Pkg.jl

October 16, 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`.

Let's 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`.

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
```

> **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. 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:

```
(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 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 shows all the packages in the environment, including recursive dependencies:

```
(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 a branch (or a certain commit) of `Example` has a hotfix that is not yet included in a registered version, we can explicitly track that branch (or commit) by appending `#branchname` (or `#commitSHA1`) 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 be added by specifying a URL to the repository:

```
(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 name (or commit SHA1) 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 similar 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 tracking a package through `add` is distinct from `develop`: 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.

In addition, it is possible to add packages relatively to the `Manifest.toml` file, see Developing Packages for an example.

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 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. When tracking a path, the package manager will never modify the files at that 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 synchronize the Manifest, use the REPL command `resolve`.

In addition to absolute paths, add and dev can accept relative paths to packages. In this case, the relative path from the active project to the package is stored. This approach is useful when the relative location of tracked dependencies is more important than their absolute location. For example, the tracked dependencies can be stored inside of the active project directory. The whole directory can be moved and Pkg will still be able to find the dependencies because their path relative to the active project is preserved even though their absolute path has changed.

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 depend on `DepPackage`.

Chapter 8

Updating packages

When new versions of packages that 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 local 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`, 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 package versions and artifacts 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/artifacts those projects used. If a package or artifact is not marked as used by any project, it is added to a list of orphaned packages. Packages and artifacts that are in the orphan list for 30 days without being used again are deleted from the system on the next garbage collection. This timing is configurable via the `collect_delay` keyword argument to `Pkg.gc()`. A value of `0` will cause anything currently not in use immediately, skipping the orphans list entirely; If you are short on disk space and want to clean out as many unused packages and artifacts as possible, you may want to try this, but if you need these versions again, you will have to download them again. To run a typical garbage collection with default arguments, simply use the `gc` command at the `pkg>` REPL:

```
(v1.0) pkg> gc
    Active manifests at:
        `~/BinaryProvider/Manifest.toml`
        ...
        `~/Compat.jl/Manifest.toml`
    Active artifacts:
        `~/src/MyProject/Artifacts.toml`

    Deleted ~/.julia/packages/BenchmarkTools/1cAj: 146.302 KiB
    Deleted ~/.julia/packages/Cassette/BXVB: 795.557 KiB
   ...
   Deleted `~/.julia/artifacts/e44cdf2579a92ad5cbacd1cddb7414c8b9d2e24e` (152.253 KiB)
   Deleted `~/.julia/artifacts/f2df5266567842bbb8a06acca56bcabf813cd73f` (21.536 MiB)

   Deleted 36 package installations (113.205 MiB)
   Deleted 15 artifact installations (20.759 GiB)
```

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

Part IV

4. Working with Environments

Chapter 13

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 that when two projects use 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 shows that 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 14

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 15

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.

> **Note**
>
> If you have an existing package from an older version of Julia (using a `REQUIRE` file rather than `Project.toml`), then you can generate a `Project.toml` file by first creating a `gen_project.jl` file in your desired package (create a new file called `gen_project.jl` and then paste in the code from here). Next, naviagte to the development directory of your package and then run the script with `julia gen_project.jl` from the terminal. This has to be done (once) before a new release can be registered for an older package. (Delete `REQUIRE` and commit the resulting `Project.toml` after checking it for correctness and adding a `version = "..."` line.)

15.1 Generating files for a package

> **Note**
>
> The PkgTemplates package offers a very easy, repeatable, and customizable way to generate the files for a new package. We recommend that you use PkgTemplates for creating new packages instead of using the minimal `pkg> generate` functionality described below.

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 potential 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!
```

15.2 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 `activate`d it):

```
(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 enforces 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 called:

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

15.3 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
```

15.4 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, are 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.

Note

The exact interaction between `Project.toml`, `test/Project.toml` and their corresponding `Manifest.tomls` are not fully worked out, and may be subject to change in future versions. The old method of adding test-specific dependencies, described in the next section, will therefore be supported throughout all Julia 1.X releases.

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

> **Note**
>
> The method of adding test-specific dependencies described in this section will be replaced by the method from the previous section in future Julia versions. The method in this section will, however, be supported throughout all Julia 1.X releases.

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"]
```

15.5 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.

