PHP-MAVEN 2.1

2012

An analysis on php project types and the way PHP-MAVEN could support them in a flexible way.

Analysis draft

Table of contents

Version 0.0.1 – Martin Eisengardt

Prolog/ License

This document is not meant to be discussed in public at this point. However the document is focused on the open source project PHP-Maven. So you may copy and publish it as long as you do not change it. See the terms of Creative Commons. See http://creativecommons.org/licenses/by-nc-nd/3.0/legalcode for details. As soon as php-maven.org published the licensing may be changed to a less restrictive version-

You are free:

- **to Share** — to copy, distribute and transmit the work

Under the following conditions:

- **Attribution** — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Noncommercial** — You may not use this work for commercial purposes.
- **No Derivative Works** — You may not alter, transform, or build upon this work.

With the understanding that:

- **Waiver**— Any of the above conditions can be waived if you get permission from the copyright holder.
- **Public Domain**— Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.
- **Other Rights**— In no way are any of the following rights affected by the license:
  - Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
  - The author's moral rights;
  - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.
- **Notice**— For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Support in PHP Maven 2.0

Within PHP Maven version 2.0 we are focused on the basics. So we do not specify what type of PHP project we are developing. A simple and small PHP Maven project provides a couple of PHP files and resource files that are bundled together within the PHAR.

This PHAR is installed in repositories. As soon as anyone needs another artifact the PHAR is simply uncompressed within a special target directory.

Lifecycle on simple projects

PHP Maven 2.0 assumes the following lifecycle mappings:

- ***Compile/Prepare***: Copy the php files and resources to the target/classes directory. Extract every dependency in target/php-deps.
- ***Test***: Extract test dependencies (therefore PHPUnit as dependency is required). Start the test cases by setting up the php inclusion path (add target/classes and target/php-deps as well as the testing folders).
- ***Package***: Create a phar from generated target/classes files.
- ***Install***: Install the phar and pom without dependencies into the local maven repository
- ***Deploy***: Install the phar and pom into a remote repository, for example the PHP-Maven nexus.
- ***Site***: generate sites and site reports.

As we can see PHP-Maven cleanly uses the maven suggestions on the common lifecycle. That is fair and it is flexible enough. But it is hard to build projects “out-of-the-box”.

Problems with PEAR imports

PEAR is somehow difficult to understand for PHP-Maven. We will focus on PHP projects first and do not cover any PECL projects.

A classic PHP project is some kind of cloud. You do not know what inside it and you do not know how to handle it. So it is with PEAR. Some projects at PEAR are small and usable class libraries. For example PHPUnit is a clean library. Although it supports a batch file that is installed it will mainly exist of several class files and packed resource files.

But there are other projects that are made of DATA-Files and of DOC-Files. Maybe there are more types of files.

Another type of project requires additional handling to be “installed” on a local machine. That may be a simple generator for configuration files. But there may be additional handling of – for example – database installation and configuration.

Currently PEAR packages are not that strict as maven projects are. We face many problems related to PEAR:

- Non-case-sensitive. For PEAR a module called “FOO” is the same than “Foo”. That will be a problem with maven because it expects case sensitive groupIds and artifactIds as well as versions.
- Versions do not always follow the maven conventions. However there may be conventions in PEAR. But some strange things happen for PEAR projects:
    - Version “1.0.0RC1” is a release candidate. That is fairly changeable to “1.0.0-RC1”. Similar way for versions “1.0.0a1” (means 1.0.0-alpha-1) and “1.0.0b2” (1.0.0-beta-2). But there was also a version called “0.9.4dev”. There are other strange things happening.
    - Version “1.00” seems to be a valid PEAR version. This will be a valid maven version too although being somehow strange.
    - If PEAR does not know a version but it knows a module it will decide “Oh, yep, I know it, let us install the other version”. It does not mind if it is newer or older than the desired ones. That’s really strange and this problem results in Packages that contain dead dependencies. Maven would never do this. If maven is not able to resolve a dependency version it would never decide to install another version on its own.
    - Most PEAR modules say something like “Dependency to Package A, version 1.2 or above”. A classic maven dependency is built to exactly one version. There is some information around the web that Maven may support version ranges these days. http://docs.codehaus.org/display/MAVEN/Dependency+Mediation+and+Conflict+Resolution. We should test it since this is even mentioned in the maven reference: http://www.sonatype.com/books/mvnref-book/reference/pom-relationships-sect-project-dependencies.html#pom-relationships-sect-version-ranges
    - PEAR seems not to handle development versions. Maven contains some logic to build SNAPSHOT versions. That are versions built from a development snapshot or a version system commit. Classic releases are those described as “alpha, beta, release candidate and final release”. But this may be pretty clear with both, maven and PEAR. The reason is that the term “RC” will always be greater than the term “alpha” or “beta”. PEAR currently does not know about handling development snapshots. See http://docs.codehaus.org/display/MAVEN/Versioning
    - One problem might be the question if version “1.0” is equal to “1.0.0”. However: Maven says: Yes it is. There should someone take a closer look on this topic.
    - Maven does never strictly guarantee that the dependencies match. Many packages are configured once and then never upgraded. However maven won’t allow it since it forces the usage of well known versions. Upgrades to newer versions of a dependency should be made manually. That means: A developer should decide that the module know uses a newer dependency. There are indeed some modules of third party packages (ezComponents for example) that use strict version ranges. So it is mostly a task of the module maintainer to force the usage of a good dependency and a clean version range. That’s ok for us because it is not a fault in PEAR. PEAR itself will support a clean dependency management.
- PEAR itself seems only to have a CLI interface. The XML files contain various information but they are not directly accessible. Meaning that we are not able to call something like “PEAR info –-xml”

Standard-Project

In PHP-Maven 2.0 we focused on standard PHP projects. Our maven plugin does neither know what to do with all the artifacts nor does it prepare anything for a runtime environment.

