Pkg.jl

November 1, 2025

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.

> **Note**
>
> The documentation is also available in PDF format: Pkg.jl.pdf.

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

Chapter 1

Background and Design

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 from the REPL 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.

Part II

2. Getting Started

What follows is a quick overview of the basic features of Pkg. It should help new users become familiar with basic Pkg features such as adding and removing packages and working with environments.

> **Note**
>
> Some Pkg output is omitted in this section in order to keep this basic 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.

> **Note**
>
> This guide uses 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.

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 `Ctrl+C` or backspace (when the REPL cursor is at the beginning of the input).

Upon entering the Pkg REPL, you should see the following prompt:

```
(@v1.10) pkg>
```

To add a package, use add:

```
(@v1.10) pkg> add Example
   Resolving package versions...
   Installed Example ─ v0.5.3
    Updating `~/.julia/environments/v1.10/Project.toml`
  [7876af07] + Example v0.5.3
    Updating `~/.julia/environments/v1.10/Manifest.toml`
  [7876af07] + Example v0.5.3
```

After the package is installed, it can be loaded into the Julia session:

```
julia> import Example

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

We can also specify multiple packages at once to install:

```
(@v1.10) pkg> add JSON StaticArrays
```

The `status` command (or the shorter `st` command) can be used to see installed packages.

```
(@v1.10) pkg> st
Status `~/.julia/environments/v1.10/Project.toml`
  [7876af07] Example v0.5.3
  [682c06a0] JSON v0.21.3
  [90137ffa] StaticArrays v1.5.9
```

Note

Some Pkg REPL commands have a short and a long version of the command, for example `status` and `st`.

To remove packages, use `rm` (or `remove`):

```
(@v1.10) pkg> rm JSON StaticArrays
```

Use `up` (or `update`) to update the installed packages

```
(@v1.10) pkg> up
```

If you have been following this guide it is likely that the packages installed are at the latest version so `up` will not do anything. Below we show the status output in the case where we deliberately have installed an old version of the Example package and then upgrade it:

```
(@v1.10) pkg> st
Status `~/.julia/environments/v1.10/Project.toml`
^ [7876af07] Example v0.5.1
Info Packages marked with ^ have new versions available and may be upgradable.

(@v1.10) pkg> up
    Updating `~/.julia/environments/v1.10/Project.toml`
  [7876af07] ↑ Example v0.5.1 ⇒ v0.5.3
```

We can see that the status output tells us that there is a newer version available and that `up` upgrades the package.

For more information about managing packages, see the Managing Packages section of the documentation.

Chapter 3

Getting Started with Environments

Up to this point, we have covered basic package management: adding, updating, and removing packages.

You may have noticed the `(@v1.10)` in the REPL prompt. This lets us know that `v1.10` is the **active environment**. Different environments can have totally different packages and versions installed from another 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`:

```
(@v1.10) pkg> activate tutorial
[ Info: activating new environment at `~/tutorial/Project.toml`.
```

Pkg lets us know we are creating a new environment and that this environment will be stored in the `~/tutorial` directory. The path to the environment is created relative to the current working directory of the REPL.

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 `~/tutorial/Project.toml`
   (empty environment)
```

`~/tutorial/Project.toml` is the location of the active environment's **project file**. A project file is a TOML file where Pkg stores the packages that have been explicitly installed. Notice this new environment is empty. Let us add some packages and observe:

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

(tutorial) pkg> status
    Status `~/tutorial/Project.toml`
  [7876af07] Example v0.5.3
  [682c06a0] JSON v0.21.3
```

We can see that the `tutorial` environment now contains `Example` and `JSON`.

> **Note**
>
> If you have the same package (at the same version) installed in multiple environments, the package will only be downloaded and stored on the hard drive once. This makes environments very lightweight and effectively free to create. Using only the default environment with a huge number of packages in it is a common beginners mistake in Julia. Learning how to use environments effectively will improve your experience with Julia packages.

When you're done working in a specific environment and want to return to the default environment, use `activate` with no arguments:

```
(tutorial) pkg> activate
  Activating project at `~/.julia/environments/v1.10`

(@v1.10) pkg>
```

This returns you to the default `@v1.10` environment. There is no separate "deactivate" command—`activate` with no arguments serves this purpose.

For more information about environments, see the Working with Environments section of the documentation.

Chapter 4

Asking for Help

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

```
(@v1.10) pkg> ?
```

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

```
(@v1.10) pkg> ?develop
```

This guide should help you get started with Pkg. Pkg has much more to offer in terms of powerful package management. For more advanced topics, see Managing Packages, Working with Environments, and Creating Packages.

Part III

3. Managing Packages

Chapter 5

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.

5.1 Adding registered packages

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

```
(@v1.13) pkg> add JSON
   Resolving package versions...
    Updating `~/.julia/environments/v1.13/Project.toml`
  [682c06a0] + JSON v0.21.4
    Updating `~/.julia/environments/v1.13/Manifest.toml`
  [682c06a0] + JSON v0.21.4
  [69de0a69] + Parsers v2.8.3
  [aea7be01] + PrecompileTools v1.3.2
  [21216c6a] + Preferences v1.5.0
  [ade2ca70] + Dates v1.11.0
  [a63ad114] + Mmap v1.11.0
  [de0858da] + Printf v1.11.0
  [9a3f8284] + Random v1.11.0
  [ea8e919c] + SHA v0.7.0
  [fa267f1f] + TOML v1.0.3
  [cf7118a7] + UUIDs v1.11.0
  [4ec0a83e] + Unicode v1.11.0
```

Here we added the package `JSON` to the current environment (which is the default `@v1.10` environment). In this example, we are using a fresh Julia installation, and this is our first time adding a package using Pkg. By default, Pkg installs the 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. Finally, the newly installed packages are "precompiled".

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

The status output contains the packages you have added yourself, in this case, `JSON`:

```
(@v1.13) pkg> st
Status `~/.julia/environments/v1.13/Project.toml`
  [682c06a0] JSON v0.21.4
```

The manifest status shows all the packages in the environment, including recursive dependencies:

```
(@v1.13) pkg> st -m
Status `~/.julia/environments/v1.13/Manifest.toml`
  [682c06a0] JSON v0.21.4
  [69de0a69] Parsers v2.8.3
  [aea7be01] PrecompileTools v1.3.2
  [21216c6a] Preferences v1.5.0
  [ade2ca70] Dates v1.11.0
  [a63ad114] Mmap v1.11.0
  [de0858da] Printf v1.11.0
  [9a3f8284] Random v1.11.0
  [ea8e919c] SHA v0.7.0
  [fa267f1f] TOML v1.0.3
  [cf7118a7] UUIDs v1.11.0
  [4ec0a83e] Unicode v1.11.0
```

Note that before 1.11 standard libraries (e.g. `Dates`) did not have dedicated version numbers.

To specify that you want a particular version (or set of versions) of a package, use the `compat` command. For example, to require any patch release of the v0.21 series of JSON after v0.21.4, call `compat JSON 0.21.4`:

```
(@v1.10) pkg> compat JSON 0.21.4
      Compat entry set:
  JSON = "0.21.4"
     Resolve checking for compliance with the new compat rules...
       Error empty intersection between JSON@0.21.3 and project compatibility 0.21.4 - 0.21
  Suggestion Call `update` to attempt to meet the compatibility requirements.

(@v1.10) pkg> update
    Updating registry at `~/.julia/registries/General.toml`
    Updating `~/.julia/environments/v1.10/Project.toml`
  [682c06a0] ↑ JSON v0.21.3 ⇒ v0.21.4
    Updating `~/.julia/environments/v1.10/Manifest.toml`
  [682c06a0] ↑ JSON v0.21.3 ⇒ v0.21.4
```

See the section on Compatibility for more on using the compat system.

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

```
julia> using JSON

julia> JSON.json(Dict("foo" => [1, "bar"])) |> print
{"foo":[1,"bar"]}
```

Note

Only packages that have been added with add can be loaded (which are packages that are shown when using `st` in the Pkg REPL). Packages that are pulled in only as dependencies (for example the `Parsers` package above) can not be loaded.

A specific version of a package can be installed by appending a version after a @ symbol to the package name:

```
(@v1.10) pkg> add JSON@0.21.1
   Resolving package versions...
    Updating `~/.julia/environments/v1.10/Project.toml`
^ [682c06a0] + JSON v0.21.1
    Updating `~/.julia/environments/v1.10/Manifest.toml`
^ [682c06a0] + JSON v0.21.1
⌅ [69de0a69] + Parsers v1.1.2
  [ade2ca70] + Dates
  [a63ad114] + Mmap
  [de0858da] + Printf
  [4ec0a83e] + Unicode
        Info Packages marked with ^ and ⌅ have new versions available, but those with ⌅ are
↪  restricted by compatibility constraints from upgrading. To see why use `status --outdated -m`
```

As seen above, Pkg gives some information when a package is not installed at its latest version.

If not all three numbers are given for the version, for example, `0.21`, then the latest registered version of `0.21.x` would be installed.

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:

```
(@v1.10) pkg> add Example#master
     Cloning git-repo `https://github.com/JuliaLang/Example.jl.git`
   Resolving package versions...
    Updating `~/.julia/environments/v1.10/Project.toml`
  [7876af07] + Example v0.5.4 `https://github.com/JuliaLang/Example.jl.git#master`
    Updating `~/.julia/environments/v1.10/Manifest.toml`
  [7876af07] + Example v0.5.4 `https://github.com/JuliaLang/Example.jl.git#master`
```

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

> **Note**
>
> If we would specify a commit id instead of a branch name, e.g. `add Example#025cf7e`, then we would effectively "pin" the package to that commit. This is because the commit id always points to the same thing unlike a branch, which may be updated.

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

```
(@v1.10) pkg> free Example
   Resolving package versions...
   Installed Example ─ v0.5.3
    Updating `~/.julia/environments/v1.10/Project.toml`
  [7876af07] ~ Example v0.5.4 `https://github.com/JuliaLang/Example.jl.git#master` ⇒ v0.5.3
    Updating `~/.julia/environments/v1.10/Manifest.toml`
  [7876af07] ~ Example v0.5.4 `https://github.com/JuliaLang/Example.jl.git#master` ⇒ v0.5.3
```

5.2 Adding unregistered packages

If a package is not in a registry, it can be added by specifying a URL to the Git repository:

```
(@v1.10) pkg> add https://github.com/fredrikekre/ImportMacros.jl
     Cloning git-repo `https://github.com/fredrikekre/ImportMacros.jl`
   Resolving package versions...
    Updating `~/.julia/environments/v1.10/Project.toml`
  [92a963f6] + ImportMacros v1.0.0 `https://github.com/fredrikekre/ImportMacros.jl#master`
    Updating `~/.julia/environments/v1.10/Manifest.toml`
  [92a963f6] + ImportMacros v1.0.0 `https://github.com/fredrikekre/ImportMacros.jl#master`
```

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.

If you want to add a package using the SSH-based `git` protocol, you have to use quotes because the URL contains a `@`. For example,

```
(@v1.10) pkg> add "git@github.com:fredrikekre/ImportMacros.jl.git"
    Cloning git-repo `git@github.com:fredrikekre/ImportMacros.jl.git`
   Updating registry at `~/.julia/registries/General`
  Resolving package versions...
Updating `~/.julia/environments/v1/Project.toml`
  [92a963f6] + ImportMacros v1.0.0 `git@github.com:fredrikekre/ImportMacros.jl.git#master`
Updating `~/.julia/environments/v1/Manifest.toml`
  [92a963f6] + ImportMacros v1.0.0 `git@github.com:fredrikekre/ImportMacros.jl.git#master`
```

Adding a package in a subdirectory of a repository

If the package you want to add by URL is not in the root of the repository, then you need pass that subdirectory using `:`. For instance, to add the `SnoopCompileCore` package in the SnoopCompile repository:

```
pkg> add https://github.com/timholy/SnoopCompile.jl.git:SnoopCompileCore
    Cloning git-repo `https://github.com/timholy/SnoopCompile.jl.git`
   Resolving package versions...
    Updating `~/.julia/environments/v1.10/Project.toml`
  [e2b509da] + SnoopCompileCore v2.9.0
↪  `https://github.com/timholy/SnoopCompile.jl.git:SnoopCompileCore#master`
    Updating `~/.julia/environments/v1.8/Manifest.toml`
  [e2b509da] + SnoopCompileCore v2.9.0
↪  `https://github.com/timholy/SnoopCompile.jl.git:SnoopCompileCore#master`
  [9e88b42a] + Serialization
```

5.3 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.

> **Warning**
>
> Note that tracking a package through add is distinct from `develop` (which is described in the next section). When using add on a local git repository, 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 the majority of cases, you want to use `develop` on a local path, **not add**.

5.4 Developing packages

By only using add your environment always has 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 environment. This has the advantage that you can send your environment (`Project.toml` and `Manifest.toml`) to someone else and they can `Pkg.instantiate` that environment 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:

```
(@v1.10) pkg> dev Example
  Updating git-repo `https://github.com/JuliaLang/Example.jl.git`
   Resolving package versions...
    Updating `~/.julia/environments/v1.10/Project.toml`
  [7876af07] + Example v0.5.4 `~/.julia/dev/Example`
    Updating `~/.julia/environments/v1.8/Manifest.toml`
  [7876af07] + Example v0.5.4 `~/.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`, the default being `joinpath(DEPOT_PATH[1],"dev")`). 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 will simply re-use the existing 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.

Let's try modify the file at `~/.julia/dev/Example/src/Example.jl` and add a simple function:

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

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

```julia
julia> import Example
[ Info: Precompiling Example [7876af07-990d-54b4-ab0e-23690620f79a]

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

Warning

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 to see the changes to Example. Revise.jl can make this process significantly more pleasant, but setting it up is beyond the scope of this guide.

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

```
(@v1.10) pkg> free Example
   Resolving package versions...
    Updating `~/.julia/environments/v1.10/Project.toml`
  [7876af07] ~ Example v0.5.4 `~/.julia/dev/Example` ⇒ v0.5.3
    Updating `~/.julia/environments/v1.8/Manifest.toml`
  [7876af07] ~ Example v0.5.4 `~/.julia/dev/Example` ⇒ v0.5.3
```

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 6

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 (recursively) depend on `DepPackage`.

Chapter 7

Updating packages

When new versions of packages 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:

```
(@v1.10) pkg> up Example
```

This will only allow Example do upgrade. If you also want to allow dependencies of Example to upgrade (with the exception of packages that are in the project) you can pass the `--preserve=direct` flag.

```
(@v1.10) pkg> up --preserve=direct Example
```

And if you also want to allow dependencies of Example that are also in the project to upgrade, you can use `--preserve=none`:

```
(@v1.10) pkg> up --preserve=none Example
```

Chapter 8

Pinning a package

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

```
(@v1.10) pkg> pin Example
 Resolving package versions...
  Updating `~/.julia/environments/v1.10/Project.toml`
  [7876af07] ~ Example v0.5.3 ⇒ v0.5.3
  Updating `~/.julia/environments/v1.8/Manifest.toml`
  [7876af07] ~ Example v0.5.3 ⇒ v0.5.3
```

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

```
(@v1.10) pkg> free Example
  Updating `~/.julia/environments/v1.10/Project.toml`
  [7876af07] ~ Example v0.5.3  ⇒ v0.5.3
  Updating `~/.julia/environments/v1.8/Manifest.toml`
  [7876af07] ~ Example v0.5.3  ⇒ v0.5.3
```

Chapter 9

Testing packages

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

```
(@v1.10) pkg> test Example
...
   Testing Example
   Testing Example tests passed
```

Chapter 10

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:

```
(@v1.10) pkg> build IJulia
    Building Conda ─→ `~/.julia/scratchspaces/44cfe95a-1eb2-52ea-b672-
↪  e2afdf69b78f/6e47d11ea2776bc5627421d59cdcc1296c058071/build.log`
    Building IJulia → `~/.julia/scratchspaces/44cfe95a-1eb2-52ea-b672-
↪  e2afdf69b78f/98ab633acb0fe071b671f6c1785c46cd70bb86bd/build.log`

julia> print(read(joinpath(homedir(), ".julia/scratchspaces/44cfe95a-1eb2-52ea-b672-
↪  e2afdf69b78f/98ab633acb0fe071b671f6c1785c46cd70bb86bd/build.log"),
↪  String))
[ Info: Installing Julia kernelspec in /home/kc/.local/share/jupyter/kernels/julia-1.8
```

Chapter 11

Interpreting and resolving version conflicts

An environment consists of a set of mutually-compatible packages. Sometimes, you can find yourself in a situation in which two packages you'd like to use simultaneously have incompatible requirements. In such cases you'll get an "Unsatisfiable requirements" error:

```
pkg> add A
Unsatisfiable requirements detected for package C [c99a7cb2]:
 C [c99a7cb2] log:
 ├─possible versions are: 0.1.0 - 0.2.0 or uninstalled
 ├─restricted by compatibility requirements with A [29c70717] to versions: 0.2.0
 │ └─A [29c70717] log:
 │   ├─possible versions are: 1.0.0 or uninstalled
 │   └─restricted to versions * by an explicit requirement, leaving only versions: 1.0.0
 └─restricted by compatibility requirements with D [756980fe] to versions: 0.1.0 - 0.1.1 or
 ↪  uninstalled — no versions left
   └─D [756980fe] log:
     ├─possible versions are: 0.1.0 - 0.2.1 or uninstalled
     └─restricted by compatibility requirements with B [f4259836] to versions: 0.1.0
       └─B [f4259836] log:
         ├─possible versions are: 1.0.0 or uninstalled
         └─restricted to versions * by an explicit requirement, leaving only versions: 1.0.0
```

This message means that a package named `D` has a version conflict. Even if you have never added `D` directly, this kind of error can arise if `D` is required by other packages that you are trying to use.

> **Note**
>
> When tackling these conflicts, first consider that the bigger a project gets, the more likely this is to happen. Using targeted projects for a given task is highly recommended, and removing unused dependencies is a good first step when hitting these issues. For instance, a common pitfall is having more than a few packages in your default (i.e. `(@1.8)`) environment, and using that as an environment for all tasks you're using julia for. It's better to create a dedicated project for the task you're working on, and keep the dependencies there minimal. To read more see Working with Environments

The error message has a lot of crucial information. It may be easiest to interpret piecewise:

```
Unsatisfiable requirements detected for package D [756980fe]:
 D [756980fe] log:
 ├─possible versions are: [0.1.0, 0.2.0-0.2.1] or uninstalled
```

means that D has three released versions, v0.1.0, v0.2.0, and v0.2.1. You also have the option of not having it installed at all. Each of these options might have different implications for the set of other packages that can be installed.

Crucially, notice the stroke characters (vertical and horizontal lines) and their indentation. Together, these connect messages to specific packages. For instance the right stroke of ├─ indicates that the message to its right (possible versions...) is connected to the package pointed to by its vertical stroke (D). This same principle applies to the next line:

```
 ├─restricted by compatibility requirements with B [f4259836] to versions: 0.1.0
```

The vertical stroke here is also aligned under D, and thus this message is in reference to D. Specifically, there's some other package B that depends on version v0.1.0 of D. Notice that this is not the newest version of D.

Next comes some information about B:

```
 │ └─B [f4259836] log:
 │   ├─possible versions are: 1.0.0 or uninstalled
 │   └─restricted to versions * by an explicit requirement, leaving only versions 1.0.0
```

The two lines below the first have a vertical stroke that aligns with B, and thus they provide information about B. They tell you that B has just one release, v1.0.0. You've not specified a particular version of B (restricted to versions * means that any version will do), but the explicit requirement means that you've asked for B to be part of your environment, for example by pkg> add B. You might have asked for B previously, and the requirement is still active.

The conflict becomes clear with the line

```
└─restricted by compatibility requirements with C [c99a7cb2] to versions: 0.2.0 — no versions left
```

Here again, the vertical stroke aligns with D: this means that D is also required by another package, C. C requires v0.2.0 of D, and this conflicts with B's need for v0.1.0 of D. This explains the conflict.

But wait, you might ask, what is C and why do I need it at all? The next few lines introduce the problem:

```
   └─C [c99a7cb2] log:
     ├─possible versions are: [0.1.0-0.1.1, 0.2.0] or uninstalled
     └─restricted by compatibility requirements with A [29c70717] to versions: 0.2.0
```

These provide more information about C, revealing that it has 3 released versions: v0.1.0, v0.1.1, and v0.2.0. Moreover, C is required by another package A. Indeed, A's requirements are such that we need v0.2.0 of C. A's origin is revealed on the next lines:

```
       └─A [29c70717] log:
         ├─possible versions are: 1.0.0 or uninstalled
         └─restricted to versions * by an explicit requirement, leaving only versions 1.0.0
```

So we can see that `A` was explicitly required, and in this case, it's because we were trying to add it to our environment.

In summary, we explicitly asked to use `A` and `B`, but this gave a conflict for `D`. The reason was that `B` and `C` require conflicting versions of `D`. Even though `C` isn't something we asked for explicitly, it was needed by `A`.

To fix such errors, you have a number of options:

- try updating your packages. It's possible the developers of these packages have recently released new versions that are mutually compatible.
- remove either `A` or `B` from your environment. Perhaps `B` is left over from something you were previously working on, and you don't need it anymore. If you don't need `A` and `B` at the same time, this is the easiest way to fix the problem.
- try reporting your conflict. In this case, we were able to deduce that `B` requires an outdated version of `D`. You could thus report an issue in the development repository of `B.jl` asking for an updated version.
- try fixing the problem yourself. This becomes easier once you understand `Project.toml` files and how they declare their compatibility requirements. We'll return to this example in Fixing conflicts.

Chapter 12

Yanked packages

Package registries can mark specific versions of packages as "yanked". A yanked package version is one that should no longer be used, typically because it contains serious bugs, security vulnerabilities, or other critical issues. When a package version is yanked, it becomes unavailable for new installations but remains accessible (i.e. via `instantiate`) to maintain reproducibility of existing environments.

When you run `pkg> status`, yanked packages are clearly marked with a warning symbol:

```
(@v1.13) pkg> status
    Status `~/.julia/environments/v1.13/Project.toml`
  [682c06a0] JSON v0.21.3
  [f4259836] Example v1.2.0 [yanked]
```

The `[yanked]` annotation indicate that version `v1.2.0` of the `Example` package has been yanked and should be updated or replaced.

When resolving dependencies, Pkg will warn you if yanked packages are present and may provide guidance on how to resolve the situation. It's important to address yanked packages promptly to ensure the security and stability of your Julia environment.

Chapter 13

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 to be collected 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:

```
(@v1.10) 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.

Chapter 14

Offline Mode

In offline mode, Pkg tries to do as much as possible without connecting to internet. For example, when adding a package Pkg only considers versions that are already downloaded in version resolution.

To work in offline mode use `import Pkg; Pkg.offline(true)` or set the environment variable `JULIA_PKG_OFFLINE` to `"true"`.

Chapter 15

Pkg client/server

When you add a new registered package, usually three things would happen:

1. update registries,
2. download the source code of the package,
3. if not available, download artifacts required by the package.

The General registry and most packages in it are developed on GitHub, while the artifacts data are hosted on various platforms. When the network connection to GitHub and AWS S3 is not stable, it is usually not a good experience to install or update packages. Fortunately, the pkg client/server feature improves the experience in the sense that:

1. If set, the pkg client would first try to download data from the pkg server,
2. if that fails, then it falls back to downloading from the original sources (e.g., GitHub).

By default, the client makes upto 8 concurrent requests to the server. This can set by the environment variable `JULIA_PKG_CONCURRENT_DOWNLOADS`.

Since Julia 1.5, `https://pkg.julialang.org` provided by the JuliaLang organization is used as the default pkg server. In most cases, this should be transparent, but users can still set/unset a pkg server upstream via the environment variable `JULIA_PKG_SERVER`.

```
# manually set it to some pkg server
julia> ENV["JULIA_PKG_SERVER"] = "pkg.julialang.org"
"pkg.julialang.org"

# unset to always download data from original sources
julia> ENV["JULIA_PKG_SERVER"] = ""
""
```

For clarification, some sources are not provided by Pkg server

- packages/registries fetched via `git`
  - `]add https://github.com/JuliaLang/Example.jl.git`

  - `]add Example#v0.5.3` (Note that this is different from `]add Example@0.5.3`)
  - `]registry add https://github.com/JuliaRegistries/General.git`, including registries installed by Julia before 1.4.

- artifacts without download info
  - TestImages

> **Note**
>
> If you have a new registry installed via pkg server, then it's impossible for old Julia versions to update the registry because Julia before 1.4 doesn't know how to fetch new data. Hence, for users that frequently switch between multiple Julia versions, it is recommended to still use git-controlled registries.

For the deployment of pkg server, please refer to PkgServer.jl.

Part IV

4. Working with Environments

The following discusses Pkg's interaction with environments. For more on the role, environments play in code loading, including the "stack" of environments from which code can be loaded, see this section in the Julia manual.