15.6 Registering packages

Once a package is ready it can be registered with the General Registry. Currently packages are submitted via `Registrator`. In addition to `Registrator`, `TagBot` helps manage the process of tagging releases.

Part VI

6. Compatibility

Compatibility refers to the ability to restrict the versions 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 16

Version specifier format

Similar to other package managers, the Julia package manager respects semantic versioning (semver). As an example, a version specifier 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 as a **tilde specifier**, e.g. `~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)`.

16.1 Behavior of versions with leading zeros (0.0.x and 0.x.y)

While the semver specification says that all versions with a major version of 0 (versions before 1.0.0) are incompatible with each other, we have decided to only apply that for when both the major and minor versions are zero. In other words, 0.0.1 and 0.0.2 are considered incompatible. A pre-1.0 version with non-zero minor version (`0.a.b` with `a != 0`) is considered compatible with versions with the same minor version and smaller or equal patch versions (`0.a.c` with `c <= b`); i.e., the versions 0.2.2 and 0.2.3 are compatible with 0.2.1 and 0.2.0. Versions with a major version of 0 and different minor versions are not considered compatible, so the version 0.3.0 might have breaking changes from 0.2.0. To that end, the `[compat]` entry:

```
[compat]
```

results in a versionbound on `Example` as `[0.0.1, 0.02)` (which is equivalent to only the version 0.0.1), while the `[compat]` entry:

```
[compat]
```

results in a versionbound on Example as `[0.2.1, 0.3.0)`.

In particular, a package may set `version = "0.2.4"` when it has feature additions compared to 0.2.3 as long as it remains backward compatible with 0.2.0. See also The `version` field.

16.2 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)
```

16.3 Tilde specifiers

A tilde specifier provides more limited upgrade possibilities. When specifying major, minor and patch versions, or when specifying major and minor versions, only the 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). For 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)
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)
```

For all versions with a major version of 0 the tilde and caret specifiers are equivalent.

16.4 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 and dependencies, and where they can be downloaded. The `General` registry (https://github.com/JuliaRegistries/General) is the default one, and is installed automatically.

Chapter 17

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.

17.1 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` (cf. the Glossary section).

17.2 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
```

17.3 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 registries just do:

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

Part VIII

8. Artifacts

Pkg can install and manage containers of data that are not Julia packages. These containers can contain platform-specific binaries, datasets, text, or any other kind of data that would be convenient to place within an immutable, life-cycled datastore. These containers, (called "Artifacts") can be created locally, hosted anywhere, and automatically downloaded and unpacked upon installation of your Julia package. This mechanism is also used to provide the binary dependencies for packages built with `BinaryBuilder.jl`.

Chapter 18

`Artifacts.toml` files

Pkg provides an API for working with artifacts, as well as a TOML file format for recording artifact usage in your packages, and to automate downloading of artifacts at package install time. Artifacts can always be referred to by content hash, but are typically accessed by a name that is bound to a content hash in an `Artifacts.toml` file that lives in a project's source tree.

> **Note**
>
> It is possible to use the alternate name `JuliaArtifacts.toml`, similar to how it is possible to use `JuliaProject.toml` and `JuliaManifest.toml` instead of `Project.toml` and `Manifest.toml`, respectively.

An example `Artifacts.toml` file is shown here:

```
# Example Artifacts.toml file
[socrates]
git-tree-sha1 = "43563e7631a7eafae1f9f8d9d332e3de44ad7239"
lazy = true

    [[socrates.download]]
    url = "https://github.com/staticfloat/small_bin/raw/master/socrates.tar.gz"
    sha256 = "e65d2f13f2085f2c279830e863292312a72930fee5ba3c792b14c33ce5c5cc58"

    [[socrates.download]]
    url = "https://github.com/staticfloat/small_bin/raw/master/socrates.tar.bz2"
    sha256 = "13fc17b97be41763b02cbb80e9d048302cec3bd3d446c2ed6e8210bddcd3ac76"

[[c_simple]]
arch = "x86_64"
git-tree-sha1 = "4bdf4556050cb55b67b211d4e78009aaec378cbc"
libc = "musl"
os = "linux"

