Chapter 15. Packages and Extensions

PHP is a high-level language that uses dynamic typing and memory management to make software development easier for end users. Unfortunately, computers are not very good at handling high-level concepts, so any high-level system must itself be built atop lower-level building blocks. In the case of PHP, the entire system is written in and built atop C.

Since PHP is open source, you can download the entire source code for the language directly from GitHub. Then you can build the language from source on your own system, make changes to it, or write your own native (C-level) extensions.

In any environment, you'll need various other packages available in order to build PHP from source. On Ubuntu Linux, these are the packages:

`pkg-config`
: A Linux package for returning information about installed libraries

`build-essential`
: A meta-package encompassing the GNU debugger, g++ compiler, and other tools for working with C/C++ projects

`autoconf`
: Package of macros to produce shell scripts that configure code packages

`bison`
: A general-purpose parser generator

`re2c`
: A regular expression compiler and open source lexer for C and C++

`libxml2-dev`
: The C-level development headers required for XML processing

`libsqlite3-dev`

The C-level development headers for SQLite and related bindings

You can install all of them with the following `apt` command:

```
$ sudo apt install -y pkg-config build-essential autocc
                      libxml2-dev libsqlite3-dev
```

Once dependencies are available, you use the `buildconf` script to generate the configuration script, then `configure` itself will ready the build environment. Several options can be passed directly to `configure` to control how the environment will be set up. Table 15-1 lists some of the most useful.

Table 15-1. PHP `configure` options

Option flag	Description
`--enable-debug`	Compile with debugging symbols. Useful for developing changes to core PHP or writing new extensions.
`--enable-libgcc`	Allow code to explicitly link against `libgcc` .
`--enable-php-streams`	Activate support for experimental PHP streams.
`--enable-phpdbg`	Enable the interactive `phpdbg` debugger.
`--enable-zts`	Enable thread safety.
`--disable-short-tags`	Disable PHP short tag support (e.g., `<?`).

Understanding how to build PHP itself isn't a prerequisite for using it. In most environments, you can install a binary distribution directly from a standard package manager. On Ubuntu, for example, you can install PHP directly with the following:

```
$ sudo apt install -y php
```

Knowing how to build PHP from source, though, is important should you ever wish to change the behavior of the language, include a nonbundled extension, or write your own native module in the future.

Standard Modules

By default, PHP uses its own extension system to power much of the core functionality of the language. In addition to core modules, various extensions are bundled directly with PHP.[1] These include the following:

- BCMath for arbitrary-precision mathematics
- FFI (Foreign Function Interface) for loading shared libraries and calling functions within them
- PDO (PHP Data Objects) for abstracting various database interfaces
- SQLite3 for interacting directly with SQLite databases

Standard modules are bundled with PHP and available for inclusion immediately through changes to your *php.ini* configuration. External extensions, like PDO support for Microsoft SQL Server, are also available but must be installed and activated separately. Tools like PECL, discussed in Recipe 15.4, make the installation of these modules straightforward for any environment.

Libraries/Composer

In addition to native extensions to the language, you can leverage Composer, the most popular dependency manager for PHP. Any PHP project can (and likely *should*) be defined as a Composer module by including a *composer.json* file that describes the project and its structure. Including such a file has two key advantages, even if you don't leverage Composer to pull third-party code into your project:

- You (or another developer) can include your project as a dependency of another project. This makes your code portable and encourages the reuse of function and class definitions.
- Once your project has a *composer.json* file, you can leverage Composer's autoloading features to dynamically include classes and

functions within your project without explicitly using `require()` to load them directly.

The recipes in this chapter explain how to configure your project as a Composer package, as well as how to leverage Composer to find and include third-party libraries. You'll also learn how to find and include native extensions to the language through PHP Extension Community Library (PECL) and PHP Extension and Application Repository (PEAR).

15.1 Defining a Composer Project

Problem

You want to start a new project that uses Composer to dynamically load code and dependencies.

