Pkg.jl

April 5, 2019

Contents

Part I

1. Introduction

Welcome to the documentation for Pkg, Julia's package manager. The documentation covers many things, for example managing package installations, developing packages, working with package registries and more.

Throughout the manual the REPL interface to Pkg is used in the examples. There is also a functional API interface, which is preferred when not working interactively. This API is documented in the API Reference section.

Chapter 1

Background and Design

Pkg is a complete rewrite of Julia's old package manager[1] and was released together with Julia v1.0. Unlike traditional package managers, which install and manage a single global set of packages, Pkg is designed around "environments": independent sets of packages that can be local to an individual project or shared and selected by name. The exact set of packages and versions in an environment is captured in a *manifest file* which can be checked into a project repository and tracked in version control, significantly improving reproducibility of projects. If you've ever tried to run code you haven't used in a while only to find that you can't get anything to work because you've updated or uninstalled some of the packages your project was using, you'll understand the motivation for this approach. In Pkg, since each project maintains its own independent set of package versions, you'll never have this problem again. Moreover, if you check out a project on a new system, you can simply materialize the environment described by its manifest file and immediately be up and running with a known-good set of dependencies.

Since environments are managed and updated independently from each other, "dependency hell" is significantly alleviated in Pkg. If you want to use the latest and greatest version of some package in a new project but you're stuck on an older version in a different project, that's no problem – since they have separate environments they can just use different versions, which are both installed at the same time in different locations on your system. The location of each package version is canonical, so when environments use the same versions of packages, they can share installations, avoiding unnecessary duplication of the package. Old package versions that are no longer used by any environments are periodically "garbage collected" by the package manager.

Pkg's approach to local environments may be familiar to people who have used Python's `virtualenv` or Ruby's `bundler`. In Julia, instead of hacking the language's code loading mechanisms to support environments, we have the benefit that Julia natively understands them. In addition, Julia environments are "stackable": you can overlay one environment with another and thereby have access to additional packages outside of the primary environment. This makes it easy to work on a project, which provides the primary environment, while still having access to all your usual dev tools like profilers, debuggers, and so on, just by having an environment including these dev tools later in the load path.

Last but not least, Pkg is designed to support federated package registries. This means that it allows multiple registries managed by different parties to interact seamlessly. In particular, this includes private registries which can live behind corporate firewalls. You can install and update your own packages from a private registry with exactly the same tools and workflows that you use to install and manage official Julia packages. If you urgently need to apply a hotfix for a public package that's critical to your company's product, you can tag a private version of it in your company's internal registry and get a fix to your developers and ops teams quickly and easily without having to wait for an upstream patch to be accepted and published. Once an official fix is published, however, you can just upgrade your dependencies and you'll be back on an official release again.

---

[1] Often denoted Pkg2, now archived as `OldPkg` at `github.com/JuliaAttic/OldPkg.jl`.

Part II

2. Getting Started

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

Chapter 2

Basic Usage

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.

> **Note**
>
> 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
```

> **Note**
>
> 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
```

Chapter 3

Getting Started with Environments

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.

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

```
(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.

Chapter 4

Modifying 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:

```
tutorial
 dev
    Example
 Manifest.toml
 Project.toml
```

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

Lets 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
```

> **Warn**
>
> 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. Lets 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:

```
(tutorial) pkg> activate

(v1.1) pkg>
```

Chapter 5

Asking for Help

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!

Part III

3. Managing Packages

Chapter 6

Adding packages

There are two ways of adding packages, either using the add command or the dev command. The most frequently used one is add and its usage is described first.

6.1   Adding registered packages

In the Pkg REPL packages can be added with the add command followed by the name of the package, for example:

```
(v1.0) pkg> add Example
   Cloning default registries into /Users/kristoffer/.julia/registries
   Cloning registry General from "https://github.com/JuliaRegistries/General.git"
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `~/.julia/environments/v1.0/Project.toml`
  [7876af07] + Example v0.5.1
  Updating `~/.julia/environments/v1.0/Manifest.toml`
  [7876af07] + Example v0.5.1
  [8dfed614] + Test
```

Here we added the package Example to the current project. In this example, we are using a fresh Julia installation, and this is our first time adding a package using Pkg. By default, Pkg clones Julia's General registry, and uses this registry to look up packages requested for inclusion in the current environment. The status update shows a short form of the package UUID to the left, then the package name, and the version. Since standard libraries (e.g. Test) are shipped with Julia, they do not have a version. The project status contains the packages you have added yourself, in this case, Example:

```
(v1.0) pkg> st
    Status `Project.toml`
  [7876af07] Example v0.5.1
```

The manifest status, in addition, includes the dependencies of explicitly added packages.

```
(v1.0) pkg> st --manifest
    Status `Manifest.toml`
  [7876af07] Example v0.5.1
  [8dfed614] Test
```

It is possible to add multiple packages in one command as pkg> add A B C.

After a package is added to the project, it can be loaded in Julia:

```
julia> using Example

julia> Example.hello("User")
"Hello, User"
```

A specific version can be installed by appending a version after a @ symbol, e.g. @v0.4, to the package name:

```
(v1.0) pkg> add Example@0.4
 Resolving package versions...
  Updating `~/.julia/environments/v1.0/Project.toml`
  [7876af07] + Example v0.4.1
  Updating `~/.julia/environments/v1.0/Manifest.toml`
  [7876af07] + Example v0.4.1
```

If the master branch (or a certain commit SHA) of Example has a hotfix that has not yet included in a registered version, we can explicitly track a branch (or commit) by appending #branch (or #commit) to the package name:

```
(v1.0) pkg> add Example#master
  Updating git-repo `https://github.com/JuliaLang/Example.jl.git`
 Resolving package versions...
  Updating `~/.julia/environments/v1.0/Project.toml`
  [7876af07] ~ Example v0.5.1  v0.5.1+ #master (https://github.com/JuliaLang/Example.jl.git)
  Updating `~/.julia/environments/v1.0/Manifest.toml`
  [7876af07] ~ Example v0.5.1  v0.5.1+ #master (https://github.com/JuliaLang/Example.jl.git)
```

The status output now shows that we are tracking the master branch of Example. When updating packages, we will pull updates from that branch.

To go back to tracking the registry version of Example, the command free is used:

```
(v1.0) pkg> free Example
 Resolving package versions...
  Updating `~/.julia/environments/v1.0/Project.toml`
  [7876af07] ~ Example v0.5.1+ #master (https://github.com/JuliaLang/Example.jl.git)  v0.5.1
  Updating `~/.julia/environments/v1.0/Manifest.toml`
  [7876af07] ~ Example v0.5.1+ #master )https://github.com/JuliaLang/Example.jl.git)  v0.5.1
```

6.2   Adding unregistered packages

If a package is not in a registry, it can still be added by instead of the package name giving the URL to the repository to add.

```
(v1.0) pkg> add https://github.com/fredrikekre/ImportMacros.jl
  Updating git-repo `https://github.com/fredrikekre/ImportMacros.jl`
 Resolving package versions...
Downloaded MacroTools  v0.4.1
  Updating `~/.julia/environments/v1.0/Project.toml`
  [e6797606] + ImportMacros v0.0.0 # (https://github.com/fredrikekre/ImportMacros.jl)
  Updating `~/.julia/environments/v1.0/Manifest.toml`
  [e6797606] + ImportMacros v0.0.0 # (https://github.com/fredrikekre/ImportMacros.jl)
  [1914dd2f] + MacroTools v0.4.1
```

The dependencies of the unregistered package (here MacroTools) got installed. For unregistered packages we could have given a branch (or commit SHA) to track using #, just like for registered packages.

6.3   Adding a local package

Instead of giving a URL of a git repo to add we could instead have given a local path to a git repo. This works similarly to adding a URL. The local repository will be tracked (at some branch) and updates from that local repo are pulled when packages are updated. Note that changes to files in the local package repository will not immediately be reflected when loading that package. The changes would have to be committed and the packages updated in order to pull in the changes.

