Joomla! Whitepaper

Update logic

Author:
Wilco Jansen

Subject:
Current installer offers methods for installation and un-installation of Joomla! packages. Installation of extensions in Joomla! can be done in the installer with three methods: uploading a package from the client browser, installing from a selected (local) directory and downloading from a selected URL. This whitepaper describes a proposal for development of update functionality to the Joomla! installer.

Change management:

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
<th>Change description</th>
</tr>
</thead>
<tbody>
<tr>
<td>04-03-2008</td>
<td>1.0</td>
<td>Initial version</td>
<td></td>
</tr>
</tbody>
</table>
Table of contents

1. **INTRODUCTION** ...................................................................................................................... 3

1.1. Scope........................................................................................................................................... 3

1.2. Objective of this document.......................................................................................................... 3

1.3. General remarks.......................................................................................................................... 3

1.4. Definitions.................................................................................................................................... 3

1.5. License......................................................................................................................................... 3

1.6. Setup of this white paper ............................................................................................................. 4

2. **UPDATE CONCEPT** ................................................................................................................ 5

2.1. Current installation procedure .................................................................................................... 5

2.2. New installation and update logic ............................................................................................... 5

2.3. Required changes to the installer ............................................................................................... 8

3. **DESCRIPTION OF THE XML ADDITIONS** .......................................................................... 9

3.1. Description of basic additions to the installer packages ............................................................. 9

3.2. Homing URL XML definition .................................................................................................... 9

3.3. Homing XML specification ........................................................................................................ 9

3.4. Dependencies XML definition ................................................................................................... 10

APPENDIX 1: REFERENCES............................................................................................................ 13
1. Introduction

1.1. Scope

The scope of this document is a full design on improvement in the installer of Joomla! This whitepaper strives to define a methodology for extending the current installation logic with the possibility to update Joomla! extensions.

1.2. Objective of this document

Proposal is created as part of the call for white paper for Joomla! 1.6, see http://forum.joomla.org/viewforum.php?f=500 for details of this call for white papers.

1.3. General remarks

This white paper is based upon the existing installation logic available within Joomla! and research performed during the Google Summer of Code 2006. There are more proposals in the Joomla! 1.6 white paper forum that describe (parts) of the required logic, within this white paper we try to incorporate most of those proposals.

Within the research project of the Summer Of Code 2006 we also defined a link to the Joomla! extensions directory. In this white paper this logic is out of scope, and could be subject to implementation in future versions of Joomla!

1.4. Definitions

- **Extensions** – components, modules, language packs, plug ins and templates.
- **Installer** – The Joomla! installation option, including all the application -and CMS framework code.
- **Package** – a compressed collection of files that hold an extension that can be installed using the Joomla! installer.

1.5. License

GNU GPL
1.6.  Setup of this white paper

Chapter 1, *introduction*, contains a brief introduction to this whitepaper.

In chapter 2, *update concepts*, a basic description is given of the conceptual pieces of the updater.

The XML definitions of all components that make the updater work are described in chapter 3, *description of XML additions*.
2. Update concept

Current installer offers methods for installation and un-installation of Joomla! packages. Installation of extensions in Joomla! can be done in the installer with three methods: uploading a package from the client browser, installing from a selected (local) directory and downloading from a selected URL.

Joomla! 1.5 provides options for installing, and un-installing, but not for 'patches' or 'upgrades'. To move from one version of an extension to another version you currently need to un-install, and re-install. With this you lose data, or you need to manually update (including altering tables).

Joomla! update aims to provide a framework to streamline updating. This can be done manually by one of the current installation methods, or by performing a check on updateable extensions. This check is done by scanning the installed extensions and seek for the remote status XML file. This file is defined by the 3rd party developers and provides information about available versions, including the download URL of those extensions. The updater will build a list with possible upgrade candidates, and offers the user the possibility to update.

2.1. Current installation procedure

The current installation process is made out of a series of sequentially performed steps. Each step can fail, and when that happens the current installation is rolled back. This mechanism of rolling back the steps can be improved because there have been reports in Joomla! 1.5 that the rollback is not always working as designed. Suggestion is to optimize the installer code during the implementation of this whitepaper.

2.2. New installation and update logic

It is obvious that an update has some additional steps to perform. First of all you need to update existing code and possible changes to the database structure. Basically this is a simple addition to the current installation code (it even has an update method available already), but the challenge is the rollback logic here.

Imagine the situation that the new updated code is copied, the table update is executed and in the final stage an error occurs. A rollback would include restoring the old code, and also the database tables. To solve this problem we need to implement a backup solution that supports this.
We propose to improve the current installer, in general we propose to add the following general logic (implemented for new installations and updates):

- Dependency validation.
- Implement new triggers. We currently have install and un-install, we propose to also add some new triggers to improve the flexibility of the installer.
- Possibility to update installed extensions manually.
- Add an option to automatically detect the availability of new versions of the extensions.

