Making extensions distribution-ready

In this chapter, you learned how to create various types of Yii extensions. Now we’ll talk about how to

share your results with people and why it’s important.

Getting ready

Let’s form a checklist for a good extension first. A good programming product should follow these points:

* Good coding style
* People should be able to find it
* A consistent, easy to read, and easy to use API
* Good documentation
* Extension should apply to the most common use cases
* Should be maintained
* Well-tested code, ideally with unit tests
* You need to provide support for it

Of course, having all these requires a lot of work, but these are necessary to create a good product.

How to do it...

1. Every modern PHP product must follow the PSR4 standards of autoloading and the PSR1 and  
   PSR2 standards of the coding style from the <http://www.php-fig.org/psr/>guide.
2. Let’s review our list in more detail, starting with the API. The API should be consistent, easy to  
   read, and easy to use. Consistent means that the overall style should not change, so no different  
   variable naming, no inconsistent names such as isFlagi() and isNotFlag2(), and so on.  
   Everything should obey the rules you’ve defined for your code. This allows less checking of  
   documentation and allows you to focus on coding.
3. A code without any documentation is almost useless. An exception is a relatively simple code, but  
   even if it’s only a few lines, it doesn’t feel right if there is not a single word about how to install and  
   use it. What makes good documentation? The purpose of the code and its pros should be as visible  
   as possible and should be written loud and clear.
4. A code is useless if developers don’t know where to put it and what should be in the application  
   configuration. Don’t expect that people know how to do framework-specific things. The installation  
   guide should be verbose. A step-by-step form is preferred by a majority of developers. If the code  
   needs SQL schema to work, provide it.
5. Even if your API methods and properties are named properly, you still need to document them with  
   PHPDoc comments specifying argument types and return types, providing a brief description for  
   each method. Don’t forget protected and private methods and properties since sometimes it’s  
   necessary to read these to understand the details of how code works. Also, consider listing public  
   methods and properties in documentation so it can be used as a reference.
6. Provide use case examples with well-commented code. Try to cover the most common ways of  
   extension usage.
7. In an example, don’t try to solve multiple problems at a time since it can be confusing.
8. It’s important to make your code flexible so it will apply to many use cases. However, since it’s not  
   possible to create code for every possible use case, try to cover the most common ones.
9. It’s important to make people feel comfortable. Providing a good documentation is a first step. The  
   second is providing a proof that your code works as expected and will work with further updates.  
   The best way to do it is a set of unit tests.
10. Extension should be maintained, at least until it’s stable and there are no more feature requests and  
    bug reports. So expect questions and reports, and reserve some time to work on the code further. If  
    you can’t devote more time to maintain extensions, but it’s very innovative and no one did it before,  
    it’s still worth sharing. If the community likes it, someone will definitely offer his or her help.
11. Finally, you need to make extensions available. Create the Composer package from your extension,  
    push it on GitHub or other shared repository storage, and publish it on the <https://packagist.org> site.
12. Each extension should have a version number and a change log. It will allow the community to  
    check if they have the latest version and check what is changed before upgrading. We recommend  
    to follow the Semantic Versioning rules from the <http://semver.org> site.
13. Even if your extension is relatively simple and documentation is good, there could be questions, and  
    for the first time, the only person who can answer them is you. Typically, questions are asked at  
    official forums, so it is better to create a topic where people can discuss your code and provide a  
    link at the extension page.

How it works...

If you want to share an extension with the community and be sure it will be useful and popular, you need  
to do more than just write code. Making extensions distribution-ready is much more work to do. It can be  
even more than creating an extension itself. So, why is it good to share extensions with the community in  
the first place?

Making the code you use in your own projects open source has its pros. You are getting people, a lot more  
people than you can get to test your closed source project. People who are using your extension are  
testing it, giving valuable feedback, and reporting bugs. If your code is popular, there will be passionate  
developers who will try to improve your code, to make it more extensive, more stable, and reusable.  
Moreover, it just feels good because you are doing a good thing.

We have covered the most important things. Still, there are more things to check out. Try existing  
extensions before writing your own. If an extension almost fits, try contacting the extension author and  
contributing ideas you have. Reviewing existing code helps you find out useful tricks, dos, and don’ts. Also,  
check wiki articles and the official forum from time to time; there is a lot of useful information about  
creating extensions and developing using Yii in general.

See also

* For modern information about PHP coding standards, refer to [http://www.php-fig. org/psr/](http://www.php-fig.org/psr/)
* To learn more about semantic versioning, refer to <http://semver.org>