6.4   Developing packages

By only using add your Manifest will always have a "reproducible state", in other words, as long as the repositories and registries used are still accessible it is possible to retrieve the exact state of all the dependencies in the project. This has the advantage that you can send your project (`Project.toml` and `Manifest.toml`) to someone else and they can "instantiate" that project in the same state as you had it locally. However, when you are developing a package, it is more convenient to load packages at their current state at some path. For this reason, the dev command exists.

Let's try to dev a registered package:

```
(v1.0) pkg> dev Example
  Updating git-repo `https://github.com/JuliaLang/Example.jl.git`
 Resolving package versions...
  Updating `~/.julia/environments/v1.0/Project.toml`
  [7876af07] + Example v0.5.1+ [`~/.julia/dev/Example`]
  Updating `~/.julia/environments/v1.0/Manifest.toml`
  [7876af07] + Example v0.5.1+ [`~/.julia/dev/Example`]
```

The dev command fetches a full clone of the package to ~/.julia/dev/ (the path can be changed by setting the environment variable JULIA_PKG_DEVDIR). When importing Example julia will now import it from ~/.julia/dev/Example and whatever local changes have been made to the files in that path are consequently reflected in the code loaded. When we used add we said that we tracked the package repository, we here say that we track the path itself. Note that the package manager will never touch any of the files at a tracked path. It is therefore up to you to pull updates, change branches etc. If we try to dev a package at some branch that already exists at ~/.julia/dev/ the package manager we will simply use the existing path. For example:

```
(v1.0) pkg> dev Example
  Updating git-repo `https://github.com/JuliaLang/Example.jl.git`
[ Info: Path `/Users/kristoffer/.julia/dev/Example` exists and looks like the correct package,
    using existing path instead of cloning
```

Note the info message saying that it is using the existing path. As a general rule, the package manager will never touch files that are tracking a path.

If dev is used on a local path, that path to that package is recorded and used when loading that package. The path will be recorded relative to the project file, unless it is given as an absolute path.

To stop tracking a path and use the registered version again, use free

```
(v1.0) pkg> free Example
 Resolving package versions...
  Updating `~/.julia/environments/v1.0/Project.toml`
  [7876af07]  Example v0.5.1+ [`~/.julia/dev/Example`]  v0.5.1
  Updating `~/.julia/environments/v1.0/Manifest.toml`
  [7876af07]  Example v0.5.1+ [`~/.julia/dev/Example`]  v0.5.1
```

It should be pointed out that by using dev your project is now inherently stateful. Its state depends on the current content of the files at the path and the manifest cannot be "instantiated" by someone else without knowing the exact content of all the packages that are tracking a path.

Note that if you add a dependency to a package that tracks a local path, the Manifest (which contains the whole dependency graph) will become out of sync with the actual dependency graph. This means that the package will not be able to load that dependency since it is not recorded in the Manifest. To update sync the Manifest, use the REPL command `resolve`.

Chapter 7

Removing packages

Packages can be removed from the current project by using `pkg> rm Package`. This will only remove packages that exist in the project, to remove a package that only exists as a dependency use `pkg> rm --manifest DepPackage`. Note that this will remove all packages that depends on `DepPackage`.

Chapter 8

Updating packages

When new versions of packages the project is using are released, it is a good idea to update. Simply calling up will try to update *all* the dependencies of the project to the latest compatible version. Sometimes this is not what you want. You can specify a subset of the dependencies to upgrade by giving them as arguments to up, e.g:

```
(v1.0) pkg> up Example
```

If `Example` has a dependency which is also a dependency for another explicitly added package, that dependency will not be updated. If you only want to update the minor version of packages, to reduce the risk that your project breaks, you can give the `--minor` flag, e.g:

```
(v1.0) pkg> up --minor Example
```

Packages that track a repository are not updated when a minor upgrade is done. Packages that track a path are never touched by the package manager.

Chapter 9

Pinning a package

A pinned package will never be updated. A package can be pinned using `pin` as for example

```
(v1.0) pkg> pin Example
 Resolving package versions...
  Updating `~/.julia/environments/v1.0/Project.toml`
  [7876af07] ~ Example v0.5.1  v0.5.1
  Updating `~/.julia/environments/v1.0/Manifest.toml`
  [7876af07] ~ Example v0.5.1  v0.5.1
```

Note the pin symbol showing that the package is pinned. Removing the pin is done using `free`

```
(v1.0) pkg> free Example
  Updating `~/.julia/environments/v1.0/Project.toml`
  [7876af07] ~ Example v0.5.1   v0.5.1
  Updating `~/.julia/environments/v1.0/Manifest.toml`
  [7876af07] ~ Example v0.5.1   v0.5.1
```

Chapter 10

Testing packages

The tests for a package can be run using `test`command:

```
(v1.0) pkg> test Example
   Testing Example
   Testing Example tests passed
```

Chapter 11

Building packages

The build step of a package is automatically run when a package is first installed. The output of the build process is directed to a file. To explicitly run the build step for a package the `build` command is used:

```
(v1.0) pkg> build MbedTLS
  Building MbedTLS  `~/.julia/packages/MbedTLS/h1Vu/deps/build.log`

shell> cat ~/.julia/packages/MbedTLS/h1Vu/deps/build.log
 Warning: `wait(t::Task)` is deprecated, use `fetch(t)` instead.
   caller = macro expansion at OutputCollector.jl:63 [inlined]
 @ Core OutputCollector.jl:63
...
[ Info: using prebuilt binaries
```

Chapter 12

Garbage collecting old, unused packages

As packages are updated and projects are deleted, installed packages that were once used will inevitably become old and not used from any existing project. Pkg keeps a log of all projects used so it can go through the log and see exactly which projects still exist and what packages those projects used. The rest can be deleted. This is done with the gc command:

```
(v1.0) pkg> gc
    Active manifests at:
        `/Users/kristoffer/BinaryProvider/Manifest.toml`
        ...
        `/Users/kristoffer/Compat.jl/Manifest.toml`
  Deleted /Users/kristoffer/.julia/packages/BenchmarkTools/1cAj: 146.302 KiB
  Deleted /Users/kristoffer/.julia/packages/Cassette/BXVB: 795.557 KiB
  ...
  Deleted /Users/kristoffer/.julia/packages/WeakRefStrings/YrK6: 27.328 KiB
  Deleted 36 package installations: 113.205 MiB
```

Note that only packages in ~/.julia/packages are deleted.

Chapter 13

Preview mode

If you just want to see the effects of running a command, but not change your state you can `preview` a command. For example:

```
(HelloWorld) pkg> preview add Plots
```

or

```
(HelloWorld) pkg> preview up
```

will show you the effects of adding `Plots`, or doing a full upgrade, respectively, would have on your project. However, nothing would be installed and your `Project.toml` and `Manifest.toml` are untouched.

Part IV

4. Working with Environments

Chapter 14

Creating your own projects

So far we have added packages to the default project at ~/.julia/environments/v1.0, it is, however, easy to create other, independent, projects. It should be pointed out if two projects uses the same package at the same version, the content of this package is not duplicated. In order to create a new project, create a directory for it and then activate that directory to make it the "active project" which package operations manipulate:

```
shell> mkdir MyProject

shell> cd MyProject
/Users/kristoffer/MyProject

(v1.0) pkg> activate .

(MyProject) pkg> st
    Status `Project.toml`
```

Note that the REPL prompt changed when the new project is activated. Since this is a newly created project, the status command show it contains no packages, and in fact, it has no project or manifest file until we add a package to it:

```
shell> ls -l
total 0

(MyProject) pkg> add Example
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `Project.toml`
  [7876af07] + Example v0.5.1
  Updating `Manifest.toml`
  [7876af07] + Example v0.5.1
  [8dfed614] + Test

shell> ls -l
total 8
-rw-r--r-- 1 stefan staff 207 Jul  3 16:35 Manifest.toml
-rw-r--r-- 1 stefan staff  56 Jul  3 16:35 Project.toml

shell> cat Project.toml
[deps]
Example = "7876af07-990d-54b4-ab0e-23690620f79a"

shell> cat Manifest.toml
```

```
[[Example]]
deps = ["Test"]
git-tree-sha1 = "8eb7b4d4ca487caade9ba3e85932e28ce6d6e1f8"
uuid = "7876af07-990d-54b4-ab0e-23690620f79a"
version = "0.5.1"

[[Test]]
uuid = "8dfed614-e22c-5e08-85e1-65c5234f0b40"
```

This new environment is completely separate from the one we used earlier.