Solution

Use Composer's `init` command at the command line to bootstrap a new project with a *composer.json* file. For example:

```
$ composer init --name ericmann/cookbook --type project
```

After walking through the interactive prompts (requesting a description, author, minimum stability, etc.), you'll be left with a well-defined *composer.json* for your project.

Discussion

Composer works by defining information about your project in a JSON document and using that information to build out additional script loaders and integrations. A newly initialized project won't have much detail in this document at all. The *composer.json* file generated by the `init` command in the Solution example will initially look like the following:

```
{
    "name": "ericmann/cookbook",
```

```
    "type": "project",
    "license": "MIT",
    "require": {}
}
```

This configuration file defines no dependencies, no additional scripts, and no autoloading. In order to be useful for something other than identifying the project and the license, you need to start adding to it. First, you need to define the autoloader to pull in your project code.

For the sake of this project, use the default namespace `Cookbook` and place all of your code in a directory called *src/* within the project. Then, update your *composer.json* to map that namespace to that directory as follows:

```
{
    "name": "ericmann/cookbook",
    "type": "project",
    "license": "MIT",
    "require": {},
    "autoload": {
        "psr-4": {
            "Cookbook\\": "src/"
        }
    }
}
```

Once you've updated your Composer config, run `composer dumpautoload` at the command line to force Composer to reload the configuration and define the automated source mappings. Once that's complete, Composer will have created a new *vendor/* directory in your project. It contains two critical components:

- An *autoload.php* script that you'll need to `require()` when you load your application
- A *composer* directory that contains Composer's code loading routines to dynamically pull in your scripts

To further illustrate how autoloading works, create two new files. First, create a file called *Hello.php* in the *src/* directory containing the `Hello` class defined in Example 15-1.

Example 15-1. Simple class definition for Composer autoloading

```
<?php
namespace Cookbook;

class Hello
{
    public function __construct(private string $greet)

    public function greet(): string
    {
        return "Hello, {$this->greet}!";
    }
}
```

Then, in the root of your project create an *app.php* file with the following contents to bootstrap the execution of the preceding snippet:

```
<?php

require_once 'vendor/autoload.php';

$intro = new Cookbook\Hello('world');

echo $intro . PHP_EOL;
```

Finally, return to the command line. Since you've added a new class to the project, you need to run `composer dumpautoload` once again so Composer is aware of the class. Then, you can run `php app.php` to invoke the application directly and produce the following output:

```
$ php app.php
Hello, world!
$
```

Any class definitions you need for your project or application can be defined the same way. The base `Cookbook` namespace will always be the root of the *src/* directory. If you want to define a nested namespace for objects, say `Cookbook\Recipes`, then create a similarly named directory (e.g.,

Recipes/) within *src/* so Composer knows where to find your class definitions when they're used later within the application.

Similarly, you can leverage Composer's `require` command to import third-party dependencies into your application.[2] These dependencies will be loaded into your application at runtime the same way your custom classes are.

See Also

Composer documentation on the `init` command and on PSR-4 autoloading.

15.2 Finding Composer Packages

Problem

You want to find a library to accomplish a particular task so you don't need to spend time reinventing the wheel by writing your own implementation.

Solution

Use the PHP Package Repository at Packagist to find the appropriate library and use Composer to install it in your application.

Discussion

Many developers find they spend the majority of their time reimplementing logic or systems they've built before. Different applications serve different purposes but often leverage the same basic building blocks and foundations in order to operate.

This is one of the key drivers behind paradigms like object-oriented programming, where you encapsulate the logic in your application within objects that can be individually manipulated, updated, or even reused. Instead of rewriting the same code over and over again, you encapsulate it within an object that can be reused within the application or even transported into your next project.[3]

In PHP, these reusable code components are often redistributed as standalone libraries that can be imported with Composer. Just as Recipe 15.1 demonstrated how to define a Composer project and automatically import your class and function definitions, the same system can be used to add third-party logic to your system as well.[4]

First, identify the need of a particular operation or piece of logic. Assume, for example, your application needs to integrate with a time-based one-time password (TOTP) system like Google Authenticator. You'll need a TOTP library to do so. To find it, navigate in your browser to packagist.org, the PHP Package Repository. The home page will look somewhat like Figure 15-1 and prominently features a search bar prominently in the header.