    [[c_simple.download]]
    sha256 = "411d6befd49942826ea1e59041bddf7dbb72fb871bb03165bf4e164b13ab5130"
    url = "https://github.com/JuliaBinaryWrappers/c_simple_jll.jl/releases/download/c_simple+v1
     .2.3+0/c_simple.v1.2.3.x86_64-linux-musl.tar.gz"

[[c_simple]]
arch = "x86_64"
git-tree-sha1 = "51264dbc770cd38aeb15f93536c29dc38c727e4c"
```

```
os = "macos"

    [[c_simple.download]]
    sha256 = "6c17d9e1dc95ba86ec7462637824afe7a25b8509cc51453f0eb86eda03ed4dc3"
    url = "https://github.com/JuliaBinaryWrappers/c_simple_jll.jl/releases/download/c_simple+v1
     .2.3+0/c_simple.v1.2.3.x86_64-apple-darwin14.tar.gz"

[processed_output]
git-tree-sha1 = "1c223e66f1a8e0fae1f9fcb9d3f2e3ce48a82200"
```

This `Artifacts.toml` binds three artifacts; one named `socrates`, one named `c_simple` and one named `processed_output`. The single required piece of information for an artifact is its `git-tree-sha1`. Because artifacts are addressed only by their content hash, the purpose of an `Artifacts.toml` file is to provide metadata about these artifacts, such as binding a human-readable name to a content hash, providing information about where an artifact may be downloaded from, or even binding a single name to multiple hashes, keyed by platform-specific constraints such as operating system or libgfortran version.

Chapter 19

Artifact types and properties

In the above example, the `socrates` artifact showcases a platform-independent artifact with multiple download locations. When downloading and installing the `socrates` artifact, URLs will be attempted in-order until one succeeds. The `socrates` artifact is marked as `lazy`, which means that it will not be automatically downloaded when the containing package is installed, but rather will be downloaded on-demand when the package first attempts to use it.

The `c_simple` artifact showcases a platform-dependent artifact, where each entry in the `c_simple` array contains keys that help the calling package choose the appropriate download based on the particulars of the host machine. Note that each artifact contains both a `git-tree-sha1` and a `sha256` for each download entry. This is to ensure that the downloaded tarball is secure before attempting to unpack it, as well as enforcing that all tarballs must expand to the same overall tree hash.

The `processed_output` artifact contains no `download` stanza, and so cannot be installed. An artifact such as this would be the result of code that was previously run, generating a new artifact and binding the resultant hash to a name within this project.

Chapter 20

Using Artifacts

Artifacts can be manipulated using convenient APIs exposed from the `Pkg.Artifacts` namespace. As a motivating example, let us imagine that we are writing a package that needs to load the Iris machine learning dataset. While we could just download the dataset during a build step into the package directory, and many packages currently do precisely this, that has some significant drawbacks:

- First, it modifies the package directory, making package installation stateful, which we want to avoid. In the future, we would like to reach the point where packages can be installed completely read-only, instead of being able to modify themselves after installation.
- Second, the downloaded data is not shared across different versions of our package. If we have three different versions of the package installed for use by various projects, then we need three different copies of the data, even if it hasn't changed between those versions. Moreover, each time we upgrade or downgrade the package, unless we do something clever (and probably brittle), we have to download the data again.

With artifacts, we will instead check to see if our `iris` artifact already exists on-disk and only if it doesn't will we download and install it, after which we can bind the result into our `Artifacts.toml` file:

```
using Pkg.Artifacts

# This is the path to the Artifacts.toml we will manipulate
artifact_toml = joinpath(@__DIR__, "Artifacts.toml")

# Query the `Artifacts.toml` file for the hash bound to the name "iris"
# (returns `nothing` if no such binding exists)
iris_hash = artifact_hash("iris", artifact_toml)

# If the name was not bound, or the hash it was bound to does not exist, create it!
if iris_hash == nothing || !artifact_exists(iris_hash)
    # create_artifact() returns the content-hash of the artifact directory once we're finished
    ↪  creating it
    iris_hash = create_artifact() do artifact_dir
        # We create the artifact by simply downloading a few files into the new artifact directory
        iris_url_base = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris"
        download("$(iris_url_base)/iris.data", joinpath(artifact_dir, "iris.csv"))
        download("$(iris_url_base)/bezdekIris.data", joinpath(artifact_dir, "bezdekIris.csv"))
        download("$(iris_url_base)/iris.names", joinpath(artifact_dir, "iris.names"))
    end
```