Chapter 15

Precompiling a project

The REPL command `precompile` can be used to precompile all the dependencies in the project. You can for example do

```
(HelloWorld) pkg> update; precompile
```

to update the dependencies and then precompile them.

Chapter 16

Using someone else's project

Simply clone their project using e.g. `git clone`, `cd` to the project directory and call

```
(v1.0) pkg> activate .

(SomeProject) pkg> instantiate
```

If the project contains a manifest, this will install the packages in the same state that is given by that manifest. Otherwise, it will resolve the latest versions of the dependencies compatible with the project.

Part V

5. Creating Packages

A package is a project with a name, uuid and `version` entry in the `Project.toml` file, and a `src/PackageName.jl` file that defines the module `PackageName`. This file is executed when the package is loaded.

16.1 Generating files for a package

To generate files for a new package, use `pkg> generate`.

```
(v1.0) pkg> generate HelloWorld
```

This creates a new project `HelloWorld` with the following files (visualized with the external tree command):

```
shell> cd HelloWorld

shell> tree .
.
 Project.toml
 src
     HelloWorld.jl

1 directory, 2 files
```

The `Project.toml` file contains the name of the package, its unique UUID, its version, the author and eventual dependencies:

```
name = "HelloWorld"
uuid = "b4cd1eb8-1e24-11e8-3319-93036a3eb9f3"
version = "0.1.0"
author = ["Some One <someone@email.com>"]

[deps]
```

The content of `src/HelloWorld.jl` is:

```
module HelloWorld

greet() = print("Hello World!")

end # module
```

We can now activate the project and load the package:

```
pkg> activate .

julia> import HelloWorld

julia> HelloWorld.greet()
Hello World!
```

16.2 Adding dependencies to the project

Let's say we want to use the standard library package Random and the registered package JSON in our project. We simply add these packages (note how the prompt now shows the name of the newly generated project, since we are inside the `HelloWorld` project directory):

```
(HelloWorld) pkg> add Random JSON
 Resolving package versions...
```

```
 Updating "~/Documents/HelloWorld/Project.toml"
[682c06a0] + JSON v0.17.1
[9a3f8284] + Random
 Updating "~/Documents/HelloWorld/Manifest.toml"
[34da2185] + Compat v0.57.0
[682c06a0] + JSON v0.17.1
[4d1e1d77] + Nullables v0.0.4
...
```

Both Random and JSON got added to the project's `Project.toml` file, and the resulting dependencies got added to the `Manifest.toml` file. The resolver has installed each package with the highest possible version, while still respecting the compatibility that each package enforce on its dependencies.

We can now use both Random and JSON in our project. Changing `src/HelloWorld.jl` to

```
module HelloWorld

import Random
import JSON

greet() = print("Hello World!")
greet_alien() = print("Hello ", Random.randstring(8))

end # module
```

and reloading the package, the new `greet_alien` function that uses Random can be used:

```
julia> HelloWorld.greet_alien()
Hello aT157rHV
```

16.3   Adding a build step to the package.

The build step is executed the first time a package is installed or when explicitly invoked with `build`. A package is built by executing the file `deps/build.jl`.

```
shell> cat deps/build.jl
println("I am being built...")

(HelloWorld) pkg> build
  Building HelloWorld  `deps/build.log`
 Resolving package versions...

shell> cat deps/build.log
I am being built...
```

If the build step fails, the output of the build step is printed to the console

```
shell> cat deps/build.jl
error("Ooops")

(HelloWorld) pkg> build
  Building HelloWorld  `deps/build.log`
 Resolving package versions...
 Error: Error building `HelloWorld`:
 ERROR: LoadError: Ooops
 Stacktrace:
  [1] error(::String) at ./error.jl:33
```

```
  [2] top-level scope at none:0
  [3] include at ./boot.jl:317 [inlined]
  [4] include_relative(::Module, ::String) at ./loading.jl:1071
  [5] include(::Module, ::String) at ./sysimg.jl:29
  [6] include(::String) at ./client.jl:393
  [7] top-level scope at none:0
 in expression starting at /Users/kristoffer/.julia/dev/Pkg/HelloWorld/deps/build.jl:1
@ Pkg.Operations Operations.jl:938
```

16.4   Adding tests to the package

When a package is tested the file `test/runtests.jl` is executed.

```
shell> cat test/runtests.jl
println("Testing...")
(HelloWorld) pkg> test
   Testing HelloWorld
 Resolving package versions...
Testing...
   Testing HelloWorld tests passed
```

Tests are run in a new Julia process, where the package itself, and any test-specific dependencies available, see below.

Test-specific dependencies in Julia 1.2 and above

> **Julia 1.2**
>
> This section only applies to Julia 1.2 and above. For specifying test dependencies on previous Julia versions, see Test-specific dependencies in Julia 1.0 and 1.1.

In Julia 1.2 and later the test environment is given by `test/Project.toml`. Thus, when running tests this will be the active project, and only dependencies to the `test/Project.toml` project can be used. Note that Pkg will add the tested package itself implictly.

> **Note**
>
> If no `test/Project.toml` exists Pkg will use the old style test-setup, as described in Test-specific dependencies in Julia 1.0 and 1.1.

To add a test-specific dependency, i.e. a dependency that is available only when testing, it is thus enough to add this dependency to the `test/Project.toml` project. This can be done from the Pkg REPL by activating this environment, and then use add as one normally does. Lets add the `Test` standard library as a test dependency:

```
(HelloWorld) pkg> activate ./test
[ Info: activating environment at `~/HelloWorld/test/Project.toml`.

(test) pkg> add Test
 Resolving package versions...
  Updating `~/HelloWorld/test/Project.toml`
  [8dfed614] + Test
  Updating `~/HelloWorld/test/Manifest.toml`
  [...]
```

We can now use `Test` in the test script and we can see that it gets installed when testing:

```
shell> cat test/runtests.jl
using Test
@test 1 == 1

(HelloWorld) pkg> test
   Testing HelloWorld
 Resolving package versions...
  Updating `/var/folders/64/76tk_g152sg6c6t0b4nkn1vw0000gn/T/tmpPzUPPw/Project.toml`
  [d8327f2a] + HelloWorld v0.1.0 [`~/.julia/dev/Pkg/HelloWorld`]
  [8dfed614] + Test
  Updating `/var/folders/64/76tk_g152sg6c6t0b4nkn1vw0000gn/T/tmpPzUPPw/Manifest.toml`
  [d8327f2a] + HelloWorld v0.1.0 [`~/.julia/dev/Pkg/HelloWorld`]
   Testing HelloWorld tests passed```
```

Test-specific dependencies in Julia 1.0 and 1.1

In Julia 1.0 and Julia 1.1 test-specific dependencies are added to the main `Project.toml`. To add `Markdown` and `Test` as test-dependencies, add the following:

```
[extras]
Markdown = "d6f4376e-aef5-505a-96c1-9c027394607a"
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"

[targets]
test = ["Markdown", "Test"]
```

16.5   Package naming guidelines

Package names should be sensible to most Julia users, *even to those who are not domain experts*. The following guidelines applies to the `General` registry, but may be useful for other package registries as well.

Since the `General` registry belongs to the entire community people may have opinions about your package name when you publish it, especially if it's ambiguous or can be confused with something other than what it is. Usually you will then get suggestions for a new name that may fit your package better.