**Dependencies**

In general there are two types of dependencies:

- Version dependencies. This is a dependency when updating your component, this is important when there is a specific upgrade path from one version to another.
- Extensions dependencies. In this situation an extensions depends on the availability of another extension. Of course these kind of dependencies also apply to a specific version of the extensions that is defined in the dependency.

Within Joomla! 1.5 there is a basic validation on dependencies implemented. The version tag is validated, and when there is a pre-1.5 install file, and the legacy plug-in is not enabled the installation fails.

Beside this the current installer has no mechanisms in place to validate dependencies. The dependencies are defined in a new section in the XML installation file. Basically it is an addition to the current XML syntax so the installer knows what to validate during installation and updating. We describe the method that defines dependencies between extension-versions.

![Diagram](image)

The white version boxes relate to a specific component. The red boxes relate to the corresponding module. The component starts with version 1.0 and progresses to the next released versions 1.0.1, 1.0.2, 1.1 and 1.2. Parallel at version 1.0 release of the component version 1.0 of the module is published. In this simplified example the dependencies of the module keep intact until version 1.2 of the component is released. In the appendix we will show the example XML files for every released version.
Manual update

The installer offers three methods of installation currently:

1. Uploading a package from the client browser.
2. Installing from a selected (local) directory.
3. Downloading from a selected URL.

When an administrator wants to update an extension, these three options are also available when updating. Instead of the “install” option the administrator can select the “update” option.

Automated detection of new versions

Besides manual updating we describe the method on automatic detection of new versions. The key concept to make this work is a new XML tag that holds a “homing” URL. This URL points to a location where the installer can validate for a XML definition file.

This XML definition file contains a reference to all available versions for the specific extension. It looks a lot like the dependency definition in the installer XML, but it will only describe the version dependencies, and not dependencies between other extensions and/or extension types.

Triggers

Current installer has php file triggers that reside in the packages; {package}.install.php and {package}.uninstall.php. For backward compatibility reasons these files get triggered during installation or uninstallation unless the following situations are matched:

- The {package}.install.php and {package}.uninstall.php are not available in the extension package.
- The existence of the installer manifest file; {package}.manifest.php

The manifest file is a PHP class holding the following “trigger” methods:

- preflight – this method is executed before any installs or updates, but NOT uninstalls. It is typically used to check whether the package can be installed.
- install – this is the install method, without this method the user won't be able to install the package.
- update – this method is fired when the users updates the package, if it is not defined, the installer uses the install method to update the packages, effectively overwriting the old package (including tables). This is useful for smaller packages that don't really need a separate update script.
- postflight – this method is executed after any install or update of the package, but NOT uninstalls.
- uninstall – this method is executed when the user chooses to uninstall the package. Although it is optional, it is HIGHLY recommended that you define it so the package can be uninstalled. If this method is not defined, the uninstall option will not be shown up in the installer.

Snapshot determination

We need to be able to roll back during installation when an error occurs. For component,
modules and templates the installation XML file holds a list with files and/or directories that are used within the installation procedure. Language files don’t hold this specification, but determining which directory needs to be secured is pretty straightforward.

Because of this the installer is able to determine which files belong to the extension, and before an update the installer can make a snapshot (by creating a backup). The only challenge is the state of the database tables. The installer will also create a snapshot of the database tables. The installer will determine the tables by scanning the installation option within the installer XML.

### 2.3. Required changes to the installer

- New options in the backend of the installer: update, check for new versions.
- Implementation of a back method: snapshot of the current extension (code + tables).
- Implementation of dependency validation.
3. Description of the XML additions

The installer XML holds all specifications for package definition. This chapter described the XML definitions.

3.1. Description of basic additions to the installer packages

The installer will have the following additions in general:

- Homing URL XML definition. Specifies the location of the XML file that defines the available versions.
- Homing XML specification. Contains the specifications on what versions are available. This XML specification resides on the URL provided in the homing URL XML definition.
- Dependencies XML definition. This is a description of the dependencies that are valid for the extension you want to install or update.

3.2. Homing URL XML definition

When the extension developer wants to support the automatic detection for availability of new versions a new tag is introduced to define the homing URL. When this option is defined the installer knows that the automatic detection of updates is enabled, and it will look for an XML file on the defined URL. This XML file holds a definition of available versions of the specified extension, extensions types and the URL where the package can be found.

The syntax is a one-line addition to the installer XML of the package:

<homing url="http://sitewhereextensionresided.org"/></homing>

3.3. Homing XML specification

The version dependency defines the previous compatible versions. This XML specification needs to define not only which versions can be upgraded to another, but also what the next possible upgrade candidate is for the current extension.

Let us give an example: We have a module that is released with the following version information: 1.0, 1.1, 2.0

When version 1.0 is released, we do not know what the next version will be. So when 1.1. is released the XML specification needs to tell us that there is an upgrade available for version 1.0. The same needs to happen when version 2.0 is released. But what if the migration to 2.0 is only possible when someone has upgraded to 1.1, you need to specify this in the XML definition. So when you specify version dependencies, you only need to define that the current installation code support the upgrade, for the homing option you need to define more.