```
    # Now bind that hash within our `Artifacts.toml`.  `force = true` means that if it already
    ↪  exists,
    # just overwrite with the new content-hash.  Unless the source files change, we do not expect
    # the content hash to change, so this should not cause unnecessary version control churn.
    bind_artifact!(artifact_toml, "iris", iris_hash)
end

# Get the path of the iris dataset, either newly created or previously generated.
# this should be something like `~/.julia/artifacts/dbd04e28be047a54fbe9bf67e934be5b5e0d357a`
```

For the specific use case of using artifacts that were previously bound, we have the shorthand notation `artifact"name"` which will automatically search for the `Artifacts.toml` file contained within the current package, look up the given artifact by name, install it if it is not yet installed, then return the path to that given artifact. An example of this shorthand notation is given below:

```
using Pkg.Artifacts

# For this to work, an `Artifacts.toml` file must be in the current working directory
# (or in the root of the current package) and must define a mapping for the "iris"
# artifact.  If it does not exist on-disk, it will be downloaded.
```

Chapter 21

The `Pkg.Artifacts` API

The `Artifacts` API is broken up into three levels: hash-aware functions, name-aware functions and utility functions.

- **Hash-aware** functions deal with content-hashes and essentially nothing else. These methods allow you to query whether an artifact exists, what its path is, to verify that an artifact satisfies its content hash on-disk, etc. Hash-aware functions include: `artifact_exists()`, `artifact_path()`, `remove_artifact()`, `verify_artifact()` and `archive_artifact()`. Note that in general you should not use `remove_artifact()` and should instead use `Pkg.gc()` to cleanup artifact installations.
- **Name-aware** functions deal with bound names within an `Artifacts.toml` file, and as such, typically require both a path to an `Artifacts.toml` file as well as the artifact name. Name-aware functions include: `artifact_meta()`, `artifact_hash()`, `bind_artifact!()`, `unbind_artifact!()`, `download_artifact()` and `ensure_artifact_installed()`.
- **Utility** functions deal with miscellaneous aspects of artifact life, such as `create_artifact()`, `ensure_all_artifacts_inst` and even the `@artifact_str` string macro.

For a full listing of docstrings and methods, see the Artifacts Reference section.

Chapter 22

Overriding artifact locations

It is occasionally necessary to be able to override the location and content of an artifact. A common use case is a computing environment where certain versions of a binary dependency must be used, regardless of what version of this dependency a package was published with. While a typical Julia configuration would download, unpack and link against a generic library, a system administrator may wish to disable this and instead use a library already installed on the local machine. To enable this, Pkg supports a per-depot `Overrides.toml` file placed within the `artifacts` depot directory (e.g. `~/.julia/artifacts/Overrides.toml` for the default user depot) that can override the location of an artifact either by content-hash or by package UUID and bound artifact name. Additionally, the destination location can be either an absolute path, or a replacement artifact content hash. This allows sysadmins to create their own artifacts which they can then use by overriding other packages to use the new artifact.

```
# Override single hash to absolute path
78f35e74ff113f02274ce60dab6e92b4546ef806 = "/path/to/replacement"

# Override single hash to new artifact content-hash
683942669b4639019be7631caa28c38f3e1924fe = "d826e316b6c0d29d9ad0875af6ca63bf67ed38c3"

# Override package bindings by specifying the package UUID and bound artifact name
# For demonstration purposes we assume this package is called `Foo`
[d57dbccd-ca19-4d82-b9b8-9d660942965b]
libfoo = "/path/to/libfoo"
libbar = "683942669b4639019be7631caa28c38f3e1924fe"
```

Due to the layered nature of Pkg depots, multiple `Overrides.toml` files may be in effect at once. This allows the "inner" `Overrides.toml` files to override the overrides placed within the "outer" `Overrides.toml` files. To remove an override and re-enable default location logic for an artifact, insert an entry mapping to the empty string:

```
78f35e74ff113f02274ce60dab6e92b4546ef806 = "/path/to/new/replacement"
683942669b4639019be7631caa28c38f3e1924fe = ""