1. Avoid jargon. In particular, avoid acronyms unless there is minimal possibility of confusion.

    – It's ok to say USA if you're talking about the USA.

    – It's not ok to say PMA, even if you're talking about positive mental attitude.

2. Avoid using `Julia` in your package name.

    – It is usually clear from context and to your users that the package is a Julia package.

    – Having Julia in the name can imply that the package is connected to, or endorsed by, contributors to the Julia language itself.

3. Packages that provide most of their functionality in association with a new type should have pluralized names.

    – `DataFrames` provides the `DataFrame` type.

    – `BloomFilters` provides the `BloomFilter` type.

    – In contrast, `JuliaParser` provides no new type, but instead new functionality in the `JuliaParser.parse()` function.

4. Err on the side of clarity, even if clarity seems long-winded to you.

   – `RandomMatrices` is a less ambiguous name than `RndMat` or RMT, even though the latter are shorter.

5. A less systematic name may suit a package that implements one of several possible approaches to its domain.

   – Julia does not have a single comprehensive plotting package. Instead, `Gadfly`, `PyPlot`, `Winston` and other packages each implement a unique approach based on a particular design philosophy.

   – In contrast, `SortingAlgorithms` provides a consistent interface to use many well-established sorting algorithms.

6. Packages that wrap external libraries or programs should be named after those libraries or programs.

   – `CPLEX.jl` wraps the `CPLEX` library, which can be identified easily in a web search.

   – `MATLAB.jl` provides an interface to call the MATLAB engine from within Julia.

Part VI

6. Compatibility

Compatibility refers to the ability to restrict what version of the dependencies that your project is compatible with. If the compatibility for a dependency is not given, the project is assumed to be compatible with all versions of that dependency.

Compatibility for a dependency is entered in the `Project.toml` file as for example:

```
[compat]
julia = "1.0"
Example = "0.4.3"
```

After a compatibility entry is put into the project file, up can be used to apply it.

The format of the version specifier is described in detail below.

> **Info**
>
> There is currently no way to give compatibility from the Pkg REPL mode so for now, one has to manually edit the project file.

Chapter 17

Version specifier format

Similar to other package managers, the Julia package manager respects semantic versioning (semver). As an example, a version specifier is given as e.g. 1.2.3 is therefore assumed to be compatible with the versions [1.2.3 - 2.0.0) where) is a non-inclusive upper bound. More specifically, a version specifier is either given as a **caret specifier**, e.g. ^1.2.3 or a **tilde specifier** ~1.2.3. Caret specifiers are the default and hence 1.2.3 == ^1.2.3. The difference between a caret and tilde is described in the next section. The union of multiple version specifiers can be formed by comma separating individual version specifiers, e.g.

```
[compat]
Example = "1.2, 2"
```

will result in [1.2.0, 3.0.0).

17.1   Caret specifiers

A caret specifier allows upgrade that would be compatible according to semver. An updated dependency is considered compatible if the new version does not modify the left-most non zero digit in the version specifier.

Some examples are shown below.

```
[compat]
PkgA = "^1.2.3" # [1.2.3, 2.0.0)
PkgB = "^1.2"   # [1.2.0, 2.0.0)
PkgC = "^1"     # [1.0.0, 2.0.0)
PkgD = "^0.2.3" # [0.2.3, 0.3.0)
PkgE = "^0.0.3" # [0.0.3, 0.0.4)
PkgF = "^0.0"   # [0.0.0, 0.1.0)
PkgG = "^0"     # [0.0.0, 1.0.0)
```

While the semver specification says that all versions with a major version of 0 are incompatible with each other, we have made that choice that a version given as 0.a.b is considered compatible with 0.a.c if a != 0 and c >= b.

17.2   Tilde specifiers

A tilde specifier provides more limited upgrade possibilities. When specifying major, minor and patch version, or when specifying major and minor version, only patch version is allowed to change. If you only specify a major version, then both minor and patch versions are allowed to be upgraded (~1 is thus equivalent to ^1). This gives the following example.

```
[compat]
PkgA = "~1.2.3" # [1.2.3, 1.3.0)
```

```
PkgB = "~1.2"   # [1.2.0, 1.3.0)
PkgC = "~1"     # [1.0.0, 2.0.0)
```

17.3 Inequality specifiers

Inequalities can also be used to specify version ranges:

```
[compat]
PkgA = ">= 1.2.3" # [1.2.3,  ∞)
PkgB = ≥" 1.2.3"  # [1.2.3,  ∞)
PkgC = "= 1.2.3"  # [1.2.3, 1.2.3]
PkgD = "< 1.2.3"  # [0.0.0, 1.2.2]
```

Part VII

7. Registries

Registries contain information about packages, such as available releases, dependencies and where it can be downloaded. The `General` registry (https://github.com/JuliaRegistries/General) is the default one, and is installed automatically.

Chapter 18

Managing registries

Julia 1.1

Pkg's registry handling requires at least Julia 1.1.

Registries can be added, removed and updated from either the Pkg REPL or by using the function based API. In this section we will describe the REPL interface. The registry API is documented in the Registry API Reference section.

18.1   Adding registries

A custom registry can be added with the `registry add` command from the Pkg REPL. Usually this will be done with a URL to the registry. Here we add the `General` registry

```
pkg> registry add https://github.com/JuliaRegistries/General
   Cloning registry from "https://github.com/JuliaRegistries/General"
     Added registry `General` to `~/.julia/registries/General`
```

and now all the packages registered in `General` are available for e.g. adding. To see which registries are currently installed you can use the `registry status` (or `registry st`) command

```
pkg> registry st
Registry Status
 [23338594] General (https://github.com/JuliaRegistries/General.git)
```

Registries are always added to the user depot, which is the first entry in DEPOT_PATH.

18.2   Removing registries

Registries can be removed with the `registry remove` (or `registry rm`) command. Here we remove the `General` registry

```
pkg> registry rm General
   Removing registry `General` from ~/.julia/registries/General

pkg> registry st
Registry Status
   (no registries found)
```

In case there are multiple registries named `General` installed you have to disambiguate with the `uuid`, just as when manipulating packages, e.g.

```
pkg> registry rm General=23338594-aafe-5451-b93e-139f81909106
  Removing registry `General` from ~/.julia/registries/General
```

18.3   Updating registries

The `registry update` (or `registry up`) command is available to update registries. Here we update the `General` registry,

```
pkg> registry up General
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General`
```

and to update all installed user registries just do

```
pkg> registry up
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General`
```

Part VIII

8. Glossary

Project: a source tree with a standard layout, including a `src` directory for the main body of Julia code, a `test` directory for testing the project, `docs` for documentation files, and optionally a `deps` directory for a build script and its outputs. A project will typically also have a project file and may optionally have a manifest file:

- **Project file:** a file in the root directory of a project, named `Project.toml` (or `JuliaProject.toml`) describing metadata about the project, including its name, UUID (for packages), authors, license, and the names and UUIDs of packages and libraries that it depends on.

- **Manifest file:** a file in the root directory of a project, named `Manifest.toml` (or `JuliaManifest.toml`) describing a complete dependency graph and exact versions of each package and library used by a project.

Package: a project which provides reusable functionality that can be used by other Julia projects via `import X` or `using X`. A package should have a project file with a `uuid` entry giving its package UUID. This UUID is used to identify the package in projects that depend on it.

> **Note**
>
> For legacy reasons it is possible to load a package without a project file or UUID from the REPL or the top-level of a script. It is not possible, however, to load a package without a project file or UUID from a project with them. Once you've loaded from a project file, everything needs a project file and UUID.

Application: a project which provides standalone functionality not intended to be reused by other Julia projects. For example a web application or a commmand-line utility, or simulation/analytics code accompanying a scientific paper. An application may have a UUID but does not need one. An application may also provide global configuration options for packages it depends on. Packages, on the other hand, may not provide global configuration since that could conflict with the configuration of the main application.

