Joomla! Extensions Installer White Paper

How to design an extensions installer which better serves our target audience

Workflow of the extensions installer

The workflow is not changed radically, providing the users and developers with strong backwards compatibility. The major change is in how the workflow is executed.

The current com_installer will try to run all steps in a single page load. The downside is that on slow, shared hosting these steps may take too long and use up too much resources leading to PHP timeout, PHP memory exhaustion or CPU time allocation being exceeded. In all cases the result is a blank page and failed or, worse, half installed extensions. In the former case the user is simply left frustrated. In the latter case they are left with a broken site, blaming either the innocent developer of the extension they tried to install or Joomla! itself. In the end of the day it's lost revenue and market share for Joomla! and its developers. This gives Joomla! a very bad reputation among users.

The solution is brilliantly simple, as has been already demonstrated by the Joomla! Update component. Lengthy operations, if possible to be split, must be split in separate page loads and/or AJAX calls. Even better, by separating the execution flow into self-contained steps we can provide new features such as avoiding the need to re-download / re-upload packages, dependency handling and so forth.

Step 1. Getting the packages

The first thing we need is getting all the packages we will be installing. There are four different methods.

Case 1.a. Upload

No difference to the existing installer. The file is uploaded and then moved to its final destination inside the site's temporary directory.

Improvements required: provide intelligible error messages about the reasons the upload has failed, along with instructions of fixing it. The actions can be semi-automated. For example, if the temporary directory doesn't exist, propose to change it to the default one. If it's not writeable, propose to fix its permissions / ownership. If the file was too big, say so.

Case 1.b Import from URL

Radically different to the existing installer. Right now we are trying to download each package archive at once. This is prone to failure if the package archive is too big and the connection too slow. Instead, we can use an AJAX-powered, split step downloader which downloads 1Mb at a time and checks the time elapsed. If too much time has elapsed the download continues in a separate AJAX call. This is already implemented in Akeeba CMS Update, see https://github.com/nikosdion/cmsupdate/tree/master/component/backend/lib

Improvements required:
1. Provide intelligible error messages about the reasons the download has failed.
2. Allow more than one package URLs to be specified. They will be downloaded one after the other. This is useful for updating multiple extensions at once and for downloading dependencies (see Step 3)

Case 1.c Import from directory

No further action required. We can skip directly to Step 3.

Case 1.d Install over the web

This is a variation of case 1.b, where the download URL is provided by a plugin. The plugin is executed in the setup phase of the URL importer, therefore we can reuse the same view (DRY for the win!).

Step 2. Extracting the packages

That's a major design flaw of our current architecture. When it was designed ten years ago the average extension was a few Kilobytes big, making it possible to extract a whole package directly. Nowadays it is quite common to have extension packages in the multi-megabyte range. Think about CiviCRM. Clearly, extracting each package in one page load will not work on shared hosting.

For this reason we propose using the AJAX-powered extraction code found in com_joomlaupdate's restore.php file (a.k.a. Akeeba Restore).

Improvements required:
1. Handling tar.gz / tar.bz2 archives. Each one of those files is a tar file which is compressed in whole using GZip / BZip2. We need a pre-step where we extract the tar.gz / tar.bz2 archive into a tar archive. Then restore.php can handle the extraction of the uncompressed tar archive. restore.php might be able to do that automatically, as the first extraction step of a .tar.gz. .tgz, .tar.bz2, .tbz archive.
2. tar archives are currently not supported by restore.php, but we can easily add support for that.
3. "Package" extensions are archives which contain other archives. We need to parse the manifest XML file right after extracting an archive. If it's a package, loop all of its contents and extract them into separate directories. A database table can hold the mapping between each extracted extension, its version and its location in the site's temporary directory. We propose the use of a table instead of the session due to the limited maximum size (64K) of session storage when using the default database session handler.

Step 3. Dependency management

This is something not currently implemented by Joomla!. We can easily add dependency management by providing an aptly-named <dependencies> section in the XML manifest.

Step 3.1. Core dependencies

This step checks the dependencies of the package to core aspects of the site:

- Joomla! version (min / max versions)
- PHP version (min / max versions)
- Database technology (list of supported DB server types)
- Database server version (min / max versions)

If any of these dependencies is not satisfied the installation halts, Step 4.6.b is executed and a descriptive error is printed on the page to let the user know why the installation could not proceed.

If a dependency is not specified we consider it satisfied by default (backwards compatible behaviour).

Step 3.2. Extension dependencies

The XML manifest provides a list of required and optional extensions, along with their min / max version numbers and, optionally, their download URL.

