Migration

The introduction of plugins significantly eases migration issues because of the tight coupling between the software version of a plugin and the data model it uses. The plugin itself is responsible for implementing proper data model migration and XperienCentral offers an easy API to implement this.

In This Topic

Version number

Each plugin has a version number that consists of three numbers. The version number has a meaning and it is very important to use it properly because it tells users of the plugin the kind of changes they can expect between two separate versions of the plugin. The three parts of the version number have the following meaning:

- **Major version** - A major software update of the plugin. Such an update may change exposed APIs which may cause incompatibility with plugins that depend on it.
- **Minor version** - A backwards compatible plugin update. The exposed API has not been changed and plugins that depend on this plugin will therefore be compatible with the updated version. The API can be extended with additional functions.
- **Micro version** - Update without any change to the interface or published services.

If you properly use this version numbering system, other developers that may use services or classes exposed by this plugin will know in advance whether they should expect problems when upgrading to a later version of the plugin.

The version number of the plugin should be defined in the `pom.xml` of the plugin. For example to define the version to be 1.0.2:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="…" ...>
        ...
        <groupId>...</groupId>
        <artifactId>...</artifactId>
        <version>1.0.2</version>
        ...
</project>
```

When defining version numbers for your plugin, be sure that they conform to the development guidelines (G017 and G019 in particular).

Back to Top

Plugin Updates

When updating a plugin, the version number of the plugin must always be updated. Depending on the type of change the major, minor or micro version number must be updated:

- **Major update** - Property definitions of node types may be changed or removed, methods of exposed interfaces and classes may be changed in signature.
- **Minor update** - Additional property definitions may be added to node types but existing property definitions are not be changed or removed. Additional methods may be added to exposed interfaces and classes but the signature of existing methods are not be changed.
- **Micro update** - Node types are unaffected. The implementation of methods of exposed interfaces and classes may be changed, but no new methods have been added and the signature of existing methods are not changed.

Proper versioning of your plugin is a required guideline - see G017 and G019.

Back to Top

Changes in the Data Model

The version number plays an important role in data model migration. A plugin version is tied to a particular data model and when the data model is updated, the plugin's version should also change. Using the version number it is possible to implement the migration logic that is needed to upgrade content managed by the entity manager created from an x.y.z version of the plugin to x.y.z+1.

In order to implement the migration of the data model from x.y.z to x.y.z+1 you should implement the `upgradeContent` method in the class defining the data model. Arguments passed to this method are the `fromVersion`, which indicates the current version of the data model and a collection of nodes on which the migration must be performed. This method is invoked by the framework automatically if a new version of the plugin is installed. Note that a plugin doesn't necessarily have to define this method - it's only invoked, using reflection, if it exists.

The implementation of `upgradeContent` is up to the developer. It may be implemented in a way that it is capable of upgrading from any version to the latest version or that it requires previous updates first. In the latter case, the method should throw an `IncompatibeUpdateException` when the implementation of `upgradeContent` is not capable of migrating from the `fromVersion` to this version. The next part clarifies the migration implementation further with an example.

- The `upgradeContent` method will only be invoked on nodes managed by the Entity Manager.
- The `upgradeContent` method is static. To use the Entity Manager (or any other service) in a static method, see Using Services in Static Methods.
- Proper handling of content migration is a required guideline - see G020.

Explicitly setting the Platform Dependency

By default, the minimum and maximum version of XperienCentral that a plugin can run on is calculated as follows: If the version of XperienCentral for which you build the plugin is x.y.z, then the minimum version of XperienCentral on which the plugin can run is x.y and the maximum version is (x+1).0. It is also possible to explicity set the minimum and maximum versions of XperienCentral on which a plugin can run in the POM file using the setting properties `webmanager.wcb-min-version` and `webmanager.wcb-max-version`. For example:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>nl.gx.webmanager.wcbs</groupId>
                <artifactId>XperienCentral-wcbs</artifactId>
        <version>10.6.0-SNAPSHOT</version>
        </parent>
        <packaging>osgi-bundle</packaging>
        <artifactId>wmpjcrbrowser</artifactId>
        <name>JCR Browser</name>
        <properties>
                <Webmanager.wcb.max-version>10.4</Webmanager.wcb.max-version>
                <Webmanager.wcb.min-version>10.0</Webmanager.wcb.min-version>
        </properties>
...
</project>
```

If `webmanager.wcb.min-version` and `webmanager.wcb.max-version` are not specified, the default versions as described above are calculated and used.

Example

An example of a "Keyword" plugin that changes its node type definition (data model) over time is presented here. The following updates to the node type definition are performed:

- The 1.0 version of the plugin contains a `keyword` property
- The 1.1 version adds an additional `secondaryKeyword` property
- The 1.2 version adds an additional `keywords` property is added. The properties `keyword` and `secondaryKeyword` are deprecated
- The 2.0 version removes the `keyword` and `secondaryKeyword` properties

The `fromVersion` argument of the `upgradeContent` method is the version of the plugin currently installed. This example uses an approach that requires previous version updates, which means that each version must be installed separately. So to update from 1.0 to 2.0, the system administrator first has to deploy 1.1, then 1.2 and finally 2.0. The code snippets below provide examples of how the code looks like for the several versions.

Version 1.0

```
@Property
public String getKeyword(){...}
public void setKeyword(String keyword) {...}
```

Version 1.1

```
@Property
public String getKeyword(){...}
public void setKeyword(String keyword) {...}
@Property
public String getSecondaryKeyword(){...}
public void setSecondaryKeyword(String keyword) {...}
```

Version 1.2

```
@Property
@Deprecated
public String getKeyword(){
        return getKeywords()[0];
}
public void setKeyword(String keyword) {
        setKeywords(new String[]{keyword, “”});
}
@Property
@Deprecated
public String getSecondaryKeyword(){
        return getKeywords()[1];
}
public void setSecondaryKeyword(String keyword) {
        setKeywords(new String[]{“”, keyword});
}

@Property
public String[] getKeywords() {...}
public void setKeywords(String[] keywords) {...}

public static void upgradeContent(String fromVersion,
Node[] nodes) throws
IncompatibeUpdateException {
        if (fromVersion.compareTo(“1.1”) != 0) {
                String msg = “This update requires 1.1. “;
                msg += “Current Plugin version is “ + fromVersion;
                throw new IncompatibeUpdateException(msg);
    } else {
                updateKeywords(nodes);
    }
}

public static void updateKeywords(Node[] nodes) {
        for (int i=0; i<nodes.length;i++) {
                String pKeyword = nodes[i].getProperty("keyword").getString();
                String sKeyword = nodes[i].getProperty("secondaryKeyword").getString();
                String[] keywords = new String[]{pKeyword, sKeyword};
                nodes[i].setProperty(“keywords”, keywords);
        }
        nodes[0].getSession().save();
}
```

Version 2.0

```
@Property
public String[] getKeywords() {...}
public void setKeywords(String[] keywords) {...}

public static void upgradeContent(String fromVersion, Node[] nodes) throws IncompatibeUpdateException {
        if (fromVersion.compareTo(“1.2”) != 0 {
                String msg = “This update requires 1.2. “;
                msg += “Current Plugin version is “ + fromVersion;
                throw new IncompatibeUpdateException(msg);
        }
}
```