It is up to the developer to prepare it. So we do not focus on the needs of developers. But it is fair and flexible because at least everything can be done with PHP-Maven. It is only a matter of how to access the sources and how to access the runtime environment.

Persons and their role

Every person that interacts with PHP-Maven has a role and special needs. We do not matter if one person has more than one role. Our focus is to guess which roles may interact with PHP-Maven.

Relevant role summary

The developer

The developer is focus on development. That means he wants to fix a bug, implement a feature or anything else. The most important thing is that he wants to do it within his favorite IDE. The project must be prepared to run within the personal computer and operating system.

Normally a developer checks out the source project from a versioning system, such as SVN or GIT. After checking out a command “mvn compile” must be working so that everything works fine.

The tester

A tester normally does not interact directly with maven. But a tester needs to test on a valid installation. So either the tester behaves like a developer (having his own environment and test it there) or he behaves like a configuration manager (installing a version on a special testing server).

The configuration manager

A configuration manager is responsible for the project package, deployment and server preparation. Packaging an artifact and deploying it on repositories is a task may be performed by either the developer or the configuration manager. However this task is covered by the maven lifecycle.

The most interesting part is the question how to install it on a production or testing server. This requires the configuration manager to know how to configure the project. And it requires the developer to know the whole infrastructure. The developer is focused on running the project locally. There may be conflicts saying that a configuration won’t ever work on a production server.

The release manager

The release manager is responsible to bundle artifacts to packages and to send working packages to production.

The production

Those people are responsible to manage production servers. Maybe some modules do not have any production servers. Libraries and frameworks do normally not have any production server. But at least there are homepages and most of the time the projects are involved in those homepages.

But the most interesting part is the question “How do we configure the project?”

Documentation

Some people write amazing project documentation. Maven itself already covers the technical development configuration. It introduces a way to write amazing development docu. This includes Javadoc/PHPDoc and code coverage reports. However that’s not the only type of documentation. There may be something like user manuals, system manuals and a online help.

Support

There will be some kind of support. However the support won't interact with maven directly. But similar to tester they need servers where the project is installed.

Even the supporters that are handling the failing installation of one customer are using another role (maybe the configuration manager role) to solve the issue.

How does PHP-Maven 2.0 support the roles?

The simple answer: It does not directly support it. Every project, every person is responsible to solve it on their own. In each project the configuration manager needs to reinvent the wheel. He must to prepare a project infrastructure that supports easy setups for developers, testing servers and production, as well as easy setups for customers that want to install the project on their own servers.

Project types and their influence

There are multiple project types around. PHP is a well known language but there is some kind of scripting and hacking around. It is not our task to condemn PHP or any developer. But we must understand it because we want to find a way to handle this situation.

Sometimes it is not clear which kind of project it is. It is not clear how to use a project. But we want to try it.

Classic project types

Legay

Let us think of legacy artifacts for everything we cannot really handle or imagine of. Legacy projects may be handled differently. They may require installations and configurations or they don't. However: legay projects may be of really good quality. But they have their own style.

We only want to focus on something that we cannot influence but that influences PHP-Maven.

If we think of version 2.0 we already have a way how to handle legacy project. We use the standard maven way and as for php we simply extract the legacy project dependencies to a common place (target/php-deps). It is up to the developer to decide how we can handle this one.

Library

A common PHP library provides a set of classes and static resources. That's the original meaning of a library. We want to focus on that one. There should be a well known identification of the class files as well as a way a standard class loader accepts them.

So we want to provide some good practices or more restrictive validations to verify a library.

CLI application

A cli application provides cli scripts. It is most of the time a compilation of several libraries. The cli scripts are used to handle things, for example a phar tooling application.

WEB application

A web application is more difficult. It is handled via web browser. At first it sounds easy but there may be a problem with it. At least we would require an apache. As far as we can say maven did not yet interact directly with apache http servers. This is caused by the fact that maven is a java product. As such it interacts with tomcat and java web applications/ java web servlets. Maybe we need to develop some plugins to instrument an apache with php integration.

Mixtures

We expect mixtures of all types very often. That means a web application typically is shipped with some classes. That means it is bundled with a library in the same artifact. Some web applications are shipped with CLI scripts, for example to perform cron jobs.

Extended project types

While thinking of the project types we found out that there is a problem. Let us think of a web application that uses smarty as a template engine. Smarty can be configured to compile the

templates as php scripts. Those compiled versions of the scripts are “cached” into a directory. But what problem do we face?

The problem of configuration

Let us think of another situation. Developers should be able to use their own configuration. Flow3 uses (per default) a htaccess file alongside the index.php (webroot). Within this htaccess file one sets the type of environment to be used. During development it contains “SetEnv FLOW3_CONTEXT Development”. And on production servers it may contain “SetEnv FLOW3_CONTEXT Production”.

So we do face a configuration problem. We have two examples: The configuration where to find the smarty cache dir and the configuration that directly will influence our built package. Keep in mind: If we would use a static .htaccess file and even if we generate it on fly we would always have the Development configuration within it. Building a phar and deploying it would always contain this development configuration.

The problem of file type information

Looking closer to pear shows us that pear is already dividing the package into several file types. PHP files are the classic scripts and classes a user of the package might depend on. But there is more. Some packages (for example localization packages) contain thousands of so called data files. What should PHP-Maven do with it? Or better question: Why does PEAR behave in this way?

The answer is simple: It is very clever. Data files are not meant to be present on include path. But they are meant to be found by the corresponding module, for example to load them via file_get_contents.