The optional extensions list is reserved for future use. The intention is to create, in due time, a system similar to apt or PEAR which allows users to install optional (suggested) packages. This is outside the scope of the current effort of this Working Group.

The required extensions are first searched in the #__extensions table of the site. We can have the following cases:

- The extension is installed and of an acceptable version. The dependency is satisfied.
- The extension is not installed or installed but of a lower version then the min requirement. A “must download” item is created.
- The extension is installed but of a higher version than the max requirement. We halt the installation, Step 4.6.b is executed and a descriptive error is printed on the page to let the user know why the installation could not proceed.

If there are “must download” items we need to go through them.

- If a “must download” item is present but the extension doesn’t provide its XML update stream location we halt the installation, Step 4.6.b is executed and a descriptive error is printed on the page to let the user know why the installation could not proceed.
- If the “must download” item is present in the extracted “package” extension we remove it from the list.
- Otherwise we fetch the update information of the “must download” dependencies and keep the latest version and download location of each one.
- Next up, we scan the dependencies of already installed extensions on the site. If their dependencies would not be satisfied after installing the latest versions of the “must download” dependencies we halt the installation, Step 4.6.b is executed and a descriptive error is printed on the page to let the user know why the installation could not proceed.

If there are dependencies to download we ask the user for confirmation. If they agree we redirect them to step 1.b, putting the list of dependencies to install in the download URL stack. The process repeats until there are no more dependencies to be downloaded.

Step 3.3. External dependencies (e.g. Composer)

This is added as a forwards compatibility provision. Since there is no integrated Composer support in Joomla! at this point we cannot implement this step. Ideally, each extension would have a composer.json in a predictable location and its contents would be integrated in the site's composer.json, then Composer would run in its own step to fetch the dependencies. Since Composer can only download and install dependencies in a single go with a limited number of methods we feel it's inappropriate for mass-distributed extensions – the main reasons being that it can cause timeout/memory exhaustion and makes extension installation rollback impossible.

Step 4. Installation

This is where the rubber meets the road. We have our packages downloaded and extracted and we need to put all that code into our production environment. However, this gets us to an interesting Catch-22 situation.

If you try to apply production code changes in smaller steps you can end up with half installed libraries or extensions between page loads. This means that your site may not be able to work properly, making it impossible to call the next page load which will copy the rest of the files. But if you try copying all the files at once it takes too long and too much PHP memory which leads to white pages and half-installed extensions on shared hosts which is the build of Joomla!'s target audience.

Interestingly, there is a fairly simple solution: a new entry point file (directly accessible from the web) which doesn't load plugins and modules and which can only be used to execute the necessary AJAX requests to perform the file copying required by the extensions installer. This could be located in administrator/index_installer.php. Since it requires the session cookie of a user with adequate permissions to be present it does not pose a security threat, just like the main administrator/index.php file. All of the following steps take place through AJAX calls to that special file.

Step 4.1. Pre-installation hooks

Here we run the pre-installation hooks of every extension in our installation queue. If any of them decides that the installation cannot proceed we halt the installation, Step 4.6.b is executed and a descriptive error is printed on the page to let the user know why the installation could not proceed.

Step 4.2. Isolating old code and data

This is something that Joomla! doesn't currently do and it's been a long standing wish of its users. Before installing the extension we move existing files and folders into subdirectories of the site's temporary directory. The extension can indicate which folders should be moved and which should be excluded from the move in the same way Phing lets you use include and exclude tags in file lists.

The file move should intelligently use FTP or direct file writes on a file-by-file and folder-by-folder basis. This allows Joomla! to preserve the ownership of the files being copied and effectively work around the "permissions hell" (mixed ownership and permissions) which is currently preventing lots of users from installing extensions on their sites.

The file move takes place in small, AJAX-powered steps to prevent server timeouts.

After all the files have been copied over, the tables of the extension are renamed (optional). The extension can specify which tables should be renamed with a #__bak_ prefix and which should be excluded from this measure. This is great for allowing developers to use a radically different schema in future versions of their extensions while preserving their users' data, something which is onerous with the current extensions architecture.

Step 4.3. File list creation and file copy

This is done in a single step by Joomla! as they're entangled in each extension type's adapter. We will be rewriting this to create a list of all files to be copied (from / to). That list will then be executed in small steps using AJAX. The idea is that each file copy step should last 5±2 seconds to prevent server timeouts. Each file will be copied by FTP or direct file writes, decided on a file-by-file basis.

If file copy is impossible we halt the installation, Step 4.6.b is executed and a descriptive error is printed on the page to let the user know why the installation could not proceed.