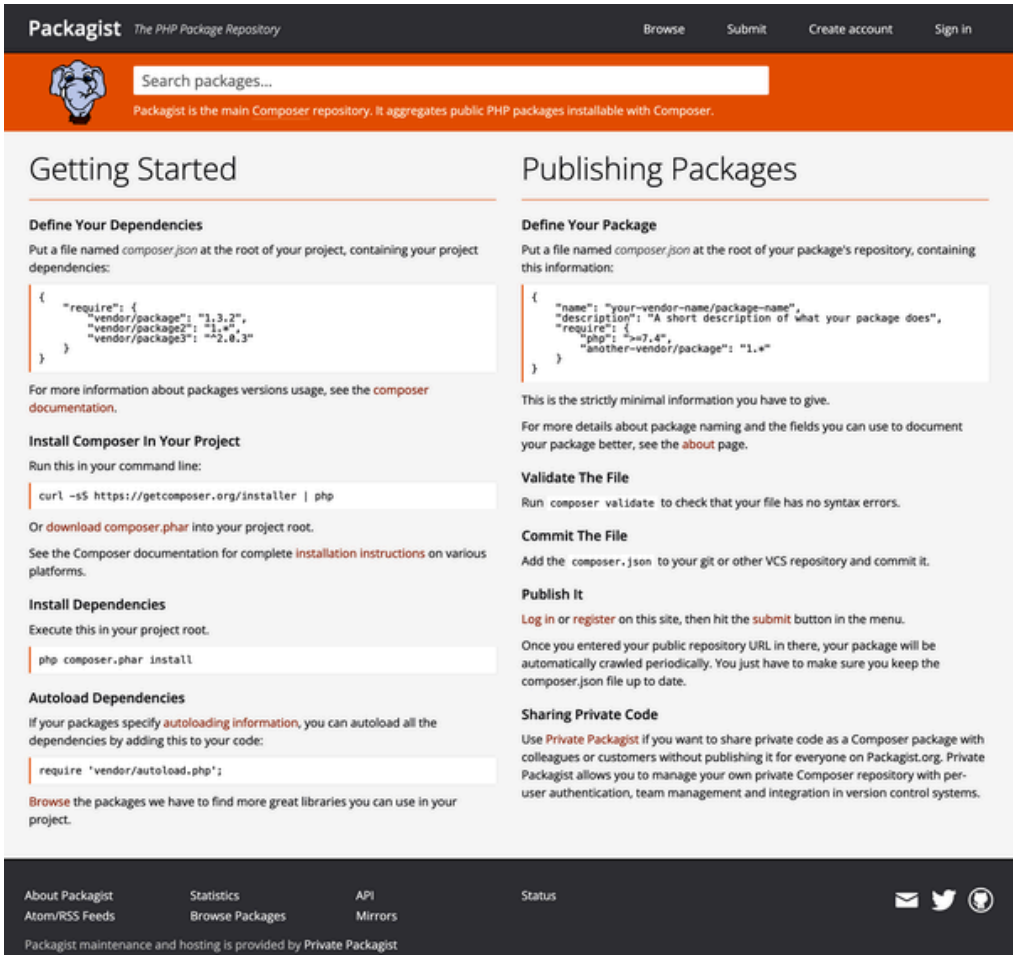

Figure 15-1. Packagist is a free distribution method for PHP packages installable via Composer

Then search for the tool you need—in this case, TOTP. You'll be rewarded with a list of available projects, sorted by popularity. You can further leverage the package type and various tags affixed to each library to pare your search results down to a handful of possible libraries.

NOTE

Popularity on Packagist is defined by both package downloads and GitHub stars. It's a good way to measure how frequently a project is being used in the wild but is by no means the only measure you should leverage. Many developers still copy and paste third-party code into their systems, so there are potentially millions of "downloads" not reflected in Packagist's metrics. Likewise, merely being popular or widely used does not mean a package is secure or the right fit for your project. Take time to carefully review each potential library to ensure that it doesn't introduce unnecessary risk into your application.