There are multiple aspects filetypes can be used for.

Framework project types

Some frameworks behave simple. For example libraries for Zend applications are simple class libraries. A typical Zend web application is a compilation of php class files and scripts (for example view scripts). They are organized in a well known directory structure.

But there are some frameworks that cannot be used directly. FLOW3 is one of them. It introduces a package manager that is responsible to load a module. And as another feature it introduces aspect oriented programming and object databases.

A similar technique is known by doctrine. The object relational mapper requires additional steps so that SQL DDLs are built from the php files.

Doctrine should be very static. The generated classes are equal as long as the source class is. So they might be created as one step during the build cycle and right before packaging the application.

But FLOW3 and especially the aspect oriented programming is highly dynamic. There might be another module, not yet known by the current project that will influence our module. And the whole class library we are developing might be unusable outside FLOW3.

First solution: Extending dependency management with 2.0.1

A first solution, maybe targeted for Version 2.0.1, will be a better control over the things going on.

As we always said the target/classes folder is a compilation of the src/main/php and the src/main/resources folders. It should contain all files that will be present on the packed phar. All files that will be needed to install the module as a dependency or as a cli/web project on a test or production server.

Maven itself depends on this logic. Many plugins depend on it. So we should never change this behaviour.

But let us have a closer look at the dependencies. As of PHP-Maven 2.0 they are simply extracted to target/php-deps. That's fair and ok. A module depending on other modules should always be able to load the dependent classes and scripts from a configured location. So that's ok.

There may be situations where one wants to get a better control about the dependency extraction process. We will support it the following ways:

- Extract a phar to custom location
- Copy the phar to custom location (do not extract) and (optional) add it to include path
- Extract only parts of the phar to a custom location
- Set it on include path (staying within the local repository)
- Invoke a custom script

Ordering of control information

The following sub chapters are meant to be the ordering of control information. That means that a later chapter overwrites the previous one.

The default behavior

If nothing else is written down in the pom we would guess that the whole phar has to be extracted to target/php-deps. As this path will be set for include path they are automatically present on include path.

Let the module decide what to do with it

Think of PHPUnit. It will contain a script file that can be invoked. As PHP-Maven controls the invocation of unit tests we don't need it. For us it is a huge nice compilation of classes. It is only needed on the include path. So it should even be possible to not extract PHPUnit but to add the phar file directly in the include path.

So PHPUnit may say "Let ***me*** be present on include path" and that is really fair.

Set it in parent.pom

Someone wants to create a parent.pom that declares the way a dependency is used in all child projects. Every time PHP-Maven guesses the way it should behave it has to merge the dependency

configuration from parent poms to the active poms. TODO: Can we use the effective POM here? Do not think so.

Set it in current.pom

The primary information on how to behave will be the current project pom.

Solving the problem with deep dependencies

Let us guess the current pom (merged with parent and with the modules own choices) contains now useful information what to do with dependencies. We now deploy our module.

Another user (we will call him “end-user”) now uses our module but wants to change the behavior of the deep dependencies. Our module says: We only need the dependency being present on include path. But the end-user decides: No: I want to extract it at a custom location.

We will solve this problem to first look at the current pom. If this pom contains information how a dependency should be handled it is used. If it does not contain it the dependencies are asked. Only if none of the dependencies contains any useful information what to do with our dependency we will use the default saying “extract it to target/php-deps”. We do not guarantee any order. So there might be concurrent information received from two depedencies (call them A and B) about dependency C. To solve them the developer should write it in the current pom and not depend on what the other dependencies are saying.

Best practice for module developers

1. We advice doing the following: Write down the information what to do with a package. Extract it? Put it as phar on class path?

2. Guess that other modules, especially libraries, are always found on php include path. Do neither depend on being extracted, nor use relative paths to find them.

3. Special modules containing files for the web root can be extracted to the target/classes path to let the developer test the whole project in a webserver. But they should be excluded from packed phar file.

Best practice for PHP-Maven

1. PHP-Maven should set a warning if it finds concurrent information what to do with a dependency.

2. We must not assume that extracted phars will be put on the include path. We do only put target/classes and target/php-deps on include path. All other paths must be added manually.

Second solution: Performance speed and project management in 2.0.1

Currently the PHP-Maven plugin is not that clever. For example it will every time unpack the dependencies. We will introduce some kind of project management.

The project management is able to detect changes and not doing every single step again and again. That means:

- Detect the PHP version and PHP installation. As changed the project itself will be dirty.
- Detect the dependencies. As changed the dependencies are dirty and this may cause in rebuilding the target/deps folders.
- Detect the pom itself. As changed the project is dirty and must be rebuilt.

We are using this project management in 2.0.1 to:

- Detect if a dependency is successfully installed (no re-install as long as nobody performs a clean)
- Detect if a file was changed (To not do a lint check every time)