[d57dbccd-ca19-4d82-b9b8-9d660942965b]
libfoo = ""
```

If the two `Overrides.toml` snippets as given above are layered on top of eachother, the end result will be mapping the content-hash `78f35e74ff113f02274ce60dab6e92b4546ef806` to `"/path/to/new/replacement"`, and mapping `Foo.libbar` to the artifact identified by the content-hash `683942669b4639019be7631caa28c38f3e1924fe`. Note that while that hash was previously overridden, it is no longer, and therefore `Foo.libbar` will look directly at locations such as `~/.julia/artifacts/683942669b4639019be7631caa28c38f3e1924fe`.

Most methods that are affected by overrides have the ability to ignore overrides by setting `honor_overrides=false` as a keyword argument within them. For UUID/name based overrides to work, `Artifacts.toml` files must be loaded with the knowledge of the UUID of the loading package. This is deduced automatically by the `artifacts""` string macro, however if you are for some reason manually using the `Pkg.Artifacts` API within your package and you wish to honor overrides, you must provide the package UUID to API calls like `artifact_meta()` and `ensure_artifact_installed()` via the `pkg_uuid` keyword argument.

Part IX

9. 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, a `docs` directory 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 X

10. `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 23

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.

23.1 The `name` field

The name of the package/project is determined by the `name` field, for 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.

23.2 The `uuid` field

`uuid` is a string with a universally unique identifier for the package/project, for example:

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

The `uuid` field is mandatory for packages.

23.3 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 `.`, for 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.

Note that Pkg.jl deviates from the SemVer specification when it comes to versions pre-1.0.0. See Pre-1.0 behavior for more details.

23.4 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.

23.5 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 24

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.

24.1 `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/Juli`
- `repo-rev`: a git revision, for example a branch `repo-rev = "master"` or a commit `repo-rev = "66607a62a83cb07ab18c0b`
- `git-tree-sha1`: a content hash of the source tree, for example `git-tree-sha1 = "ca3820cc4e66f473467d912c4b2b3ae5c`

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#cf6ba6cc0be0bb5f56840188563`
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 XI

11. 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 25

package commands

```
add [--preserve=<opt>] 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 to add. Version specifications are of the form `@1`, `@1.2` or `@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 it would track a remote repository online.

Pkg resolves the set of packages in your environment using a tiered approach. The `--preserve` command line option allows you to key into a specific tier in the resolve algorithm. The following table describes the command line arguments to `--preserve` (in order of strictness).

Argument	Description
`all`	Preserve the state of all existing dependencies (including recursive dependencies)
`direct`	Preserve the state of all existing direct dependencies
`semver`	Preserve semver-compatible versions of direct dependencies
`none`	Do not attempt to preserve any version information
`tiered`	Use the tier which will preserve the most version information (this is the default)

Examples

```
pkg> add Example
pkg> add --preserve=all 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 `pkg` and all of its dependencies in depth-first recursive order. If no packages are given, run 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 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`.

Examples

```
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.

Examples

```
pkg> pin Example
pkg> pin Example@0.5.0
pkg> pin Example=7876af07-990d-54b4-ab0e-23690620f79a@0.5.0
```

```
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 match. 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 [--coverage] pkg[=uuid] ...
```

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 pkg within the constraints of the indicated version specifications. These specifications are of the form `@1`, `@1.2` or `@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 26

`registry` commands

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

Add 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 registries `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 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 27

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.

```
gc
```

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 [-v|--verbose]
instantiate [-v|--verbose] [-m|--manifest]
instantiate [-v|--verbose] [-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 [-d|--diff] [pkgs...]
status [-d|--diff] [-p|--project] [pkgs...]
status [-d|--diff] [-m|--manifest] [pkgs...]
```

Show the status of the current environment. In `--project` mode (default), the status of the project file is summarized. In `--manifest` mode the output also includes the recursive dependencies of added packages given in the manifest. If there are any packages listed as arguments the output will be limited to those packages. The `--diff` option will, if the environment is in a git repository, limit the output to the difference as compared to the last git commit.

Julia 1.1

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

Julia 1.3

The `--diff` option requires Julia 1.3. In earlier versions `--diff` is the default for environments in git repositories.

Part XII

12. 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, for example in scripts.

Chapter 28

Package API Reference

In the REPL mode, packages (with associated version, UUID, URL etc) are parsed from strings, for example `"Package#master"`,`"Package@v0.1"`, `"www.mypkg.com/MyPkg#my/feature"`.

In the API mode, it is possible to use strings as arguments for simple commands (like `Pkg.add(["PackageA", "PackageB"])`, but more complicated commands, which e.g. specify URLs or version range, require the use of a more structured format over strings. This is done by creating an instance of `PackageSpec` which is passed in to functions.

`Pkg.PackageSpec` – Function.