> **Note**
>
> **Projects *vs.* Packages *vs.* Applications:**
>
> 1. **Project** is an umbrella term: packages and applications are kinds of projects.
> 2. **Packages** should have UUIDs, applications can have a UUIDs but don't need them.
> 3. **Applications** can provide global configuration, whereas packages cannot.

Library (future work): a compiled binary dependency (not written in Julia) packaged to be used by a Julia project. These are currently typically built in- place by a `deps/build.jl` script in a project's source tree, but in the future we plan to make libraries first-class entities directly installed and upgraded by the package manager.

Environment: the combination of the top-level name map provided by a project file combined with the dependency graph and map from packages to their entry points provided by a manifest file. For more detail see the manual section on code loading.

- **Explicit environment:** an environment in the form of an explicit project file and an optional corresponding manifest file together in a directory. If the manifest file is absent then the implied dependency graph and location maps are empty.

- **Implicit environment:** an environment provided as a directory (without a project file or manifest file) containing packages with entry points of the form `X.jl`, `X.jl/src/X.jl` or `X/src/X.jl`. The top-level name map is implied by these entry points. The dependency graph is implied by the existence of project files inside of these package directories, e.g. `X.jl/Project.toml` or `X/Project.toml`. The dependencies of the X package are the dependencies in the corresponding project file if there is one. The location map is implied by the entry points themselves.

Registry: a source tree with a standard layout recording metadata about a registered set of packages, the tagged versions of them which are available, and which versions of packages are compatible or incompatible with each other. A registry is indexed by package name and UUID, and has a directory for each registered package providing the following metadata about it:

- name – e.g. `DataFrames`

- UUID – e.g. `a93c6f00-e57d-5684-b7b6-d8193f3e46c0`

- authors – e.g. `Jane Q. Developer <jane@example.com>`

- license – e.g. MIT, BSD3, or GPLv2

- repository – e.g. `https://github.com/JuliaData/DataFrames.jl.git`

- description – a block of text summarizing the functionality of a package

- keywords – e.g. `data`, `tabular`, `analysis`, `statistics`

- versions – a list of all registered version tags

For each registered version of a package, the following information is provided:

- its semantic version number – e.g. `v1.2.3`

- its git tree SHA-1 hash – e.g. `7ffb18ea3245ef98e368b02b81e8a86543a11103`

- a map from names to UUIDs of dependencies

- which versions of other packages it is compatible/incompatible with

Dependencies and compatibility are stored in a compressed but human-readable format using ranges of package versions.

Depot: a directory on a system where various package-related resources live, including:

- `environments`: shared named environments (e.g. `v1.0`, `devtools`)

- `clones`: bare clones of package repositories

- `compiled`: cached compiled package images (`.ji` files)

- `config`: global configuration files (e.g. `startup.jl`)

- `dev`: default directory for package development

- `logs`: log files (e.g. `manifest_usage.toml`, `repl_history.jl`)

- `packages`: installed package versions

- `registries`: clones of registries (e.g. `General`)

Load path: a stack of environments where package identities, their dependencies, and entry-points are searched for. The load path is controlled in Julia by the `LOAD_PATH` global variable which is populated at startup based on the value of the `JULIA_LOAD_PATH` environment variable. The first entry is your primary environment, often the current project, while later entries provide additional packages one may want to use from the REPL or top-level scripts.

Depot path: a stack of depot locations where the package manager, as well as Julia's code loading mechanisms, look for registries, installed packages, named environments, repo clones, cached compiled package images, and configuration files. The depot path is controlled by the Julia `DEPOT_PATH` global variable which is populated at startup based on the value of the `JULIA_DEPOT_PATH` environment variable. The first entry is the "user depot" and should be writable by and owned by the current user. The user depot is where: registries are cloned, new package versions are installed, named environments are created and updated, package repos are cloned, newly compiled package image files are saved, log files are written, development packages are checked out by default, and global configuration data is saved. Later entries in the depot path are treated as read-only and are appropriate for registries, packages, etc. installed and managed by system administrators.

Part IX

9. `Project.toml` and `Manifest.toml`

Two files that are central to Pkg are `Project.toml` and `Manifest.toml`. `Project.toml` and `Manifest.toml` are written in TOML (hence the `.toml` extension) and include information about dependencies, versions, package names, UUIDs etc.

> **Note**
>
> The `Project.toml` and `Manifest.toml` files are not only used by the package manager, they are also used by Julia's code loading, and determines e.g. what `using Example` should do. For more details see the section about Code Loading in the Julia manual.

Chapter 19

`Project.toml`

The project file describes the project on a high level, for example the package/project dependencies and compatibility constraints are listed in the project file. The file entries are described below.

19.1   The `name` field

The name of the package/project is determined by the name field. Example:

```
name = "Example"
```

The name can contain word characters `[a-zA-Z0-9_]`, but can not start with a number. For packages it is recommended to follow the package naming guidelines. The name field is mandatory for packages.

19.2   The `uuid` field

uuid is a string with a universally unique identifier for the package/project. Example:

```
uuid = "7876af07-990d-54b4-ab0e-23690620f79a"
```

The uuid field is mandatory for packages.

19.3   The `version` field

version is a string with the version number for the package/project. It should consist of three numbers, major version, minor version and patch number, separated with a `.`. Example:

```
version = "1.2.5"
```

Julia uses Semantic Versioning (SemVer) and the version field should follow SemVer. The basic rules are:

- Before 1.0.0, anything goes, but when you make breaking changes the minor version should be incremented.

- After 1.0.0 only make breaking changes when incrementing the major version.

- After 1.0.0 no new public API should be added without incrementing the minor version. This includes, in particular, new types, functions, methods and method overloads, from Base or other packages.

See also the section on Compatibility.

19.4   The [deps] section

All dependencies of the package/project are listed in the [deps] section. Each dependency is listed as a name-uuid pair, for example:

```
[deps]
Example = "7876af07-990d-54b4-ab0e-23690620f79a"
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"
```

Typically it is not needed to manually add entries to the [deps] section, this is instead handled by Pkg operations such as add.

19.5   The [compat] section

Compatibility constraints for the dependencies listed under [deps] can be listed in the [compat] section. Example:

```
[deps]
Example = "7876af07-990d-54b4-ab0e-23690620f79a"

[compat]
Example = "1.2"
```

The Compatibility section describes the different possible compatibility constraints in detail. It is also possible to list constraints on julia itself, although julia is not listed as a dependency in the [deps] section:

```
[compat]
julia = "1.1"
```

Chapter 20

`Manifest.toml`

The manifest file is an absolute record of the state of the packages in the environment. It includes exact information about (direct and indirect) dependencies of the project, and given a `Project.toml` + `Manifest.toml` pair it is possible to instantiate the exact same package environment, which is very useful for reproducibility, see `Pkg.instantiate`.

> **Note**
>
> The `Manifest.toml` file is generated and maintained by Pkg and, in general, this file should *never* be modified manually.

20.1 `Manifest.toml` entries

Each dependency has its own section in the manifest file, and its content varies depending on how the dependency was added to the environment, see the examples below. Every dependency section includes a combination of the following entries:

- uuid: the UUID for the dependency, for example uuid = "7876af07-990d-54b4-ab0e-23690620f79a".

- deps: a vector listing the dependencies of the dependency, for example deps = ["Example", "JSON"].

- version: a version number, for example version = "1.2.6".

- path: a file path to the source code, for example path = /home/user/Example.

- repo-url: a URL to the repository where the source code was found, for example repo-url = "https://github.com/JuliaL

- repo-rev: a git revision, for example a branch repo-rev = "master" or a commit repo-rev = "66607a62a83cb07ab18c0b3

- git-tree-sha1: a content hash of the source tree, for example git-tree-sha1 = "ca3820cc4e66f473467d912c4b2b3ae5d

Added package

When a package is added from a package registry, for example by invoking pkg> add Example or with a specific version pkg> add Example@1.2 the resulting `Manifest.toml` entry looks like:

```
[[Example]]
deps = ["DependencyA", "DependencyB"]
git-tree-sha1 = "8eb7b4d4ca487caade9ba3e85932e28ce6d6e1f8"
uuid = "7876af07-990d-54b4-ab0e-23690620f79a"
version = "1.2.3"
```

Note, in particular, that no `repo-url` is present, since that information is included in the registry where this package were found.