Third solution: Project types and file types with 2.1

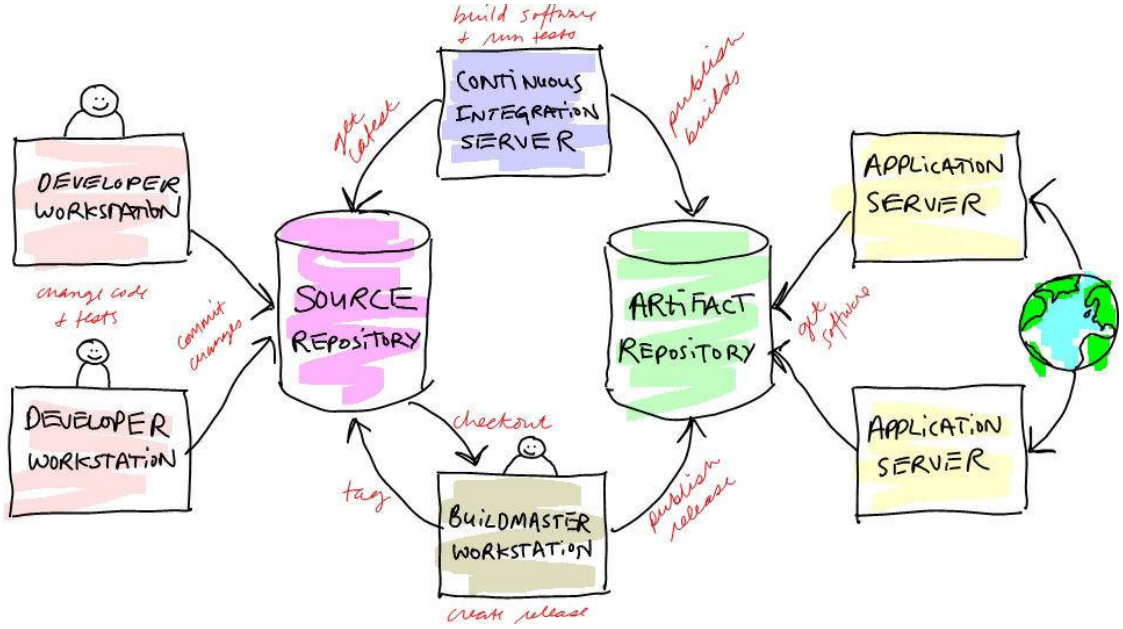
Standard-Package vs. Assemblies

An assembly is always the preferred way to build a working web or cli application that is used outside maven. It will be bundled with dependencies and should be ready-to-use.

The standard package will always be focused on the module itself. It will always be focused on having a module that can be used together with the pom. Packing the dependencies inside the resulting package is not recommended.

The package and the assemblies will only contain the files that are required to use this as a module. Any dynamic file that relies on additional data (for example caches or configuration files) will not be contained in these packages.

The extended lifecycle



(origin: http://johannesbrodwall.com/2007/09/14/lightweight-container-life-cycle/)

As of PHP-Maven 2.0 we are focused on the developer lifecycle. And that's the classical maven lifecycle from documentation. On this picture it is the left part where the two developers are sitting on their work stations.

The need two things:

1. At least access to the source repository
2. Possibility to test the module locally.

Most important thing is thinking of another person that is called the build master or in our role analysis in the previous chapters the configuration manager. He is interacting with the target artifact repository. We are using this term and not the term "buildmaster" because we do want to keep in

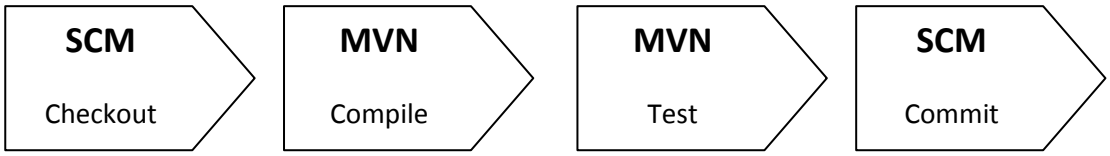
mind the preparation for a build environment and a testable environment. That means: Some person is responsible for setting up configurations and test servers.

There are several application servers. We do not matter if they are called test servers, beta servers or production server. Nor do we mind if they are located within the company or if they are setup by one of the customers. Those questions are only a matter of “Who gets access” and “Who does deployment”.

The CI servers are mainly some kind of automated build servers. We do not yet cover them to be something special. From our point of view they behave like workstations of developers or configuration managers. Except that they are not humans and that they are creating several reports.

The developer role

There is the classical development life cycle introduced by maven:



There are two main maven goals we depend on: The compile goal and the test goal. PHP-Maven must be able to create a working and testable copy of the module by doing the “compile”. It must be possible to start the Unit tests or the web pages for web applications after this goal.

The next goal would be to test the project. Usually “mvn test” starts unit tests. This won’t be changed by PHP-Maven. But we need some additional steps before being able to start the tests. The developer does not want additional steps at this point. Developers are very lazy. Any additional steps they would have to perform will be the worst thing they ever could think of.

What is inside the SCM?

- Typically everything except the target folder.
- It is up to the developer team to decide the scm organization although maven itself contains some conventions.

Phases overview

The compile goal is separated into several phases. Normally the PHP sources need no compilation. We do not provide any native code at the moment. But there may be some plugin that wants to do a compile run. We think of code obfuscators.

More deep inside we will do the following:

1. **validate**: validates the project itself. May validate the configuration.
2. **initialize**: Creates needed directories. May fork a local pear installation.

3. **generate-sources**: A placement to do any source code generation, for example to create php files from UML models.
4. **process-sources**: This feature can be used to wrap properties into php files before they are compiled.
5. **generate-resources**: Similar to the source code generation there may be features for resource generation. We can think of generating DDL files from php classes (see doctrine-project ORM).
6. **compile**: The source code compilation on a classic project means: Copy the php files to the target folder.
7. **process-classes:** The class processing is an option to add obfuscators.
8. **generate-test-sources**: Similar to generate-sources (but for tests).
9. **process-test-sources**: Similar to process-sources (but for tests).
10. **generate-test-resources**: Similar to generate-resources (but for tests).
11. **test-compile**: Similar to compile (but for tests).
12. **process-test-classes:** Similar to process-classes (but for tests).
13. **test**: Performs the unit tests.

The validate phase

Validation will perform basic checks. For example Maven itself verifies the pom file to be meaningful. The dependencies should be resolved and validated.