So how is this done? For every version you need to specify the “version compatibility”, so
instead of one entry for the current version, you need to specify multiple entries and per entry you need to define the version dependencies (only one entry is shown below).

```xml
<?xml version="1.0" encoding="utf-8"?>
<Package>
    <Name>projectname</name>
    <Type>component|module|plugin|language|template</type>
    <Version>x.y.z</version>
    <Dependencies>
        <Dependency>x.y.z</dependency>
        <Dependency>x.y.z</dependency>
        ...
    </Dependencies>
    <url>http://locationtodownloadpackage</url>
    <crc>{checksumvalue}</crc>
</package>
```

For every version of the extension that is released an entry need to be available. With this information the installer is able to determine what versions can be used for updating to a new version.

You can define as much as needed entries in the dependency section. The version dependency given can be used in the following manner:

- **Version=“1”** – Dependency applies to all versions 1.y.z where y and z can be any minor or maintenance encoding.
- **Version=“1.6”** – Dependency applies to all versions 1.6.z where z can be any number.
- **Version=“1.6.0”** – Dependency applies only to version 1.6.0 of the extension.

The URL specification defines the download location for these extensions. The URL reference needs to point to the full URL location of the package (so directories are not allowed).

The CRC holds the MD5 hash of the package. This value will be checked after the extensions has been downloaded. This entry is optional, but strongly recommended.

---

### 3.4. Dependencies XML definition

The dependencies are an addition to the current syntax so installer can determine what dependencies are in place. Remember that package definitions only describe the relationship between extensions, but in the dependency definition it is not defined what to do with those dependencies. The installer will succeed if all dependencies are met, else a warning message with a hint will be returned to the end-user.

The dependencies are defined in the following XML syntax:

```xml
<dependencies>
    <dependency name="projectname" type="component|module|plugin|language|template" version="x.y.z.">
</dependencies>
```
The number of dependencies you want to define is unlimited. Remember this is additional to the existing XML installer code. What it does is that it creates a relation between the package that will be installed, and the dependencies it has to other extensions of your website installation. The dependencies are uniquely defined using the following fields in the XML installer file:

- **Project name.** This is the name of the extension as defined in the `<name>` tag of the XML install file. Keep in mind that the project name can occur multiple times if you define several versions and/or package types. The combination project name, package type and version may only exist once. If no project name is given, the actual name from the installer file will be used.

- **Package type.** The package types refer to existing extensions types; component, module, plug-in, template and language file. If no package type is given, the actual package type from the installer file will be used.

- **Version dependencies.** Defines what version the dependency applies to. The major, minor and maintenance encoding is obligated. The version approach is layered, this means that if a major version is defined, the dependency applies to all versions of the extension. It is possible to have the following layering:
  - Version=”1” – Dependency applies to all versions 1.y.z where y and z can be any minor or maintenance encoding.
  - Version=”1.6” – Dependency applies to all versions 1.6.z where z can be any number.
  - Version=”1.6.0” – Dependency applies only to version 1.6.0 of the extension.

**Simple Example definition**

Below you see an example dependency definition. This could be an example of a content component that is updated to version 2.0. In the dependency definition you define that the update depends on versions that have version 1.x.y. So if a user wants to update the same component from 0.5 to 2.0, the installer will fail because the dependency does not meet. If the user tries to upgrade

```xml
<install type="module" version="1.0">
  <name>content</name>
  <version>2.0</version>
  <dependencies>
    <dependency type="component" name="test" version="1"></dependency>
  </dependencies>
</install>
```

**Advanced Example definition**

A more advanced example is shown below. Here we again try to update a component to version 2.0, but this time you only can update to 2.0 from versions 1.5.0, 1.5.1, 1.6 and 1.7 to 2.0, and also the update depends on another component (com_dependent) and a plugin (plg_test).

```xml
<install type="module" version="1.0">
  <name>content</name>
  <version>2.0</version>
</install>
```
<dependencies>
  <dependency type="component" name="test" version="1.5.0"/>
  <dependency type="component" name="test" version="1.5.1"/>
  <dependency type="component" name="test" version="1.6"/>
  <dependency type="component" name="test" version="1.7"/>
  <dependency type="component" name="dependent" version="1"/>
  <dependency type="plugin" name="test" version="1"/>
</dependencies>

In the pre-installation validation the installation or update will fail if these dependencies are not met. The dependency check most of the times will be implemented for extension types “module” and “plugin” because these extension types most of the time rely on the availability of a specific version of the corresponding component.

As described, there is no limit to the number of dependency definitions in the XML file, so it is possible to create pretty advanced dependency schemes here.
Appendix 1: References

Below you see references to a variety of resources holding additional information. The title of the reference is in bold, the author(s) are marked in Italic and the version or date information is marked underlines.