Chapter 16

Creating your own environments

So far we have added packages to the default environment at `~/.julia/environments/v1.10`. It is however easy to create other, independent, projects. This approach has the benefit of allowing you to check in a `Project.toml`, and even a `Manifest.toml` if you wish, into version control (e.g. git) alongside your code. 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:

```
(@v1.10) pkg> activate MyProject
Activating new environment at `~/MyProject/Project.toml`

(MyProject) pkg> st
    Status `~/MyProject/Project.toml` (empty project)
```

Note that the REPL prompt changes when the new project is activated. Until a package is added, there are no files in this environment and the directory to the environment might not even be created:

```
julia> isdir("MyProject")
false

(MyProject) pkg> add Example
   Resolving package versions...
   Installed Example ─ v0.5.3
    Updating `~/MyProject/Project.toml`
  [7876af07] + Example v0.5.3
    Updating `~/MyProject/Manifest.toml`
  [7876af07] + Example v0.5.3
Precompiling environment...
  1 dependency successfully precompiled in 2 seconds

julia> readdir("MyProject")
2-element Vector{String}:
 "Manifest.toml"
 "Project.toml"

julia> print(read(joinpath("MyProject", "Project.toml"), String))
[deps]
Example = "7876af07-990d-54b4-ab0e-23690620f79a"
```

```
julia> print(read(joinpath("MyProject", "Manifest.toml"), String))
# This file is machine-generated - editing it directly is not advised

julia_version = "1.10.0"
manifest_format = "2.0"
project_hash = "2ca1c6c58cb30e79e021fb54e5626c96d05d5fdc"

[[deps.Example]]
git-tree-sha1 = "46e44e869b4d90b96bd8ed1fdcf32244fddfb6cc"
uuid = "7876af07-990d-54b4-ab0e-23690620f79a"
version = "0.5.3"
```

This new environment is completely separate from the one we used earlier. See `Project.toml` and `Manifest.toml` for a more detailed explanation.

Chapter 17

Using someone else's project

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

```
shell> git clone https://github.com/JuliaLang/Example.jl.git
Cloning into 'Example.jl'...
...

(@v1.10) pkg> activate Example.jl
Activating project at `~/Example.jl`

(Example) pkg> instantiate
  No packages added to or removed from `~/Example.jl/Project.toml`
  No packages added to or removed from `~/Example.jl/Manifest.toml`
```

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.

Note that `activate` by itself does not install missing dependencies. If you only have a `Project.toml`, a `Manifest.toml` must be generated by "resolving" the environment, then any missing packages must be installed and precompiled. `instantiate` does all this for you.

If you already have a resolved `Manifest.toml`, then you will still need to ensure that the packages are installed and with the correct versions. Again `instantiate` does this for you.

In short, `instantiate` is your friend to make sure an environment is ready to use. If there's nothing to do, `instantiate` does nothing.

Chapter 18

Returning to the default environment

To return to the default environment after working in a project environment, simply call `activate` with no arguments:

```
(MyProject) pkg> activate
  Activating project at `~/.julia/environments/v1.10`

(@v1.10) pkg>
```

This deactivates the current project and returns you to the default shared environment (typically `@v#.#`). There is no separate "deactivate" command—calling `activate()` with no arguments is how you return to your base package setup. This only affects the current Julia session; the change does not persist when you restart Julia.

Specifying project on startup

Instead of using `activate` from within Julia, you can specify the project on startup using the `--project=<path>` flag. For example, to run a script from the command line using the environment in the current directory you can run

```
$ julia --project=. myscript.jl
```

Chapter 19

Temporary environments

Temporary environments make it easy to start an environment from a blank slate to test a package or set of packages, and have Pkg automatically delete the environment when you're done. For instance, when writing a bug report, you may want to test your minimal reproducible example in a 'clean' environment to ensure it's actually reproducible as written. You might also want a scratch space to try out a new package, or a sandbox to resolve version conflicts between several incompatible packages.

```
(@v1.10) pkg> activate --temp # requires Julia 1.5 or later
  Activating new environment at
↪  `/var/folders/34/km3mmt5930gc4pzq1d08jvjw0000gn/T/jl_a31egx/Project.toml`

(jl_a31egx) pkg> add Example
    Updating registry at `~/.julia/registries/General`
   Resolving package versions...
    Updating `/private/var/folders/34/km3mmt5930gc4pzq1d08jvjw0000gn/T/jl_a31egx/Project.toml`
  [7876af07] + Example v0.5.3
    Updating `/private/var/folders/34/km3mmt5930gc4pzq1d08jvjw0000gn/T/jl_a31egx/Manifest.toml`
  [7876af07] + Example v0.5.3
```

Chapter 20

Shared environments

A "shared" environment is simply an environment that exists in `~/.julia/environments`. The default `v1.10` environment is therefore a shared environment:

```
(@v1.10) pkg> st
Status `~/.julia/environments/v1.10/Project.toml`
```

Shared environments can be activated with the `--shared` flag to `activate`:

```
(@v1.10) pkg> activate --shared mysharedenv
  Activating project at `~/.julia/environments/mysharedenv`

(@mysharedenv) pkg>
```

Shared environments have a `@` before their name in the Pkg REPL prompt.

Chapter 21

Environment Precompilation

Before a package can be imported, Julia will "precompile" the source code into an intermediate more efficient cache on disc. This precompilation can be triggered via code loading if the un-imported package is new or has changed since the last cache

```
julia> using Example
[ Info: Precompiling Example [7876af07-990d-54b4-ab0e-23690620f79a]
```

or using Pkg's precompile option, which can precompile the entire environment, or a given dependency, and do so in parallel, which can be significantly faster than the code-load route above.

```
(@v1.10) pkg> precompile
Precompiling environment...
  23 dependencies successfully precompiled in 36 seconds
```

However, neither of these should be routinely required thanks to Pkg's automatic precompilation.

21.1 Automatic Precompilation

By default, any package that is added to a project or updated in a Pkg action will be automatically precompiled, along with its dependencies.

```
(@v1.10) pkg> add Images
   Resolving package versions...
    Updating `~/.julia/environments/v1.10/Project.toml`
  [916415d5] + Images v0.25.2
    Updating `~/.julia/environments/v1.10/Manifest.toml`
    ...
Precompiling environment...
  Progress [===================>                     ]  45/97
  ✓ NaNMath
  ✓ IntervalSets
  ◐ CoordinateTransformations
  ◑ ArnoldiMethod
  ◑ IntegralArrays
  ◒ RegionTrees
  ◐ ChangesOfVariables
  ◓ PaddedViews
```

The exception is the `develop` command, which neither builds nor precompiles the package. When that happens is left up to the user to decide.

If a given package version errors during auto-precompilation, Pkg will remember for the following times it automatically tries and will skip that package with a brief warning. Manual precompilation can be used to force these packages to be retried, as `pkg> precompile` will always retry all packages.

The indicators next to the package names displayed during precompilation indicate the status of that package's precompilation.

- `[◐, ◓, ◑, ◒]` Animated "clock" characters indicate that the package is currently being precompiled.
- ✓ A green checkmark indicates that the package has been successfully precompiled (after which that package will disappear from the list). If the checkmark is yellow it means that the package is currently loaded so the session will need to be restarted to access the version that was just precompiled.
- ? A question mark character indicates that a `PrecompilableError` was thrown, indicating that precompilation was disallowed, i.e. `__precompile__(false)` in that package.
- ✗ A cross indicates that the package failed to precompile.

Controlling Auto-precompilation

Auto-precompilation can be controlled in several ways:

- **Environment variable**: Set `ENV["JULIA_PKG_PRECOMPILE_AUTO"]=0` to disable auto-precompilation globally.
- **Programmatically**: Use `Pkg.autoprecompilation_enabled(false)` to disable auto-precompilation for the current session, or `Pkg.autoprecompilation_enabled(true)` to re-enable it.
- **Scoped control**: Use `Pkg.precompile(f, args...; kwargs...)` to execute a function `f` with auto-precompilation temporarily disabled, then automatically trigger precompilation afterward if any packages were modified during the execution.

Julia 1.13

The `Pkg.autoprecompilation_enabled()` function and `Pkg.precompile()` do-block syntax require at least Julia 1.13.

For example, to add multiple packages without triggering precompilation after each one:

```
julia> Pkg.precompile() do
           Pkg.add("Example")
           Pkg.dev("JSON")
           Pkg.update("HTTP")
       end
   Resolving package versions...
   ...
Precompiling environment...
  14 dependencies successfully precompiled in 25 seconds
```

Or to temporarily disable auto-precompilation:

```
julia> Pkg.autoprecompilation_enabled(false)
false

julia> Pkg.add("Example")  # No precompilation happens
   Resolving package versions...
   ...

julia> Pkg.autoprecompilation_enabled(true)
true
```

21.2 Precompiling new versions of loaded packages

If a package that has been updated is already loaded in the session, the precompilation process will go ahead and precompile the new version, and any packages that depend on it, but will note that the package cannot be used until session restart.

Part V

5. Creating Packages

Chapter 22

Generating files for a package

Note

The PkgTemplates package offers an easy, repeatable, and customizable way to generate the files for a new package. It can also generate files needed for Documentation, CI, etc. We recommend that you use PkgTemplates for creating new packages instead of using the minimal pkg> `generate` functionality described below.