The initialize phase

The target directories are created. If we are using pear a local pear installation is forked (just an example about the meaning in our environment).

The compile phase

Compiling will do several things. At first it simply copies the php files to the target folder. At second it will do a validation check of the files. There may be several checks, the most important one is a lint-check.

The test phase

This phase will start a validation check by invoking PHPUnit or other test frameworks.

But we will need a closer look in the test phase. What does it mean? The most important question is: How do we configure the module for running the PHPUnit tests?

Every kind of configuration is an application specific process. But PHP-Maven 2.1 will introduce a way to apply configurations to the target folders. That means: During compile phase there will be plugins to create a working configuration.

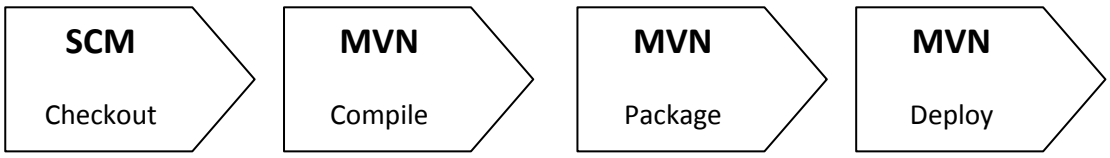
It is important to understand that configuration files may be part of the target folder in order to let the PHPUnit tests and other manual test be running successfully. But they will not become part of the package or part of the assemblies.

The configuration manager role

At first the configuration manager is responsible to build releases. A release is built from all target files. But it may be not the same artifact the developer built. The most important change is the directory layout and the configuration.

We will force anyone to use assemblies to build some kind downloadable standalone release for their homepages. So if anyone feels the need to build a release that is used outside maven they will be forced to use an assembly.

For assemblies it is important that the correct configuration is used. And in order to use the correct configuration we must tell PHP-Maven what to do. We are always keeping in mind that the developer and the configuration manager are the same person and are using the same computer. So there is a conflict: Being the developer will cause a configuration compatible to the local development machine, being the configuration manager will cause to use an anonymous configuration and create some installation script.



As we see there is one additional step: the “mvn package”. This step is only performed by a configuration manager. It builds a snapshot or a release version. It is immediately followed by “mvn install” to copy the phar to the local repository and by “mvn deploy” to copy it to a remote repository.

As of PHP-Maven 2.0 the package goal simply takes the target folder and creates a phar. But as we already explained this may cause problems. It may fetch any local configuration file that may contain database passwords of the local development machine.

Or solution for PHP-Maven 2.1 will cause to divide file types. Configuration files are meant to be not loaded into a packed phar file.

The results are phar files that are

- without any hint of developer local configuration files
- only usable together with maven (see the introduction: standard package vs. assemblies)

- usable for anyone that wants to build a project depending on that phar
- usable for anyone that wants to setup a server by using PHP-Maven

The configuration manager role (non-maven phar)

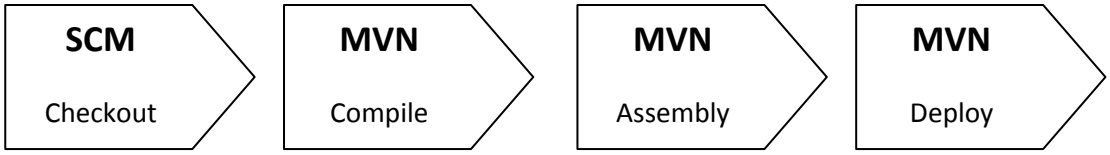
Building a non-maven phar is useful to provide external download links. There is always a difference between two kinds of end users.

The first group is a developer that depends on a project module. They are normally interested in having a build environment and good documentation. They may use PHP-Maven. Maybe the project decides to force everyone depending on their work to use PHP-Maven. But they may not want to use PHP-Maven.

The second group simply uses the resulting cli or web application. They are normally not interested in single libraries. They mostly do never want to use the "overhead" of PHP-Maven as they never develop anything. Maybe they should be forced to use PHP-Maven.

As a result we must support both, installable archives for external websites and installation as well as installations by the help of maven.

Variant A: Building from SCM and deploy



The assemblies are built on top of an existing PHP-Maven build. However this is difficult as we face the same problems for configuration that the packaging faces. We should not embed the developer configuration within the final releases.

The assembly plugin expects to create a new file (separated from original phar that only contains the module itself). This additional file will be added to the build and can be deployed to the repository. As an additional step we will have to influence the creation of assemblies. This is mainly caused by the dependency problem. We expect the phar files to only use the module itself and we expect to get the other dependencies by module and not as part of the assembly.

Variant B: Building from SCM and do not deploy

However the assembly plugin itself is able to skip the deployment. The configuration option "appendAssemblyId" controls this task.

The continuous integration server

There is nothing special on it. Except that it behaves as a developer and configuration manager. It simply tries to build the project and calculates some meta data. As long as we use the standard maven phases (and of course we do) this would never be a problem.

There will be some additional conventions, we will discuss later on.

The application servers

In this chapter we use the term “install” in the meaning to install an application locally. Do not think of the maven install goal.

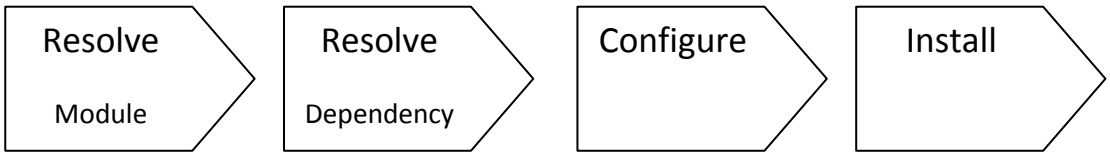
Depending on either a web application or a cli application the results of a project module will be installed on a server. If someone decides to create installable packages via assembly this is not a problem.

