Pkg · The Julia Language

Standard Library / Pkg







Pkg

Pkg is Julia's builtin package manager, and handles operations such as installing, updating and removing packages.



What follows is a very brief introduction to Pkg. For more information on Project.toml files, Manifest.toml files, package version compatibility ([compat]), environments, registries, etc., it is highly recommended to read the full manual, which is available here: https://julialang.github.io /Pkg.jl/v1/.

What follows is a quick overview of Pkg, Julia's package manager. It should help new users become familiar with basic Pkg features.

Pkg comes with a REPL. Enter the Pkg REPL by pressing] from the Julia REPL. To get back to the Julia REPL, press backspace or ^C.



This guide relies on the Pkg REPL to execute Pkg commands. For non-interactive use, we recommend the Pkg API. The Pkg API is fully documented in the API Reference section of the Pkg documentation.

Upon entering the Pkg REPL, you should see a similar prompt:

(v1.1) pkg>

To add a package, use add:

(v1.1) pkg> add Example



3/20/21, 11:23 1 of 5

Some Pkg output has been omitted in order to keep this guide focused. This will help maintain a good pace and not get bogged down in details. If you require more details, refer to subsequent sections of the Pkg manual.

We can also specify multiple packages at once:

```
(v1.1) pkg> add JSON StaticArrays
```

To remove packages, use rm:

```
(v1.1) pkg> rm JSON StaticArrays
```

So far, we have referred only to registered packages. Pkg also supports working with unregistered packages. To add an unregistered package, specify a URL:

```
(v1.1) pkg> add https://github.com/JuliaLang/Example.jl
```

Use rm to remove this package by name:

```
(v1.1) pkg> rm Example
```

Use update to update an installed package:

```
(v1.1) pkg> update Example
```

To update all installed packages, use update without any arguments:

```
(v1.1) pkg> update
```

Up to this point, we have covered basic package management: adding, updating and removing packages. This will be familiar if you have used other package managers. Pkg offers significant advantages over traditional package managers by organizing dependencies into environments.

You may have noticed the (v1.1) in the REPL prompt. This lets us know v1.1 is the active environment. The active environment is the environment that will be modified by Pkg commands such as add, rm and update.

Let's set up a new environment so we may experiment. To set the active environment, use activate:

2 of 5 3/20/21, 11:23

```
(v1.1) pkg> activate tutorial
[ Info: activating new environment at `/tmp/tutorial/Project.toml`.
```

Pkg lets us know we are creating a new environment and that this environment will be stored in the /tmp/tutorial directory.

Pkg has also updated the REPL prompt in order to reflect the new active environment:

```
(tutorial) pkg>
```

We can ask for information about the active environment by using status:

```
(tutorial) pkg> status
   Status `/tmp/tutorial/Project.toml`
   (empty environment)
```

/tmp/tutorial/Project.toml is the location of the active environment's project file. A project file is where Pkg stores metadata for an environment. Notice this new environment is empty. Let us add a package and observe:

```
(tutorial) pkg> add Example
...

(tutorial) pkg> status
    Status `/tmp/tutorial/Project.toml`
    [7876af07] Example v0.5.1
```

We can see tutorial now contains Example as a dependency.

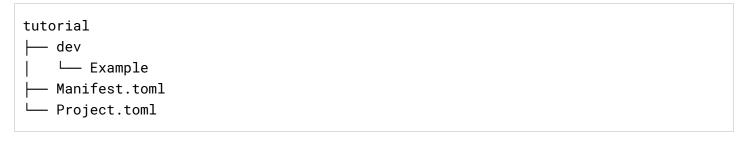
Say we are working on Example and feel it needs new functionality. How can we modify the source code? We can use develop to set up a git clone of the Example package.

```
(tutorial) pkg> develop --local Example
...

(tutorial) pkg> status
    Status `/tmp/tutorial/Project.toml`
    [7876af07] Example v0.5.1+ [`dev/Example`]
```

Notice the feedback has changed. dev/Example refers to the location of the newly created clone. If we look inside the /tmp/tutorial directory, we will notice the following files:

3 of 5



Instead of loading a registered version of Example, Julia will load the source code contained in tutorial/dev/Example.

Let's try it out. First we modify the file at tutorial/dev/Example/src/Example.jl and add a simple function:

```
plusone(x::Int) = x + 1
```

Now we can go back to the Julia REPL and load the package:

```
julia> import Example
```



A package can only be loaded once per Julia session. If you have run import Example in the current Julia session, you will have to restart Julia and rerun activate tutorial in the Pkg REPL. Revise.jl can make this process significantly more pleasant, but setting it up is beyond the scope of this guide.

Julia should load our new code. Let's test it:

```
julia> Example.plusone(1)
2
```

Say we have a change of heart and decide the world is not ready for such elegant code. We can tell Pkg to stop using the local clone and use a registered version instead. We do this with free:

```
(tutorial) pkg> free Example
```

When you are done experimenting with tutorial, you can return to the default environment by running activate with no arguments:

4 of 5 3/20/21, 11:23

```
(tutorial) pkg> activate
(v1.1) pkg>
```

If you are ever stuck, you can ask Pkg for help:

```
(v1.1) pkg> ?
```

You should see a list of available commands along with short descriptions. You can ask for more detailed help by specifying a command:

```
(v1.1) pkg> ?develop
```

This guide should help you get started with Pkg. Pkg has much more to offer in terms of powerful package management, read the full manual to learn more!

« Memory-mapped I/O

Printf »

Powered by Documenter.jl and the Julia Programming Language.

5 of 5 3/20/21, 11:23