Further, if you know of a particular module author whose work you trust, you can search for that directly by adding their username to the search. For example, a search for `Eric Mann totp` will yield a specific TOTP implementation originally created by this book's author.

Once you've identified and carefully audited available packages for extending your application, review Recipe 15.3 for instructions on how to install and manage them.

See Also

Packagist.org: the PHP Package Repository.

15.3 Installing and Updating Composer Packages

Problem

You've discovered a package on Packagist you want to include in your project.

Solution

Install the package via Composer (assume version 1.0) as follows:

```
composer require "vendor/package:1.0"
```

Discussion

Composer works with two files in your local filesystem: *composer.json* and *composer.lock*. The first is the one you define to describe your project, autoloading, and license. As a concrete example, the original *composer.json* file you defined in Recipe 15.1 is as follows:

```
{
    "name": "ericmann/cookbook",
    "type": "project",
    "license": "MIT",
    "require": {},
    "autoload": {
        "psr-4": {
            "Cookbook\\": "src/"
        }
    }
}
```

Once you run the `require` statement from the Solution example, Composer *updates* your *composer.json* file to add the specified vendor dependency. Your file will now appear as follows:

```
{
    "name": "ericmann/cookbook",
    "type": "project",
    "license": "MIT",
    "require": {
        "vendor/package": "1.0"
    },
    "autoload": {
        "psr-4": {
            "Cookbook\\": "src/"
        }
    }
}
```

When you `require` a package, Composer does three things:

1. It checks to make sure the package exists and grabs either the latest version (if no version was specified) or the version you ask for. It then

updates *composer.json* to store the package in the `require` key.

2. By default, Composer then downloads and installs your package in the *vendor/* directory within your project. It also updates the autoloader script, so the package will be available to other code within your project immediately.
3. Composer also maintains a `composer.lock` file within your project that explicitly identifies which versions of which packages you have installed.

In the Solution example, you explicitly specified version 1.0 of a package. If instead you had not specified a version, Composer would fetch the latest version available and use that in the *composer.json* file. If 1.0 is in fact the latest version, Composer would use `^1.0` as the version indicator, which would then install any potential maintenance versions down the road (like a 1.0.1 version). The *composer.lock* file keeps track of the *exact* version installed so even if you were to delete your entire *vendor/* directory, reinstalling packages via `composer install` will still fetch the same versions as before.

Composer will also endeavor to find the best version for your local environment. It does this by comparing the PHP version required for your environment (and used to run the tool) with those versions supported by the requested packages. Composer also attempts to reconcile any dependencies both explicitly declared by your project and implicitly imported through a transitive dependency declared elsewhere. Should the system fail to find a compatible version to include, it will report an error so you can manually reconcile the version numbers listed in your *composer.json* file.

WARNING

Composer follows semantic versioning in its version constraints. A requirement of `^1.0` will only permit maintenance versions (e.g., 1.0.1, 1.0.2) to be installed. A greater-than constraint (e.g., `>=1.0`) will install any stable version at or above version 1.0. Keeping track of how you define your version constraints is critical to prevent accidental import of breaking package changes introduced by major versions. For more background on how to define version constraints, reference the Composer documentation.

Packagist-hosted libraries with public code aren't the only things you can include via Composer. In addition, you can point your system at either public or private projects hosted in version control systems like GitHub.

To add a GitHub repository to your project, first add a `repositories` key to *composer.json* so the system knows where to look. Then update your `require` key to pull in the project you need. Running `composer update` will then pull the package not from Packagist but directly from GitHub and include it in your project just like any other library.

For example, assume you want to use a particular TOTP library but have uncovered a minor bug. First, fork the GitHub repository to your own account. Then, create a branch in GitHub to hold your changes. Finally, update *composer.json* to point at your custom fork and branch, as illustrated in Example 15-2.