A small helper goal will be introduced to download an artifact from maven repository. However this is optional. Either the assembly is downloaded from a web site or it is downloaded from maven repository.

After downloading it the assembly must be installed. Thinking of a non-specified project type (f.e. those coming from PHP-Maven 2.0) we will always assume that the assembly is an executable phar. Executable phars are invoked and it may lead to a valid installation.

Install via module PHAR

PHP-Maven will embed some installing helper goal. This goal will perform the following actions:



PHP-Maven needs to know how the application wants to be configured. After downloading the module and the dependencies it will ask for the missing configuration values. The configuration values might be supplied via maven-settings, command line option or directly. After getting the configuration values the configuration file is generated and the application is “installed”.

What do we mean by installing? The configuration and installation itself is explained by the module.

The configuration

A small example:

- There is a web application “MyWebApp”.
- “MyWebApp” depends on library “FooUserManagement” and on “PhpMyAdmin”.
- “PhpMyAdmin” says: I need a mysql database connection. After setting the configuration I will generate a config.php script.
- “FooUserManagement” does not need any configuration. It is a module and within the index.php boot script a configuration must be set.
- “MyWebApp” says: I need a mysql database connection and a table name prefix. I write this down as a configuration files.

What is happening? Using “PhpMyAdmin” alone and trying to install it will cause PHP-Maven to ask for the mysql Connection. Using “MyWebApp” will cause to ask for the mysql-connection twice. The first time it will ask for the web application and the second time it will ask for the phpMyAdmin.

To solve this the module “MyWebApp” can instrument the phpMyAdmin installation. It will be able to invoke the “phpMyAdmin” configurator with correct values.

The installation

The installation of cli components should be easy. The might need additional settings of environment variables. For example they need a PATH variable to find the installed shell-scripts. Or they may be integrated in crontab.

Installing web applications might be more difficult. First of all we must fetch the location of directories.

- The web application root is visible via URL. It contains bootstrap scripts.
- The library folder is not visible via URL. It contains php class files.
- The work folder contains several files representing data, for example a forum supports uploading. These uploads are not visible by URL (for security reasons). They are not stored in database but located in an additional folder. A small php script is used as a wrapper to download them. This small script does a security check.
- The cache/temp folder contains temporary files where applications tore files that can be deleted and rebuilt. For example the smarty compiled templates.

We will discuss folder and file types later on.

Installing web applications requires web servers. Installing them may result in manipulating the server configurations. It will be possible to give maven control over web servers. Similar to tomcat there could be an option for starting, stopping and configuring web servers.

For a sample via goals tomcat:deploy and tomcat:redeploy see
https://wiki.base22.com/display/btg/How+to+create+a+Maven+web+app+and+deploy+to+Tomcat+-+fast

Conventions in PHP-Maven 2.1

The configuration framework

Definition

Each project that wants to be configured needs a special maven property. Some frameworks call it stage. As stage is already used in the maven world we call it Context.

Per default the maven property “ProjectContext” may contain the following values:

- **Development**: While building the target/classes directory
- **UnitTesting**: While building the target/test-classes directory
- **Package**: While building the phar package or the assembly
- **Testing**: While installing an application on a testing server.
- **Production:** While installing an application on a productional server.

The default is set in the super pom:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.phpmaven</groupId>
      <artifactId>maven-php-config</artifactId>
      <configuration>
        <contextProperty>ProjectContext</contextProperty>
        <configFolder>${project.basedir}/src/main/config/${CONTEXT}
</configFolder>
        <configTargetFolder>${project.basedir}/target/php-config
</configTargetFolder>
        <defaultDevelopmentContext>Development</defaultDevelopmentContext>
        <defaultUnitTestingContext>UnitTesting</defaultUnitTestingContext>
        <defaultPackagingContext>Packaging</defaultPackagingContext>
        <defaultInstallContext>Production</defaultInstallContext>
      </configuration>
    </plugin>
  </plugins>
</build>
```

The configuration plugin will know perform the following steps:

While preparing the target/classes (goal process-sources...) we will add src/main/config/Development to the resources file set. This folder will be processed to build the configuration from. While creating the test-classes we will add the folder

src/main/config/UnitTesting. The files will be copied/ filtered to folder target/php-config. One might influence this behaviour so that the configuration is merged into the resulting target/classes folder. In this situation the PHP-Maven plugin must be able to detect which files are built from standard resources and which files are built from configuration. There may be situations a special file only is valid while using Development or UnitTesting context. The file might not be present on Packaging Context.

How do we solve this one? The configuration plugin is remembering the previous Context. While needing a new Context the configuration plugin forks a new maven execution on the current needs. This execution uses the ProjectContext as we need it.

Usage

One might use the Context to activate or deactivate profiles. Profiles are a maven built-in feature. You can influence the build depending on the current Context as this is a normal maven property. However it is not recommended to do it in the project pom. The only way to influence the build should be declaring properties depending on the context property.

You might add a section in settings.xml containing the mysql-server and mysql-passwort for development. It might contain a different mysql database for unit testing to not influence a local installation the developer uses. However every developer is responsible to provide good configuration files that contain these entries.

While packaging the configuration of all other Contexts are built into the resulting phar. As we install a dependency they are extracted and a filtering is done.

Skip configuration at all

It is possible to deactivate the configuration of a module. This might be the default for some project types. For example libraries should never rely on a specific configuration. Instead the library API should be made of factories or other patterns so that you get an instance of the classes with certain configurations.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.phpmaven</groupId>
      <artifactId>maven-php-config</artifactId>
      <configuration>
        <enable>false</enable>
      </configuration>
    </plugin>
  </plugins>
</build>
```

About the Testing context

Above we did not mention the Testing context. However this is not mentioned because there should be never a difference between “testing” servers and “production” servers. As we said before the only difference is the question of access. Do beta testers use the installation? Is the installation visible to your customers?