```
PackageSpec(name::String, [uuid::UUID, version::VersionNumber])
```

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

- The `name` of the package.
- The package's 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 `rev`ision. `rev` can be a branch name or a git commit SHA1.
- A local `path`. This is equivalent to using the `url` argument but can be more descriptive.
- A `mode`, which is an instance of the enum `PackageMode`, with possible values `PKGMODE_PROJECT`

(the default) or `PKGMODE_MANIFEST`. 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 API version:

source

`Pkg.PackageMode` – Type.

An enum with the instances

- `PKGMODE_MANIFEST`

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)

- PKGMODE_PROJECT

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.

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}}; preserve=PRESERVE_TIERED)
```

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

Resolution Tiers

Pkg resolves the set of packages in your environment using a tiered algorithm. The `preserve` keyword argument allows you to key into a specific tier in the resolve algorithm. The following table describes the argument values for `preserve` (in order of strictness):

Value	Description
PRESERVE_ALL	Preserve the state of all existing dependencies (including recursive dependencies)
PRESERVE_DIRECT	Preserve the state of all existing direct dependencies
PRESERVE_SEMVER	Preserve semver-compatible versions of direct dependencies
PRESERVE_NONE	Do not attempt to preserve any version information
PRESERVE_TIERED	Use the tier which will preserve the most version information (this is the default)

Examples

```
Pkg.add("Example") # Add a package from registry
Pkg.add("Example"; preserve=Pkg.PRESERVE_ALL) # Add the `Example` package and preserve existing
↪  dependencies
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
```

See also PackageSpec.

source

Pkg.develop – Function.

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

Make a package available for development by tracking it by path. If pkg is given with only a name or by a URL, the package will be downloaded to the location specified 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
```

See also PackageSpec

source

Pkg.activate – Function.

```

```

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.
- Otherwise, s is interpreted as a non-existing path, which is then activated.

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}})
```

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}})
```

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 by how much packages are allowed to be upgraded (major, minor, patch, fixed).

See also PackageSpec, PackageMode, UpgradeLevel.

source

Pkg.test – Function.

```
Pkg.test(; kwargs...)
Pkg.test(pkg::Union{String, Vector{String}; kwargs...)
```

Keyword arguments:

- coverage::Bool=false: enable or disable generation of coverage statistics.
- julia_args::Union{Cmd, Vector{String}}: options to be passed the test process.
- test_args::Union{Cmd, Vector{String}}: test arguments (ARGS) available in the test process.

Julia 1.3

julia_args and test_args requires at least Julia 1.3.

Run the tests for package pkg, or for the current project (which thus needs to be a package) if no positional argument is given to Pkg.test. 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 in it. If a manifest exists, the versions in that manifest are used, otherwise a feasible set of packages is resolved and installed.

During the tests, 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"]
```

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)
```

Run the build script in `deps/build.jl` for pkg and all of its 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 on 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}})
```

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")
```

source

`Pkg.free` – Function.

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

If pkg is pinned, remove the pin. If pkg is tracking a path, e.g. after `Pkg.develop`, go back to tracking registered versions.

Examples

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

source

`Pkg.instantiate` – Function.