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

```
(@v1.10) pkg> generate HelloWorld
```

This creates a new project `HelloWorld` in a subdirectory by the same name, with the following files (visualized with the external `tree` command):

```
shell> tree HelloWorld/
HelloWorld/
├── Project.toml
└── src
    └── HelloWorld.jl

2 directories, 2 files
```

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

```
name = "HelloWorld"
uuid = "b4cd1eb8-1e24-11e8-3319-93036a3eb9f3"
version = "0.1.0"
authors = ["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 by using the path to the directory where it is installed, and load the package:

```
pkg> activate ./HelloWorld

julia> import HelloWorld

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

For the rest of the tutorial we enter inside the directory of the project, for convenience:

```
julia> cd("HelloWorld")
```

Chapter 23

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 `activated` it):

```
(HelloWorld) pkg> add Random JSON
   Resolving package versions...
    Updating `~/HelloWorld/Project.toml`
  [682c06a0] + JSON v0.21.3
  [9a3f8284] + Random
    Updating `~/HelloWorld/Manifest.toml`
  [682c06a0] + JSON v0.21.3
  [69de0a69] + Parsers v2.4.0
  [ade2ca70] + Dates
 ...
```

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
```

Chapter 24

Defining a public API

If you want your package to be useful to other packages and you want folks to be able to easily update to newer version of your package when they come out, it is important to document what behavior will stay consistent across updates.

Unless you note otherwise, the public API of your package is defined as all the behavior you describe about public symbols. A public symbol is a symbol that is exported from your package with the `export` keyword or marked as public with the `public` keyword. When you change the behavior of something that was previously public so that the new version no longer conforms to the specifications provided in the old version, you should adjust your package version number according to Julia's variant on SemVer. If you would like to include a symbol in your public API without exporting it into the global namespace of folks who call `using YourPackage`, you should mark that symbol as public with `public that_symbol`. Symbols marked as public with the `public` keyword are just as public as those marked as public with the `export` keyword, but when folks call `using YourPackage`, they will still have to qualify access to those symbols with `YourPackage.that_symbol`.

Let's say we would like our `greet` function to be part of the public API, but not the `greet_alien` function. We could then write the following and release it as version `1.0.0`.

```
module HelloWorld

export greet

import Random
import JSON

"Writes a friendly message."
greet() = print("Hello World!")

"Greet an alien by a randomly generated name."
greet_alien() = print("Hello ", Random.randstring(8))

end # module
```

Then, if we change `greet` to

```
"Writes a friendly message that is exactly three words long."
greet() = print("Hello Lovely World!")
```

We would release the new version as `1.1.0`. This is not breaking because the new implementation conforms to the old documentation, but it does add a new feature, that the message must be three words long.

Later, we may wish to change `greet_alien` to

```
"Greet an alien by the name of \"Zork\"."
greet_alien() = print("Hello Zork")
```

And also export it by changing

```
export greet
```

to

```
export greet, greet_alien
```

We should release this new version as `1.2.0` because it adds a new feature `greet_alien` to the public API. Even though `greet_alien` was documented before and the new version does not conform to the old documentation, this is not breaking because the old documentation was not attached to a symbol that was exported at the time so that documentation does not apply across released versions.

However, if we now wish to change `greet` to

```
"Writes a friendly message that is exactly four words long."
greet() = print("Hello very lovely world")
```

we would need to release the new version as `2.0.0`. In version `1.1.0`, we specified that the greeting would be three words long, and because `greet` was exported, that description also applies to all future versions until the next breaking release. Because this new version does not conform to the old specification, it must be tagged as a breaking change.

Please note that version numbers are free and unlimited. It is okay to use lots of them (e.g. version `6.62.8`).

Chapter 25

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`.

```
julia> mkpath("deps");

julia> write("deps/build.jl",
             """
             println("I am being built...")
             """);

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

julia> print(readchomp("deps/build.log"))
I am being built...
```

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

```
julia> write("deps/build.jl",
             """
             error("Ooops")
             """);

(HelloWorld) pkg> build
    Building HelloWorld → `~/HelloWorld/deps/build.log`
ERROR: Error building `HelloWorld`:
ERROR: LoadError: Ooops
Stacktrace:
 [1] error(s::String)
   @ Base ./error.jl:35
 [2] top-level scope
   @ ~/HelloWorld/deps/build.jl:1
 [3] include(fname::String)
   @ Base.MainInclude ./client.jl:476
 [4] top-level scope
   @ none:5
in expression starting at /home/kc/HelloWorld/deps/build.jl:1
```

Warning

A build step should generally not create or modify any files in the package directory. If you need to store some files from the build step, use the Scratch.jl package.

Chapter 26

Adding tests to the package

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

```
julia> mkpath("test");

julia> write("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.

> **Warning**
>
> Tests should generally not create or modify any files in the package directory. If you need to store some files from the build step, use the Scratch.jl package.

26.1 Test-specific dependencies

Test-specific dependencies are dependencies that are not dependencies of the package itself but are available when the package is tested.

Recommended approach: Using workspaces with `test/Project.toml`

> **Compat**
>
> Workspaces require Julia 1.12+. For older Julia versions, see the legacy approaches below.

The recommended way to add test-specific dependencies is to use workspaces. This is done by:

1. Adding a `[workspace]` section to your package's `Project.toml`:

```
[workspace]
projects = ["test"]
```

1. Creating a `test/Project.toml` file with your test dependencies:

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

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

When using workspaces, the package manager resolves dependencies for all projects in the workspace together, and creates a single `Manifest.toml` next to the base `Project.toml`. This provides better dependency resolution and makes it easier to manage test-specific dependencies.

You can now use `Test` in the test script:

```
julia> write("test/runtests.jl",
             """
             using Test
             @test 1 == 1
             """);

(HelloWorld/test) pkg> activate .

(HelloWorld) pkg> test
   Testing HelloWorld
 Resolving package versions...
   Testing HelloWorld tests passed
```

Workspaces can also be used for other purposes, such as documentation or benchmarks, by adding additional projects to the workspace:

```
[workspace]
projects = ["test", "docs", "benchmarks"]
```

See the section on Workspaces in the `Project.toml` documentation for more details.

Alternative approach: Using `[sources]` with path-based dependencies

An alternative to workspaces is to use the `[sources]` section in `test/Project.toml` to reference the parent package. The `[sources]` section allows you to specify custom locations (paths or URLs) for dependencies, overriding registry information. This approach creates a **separate manifest** in the `test/` directory (unlike workspaces which create a single shared manifest).

To use this approach:

1. Create a `test/Project.toml` file and add your test dependencies:

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

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

1. Add the parent package as a dependency using `[sources]` with a relative path:

```
# In test/Project.toml
[deps]
HelloWorld = "00000000-0000-0000-0000-000000000000"  # Your package UUID
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"

[sources]
HelloWorld = {path = ".."}
```

The `[sources]` section tells Pkg to use the local path for `HelloWorld` instead of looking it up in a registry. This creates a separate `test/Manifest.toml` that tracks the resolved dependencies for your test environment independently from the main package manifest. You can now run tests directly:

```
$ julia --project=test
julia> using HelloWorld, Test

julia> include("test/runtests.jl")
```

Difference from workspaces

The key difference from workspaces is that this approach uses a **separate manifest file** (`test/Manifest.toml`) for the test environment, while workspaces create a **single shared manifest** (`Manifest.toml`) that resolves all projects together. This means:

- With `[sources]` + path: Dependencies are resolved independently for each environment
- With workspaces: Dependencies are resolved together, ensuring compatibility across all projects in the workspace

For more details on `[sources]`, see the `[sources]` section in the Project.toml documentation.

Legacy approach: `target` based test specific dependencies

Warning

This approach is legacy and maintained for compatibility. New packages should use workspaces instead.

Using this method, test-specific dependencies are added under an `[extras]` section and to a test target:

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

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

Note that the only supported targets are `test` and `build`, the latter of which (not recommended) can be used for any `deps/build.jl` scripts.

Legacy approach: `test/Project.toml` without workspace

Warning

This approach is legacy and maintained for compatibility. New packages should use workspaces instead.

In Julia 1.2 and later, test dependencies can be declared in `test/Project.toml` without using a workspace. When running tests, Pkg will automatically merge the package and test projects to create the test environment.

Note

If no `test/Project.toml` exists, Pkg will use the `target` based test specific dependencies.

This approach works similarly to the workspace approach, but without the workspace declaration in the main `Project.toml`.

Chapter 27

Compatibility on dependencies

Every dependency should in general have a compatibility constraint on it. This is an important topic so there is a separate chapter about it: Compatibility.

Chapter 28

Weak dependencies

Note

This is a somewhat advanced usage of Pkg which can be skipped for people new to Julia and Julia packages.

Compat

The described feature requires Julia 1.9+.

A weak dependency is a dependency that will not automatically install when the package is installed but you can still control what versions of that package are allowed to be installed by setting compatibility on it. These are listed in the project file under the `[weakdeps]` section:

```
[weakdeps]
SomePackage = "b3785f31-9d33-4cdf-bc73-f646780f1739"

[compat]
SomePackage = "1.2"
```

The current usage of this is almost solely limited to "extensions" which is described in the next section.