The configuration may differ from where to get it from.

Unknown mappings will be complained. They should be added to configuration:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.phpmaven</groupId>
      <artifactId>maven-php-config</artifactId>
      <configuration>
        <contextMappings>
          <contextMapping>
            <sourceMapping>Testing</sourceMapping>
            <targetMapping>Production</targetMapping>
          </contextMapping>
        </contextMappings>
      </configuration>
    </plugin>
  </plugins>
</build>
```

This configuration will cause to let the Testing context behave like the Production context. Target mappings must always be one of the mappings from the defaults.

Context in dependencies

If we are depending on other projects there might be difference. They may be contexts another project does not understand and there might be a problem because another project uses a different context property. The configuration plugin will always map the configuration. It looks at the pom of the target project and it does a mapping.

Let us assume the dependency uses the Property “ZendStage”. The configuration plugin must be clever enough to set this property while installing the dependency.

You might influence this behaviour by setting up a configuration manually.

Ignore the configuration (do not configure)

```
<build>
  <plugins>
    <plugin>
      <groupId>org.phpmaven</groupId>
      <artifactId>maven-php-config</artifactId>
      <configuration>
        <dependencyConfigurations>
          <dependencyConfiguration>
            <groupId>org.somegroup</groupId>
            <artifactId>some-artifact</artifactId>
            <action>ignore</action>
          </dependencyConfiguration>
        </dependencyConfigurations>
      </configuration>
    </plugin>
  </plugins>
</build>
```

This configuration will cause to ignore a configuration of this dependency. There will be no action and therefore the dependency is not configured. However this is not recommended but it may be useful if your project does only require a small part of another module that need not be configured.

Map and influence properties

Sometimes properties need to be mapped. However we will introduce standard properties but they may be differing from other modules. So we need some kind of property mapping between them.

The following example is mapping the properties beginning with “mysqlConfig.” to those beginning with “database.mysqli.config.. This may be caused in the fact a module only is using mysql as database. But a dependency allows us to configure various drivers and one of them might be the mysqli driver.

We will add a property called “database.driver” while installing this dependency.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.phpmaven</groupId>
      <artifactId>maven-php-config</artifactId>
      <configuration>
        <dependencyConfigurations>
          <dependencyConfiguration>
            <groupId>org.somegroup</groupId>
            <artifactId>some-artifact</artifactId>
            <propertyMappings>
              <propertyMapping>
                <fromProperty>mysqlConfig.*</fromProperty>
                <toProperty>database.mysqli.config.*</toProperty>
              </propertyMapping>
            </propertyMappings>
            <properties>
              <database.driver>mysqli</database.driver>
            </properties>
          </dependencyConfiguration>
        </dependencyConfigurations>
      </configuration>
    </plugin>
  </plugins>
</build>
```

The dependency framework (introduced in 2.0.1)

Definition

The installation framework allows us to influence the behaviour of the dependency management. It answers the question what to do with dependencies.

For example the zend framework is only needed to be on include path. So it might be added directly from include path without extracting it to php-deps.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.phpmaven</groupId>
      <artifactId>maven-php-dependency</artifactId>
      <configuration>
        <defaultAction>extractAndConfigure</defaultAction>
      </configuration>
    </plugin>
  </plugins>
</build>
```

This is the default action we will use. It extracts and configures the dependencies. The default action may be overwritten per module. The ordering is as described in the 2.0.1 analysis.

We will introduce the action to configure the plugin in version 2.1. There might be the following actions:

Action extractAndConfigure

This action causes the dependency to be extracted in php-deps or php-test-deps folder. It will be configured using the configuration framework.

Action extract

This action will only extract the dependency and do no configuration at all.

Action includeLibrary

This action will include the phar file into the include path. It may include the phar file directly. Some phar files may contain special class folders because of additional files. So it might result in adding sub folders to the include path.

Action include

This action will always include the phar file directly into the class path

Action invokeScript

This action causes a php script to be invoked. The php script is responsible to configure the dependencies.

The script to be invoked is configured via “dependencyScript” configuration option. It is invoked and receives a file name of a file containing php serialized values. It may read the file by

```
$dependencies = unserialize(file_get_contents($_SERVER['argv'][1]));
```

After reading the dependencies can be configured. The specification is not yet finished at this point.

Action execute

Phar files can be directly executed. The working directory is set to the php-deps or php-test-deps folder.

Per dependency configuration

```
<build>
  <plugins>
    <plugin>
      <groupId>org.phpmaven</groupId>
      <artifactId>maven-php-dependency</artifactId>
      <configuration>
        <dependencyConfigurations>
          <dependencyConfiguration>
            <groupId>org.somegroup</groupId>
            <artifactId>some-artifact</artifactId>
            <action>invokeScript</action>
          </dependencyConfiguration>
        </dependencyConfigurations>
        <dependencyScript>${project.basedir}/src/build/dependency.php
</dependencyScript>
      </configuration>
    </plugin>
  </plugins>
</build>
```

This example will overwrite the default behavior for exactly one dependency.

The file and folder type framework

The behavior of php-maven depends on the knowledge of file and folder types. PHP files may be cli scripts, web scripts as well as classes. We will be able to let PHP-maven configure the file types in the following way:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.phpmaven</groupId>
      <artifactId>maven-php-filetypes</artifactId>
      <configuration>
        <fileTypes>
          <fileType>
            <includes><include>classes/library/**/*.php</include></includes>
            <types>
              <type>class</type>
            </types>
          </fileType>
        </fileTypes>
        <folderTypes>
          <folderType>
            <includes><include>classes/library</include></includes>
            <types>
              <type>includePath</type>
              <type>classes</type>
            </types>
          </folderType>
        </folderTypes>
      </configuration>
    </plugin>
  </plugins>
</build>
```

The default configuration would be to declare all php files to be unspecified php scripts and to let all other files be unknown files. However there may be a project type declaration that sets both, default values and validations on file types or folder types.

