description, false, new Date());
  }

  private Todo findById(long id) {
    for (Todo todo : todos) {
      if (id == todo.getId()) {
        return todo;
      }
    }
    return null;
  }

}
```

9.5. Create a factory

Create a new class called *TodoServiceFactory* in the `com.example.e4.rcp.todo.services` package. For this you might need the following tip.

> **Tip:**In its default configuration the Eclipse IDE hides parent packages if they don't contain any classes. During the specification of your class you can define the correct package. This is depicted in the following screenshot.

The class provides access to your `ITodoService` implementation via a static method. It can be considered to be a *factory* for the `ITodoService` interface. A factory hides the creation of the concrete instance of a certain interface.

```
package com.example.e4.rcp.todo.services;

import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.services.internal.MyTodoServiceImpl;
```

/** * factory provides access to the todo service provider * */

```
public class TodoServiceFactory {

  private static ITodoService todoService = new MyTodoServiceImpl();

  public static ITodoService getInstance() {
    return todoService;
  }

}
```

9.6. Export the package in the service plug-in

Export the `com.example.e4.rcp.todo.services` package via the `MANIFEST.MF` file on the *Runtime* tab, so that it is available for other plug-ins.

> **Note:**Please note that the Eclipse tooling does not support the export of empty packages. You have to create at least one class in the package before you are able to export it.

10. Tutorial: Using the Activator and exporting your bundle

In this exercise you create another bundle which uses an `Activator`. You also run it within Eclipse At the end of this chapter you will also export your bundle to use it later in a stand-alone OSGi server.

10.1. Create a new Bundle

Create a new simple plug-in project "com.vogella.osgi.firstbundle.internal" via *File → New → Other... → Plug-in Development → Plug-in Project*.

10.2. Coding

Create the following thread class.

```java
package com.vogella.osgi.firstbundle.internal;

public class MyThread extends Thread {
  private volatile boolean active = true;

  public void run() {
    while (active) {
      System.out.println("Hello OSGi console");
      try {
        Thread.sleep(5000);
      } catch (Exception e) {
        System.out.println("Thread interrupted " + e.getMessage());
      }
    }
  }

  public void stopThread() {
    active = false;
  }
}
```

Change the class Activator.java to the following.

```java
package com.vogella.osgi.firstbundle;

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

import de.vogella.osgi.firstbundle.internal.MyThread;
```

```
public class Activator implements BundleActivator {
  private MyThread myThread;

  public void start(BundleContext context) throws Exception {
    System.out.println("Starting com.vogella.osgi.firstbundle");
    myThread = new MyThread();
    myThread.start();
  }


  public void stop(BundleContext context) throws Exception {
    System.out.println("Stopping com.vogella.osgi.firstbundle");
    myThread.stopThread();
    myThread.join();
  }

}
```

10.3. Run

Select your *MANIFEST.MF* file, right-click it and select *Run As → Run Configuration* . Create an OSGi Framework launch configuration. deselect all bundles except your de.vogella.osgi.firstbundle. Afterwards press the *Add Required bundles* . This adds the `org.eclipse.osgi` bundle to your run configuration.

Run this configuration. Every 5 second a new message is written to the console.

> **Tip:**In case you are wondering in which folder OSGi starts you find it in ${workspace_loc}/.metadata/plugins/org.eclipse.pde.core/ <runconfig> . This folder list the installed bundles via the file "dev.properties" and set the Eclipse workspace as reference in the config.ini file via the osgi.bundles=reference\:file\ statement. This way you can update your bundles in the running OSGi environment directly without any deployment.

10.4. Export your bundle

Export your bundle. This will allow you to install it into a OSGi runtime. Select your bundle and choose File -> Export -> Plug-in Development -> "Deployable plug-ins and fragment".

Unflag the option to export the source.

11. Running a stand-alone OSGi server

This chapter will show how to run Equinox as a OSGi stand-alone runtime.

In your Eclipse installation directory identify the file org.eclipse.osgi*.jar. This file should be in the "plugin" folder. Copy this jar file to a new place, e.g., c:\temp\osgi-server. Rename the file to "org.eclipse.osgi.jar".

Start your OSGi server via the following command.

```
java -jar org.eclipse.osgi.jar -console
```

You can use "install URL" to install a bundle from a certain URL. For example to install your bundle from "c:\temp\bundles" use:

```
install file:c:\temp\bundles\plugins\de.vogella.osgi.firstbundle_1.0.0.jar
```

Tip: You probably need to correct the path and the bundle name on your system.

You can start then the bundle with start and the id.

Tip: You can remove all installed bundles with the -clean parameter.

12. About this website

12.1. Donate to support free tutorials

Please consider a contribution if this article helped you. It will help to maintain our content and our Open Source activities.

12.2. Questions and discussion

Writing and updating these tutorials is a lot of work. If this free community service was helpful, you can support the cause by giving a tip as well as reporting typos and factual errors.

If you find errors in this tutorial, please notify me (see the **top of the page**). Please note that due to the high volume of feedback I receive, I cannot answer questions to your implementation. Ensure you have read the **vogella FAQ** as I don't respond to questions already answered there.

12.3. License for this tutorial and its code

13. Links and Literature

13.1. Source Code

Source Code of Examples

13.2. OSGi Resources

OSGi homepage

Equinox homepage

Equinox quick start guide

OSGi blueprint services

hawtio dependency visualization tool

13.3. vogella Resources

public and inhouse training. Whichever course you decide to take, you are guaranteed to experience what many before you refer to as **“The best IT class I have ever attended”**.