Chapter 29

Conditional loading of code in packages (Extensions)

Note

This is a somewhat advanced usage of Pkg which can be skipped for people new to Julia and Julia packages.

Compat

The described feature requires Julia 1.9+.

Sometimes one wants to make two or more packages work well together, but may be reluctant (perhaps due to increased load times) to make one an unconditional dependency of the other. A package extension is a module in a file (similar to a package) that is automatically loaded when some other set of packages are loaded into the Julia session. This is very similar to functionality that the external package Requires.jl provides, but which is now available directly through Julia, and provides added benefits such as being able to precompile the extension.

29.1 Code structure

A useful application of extensions could be for a plotting package that should be able to plot objects from a wide variety of different Julia packages. Adding all those different Julia packages as dependencies of the plotting package could be expensive since they would end up getting loaded even if they were never used. Instead, the code required to plot objects for specific packages can be put into separate files (extensions) and these are loaded only when the packages that define the type(s) we want to plot are loaded.

Below is an example of how the code can be structured for a use case in which a `Plotting` package wants to be able to display objects defined in the external package `Contour`. The file and folder structure shown below is found in the `Plotting` package.

`Project.toml`:

```
name = "Plotting"
version = "0.1.0"
uuid = "..."

[weakdeps]
```

```
Contour = "d38c429a-6771-53c6-b99e-75d170b6e991"

[extensions]
# name of extension to the left
# extension dependencies required to load the extension to the right
# use a list for multiple extension dependencies
ContourExt = "Contour"

[compat]
Contour = "0.6.2"
```

`src/Plotting.jl`:

```julia
module Plotting

function plot(x::Vector)
    # Some functionality for plotting a vector here
end

end # module
```

`ext/ContourExt.jl` (can also be in `ext/ContourExt/ContourExt.jl`):

```julia
module ContourExt # Should be same name as the file (just like a normal package)

using Plotting, Contour

function Plotting.plot(c::Contour.ContourCollection)
    # Some functionality for plotting a contour here
end

end # module
```

Extensions can have arbitrary names (here `ContourExt`), following the format of this example is likely a good idea for extensions with a single dependency. In Pkg output, extension names are always shown together with their parent package name.

Compat

Often you will want to load extension dependencies when testing your package. The recommended approach is to use workspaces and add the extension dependencies to your `test/Project.toml` (see Test-specific dependencies). For older Julia versions that don't support workspaces, you can put the extension dependencies into the `test` target, which requires you to also put the package in the `[extras]` section. The project verifier on older Julia versions will complain if this is not done.

Note

If you use a manifest generated by a Julia version that does not know about extensions with a Julia version that does know about them, the extensions will not load. This is because the manifest lacks some information that tells Julia when it should load these packages. So make sure you use a manifest generated at least the Julia version you are using.

29.2 Behavior of extensions

A user that depends only on `Plotting` will not pay the cost of the "extension" inside the `ContourExt` module. It is only when the `Contour` package actually gets loaded that the `ContourExt` extension is loaded too and provides the new functionality.

In our example, the new functionality is an additional method, which we add to an existing function from the parent package `Plotting`. Implementing such methods is among the most standard use cases of package extensions. Within the parent package, the function to extend can even be defined with zero methods, as follows:

```
function plot end
```

> **Note**
>
> If one considers `ContourExt` as a completely separate package, it could be argued that defining `Plotting.plot(c::Contour.ContourCollection)` is type piracy since `ContourExt` owns neither the function `Plotting.plot` nor the type `Contour.ContourCollection`. However, for extensions, it is ok to assume that the extension owns the functions in its parent package.

In other situations, one may need to define new symbols in the extension (types, structs, functions, etc.) instead of reusing those from the parent package. Such symbols are created in a separate module corresponding to the extension, namely `ContourExt`, and thus not in `Plotting` itself. If extension symbols are needed in the parent package, one must call `Base.get_extension` to retrieve them. Here is an example showing how a custom type defined in `ContourExt` can be accessed in `Plotting`:

```
ext = Base.get_extension(@__MODULE__, :ContourExt)
if !isnothing(ext)
    ContourPlotType = ext.ContourPlotType
end
```

On the other hand, accessing extension symbols from a third-party package (i.e. not the parent) is not a recommended practice at the moment.

29.3 Backwards compatibility

This section discusses various methods for using extensions on Julia versions that support them, while simultaneously providing similar functionality on older Julia versions.

Requires.jl

This section is relevant if you are currently using Requires.jl but want to transition to using extensions (while still having Requires be used on Julia versions that do not support extensions). This is done by making the following changes (using the example above):

- Add the following to the package file. This makes it so that Requires.jl loads and inserts the callback only when extensions are not supported

```
# This symbol is only defined on Julia versions that support extensions
if !isdefined(Base, :get_extension)
using Requires
end

@static if !isdefined(Base, :get_extension)
function __init__()
    @require Contour = "d38c429a-6771-53c6-b99e-75d170b6e991" include("../ext/ContourExt.jl")
end
end
```

or if you have other things in your `__init__()` function:

```
if !isdefined(Base, :get_extension)
using Requires
end

function __init__()
    # Other init functionality here

    @static if !isdefined(Base, :get_extension)
        @require Contour = "d38c429a-6771-53c6-b99e-75d170b6e991"
        ↪  include("../ext/ContourExt.jl")
    end
end
```

- Make the following change in the conditionally-loaded code in `ContourExt.jl`:

```
isdefined(Base, :get_extension) ? (using Contour) : (using ..Contour)
```

- Add `Requires` to `[weakdeps]` in your `Project.toml` file, so that it is listed in both `[deps]` and `[weakdeps]`. Julia 1.9+ knows to not install it as a regular dependency, whereas earlier versions will consider it a dependency.

The package should now work with Requires.jl on Julia versions before extensions were introduced and with extensions on more recent Julia versions.

Transition from normal dependency to extension

This section is relevant if you have a normal dependency that you want to transition be an extension (while still having the dependency be a normal dependency on Julia versions that do not support extensions). This is done by making the following changes (using the example above):

- Make sure that the package is **both** in the `[deps]` and `[weakdeps]` section. Newer Julia versions will ignore dependencies in `[deps]` that are also in `[weakdeps]`.
- Add the following to your main package file (typically at the bottom):

```
if !isdefined(Base, :get_extension)
  include("../ext/ContourExt.jl")
end
```

Using an extension while supporting older Julia versions

In the case where one wants to use an extension (without worrying about the feature of the extension being available on older Julia versions) while still supporting older Julia versions without workspace support, the packages under `[weakdeps]` should be duplicated into `[extras]`. This is an unfortunate duplication, but without doing this the project verifier under older Julia versions will throw an error if it finds packages under `[compat]` that is not listed in `[extras]`.

For Julia 1.13+, using workspaces is recommended and this duplication is not necessary.

Chapter 30

Package naming guidelines

Package names should be sensible to most Julia users, even to those who are not domain experts. The following guidelines apply 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 for package names to contain `DNA` if you're talking about the DNA, which has a universally agreed upon definition.
   - It's more difficult to justify package names containing the acronym `CI` for instance, which may mean continuous integration, confidence interval, etc.
   - If there is risk of confusion it may be best to disambiguate an acronym with additional words such as a lab group or field.
   - If your acronym is unambiguous, easily searchable, and/or unlikely to be confused across domains a good justification is often enough for approval.

2. Avoid using `Julia` in your package name or prefixing it with `Ju`.

   - It is usually clear from context and to your users that the package is a Julia package.
   - Package names already have a `.jl` extension, which communicates to users that `Package.jl` 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.

- Generally package names should be at least 5 characters long not including the `.jl` extension

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 can 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.

7. Avoid naming a package closely to an existing package

   - `Websocket` is too close to `WebSockets` and can be confusing to users. Rather use a new name such as `SimpleWebsockets`.

8. Avoid using a distinctive name that is already in use in a well known, unrelated project.

   - Don't use the names `Tkinter.jl`, `TkinterGUI.jl`, etc. for a package that is unrelated to the popular `tkinter` python package, even if it provides bindings to Tcl/Tk. A package name of `Tkinter.jl` would only be appropriate if the package used Python's library to accomplish its work or was spearheaded by the same community of developers.
   - It's okay to name a package `HTTP.jl` even though it is unrelated to the popular rust crate `http` because in most usages the name "http" refers to the hypertext transfer protocol, not to the `http` rust crate.
   - It's okay to name a package `OpenSSL.jl` if it provides an interface to the OpenSSL library, even without explicit affiliation with the creators of the OpenSSL (provided there's no copyright or trademark infringement etc.)