Added package by branch

The resulting dependency section when adding a package specified by a branch, e.g. pkg> add Example#master
or pkg> add https://github.com/JuliaLang/Example.jl.git looks like:

```
[[Example]]
deps = ["DependencyA", "DependencyB"]
git-tree-sha1 = "54c7a512469a38312a058ec9f429e1db1f074474"
repo-rev = "master"
repo-url = "https://github.com/JuliaLang/Example.jl.git"
uuid = "7876af07-990d-54b4-ab0e-23690620f79a"
version = "1.2.4"
```

Note that both the branch we are tracking (master) and the remote repository url ("https://github.com/JuliaLang/Example.jl.git")
are stored in the manifest.

Added package by commit

The resulting dependency section when adding a package specified by a commit, e.g. pkg> add Example#cf6ba6cc0be0bb5f56840188563579d
looks like

```
[[Example]]
deps = ["DependencyA", "DependencyB"]
git-tree-sha1 = "54c7a512469a38312a058ec9f429e1db1f074474"
repo-rev = "cf6ba6cc0be0bb5f56840188563579d67048be34"
repo-url = "https://github.com/JuliaLang/Example.jl.git"
uuid = "7876af07-990d-54b4-ab0e-23690620f79a"
version = "1.2.4"
```

The only difference from tracking a branch is the content of repo-rev.

Developed package

The resulting dependency section when adding a package with develop, e.g. pkg> develop Example or pkg>
develop /path/to/local/folder/Example looks like:

```
[[Example]]
deps = ["DependencyA", "DependencyB"]
path = "/home/user/.julia/dev/Example/"
uuid = "7876af07-990d-54b4-ab0e-23690620f79a"
version = "1.2.4"
```

Note that the path to the source code is included, and changes made to that source tree is directly reflected.

Pinned package

Pinned packages are also recorded in the manifest file, the resulting dependency section for e.g. pkg> add Example;
pin Example looks like:

```
[[Example]]
deps = ["DependencyA", "DependencyB"]
git-tree-sha1 = "54c7a512469a38312a058ec9f429e1db1f074474"
pinned = true
uuid = "7876af07-990d-54b4-ab0e-23690620f79a"
version = "1.2.4"
```

The only difference is the addition of the pinned = true entry.

Multiple package with the same name

Julia differentiates packages based on UUID, which means that the name alone is not enough to identify a package. It is possible to have multiple packages in the same environment with the same name, but with different UUID. In such a situation the `Manifest.toml` file looks a bit different. Consider for example the situation where you have added A and B to your environment, and the `Project.toml` file looks as follows:

```
[deps]
A = "ead4f63c-334e-11e9-00e6-e7f0a5f21b60"
B = "edca9bc6-334e-11e9-3554-9595dbb4349c"
```

If A now depends on B = "f41f7b98-334e-11e9-1257-49272045fb24", i.e. *another* package named B there will be two different B packages in the `Manifest.toml` file. In this case the full `Manifest.toml` file, with `git-tree-sha1` and `version` fields removed for clarity, looks like:

```
[[A]]
uuid = "ead4f63c-334e-11e9-00e6-e7f0a5f21b60"

    [A.deps]
    B = "f41f7b98-334e-11e9-1257-49272045fb24"

[[B]]
uuid = "f41f7b98-334e-11e9-1257-49272045fb24"
[[B]]
uuid = "edca9bc6-334e-11e9-3554-9595dbb4349c"
```

There is now an array of the two B packages, and the `[deps]` section for A has been expanded in order to be explicit about which B package A depends on.

Part X

10. REPL Mode Reference

This section describes available commands in the Pkg REPL. The REPL mode is mostly meant for interactive use, and for non-interactive use it is recommended to use the "API mode", see API Reference.

Chapter 21

package commands

```
add pkg[=uuid] [@version] [#rev] ...
```

Add package pkg to the current project file. If pkg could refer to multiple different packages, specifying uuid allows you to disambiguate. @version optionally allows specifying which versions of packages. Versions may be specified by @1, @1.2, @1.2.3, allowing any version with a prefix that matches, or ranges thereof, such as @1.2-3.4.5. A git-revision can be specified by #branch or #commit.

If a local path is used as an argument to add, the path needs to be a git repository. The project will then track that git repository just like if it is was tracking a remote repository online.

Examples

```
pkg> add Example
pkg> add Example@0.5
pkg> add Example#master
pkg> add Example#c37b675
pkg> add https://github.com/JuliaLang/Example.jl#master
pkg> add git@github.com:JuliaLang/Example.jl.git
pkg> add Example=7876af07-990d-54b4-ab0e-23690620f79a
```

```
build [-v|verbose] pkg[=uuid] ...
```

Run the build script in deps/build.jl for each package in pkg and all of their dependencies in depth-first recursive order. If no packages are given, runs the build scripts for all packages in the manifest. The -v/--verbose option redirects build output to stdout/stderr instead of the build.log file. The startup.jl file is disabled during building unless julia is started with --startup-file=yes.

```
develop [--shared|--local] pkg[=uuid] ...
```

Make a package available for development. If pkg is an existing local path that path will be recorded in the manifest and used. Otherwise, a full git clone of pkg at rev rev is made. The location of the clone is controlled by the --shared (default) and --local arguments. The --shared location defaults to ~/.julia/dev, but can be controlled with the JULIA_PKG_DEVDIR environment variable. When --local is given, the clone is placed in a dev folder in the current project. This operation is undone by free.

Example

```
pkg> develop Example
pkg> develop https://github.com/JuliaLang/Example.jl
pkg> develop ~/mypackages/Example
pkg> develop --local Example
```

```
free pkg[=uuid] ...
```

Free a pinned package pkg, which allows it to be upgraded or downgraded again. If the package is checked out (see `help develop`) then this command makes the package no longer being checked out.

```
generate pkgname
```

Create a project called pkgname in the current folder.

```
pin pkg[=uuid] ...
```

Pin packages to given versions, or the current version if no version is specified. A pinned package has its version fixed and will not be upgraded or downgraded. A pinned package has the symbol next to its version in the status list.

```
rm [-p|--project] pkg[=uuid] ...
```

Remove package `pkg` from the project file. Since the name `pkg` can only refer to one package in a project this is unambiguous, but you can specify a uuid anyway, and the command is ignored, with a warning if package name and UUID do not mactch. When a package is removed from the project file, it may still remain in the manifest if it is required by some other package in the project. Project mode operation is the default, so passing -p or --project is optional unless it is preceded by the -m or --manifest options at some earlier point.

```
rm [-m|--manifest] pkg[=uuid] ...
```

Remove package `pkg` from the manifest file. If the name `pkg` refers to multiple packages in the manifest, uuid disambiguates it. Removing a package from the manifest forces the removal of all packages that depend on it, as well as any no-longer-necessary manifest packages due to project package removals.

```
test [opts] pkg[=uuid] ...

opts: --coverage
```

Run the tests for package `pkg`. This is done by running the file `test/runtests.jl` in the package directory. The option --coverage can be used to run the tests with coverage enabled. The `startup.jl` file is disabled during testing unless julia is started with --startup-file=yes.

```
up [-p|project]  [opts] pkg[=uuid] [@version] ...
up [-m|manifest] [opts] pkg[=uuid] [@version] ...

opts: --major | --minor | --patch | --fixed
```

Update the indicated package within the constraints of the indicated version specifications. Versions may be specified by @1, @1.2, @1.2.3, allowing any version with a prefix that matches, or ranges thereof, such as @1.2-3.4.5. In --project mode, package specifications only match project packages, while in manifest mode they match any manifest package. Bound level options force the following packages to be upgraded only within the current major, minor, patch version; if the --fixed upgrade level is given, then the following packages will not be upgraded at all.