Example 15-2. Use Composer to pull projects from GitHub repositories

```
{
    "name": "ericmann/cookbook",
    "type": "project",
    "license": "MIT",
    "repositories": [
        {
            "type": "vcs",
            "url": "\https://github.com/phpcookbookreac
                                ❶

        }
    ],
    "require": {
        "vendor/package": "dev-bugfix"
                                ❷

    },
    "minimum-stability": "dev",
                                ❸

    "autoload": {
        "psr-4": {
            "Cookbook\\": "src/"
        }
    }
}
```

❶ Ensure that the package you want to include is one you have access to. This repository can either be public or private. If it's private, then you'll need to expose a GitHub personal access token as an environment variable so Composer has the appropriate credentials to pull in the code.

❷ Once the repository is defined, add a new branch specification to your `require` block. Since this is not a tagged or released version, prefix your branch name with `dev-` so Composer knows which branch to pull in.

❸ To include development branches in your project, you should call out the minimum stability required by the project as well to avoid any potential issues with the inclusion.

Whether a library enters your project as a public package, repository, or even as a hardcoded ZIP artifact is up to your development team. Regardless, any reusable package can be loaded via Composer with ease and exposed to the rest of your application.

See Also

Documentation on Composer's `require` command.

15.4 Installing Native PHP Extensions

Problem

You want to install a publicly available native extension for PHP, like the APC User Cache (APCu).

Solution

Find the extension in the PECL repository and install it into the system by using PEAR. For example, install the APCu as follows:

```
$ pecl install apcu
```

Discussion

The PHP community uses two pieces of technology to distribute native extensions to the language itself: PEAR and PECL. The primary difference between them is the kind of package they're used for.

PEAR itself can bundle just about anything—the packages it distributes are bundled as gzip-compressed TAR archives that are composed of PHP code. In this way, PEAR is similar to Composer and can be used for managing, installing, and updating additional PHP libraries used within your application.[5] PEAR packages are loaded differently than Composer ones, though, so take care if you choose to mix and match between the two package managers.

PECL is a library of native extensions to PHP written in C, the same base language as PHP itself. PECL uses PEAR to handle installation and management of extensions; the new functionality introduced through an extension is accessed the same ways as functions native to the language itself.

In reality, many PHP packages introduced in modern versions of the language began as PECL extensions that could be optionally installed by developers for testing and initial integrations. The sodium encryption library, for example, began as a PECL extension before being added to the core distribution of PHP as of version 7.2.[6]

Certain databases (for example, MongoDB), distribute their core drivers for PHP as native PECL extensions. Various networking, security, multimedia, and console manipulation libraries are also available. All are written in highly efficient C code and, thanks to PECL and bindings against PHP, behave as if they were a part of the language itself.

Unlike tools like Composer, which deliver userland PHP code, PECL delivers the raw C code directly to your environment. The `install` command will do the following:

1. Download the extension source
2. Compile the source for your system, leveraging the local environment, its configuration, and the system architecture to ensure compatibility
3. Create a compiled *.so* file for the extension within the extension directory defined by your environment

TIP

While some extensions have appeared to be self-enabling, it's highly likely you will need to modify your system's *php.ini* file to explicitly include the extension. It's a good idea to then restart your web server (Apache, NGINX, or similar) to ensure that PHP loads the new extension as expected.

On Linux systems, you might even want to leverage your system package manager to install a precompiled native extension. Installing APCu on an Ubuntu Linux system is usually as simple as this:

```
$ sudo apt install php-apcu
```

Whether you leverage PECL to build an extension directly or utilize a precompiled binary through a package manager, extending PHP is efficient and easy. These extensions expand the functionality of the language and make your final applications significantly more useful.

See Also

Documentation on the PECL repository and PEAR extension packaging system.

[1] A full list of bundled and external extensions can be found in the PHP Manual.

[2] For more on installing third-party libraries with Composer, see Recipe 15.3.

[3] For a deeper discussion of object-oriented programming and code reuse, review Chapter 8.

[4] The actual *installation* of third-party Composer packages will be discussed in Recipe 15.3.

[5] See Recipe 15.3 for more on installing packages via Composer.

[6] The sodium extension is discussed at length in Chapter 9.