9. Packages should follow the Stylistic Conventions.

   - The package name should begin with a capital letter and word separation is shown with upper camel case
   - Only ASCII characters are allowed in a package name
   - Packages that provide the functionality of a project from another language should use the Julia convention
   - Packages that provide pre-built libraries and executables can keep their original name, but should get `_jll`as a suffix. For example `pandoc_jll` wraps pandoc. However, note that the generation and release of most JLL packages is handled by the Yggdrasil system.

10. For the complete list of rules for automatic merging into the General registry, see these guidelines.

Chapter 31

Registering packages

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

Chapter 32

Creating new package versions

After registering your package, you'll want to release new versions as you add features and fix bugs. The typical workflow is:

1. **Update the version number** in your `Project.toml` file according to semantic versioning rules. For example:
   - Increment the patch version (1.2.3 → 1.2.4) for bug fixes
   - Increment the minor version (1.2.3 → 1.3.0) for new features that don't break existing functionality
   - Increment the major version (1.2.3 → 2.0.0) for breaking changes
2. **Commit your changes** to your package repository, including the updated version number.
3. **Tag the release** using Registrator. Comment `@JuliaRegistrator register` on a commit or pull request in your GitHub repository
4. **Automated tagging**: Once you've set up `TagBot`, it will automatically create a git tag in your repository when a new version is registered. This keeps your repository tags synchronized with registered versions.

The registration process typically takes a few minutes. Registrator will:

- Check that your package meets registry requirements (has tests, proper version bounds, etc.)
- Submit a pull request to the General registry
- Automated checks will run, and if everything passes, the PR will be automatically merged

For private registries or more advanced workflows, see the documentation for LocalRegistry.jl and RegistryCI.jl.

Chapter 33

Best Practices

Packages should avoid mutating their own state (writing to files within their package directory). Packages should, in general, not assume that they are located in a writable location (e.g. if installed as part of a system-wide depot) or even a stable one (e.g. if they are bundled into a system image by PackageCompiler.jl). To support the various use cases in the Julia package ecosystem, the Pkg developers have created a number of auxiliary packages and techniques to help package authors create self-contained, immutable, and relocatable packages:

- `Artifacts` can be used to bundle chunks of data alongside your package, or even allow them to be downloaded on-demand. Prefer artifacts over attempting to open a file via a path such as `joinpath(@__DIR__, "data", "my_dataset.csv")` as this is non-relocatable. Once your package has been precompiled, the result of `@__DIR__` will have been baked into your precompiled package data, and if you attempt to distribute this package, it will attempt to load files at the wrong location. Artifacts can be bundled and accessed easily using the `artifact"name"` string macro.
- `Scratch.jl` provides the notion of "scratch spaces", mutable containers of data for packages. Scratch spaces are designed for data caches that are completely managed by a package and should be removed when the package itself is uninstalled. For important user-generated data, packages should continue to write out to a user-specified path that is not managed by Julia or Pkg.
- `Preferences.jl` allows packages to read and write preferences to the top-level `Project.toml`. These preferences can be read at runtime or compile-time, to enable or disable different aspects of package behavior. Packages previously would write out files to their own package directories to record options set by the user or environment, but this is highly discouraged now that `Preferences` is available.

Chapter 34

See Also

- Managing Packages - Learn how to add, update, and manage package dependencies
- Working with Environments - Understand environments and reproducible development
- Compatibility - Specify version constraints for dependencies
- API Reference - Functional API for non-interactive package management

Part VI

6. Apps

Note

The app support in Pkg is currently considered experimental and some functionality and API may change.

Some inconveniences that can be encountered are:

- You need to manually make `~/.julia/bin` available on the PATH environment.
- The path to the julia executable used is the same as the one used to install the app. If this julia installation gets removed, you might need to reinstall the app.

Apps are Julia packages that are intended to be run as "standalone programs" (by e.g. typing the name of the app in the terminal possibly together with some arguments or flags/options). This is in contrast to most Julia packages that are used as "libraries" and are loaded by other files or in the Julia REPL.

Chapter 35

Creating a Julia app

A Julia app is structured similar to a standard Julia library with the following additions:

- A `@main` entry point in the package module (see the Julia help on `@main` for details)
- An `[apps]` section in the `Project.toml` file listing the executable names that the package provides.

A very simple example of an app that prints the reversed input arguments would be:

```julia
# src/MyReverseApp.jl
module MyReverseApp

function (@main)(ARGS)
    for arg in ARGS
        print(stdout, reverse(arg), " ")
    end
    return
end

end # module
```

```toml
# Project.toml

# standard fields here

[apps]
reverse = {}
```

The empty table `{}` is to allow for giving metadata about the app.

After installing this app one could run:

```
$ reverse some input string
 emos tupni gnirts
```

directly in the terminal.

Chapter 36

Multiple Apps per Package

A single package can define multiple apps by using submodules. Each app can have its own entry point in a different submodule of the package.

```julia
# src/MyMultiApp.jl
module MyMultiApp

function (@main)(ARGS)
    println("Main app: ", join(ARGS, " "))
end

include("CLI.jl")

end # module
```

```julia
# src/CLI.jl
module CLI

function (@main)(ARGS)
    println("CLI submodule: ", join(ARGS, " "))
end

end # module CLI
```

```
# Project.toml

# standard fields here

[apps]
main-app = {}
cli-app = { submodule = "CLI" }
```

This will create two executables:

- `main-app` that runs `julia -m MyMultiApp`
- `cli-app` that runs `julia -m MyMultiApp.CLI`

Chapter 37

Configuring Julia Flags

Apps can specify default Julia command-line flags that will be passed to the Julia process when the app is run. This is useful for configuring performance settings, threading, or other Julia options specific to your application.

37.1 Default Julia Flags

You can specify default Julia flags in the `Project.toml` file using the `julia_flags` field:

```
# Project.toml

[apps]
myapp = { julia_flags = ["--threads=4", "--optimize=2"] }
performance-app = { julia_flags = ["--threads=auto", "--startup-file=yes", "--depwarn=no"] }
debug-app = { submodule = "Debug", julia_flags = ["--check-bounds=yes", "--optimize=0"] }
```

With this configuration:

- `myapp` will run with 4 threads and optimization level 2
- `performance-app` will run with automatic thread detection, startup file enabled, and deprecation warnings disabled
- `debug-app` will run with bounds checking enabled and no optimization

37.2 Runtime Julia Flags

You can override or add to the default Julia flags at runtime using the `--` separator. Everything before `--` will be passed as flags to Julia, and everything after `--` will be passed as arguments to your app:

```
# Uses default flags from Project.toml
myapp input.txt output.txt

# Override thread count, keep other defaults
myapp --threads=8 -- input.txt output.txt

# Add additional flags
myapp --threads=2 --optimize=3 --check-bounds=yes -- input.txt output.txt
```

```
# Only Julia flags, no app arguments
myapp --threads=1 --
```

The final Julia command will combine:

1. Fixed flags (like `--startup-file=no` and `-m ModuleName`)
2. Default flags from `julia_flags` in Project.toml
3. Runtime flags specified before `--`
4. App arguments specified after `--`

37.3 Overriding the Julia Executable

By default, apps run with the same Julia executable that was used to install them. You can override this globally using the `JULIA_APPS_JULIA_CMD` environment variable:

```
# Use a different Julia version for all apps
export JULIA_APPS_JULIA_CMD=/path/to/different/julia
myapp input.txt

# On Windows
set JULIA_APPS_JULIA_CMD=C:\path\to\different\julia.exe
myapp input.txt
```

Chapter 38

Installing Julia apps

The installation of Julia apps is similar to installing Julia libraries but instead of using e.g. `Pkg.add` or `pkg> add` one uses `Pkg.Apps.add` or `pkg> app add` (`develop` is also available).

Part VII

7. 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 = "1.6"
Example = "0.5"
```

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

Use the command `compat` to edit the compat entries in the Pkg REPL, or manually edit the project file.

Info

The rules below apply to the `Project.toml` file; for registries, see Registry Compat.toml.

Info

Note that registration into Julia's General Registry requires each dependency to have a `[compat]` entry with an upper bound.

Chapter 39

Version specifier format

Similar to other package managers, the Julia package manager respects semantic versioning (semver), with an exception for leading zeros. 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)`. Note leading zeros are treated differently, e.g. `Example = "0.2, 1"` would only result in `[0.2.0 - 0.3.0) ∪ [1.0.0 - 2.0.0)`. See the next section for more information on versions with leading zeros.

39.1 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]
Example = "0.0.1"
```

results in a versionbound on `Example` as `[0.0.1, 0.0.2)` (which is equivalent to only the version 0.0.1), while the `[compat]` entry:

```
[compat]
Example = "0.2.1"
```

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.