```

```

If a `Manifest.toml` file exists in the active project, download all the packages declared in that manifest. Otherwise, resolve a set of feasible packages from the `Project.toml` files and install them. `verbose = true` prints the build output to `stdout`/`stderr` instead of redirecting to the `build.log` file. If no `Project.toml` exist in the current active project, create one with all the dependencies in the manifest and instantiate the resulting project.

source

`Pkg.resolve` – Function.

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

source

`Pkg.gc` – Function.

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.

Print out the status of the project/manifest. If `mode` is `PKGMODE_PROJECT`, print out status only about the packages that are in the project (explicitly added). If `mode` is `PKGMODE_MANIFEST`, print status also about those in the manifest (recursive dependencies). If there are any packages listed as arguments, the output will be limited to those packages. Setting `diff=true` will, if the environment is in a git repository, limit the output to the difference as compared to the last git commit.

Julia 1.1

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

Julia 1.3

The `diff` keyword argument requires Julia 1.3. In earlier versions `diff=true` is the default for environments in git repositories.

source

`Pkg.precompile` – Function.

Precompile all the dependencies of the project.

Julia 1.3

This function requires at least Julia 1.3. On earlier versions you can use `Pkg.API.precompile()` or the `precompile` Pkg REPL command.

Examples

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 (`protocol == nothing`). Other choices for `protocol` are `"https"` or `"git"`.

Examples

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

source

`Pkg.dependencies` – Function.

Julia 1.4

This feature requires Julia 1.4, and is considered experimental.

Query the dependecy graph. The result is a `Dict` that maps a package UUID to a `PackageInfo` struct representing the dependency (a package).

`PackageInfo` fields

Field	Description
name	The name of the package
version	The version of the package (this is `Nothing` for stdlibs)
isdeveloped	Whether a package is directly tracking a directory
ispinned	Whether a package is pinned
source	The directory containing the source code for that package
dependencies	The dependencies of that package as a vector of UUIDs

source

`Pkg.project` – Function.

Julia 1.4

This feature requires Julia 1.4, and is considered experimental.

Request a `ProjectInfo` struct which contains information about the active project.

`ProjectInfo` fields

Field	Description
name	The project's name
uuid	The project's UUID
version	The project's version
dependencies	The project's direct dependencies as a `Dict` which maps dependency name to dependency UUID
path	The location of the project file which defines the active project

source

`Pkg.undo` – Function.

Undoes the latest change to the active project. Only states in the current session are stored, up to a maximum of 50 states.

See also: `redo`.

source

`Pkg.redo` – Function.

Redoes the changes from the latest `undo`.

source

Chapter 29

Registry API Reference

!!! compat "Julia 1.1"
    Pkg's registry handling requires at least Julia 1.1.

The function API for registries uses `RegistrySpec`s, similar to `PackageSpec`.

`Pkg.RegistrySpec` – Type.

```
RegistrySpec(name::String)
```

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.

!!! compat "Julia 1.1"
    Pkg's registry handling requires at least Julia 1.1.

Examples

Below is a comparison between the REPL version and the API 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)
```

Add new package registries.

!!! compat "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"))
```

source

Pkg.Registry.rm – Function.

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

Remove registries.

Julia 1.1

Pkg's registry handling requires at least Julia 1.1.

Examples

```
Pkg.Registry.rm("General")
```

source

Pkg.Registry.update – Function.

```
Pkg.Registry.update()
Pkg.Registry.update(registry::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")
```

source

Pkg.Registry.status – Function.

```

```

Display information about available registries.

Julia 1.1

Pkg's registry handling requires at least Julia 1.1.

Examples

```

```

source

Chapter 30

Artifacts API Reference

Julia 1.3

Pkg's artifacts API requires at least Julia 1.3.

`Pkg.Artifacts.create_artifact` – Function.

Creates a new artifact by running `f(artifact_path)`, hashing the result, and moving it to the artifact store (`~/.julia/artifacts` on a typical installation). Returns the identifying tree hash of this artifact.

source

`Pkg.Artifacts.artifact_exists` – Function.

Returns whether or not the given artifact (identified by its sha1 git tree hash) exists on-disk. Note that it is possible that the given artifact exists in multiple locations (e.g. within multiple depots).

source

`Pkg.Artifacts.artifact_path` – Function.

Given an artifact (identified by SHA1 git tree hash), return its installation path. If the artifact does not exist, returns the location it would be installed to.

source

`Pkg.Artifacts.remove_artifact` – Function.

Removes the given artifact (identified by its SHA1 git tree hash) from disk. Note that if an artifact is installed in multiple depots, it will be removed from all of them. If an overridden artifact is requested for removal, it will be silently ignored; this method will never attempt to remove an overridden artifact.

In general, we recommend that you use `Pkg.gc()` to manage artifact installations and do not use `remove_artifact()` directly, as it can be difficult to know if an artifact is being used by another package.

source

Pkg.Artifacts.verify_artifact – Function.

Verifies that the given artifact (identified by its SHA1 git tree hash) is installed on- disk, and retains its integrity. If the given artifact is overridden, skips the verification unless `honor_overrides` is set to `true`.

source

Pkg.Artifacts.artifact_meta – Function.