Chapter 22

`registry` commands

```
registry add reg...
```

Adds package registries `reg...` to the user depot.

> **Julia 1.1**
>
> Pkg's registry handling requires at least Julia 1.1.

Examples

```
pkg> registry add General
pkg> registry add https://www.my-custom-registry.com
```

```
registry rm reg...
```

Remove package registres `reg...`.

> **Julia 1.1**
>
> Pkg's registry handling requires at least Julia 1.1.

Examples

```
pkg> registry rm General
```

```
registry status
```

Display information about installed registries.

> **Julia 1.1**
>
> Pkg's registry handling requires at least Julia 1.1.

Examples

```
pkg> registry status
```

```
registry up
registry up reg...
```

Update package registries `reg...`. If no registries are specified all user registries will be updated.

Julia 1.1

Pkg's registry handling requires at least Julia 1.1.

Examples

```
pkg> registry up
pkg> registry up General
```

Chapter 23

Other commands

```
activate
activate [--shared] path
```

Activate the environment at the given `path`, or the home project environment if no `path` is specified. The active environment is the environment that is modified by executing package commands. When the option `--shared` is given, `path` will be assumed to be a directory name and searched for in the `environments` folders of the depots in the depot stack. In case no such environment exists in any of the depots, it will be placed in the first depot of the stack.

Deletes packages that cannot be reached from any existing environment.

```
help
```

List available commands along with short descriptions.

```
help cmd
```

If cmd is a partial command, display help for all subcommands. If cmd is a full command, display help for cmd.

```
instantiate
instantiate [-m|--manifest]
instantiate [-p|--project]
```

Download all the dependencies for the current project at the version given by the project's manifest. If no manifest exists or the `--project` option is given, resolve and download the dependencies compatible with the project.

```
precompile
```

Precompile all the dependencies of the project by running `import` on all of them in a new process. The `startup.jl` file is disabled during precompilation unless julia is started with `--startup-file=yes`.

```
resolve
```

Resolve the project i.e. run package resolution and update the Manifest. This is useful in case the dependencies of developed packages have changed causing the current Manifest to be out of sync.

```
status [pkgs...]
status [-p|--project] [pkgs...]
status [-m|--manifest] [pkgs...]
```

Show the status of the current environment. By default, the full contents of the project file is summarized, showing what version each package is on and how it has changed since the last git commit (if in a git repo), as well as any

changes to manifest packages not already listed. In `--project` mode, the status of the project file is summarized. In `--manifest` mode the output also includes the dependencies of explicitly added packages. If there are any packages listed as arguments the output will be limited to those packages.

Julia 1.1

`pkg> status` with package arguments requires at least Julia 1.1.

Part XI

11. API Reference

This section describes the function interface, or "API mode" for interacting with Pkg.jl. The function API is recommended for non-interactive usage, in i.e. scripts.

Chapter 24

Package API Reference

In the REPL mode packages (with associated version, UUID, URL etc) are parsed from strings, for example, `"Package#master"`,`"Package@` `"www.mypkg.com/MyPkg#my/feature"`. It is possible to use strings as arguments for simple commands in the API mode (like `Pkg.add(["PackageA", "PackageB"])`, more complicated commands, that e.g. specify URLs or version range, uses a more structured format over strings. This is done by creating an instance of a `PackageSpec` which are passed in to functions.

`Pkg.PackageSpec` – Function.

```
PackageSpec(name::String, [uuid::UUID, version::VersionNumber])
PackageSpec(; name, url, path, rev, version, mode, level)
```

A PackageSpec is a representation of a package with various metadata. This includes:

- The `name` of the package.
- The package unique `uuid`.
- A `version` (for example when adding a package. When upgrading, can also be an instance of

the enum `UpgradeLevel`

- A `url` and an optional git `revision`. `rev` could be a branch name or a git commit SHA.
- A local path `path`. This is equivalent to using the `url` argument but can be more descriptive.
- A mode, which is an instance of the enum `PackageMode` which can be either PKGMODE_PROJECT or

PKGMODE_MANIFEST, defaults to PKGMODE_PROJECT. Used in e.g. `Pkg.rm`.

Most functions in Pkg take a `Vector` of `PackageSpec` and do the operation on all the packages in the vector.

Below is a comparison between the REPL version and the `PackageSpec` version:

source

`Pkg.PackageMode` – Type.

```
PackageMode
```

An enum with the instances

- `PKGMODE_MANIFEST`
- `PKGMODE_PROJECT`

REPL	API
`Package`	`PackageSpec("Package")`
`Package@0.2`	`PackageSpec(name="Package", version="0.2")`
`Package=a67d...`	`PackageSpec(name="Package", uuid="a67d...")`
`Package#master`	`PackageSpec(name="Package", rev="master")`
`local/path#feature`	`PackageSpec(path="local/path"; rev="feature")`
`www.mypkg.com`	`PackageSpec(url="www.mypkg.com")`
`--manifest Package`	`PackageSpec(name="Package", mode=PKGSPEC_MANIFEST)`
`--major Package`	`PackageSpec(name="Package", version=PKGLEVEL_MAJOR)`

Determines if operations should be made on a project or manifest level. Used as an argument to `PackageSpec` or as an argument to `Pkg.rm`.

source

`Pkg.UpgradeLevel` – Type.

```
UpgradeLevel
```

An enum with the instances

- `UPLEVEL_FIXED`
- `UPLEVEL_PATCH`
- `UPLEVEL_MINOR`
- `UPLEVEL_MAJOR`

Determines how much a package is allowed to be updated. Used as an argument to `PackageSpec` or as an argument to `Pkg.update`.

source

`Pkg.add` – Function.

```
Pkg.add(pkg::Union{String, Vector{String}})
Pkg.add(pkg::Union{PackageSpec, Vector{PackageSpec}})
```

Add a package to the current project. This package will be available using the `import` and `using` keywords in the Julia REPL and if the current project is a package, also inside that package.

Examples

```
Pkg.add("Example") # Add a package from registry
Pkg.add(PackageSpec(name="Example", version="0.3")) # Specify version; latest release in the
↪    0.3 series
Pkg.add(PackageSpec(name="Example", version="0.3.1")) # Specify version; exact release
Pkg.add(PackageSpec(url="https://github.com/JuliaLang/Example.jl", rev="master")) # From url
↪    to remote gitrepo
Pkg.add(PackageSpec(url="/remote/mycompany/juliapackages/OurPackage"))` # From path to local
↪    gitrepo
```

See also `PackageSpec`.

source

`Pkg.develop` – Function.

```
Pkg.develop(pkg::Union{String, Vector{String}})
Pkg.develop(pkgs::Union{Packagespec, Vector{Packagespec}})
```

Make a package available for development by tracking it by path. If pkg is given with only a name or by a URL the packages will be downloaded to the location by the environment variable JULIA_PKG_DEVDIR with .julia/dev as the default.

If pkg is given as a local path, the package at that path will be tracked.

Examples

```
# By name
Pkg.develop("Example")

# By url
Pkg.develop(PackageSpec(url="https://github.com/JuliaLang/Compat.jl"))

# By path
Pkg.develop(PackageSpec(path="MyJuliaPackages/Package.jl"))
```

See also PackageSpec

source

Pkg.activate – Function.

```
Pkg.activate([s::String]; shared::Bool=false)
```

Activate the environment at s. The active environment is the environment that is modified by executing package commands. The logic for what path is activated is as follows:

- If shared is true, the first existing environment named s from the depots in the depot stack will be activated. If no such environment exists, create and activate that environment in the first depot.

- If s is an existing path, then activate the environment at that path.

- If s is a package in the current project and s is tracking a path, then activate the environment at the tracked path.

- Else, s is interpreted as a non-existing path, activate that path.

If no argument is given to activate, then activate the home project. The home project is specified by either the --project command line option to the julia executable, or the JULIA_PROJECT environment variable.