Step 4.4. Database update

After all files have been copied, we run the database update for each extension. We propose supporting two different methods for database update:

Traditional (one SQL file per version, executing the update to the previous version). This is problematic in many occasions: extensions which have been around since before Joomla! 1.6, tables modified manually, updates performed by out-of-band (e.g. development, alpha, beta) releases of an extension, reinstalling after a failed installation / update etc.

Schema-aware, XML-based SQL commands. SQL commands are executed based on how the schema looks like in the database, e.g. if tables, fields or indexes are missing, type of fields and so on. This provides maximum flexibility for developers. The code is already written and battle tested: https://github.com/akeeba/fof/blob/development/fof/database/installer.php

If the schema update fails we halt the installation, Step 4.6.b is executed and a descriptive error is printed on the page to let the user know why the installation could not proceed.

Step 4.5. Post-installation hooks

The post-installation hooks of each installed extension are executed, just like the good old installer. Of course if anything returns false we halt the installation, Step 4.6.b is executed and a descriptive error is printed on the page to let the user know why the installation could not proceed.

Step 4.6 Finalisation

Here we finalise the installation effort. If it's successful we clean up, if it's failed we roll back.

Step 4.6.a. Clean-up

We have to delete any downloaded/uploaded extension archives, the subdirectories in the temporary directory holding the extracted extensions and the subdirectories holding the isolated code files from step 4.2. Finally, we remove any #__bak_ tables we created in step 4.2.

Step 4.6.b. Rollback

First we do the inverse of step 4.2. Files are moved from the temporary directories back to production and tables are renamed from #__bak_ to #__. Finally we run step 4.6 to clean up after ourselves.

Updating extensions

Extensions update workflow is largely left unchanged, however we propose a few major tweaks.

Don't load everything at once. One of the fatal flaws in the design of the extensions updater is that it tries to download the update information of all extensions at once. Many sites end up having dozens of update sites, each one requiring 1-2 seconds to respond. This leads to PHP timeouts, memory outages and much frustration. Instead of trying to fetch everything at once, allow com_installer to fetch updates using AJAX, trying to take no more than 5±2 seconds for each AJAX call. It's slightly slower but FAR MORE reliable on shared hosting.

Store the download URL with the update information. Our current updater is a bit braindead. When you fetch the updates it will fetch the update XML file, determine if there's an update available and then store the updated version and the URL *to the update XML file* in the database. If you want to update an extension you have to re-download the same update XML file you originally read to determine if there is an update(!!!), get the download URL and then download the update and execute the installation. That's just plain stupid. Store the download URL in the database and save a lengthy roundtrip to an external server.

Show which extensions are already up-to date. Right now you can only see which extensions are out of date. All right, cool, but what about seeing which extensions are already up to date and what version they are on? We can integrate the update results in the Manage tab and show which extensions are up-to-date, which are not (and what is their latest version + an update button). We can even filter by update status.

Let the user clear the update cache. Seriously, do we need to explain this?

Let the user re-enable disabled update sites. Right now if a connection cannot be made to an update site it is disabled and the ONLY way to re-enable it is by editing the database. Even worse THERE IS NO WARNING THAT THIS HAPPENED. That's just plain stupid and DANGEROUS as the user has no idea that there are updates they are not aware of! You know, maybe there was a temporary network issue, that shouldn't sentence a user's site to not being updated ever again. Just show a list of disabled update sites, let the user enable them (all or a few) and fetch the updates.

Updates to use the install from URL method. Instead of having a complicated new view for installing updates why not apply the DRY principle and use Step 1, case 1.b instead?

Backwards compatibility and provisions for CLI

Joomla! currently provides a method for pointing it to a URL or an extension package and installing it. This is used by CLI scripts which need to automatically install new versions of extensions. This should still be supported by making a few tweaks in the workflow of that backwards compatible method:

1. Download the package (if it's a URL) directly, using the JHttp package
2. Extract the package directly, using the relevant helper classes in Joomla!
3. There is a parameter which determines if dependency management should be optimistic (assume a "Yes" reply to download dependencies) or pessimistic (assume a "No" reply to download dependecies). The dependency determination is the same code.
4. All copies and updates are performed directly, without paying any attention to the elapsed time.

Due to the lengthy nature of this process, this method is best suited for CLI CRON jobs. Surely, developers can use it in code executed over the web but they have to keep in mind that they are risking timeouts and breaking the user’s site in the process.

Furthermore, we propose that developers of extensions actively bear in mind that their extension MUST be able to be installed from the CLI. As such they are suggested to not use the post-installation message for interaction with the user. Instead, use the Post-Installation Messages feature added in Joomla! 3.2 and later. The reason is that this WG will be able to provide CLI scripts to automatically apply updates to all / specific extensions or even a command line extensions installer for advanced users.