```
artifact_meta(name::String, artifacts_toml::String;
              platform::Platform = platform_key_abi(),
```

Get metadata about a given artifact (identified by name) stored within the given `(Julia)Artifacts.toml` file. If the artifact is platform-specific, use `platform` to choose the most appropriate mapping. If none is found, return `nothing`.

source

Pkg.Artifacts.artifact_hash – Function.

Thin wrapper around `artifact_meta()` to return the hash of the specified, platform- collapsed artifact. Returns `nothing` if no mapping can be found.

source

Pkg.Artifacts.bind_artifact! – Function.

```
bind_artifact!(artifacts_toml::String, name::String, hash::SHA1;
               platform::Union{Platform,Nothing} = nothing,
               download_info::Union{Vector{Tuple},Nothing} = nothing,
               lazy::Bool = false,
```

Writes a mapping of `name -> hash` within the given `(Julia)Artifacts.toml` file. If `platform` is not `nothing`, this artifact is marked as platform-specific, and will be a multi-mapping. It is valid to bind multiple artifacts with the same name, but different `platforms` and `hash`'es within the same `artifacts_toml`. If `force` is set to `true`, this will overwrite a pre-existant mapping, otherwise an error is raised.

`download_info` is an optional tuple that contains a vector of URLs and a hash. These URLs will be listed as possible locations where this artifact can be obtained. If `lazy` is set to `true`, even if download information is available, this artifact will not be downloaded until it is accessed via the `artifact"name"` syntax, or `ensure_artifact_installed()` is called upon it.

source

Pkg.Artifacts.unbind_artifact! – Function.

Unbind the given `name` from an `(Julia)Artifacts.toml` file. Silently fails if no such binding exists within the file.

source

Pkg.Artifacts.download_artifact – Function.

```
download_artifact(tree_hash::SHA1, tarball_url::String, tarball_hash::String;
```

Download/install an artifact into the artifact store. Returns `true` on success.

source

`Pkg.Artifacts.find_artifacts_toml` – Function.

```
```

Given the path to a `.jl` file, (such as the one returned by `__source__.file` in a macro context), find the `(Julia)Artifacts.toml` that is contained within the containing project (if it exists), otherwise return `nothing`.

source

`Pkg.Artifacts.ensure_artifact_installed` – Function.

```
ensure_artifact_installed(name::String, artifacts_toml::String;
                          platform::Platform = platform_key_abi(),
```

Ensures an artifact is installed, downloading it via the download information stored in `artifacts_toml` if necessary. Throws an error if unable to install.

source

`Pkg.Artifacts.ensure_all_artifacts_installed` – Function.

```
ensure_all_artifacts_installed(artifacts_toml::String;
                               platform = platform_key_abi(),
                               pkg_uuid = nothing,
                               include_lazy = false,
```

Installs all non-lazy artifacts from a given `(Julia)Artifacts.toml` file. `package_uuid` must be provided to properly support overrides from `Overrides.toml` entries in depots.

If `include_lazy` is set to `true`, then lazy packages will be installed as well.

source

`Pkg.Artifacts.@artifact_str` – Macro.

```
```

Macro that is used to automatically ensure an artifact is installed, and return its location on-disk. Automatically looks the artifact up by name in the project's `(Julia)Artifacts.toml` file. Throws an error on inability to install the requested artifact.

source

`Pkg.Artifacts.archive_artifact` – Function.

```
```

Archive an artifact into a tarball stored at `tarball_path`, returns the SHA256 of the resultant tarball as a hexidecimal string. Throws an error if the artifact does not exist. If the artifact is overridden, throws an error unless `honor_overrides` is set.

source