The above example would treat all php files inside target/classes/library as php classes.

It will cause the folder target/classes/library be present on the include path.

File type “class”

This file type is used by all files that contain a single class. The name of the file should be the same as the class file. The namespace is setting the folder names. However this may only be validated through additional validations. For example a featured project may add a validator that fails of the naming convention is not matched.

Class files are validated by executing them directly.

File type "cli-script"
The cli-script file type is used by php files that represent a cli script. It is only validated by lint checking because an invocation may lead to problems.

File type "web-script"
The web-script file type is used by php web files. It is only validated by lint checking because an invocation may lead to problems.

File type "script"
The script file type is used by anonymous php files. It is only validated by lint checking because an invocation may lead to problems. Some project types will add validators that complain about unknown php files.

File type "data"
Data files are files of various content. They are known to represent some data the php project is using. Some project types will add validators that deny data files within the classes folders because this may cause security leaks. Maybe xml files can be declared as data files.

Static configuration files will always be of filetype data.

File type "doc"
This file type represents documentation. Documentation files are not meant to be of any use by the library. For example a license file packed with the phar should be a doc file. While unpacking a phar the doc files are silently ingored.

File type "config"
This file is meant to be a dynamic configuration file. This type of files should be filtered by maven before unpacking/installing them.

File type "meta"
Special meta files that are used by the packager (PHP-Maven at this point).

File type "unit-testcase"
Unit test files. This is a parent of all files representing executable unit tests. As a child file type there will be for example "phpunit-testcase" or "jsunit-testcase"

File type "autoloader"
A special php script that installs an auto loader.

Additional file types
Frameworks may introduce additional file types. They always must depend on the existing file type. For example smarty introduces template scripts. A validator may be added to parse every "smarty-template" for validation before packaging them. The file type "smarty-template" must be declared as a child of file type "data" since they are not regular php files.

Folder type "includePath"
This adds the matching folders to the include path. Per default the whole target folder will be added to the include path.

Folder type "webroot"

The web root is the folder the framework is using as the web root. Suring installation or while running a webserver it is used to declare the document root. Normally there are web scripts within the web root folder.

Folder type "classes"

The "classes" folder contains class files.

Folder type "cli-scripts"

The "scripts" folder contains cli scripts that may be invoked.

Folder type "excluded"

This folder contains unspecified files. They are excluded from validation and phar packaging. This may be files used by the pom itself.

Folder type "meta"

Special meta folder that are used by the packager (PHP-Maven at this point).

Folder type "data"

Unspecified data folder that contains various kind of files.

Folder type "config"

This folder Contains configuration files.

Additional folder types

Frameworks may introduce additional folder types. They always must depend on the existing folder type. For example smarty introduces template script folders that are added as inclusion path for validation. A validator for smarty templates may check the existence of other templates that are included. The folder type "smarty-templates" must be declared as a child of file type "data" since they are not regular php files.

The project types and frameworks

Definition

Frameworks are supported by adding a project type to a project. Project types will be inherited from parent pom. The inherit configuration parameter may be used to completely redefine the project type and ignore the parent pom.

By applying a project type a plugin is loaded from phpmaven framework. We need to setup some kind of project registry to make the configuration very easy.

Adding project types

```
<build>
  <plugins>
    <plugin>
      <groupId>org.phpmaven</groupId>
      <artifactId>maven-php-projecttypes</artifactId>
      <configuration>
        <inherit>false</inherit>
        <projectTypes>
          <projectType>zend-framework</projectType>
        </projectTypes>
      </configuration>
    </plugin>
  </plugins>
</build>
```

The above example will declare the project to be a zend framework project. Technical this will force PHP-Maven to load a plugin. This plugin will install file types, folder types, validators and many more things.

Some frameworks may ensure that a dependency is used. The zend-framework project type will ensure that there is any dependency to the framework.

Others may influence the whole behavior of a build goal. The project type “jsunit” will ensure that the jsunit test cases get executed while testing the project.

Too complex?

Well, it is very flexible. Everyone can use PHP-Maven to build and use an unspecified project. The defaults will always behave as of PHP-Maven 2.0. So everyone can create simple projects.

The easiest way to use project types is to use a proper parent. For example we can introduce “org.phpmaven:php-zendfwk-app-parent” for all zend framework web applications.

Everything will be covered inside the parent pom and the plugins.

Getting more control of the behavior can cause the developer to add project types. This is a small example. Project types will load the plugins to handle them. Everything is now covered inside the plugins.

Only if one needs to create their own validators, project types and frameworks he might get deeper inside and use more complex configurations.

There is one benefit from complex configurations: The developer should decide to use easy configurations. As such they are forced to use some standards others may expect when using their library.

PHAR meta information

TODO (contacted PEAR group to get an open specification on this issue)

We should include some meta data into the phars we built. The meta data should contain a path that is meant to be the meta file path. Meta file paths are not meant to be extracted. But they are used to contain useful information for external systems trying to load the phar. And they may be used to create a project from existing phars.

As we are part of maven we will always use the information inside the POM.