Examples

```
Pkg.activate()
Pkg.activate("local/path")
Pkg.activate("MyDependency")
```

source

Pkg.rm – Function.

```
Pkg.rm(pkg::Union{String, Vector{String}})
Pkg.rm(pkg::Union{PackageSpec, Vector{PackageSpec}})
```

Remove a package from the current project. If the mode of pkg is PKGMODE_MANIFEST also remove it from the manifest including all recursive dependencies of pkg.

See also PackageSpec, PackageMode.

source

Pkg.update – Function.

```
Pkg.update(; level::UpgradeLevel=UPLEVEL_MAJOR, mode::PackageMode = PKGMODE_PROJECT)
Pkg.update(pkg::Union{String, Vector{String}})
Pkg.update(pkg::Union{PackageSpec, Vector{PackageSpec}})
```

Update a package pkg. If no posistional argument is given, update all packages in the manifest if mode is PKGMODE_MANIFEST and packages in both manifest and project if mode is PKGMODE_PROJECT. If no positional argument is given level can be used to control what how much packages are allowed to be upgraded (major, minor, patch, fixed).

See also PackageSpec, PackageMode, UpgradeLevel.

source

Pkg.test – Function.

```
Pkg.test(; coverage::Bool=false)
Pkg.test(pkg::Union{String, Vector{String}; coverage::Bool=false)
Pkg.test(pkgs::Union{PackageSpec, Vector{PackageSpec}}; coverage::Bool=false)
```

Run the tests for package pkg or if no positional argument is given to test, the current project is tested (which thus needs to be a package). A package is tested by running its test/runtests.jl file.

The tests are run by generating a temporary environment with only pkg and its (recursive) dependencies (recursively) in it. If a manifest exist, the versions in that manifest is used, otherwise a feasible set of package are resolved and installed.

During the test, test-specific dependencies are active, which are given in the project file as e.g.

```
[extras]
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"

[targets]
test = ["Test"]
```

Coverage statistics for the packages may be generated by passing coverage=true. The default behavior is not to run coverage.

The tests are executed in a new process with check-bounds=yes and by default startup-file=no. If using the startup file (~/.julia/config/startup.jl) is desired, start julia with --startup-file=yes. Inlining of functions during testing can be disabled (for better coverage accuracy) by starting julia with --inline=no.

source

Pkg.build – Function.

```
Pkg.build(; verbose = false)
Pkg.build(pkg::Union{String, Vector{String}}; verbose = false)
Pkg.build(pkgs::Union{PackageSpec, Vector{PackageSpec}}; verbose = false)
```

Run the build script in deps/build.jl for pkg and all of the dependencies in depth-first recursive order. If no argument is given to build, the current project is built, which thus needs to be a package. This function is called

automatically one any package that gets installed for the first time. `verbose = true` prints the build output to stdout/stderr instead of redirecting to the `build.log` file.

source

Pkg.pin – Function.

```
Pkg.pin(pkg::Union{String, Vector{String}})
Pkg.pin(pkgs::Union{PackageSpec, Vector{PackageSpec}})
```

Pin a package to the current version (or the one given in the `PackageSpec`) or to a certain git revision. A pinned package is never updated.

Examples

```
Pkg.pin("Example")
Pkg.pin(PackageSpec(name="Example", version="0.3.1"))
```

source

Pkg.free – Function.

```
Pkg.free(pkg::Union{String, Vector{String}})
Pkg.free(pkgs::Union{PackageSpec, Vector{PackageSpec}})
```

Free a package which removes a `pin` if it exists, or if the package is tracking a path, e.g. after `Pkg.develop`, go back to tracking registered versions.

Examples

```
Pkg.free("Package")
Pkg.free(PackageSpec("Package"))
```

source

Pkg.instantiate – Function.

```
Pkg.instantiate()
```

If a `Manifest.toml` file exist in the current project, download all the packages declared in that manifest. Else, resolve a set of feasible packages from the `Project.toml` files and install them.

source

Pkg.resolve – Function.

```
Pkg.resolve()
```

Update the current manifest with eventual changes to the dependency graph from packages that are tracking a path.

source

Pkg.gc – Function.

```
Pkg.gc()
```

Garbage collect packages that are no longer reachable from any project. Only packages that are tracked by version are deleted, so no packages that might contain local changes are touched.

source

`Pkg.status` – Function.

```
Pkg.status([pkgs...]; mode::PackageMode=PKGMODE_PROJECT)
```

Print out the status of the project/manifest. If mode is `PKGMODE_PROJECT` prints out status about only those packages that are in the project (explicitly added). If mode is `PKGMODE_MANIFEST` also print for those in the manifest (recursive dependencies). If there are any packages listed as arguments the output will be limited to those packages.

> **Julia 1.1**
>
> `Pkg.status` with package arguments requires at least Julia 1.1.

source

`Pkg.setprotocol!` – Function.

```
setprotocol!(;
    domain::AbstractString = "github.com",
    protocol::Union{Nothing, AbstractString}=nothing
)
```

Set the protocol used to access hosted packages when adding a url or `developing` a package. Defaults to delegating the choice to the package developer (`proto == nothing`). Other choices for `proto` are `"https"` or `"git"`.

Examples

```
julia> Pkg.setprotocol!(domain = "github.com", protocol = "ssh")

julia> Pkg.setprotocol!(domain = "gitlab.mycompany.com")
```

source

Chapter 25

Registry API Reference

Julia 1.1

Pkg's registry handling requires at least Julia 1.1.

The function API for registries uses RegistrySpecs, similar to PackageSpec.

Pkg.RegistrySpec – Type.

```
RegistrySpec(name::String)
RegistrySpec(; name, url, path)
```

A `RegistrySpec` is a representation of a registry with various metadata, much like PackageSpec.

Most registry functions in Pkg take a `Vector` of `RegistrySpec` and do the operation on all the registries in the vector.

Julia 1.1

Pkg's registry handling requires at least Julia 1.1.

Examples

Below is a comparison between the REPL version and the `RegistrySpec` version:

REPL	API
Registry	RegistrySpec("Registry")
Registry=a67d...	RegistrySpec(name="Registry", uuid="a67d...")
local/path	RegistrySpec(path="local/path")
www.myregistry.com	RegistrySpec(url="www.myregistry.com")

source

Pkg.Registry.add – Function.

```
Pkg.Registry.add(url::String)
Pkg.Registry.add(registry::RegistrySpec)
```

Add new package registries.

Julia 1.1

Pkg's registry handling requires at least Julia 1.1.

Examples

```
Pkg.Registry.add("General")
Pkg.Registry.add(RegistrySpec(uuid = "23338594-aafe-5451-b93e-139f81909106"))
Pkg.Registry.add(RegistrySpec(url = "https://github.com/JuliaRegistries/General.git"))
```

source

Pkg.Registry.rm – Function.

```
Pkg.Registry.rm(registry::String)
Pkg.Registry.rm(registry::RegistrySpec)
```

Remove registries.

> **Julia 1.1**
>
> Pkg's registry handling requires at least Julia 1.1.

Examples

```
Pkg.Registry.rm("General")
Pkg.Registry.rm(RegistrySpec(uuid = "23338594-aafe-5451-b93e-139f81909106"))
```

source

Pkg.Registry.update – Function.

```
Pkg.Registry.update()
Pkg.Registry.update(registry::RegistrySpec)
Pkg.Registry.update(registry::Vector{RegistrySpec})
```

Update registries. If no registries are given, update all available registries.

> **Julia 1.1**
>
> Pkg's registry handling requires at least Julia 1.1.

Examples

```
Pkg.Registry.update()
Pkg.Registry.update("General")
Pkg.Registry.update(RegistrySpec(uuid = "23338594-aafe-5451-b93e-139f81909106"))
```

source

Pkg.Registry.status – Function.

```
Pkg.Registry.status()
```

Display information about available registries.

> **Julia 1.1**
>
> Pkg's registry handling requires at least Julia 1.1.

Examples

```
Pkg.Registry.status()
```

source