39.2 Caret specifiers

A caret (^) specifier allows upgrade that would be compatible according to semver. This is the default behavior if no specifier is used. 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)
```

39.3 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, 0.1.0)
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.

39.4 Equality specifier

Equality can be used to specify exact versions:

```
[compat]
PkgA = "=1.2.3"           # [1.2.3, 1.2.3]
PkgA = "=0.10.1, =0.10.3" # 0.10.1 or 0.10.3
```

39.5 Inequality specifiers

Inequalities can also be used to specify version ranges:

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

39.6 Hyphen specifiers

Hyphen syntax can also be used to specify version ranges. Make sure that you have a space on both sides of the hyphen.

```
[compat]
PkgA = "1.2.3 - 4.5.6" # [1.2.3, 4.5.6]
PkgA = "0.2.3 - 4.5.6" # [0.2.3, 4.5.6]
```

Any unspecified trailing numbers in the first end-point are considered to be zero:

```
[compat]
PkgA = "1.2 - 4.5.6"   # [1.2.0, 4.5.6]
PkgA = "1 - 4.5.6"     # [1.0.0, 4.5.6]
PkgA = "0.2 - 4.5.6"   # [0.2.0, 4.5.6]
PkgA = "0.2 - 0.5.6"   # [0.2.0, 0.5.6]
```

Any unspecified trailing numbers in the second end-point will be considered to be wildcards:

```
[compat]
PkgA = "1.2.3 - 4.5"   # 1.2.3 - 4.5.* = [1.2.3, 4.6.0)
PkgA = "1.2.3 - 4"     # 1.2.3 - 4.*.* = [1.2.3, 5.0.0)
PkgA = "1.2 - 4.5"     # 1.2.0 - 4.5.* = [1.2.0, 4.6.0)
PkgA = "1.2 - 4"       # 1.2.0 - 4.*.* = [1.2.0, 5.0.0)
PkgA = "1 - 4.5"       # 1.0.0 - 4.5.* = [1.0.0, 4.6.0)
PkgA = "1 - 4"         # 1.0.0 - 4.*.* = [1.0.0, 5.0.0)
PkgA = "0.2.3 - 4.5"   # 0.2.3 - 4.5.* = [0.2.3, 4.6.0)
PkgA = "0.2.3 - 4"     # 0.2.3 - 4.*.* = [0.2.3, 5.0.0)
PkgA = "0.2 - 4.5"     # 0.2.0 - 4.5.* = [0.2.0, 4.6.0)
PkgA = "0.2 - 4"       # 0.2.0 - 4.*.* = [0.2.0, 5.0.0)
PkgA = "0.2 - 0.5"     # 0.2.0 - 0.5.* = [0.2.0, 0.6.0)
PkgA = "0.2 - 0"       # 0.2.0 - 0.*.* = [0.2.0, 1.0.0)
```

Chapter 40

Fixing conflicts

Version conflicts were introduced previously with an example of a conflict arising in a package `D` used by two other packages, `B` and `C`. Our analysis of the error message revealed that `B` is using an outdated version of `D`. To fix it, the first thing to try is to `pkg> dev B` so that you can modify `B` and its compatibility requirements. If you open its `Project.toml` file in an editor, you would probably notice something like

```
[compat]
D = "0.1"
```

Usually the first step is to modify this to something like

```
[compat]
D = "0.1, 0.2"
```

This indicates that `B` is compatible with both versions 0.1 and version 0.2; if you `pkg> up` this would fix the package error. However, there is one major concern you need to address first: perhaps there was an incompatible change in `v0.2` of `D` that breaks `B`. Before proceeding further, you should update all packages and then run B's tests, scanning the output of `pkg> test B` to be sure that `v0.2` of `D` is in fact being used. (It is possible that an additional dependency of `D` pins it to `v0.1`, and you wouldn't want to be misled into thinking that you had tested `B` on the newer version.) If the new version was used and the tests still pass, you can assume that `B` didn't need any further updating to accommodate `v0.2` of `D`; you can safely submit this change as a pull request to `B` so that a new release is made. If instead an error is thrown, it indicates that `B` requires more extensive updates to be compatible with the latest version of `D`; those updates will need to be completed before it becomes possible to use both `A` and `B` simultaneously. You can, though, continue to use them independently of one another.

Part VIII

8. Registries

Registries contain information about packages, such as available releases and dependencies, and where they can be downloaded. The `General` registry is the default one, and is installed automatically if there are no other registries installed.

Chapter 41

Managing registries

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

41.1 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.

If a custom registry has been installed causing the `General` registry to not be automatically installed, it is easy to add it manually:

```
pkg> registry add 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).

Registries from a package server

It is possible for a package server to be advertising additional available package registries. When Pkg runs with a clean Julia depot (e.g. after a fresh install), with a custom package server configured with `JULIA_PKG_SERVER`, it will automatically add all such available registries. If the depot already has some registries installed (e.g. General), the additional ones can easily be installed with the no-argument `registry add` command.

41.2 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
```

41.3 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`
```

Registries automatically update once per session when a package operation is performed so it rarely has to be done manually.

Chapter 42

Registry format

In a registry, each package gets its own directory; in that directory are the following files: `Compat.toml`, `Deps.toml`, `Package.toml`, and `Versions.toml`. The formats of these files are described below.

42.1 Registry Package.toml

The `Package.toml` file contains basic metadata about the package, such as its name, UUID, repository URL, and optional metadata.

Package metadata

The `[metadata]` table in `Package.toml` provides a location for metadata about the package that doesn't fit into the other registry files. This is an extensible framework for adding package-level metadata.

Deprecated packages

One use of the `[metadata]` table is to mark packages as deprecated using `[metadata.deprecated]`. Deprecated packages will:

- Show as `[deprecated]` in package status output
- Be excluded from tab-completion suggestions
- Still be installable and usable

The `[metadata.deprecated]` table can contain arbitrary metadata fields. Two special fields are recognized by Pkg and displayed when using `pkg> status --deprecated`:

- `reason`: A string explaining why the package is deprecated
- `alternative`: A string suggesting a replacement package

Example:

```
name = "MyPackage"
uuid = "..."
repo = "..."
```

```
[metadata.deprecated]
reason = "This package is no longer maintained"
alternative = "ReplacementPackage"
```

Other fields can be added to `[metadata.deprecated]` for use by registries or other tools.

42.2 Registry Compat.toml

The `Compat.toml` file has a series of blocks specifying version numbers, with a set of dependencies listed below. For example, part of such a file might look like this:

```
["0.8-0.8.3"]
DependencyA = "0.4-0.5"
DependencyB = "0.3-0.5"

["0.8.2-0.8.5"]
DependencyC = "0.7-0"
```

Dependencies that are unchanged across a range of versions are grouped together in these blocks. The interpretation of these ranges is given by the comment after each line below:

```
"0.7-0.8"  # [0.7.0, 0.9.0)
"0.7-0"    # [0.7.0, 1.0.0)
"0.8.6-0"  # [0.8.6, 1.0.0)
"0.7-*"    # [0.7.0, ∞)
```

So for this package, versions `[0.8.0, 0.8.3]` depend on versions `[0.4.0, 0.6.0)` of `DependencyA` and version `[0.3.0, 0.6.0)` of `DependencyB`. Meanwhile, it is also true that versions `[0.8.2, 0.8.5]` require specific versions of `DependencyC` (so that all three are required for versions `0.8.2` and `0.8.3`).

42.3 Registry formats

> **Julia 1.7**
>
> Compressed registry formats are available starting with Julia 1.7.

Registries can be installed in several different formats, each with different tradeoffs:

Compressed registries (preferred)

When using a package server (the default), registries are downloaded as compressed tarballs. This is the preferred format for the General registry because it is:

- **Fast for the initial download**: Only a single compressed file needs to be transferred
- **Fast to use**: Pkg reads data directly from the packed tarball, avoiding many small filesystem reads
- **Low disk usage**: The registry can be read directly from the compressed file without extraction

You can check if a registry is compressed by running `Pkg.Registry.status()`, which will describe it as a "packed registry" when it remains in its tarball and an "unpacked registry" when the files have been extracted to disk.

Git registries

Registries can also be installed as git clones. This format:

- **Provides immediate updates**: Running `Pkg.Registry.update()` fetches the latest changes directly from the git repository
- **Uses more disk space**: The full git history is stored locally
- **May be slower**: Cloning and updating can take longer than downloading a compressed tarball
- **Integrates with local tooling**: All registry files are present on disk, so you can inspect or customize them using familiar editors and git workflows

To install a registry as a git clone, use:

```
Pkg.Registry.add(url = "https://github.com/JuliaRegistries/General.git")
```

Converting between formats

To convert an existing registry from git to compressed (or vice versa), remove and re-add it:

```
# Convert to compressed (uses package server if available)
pkg> registry rm General

pkg> registry add General

# Convert to git
pkg> registry rm General

pkg> registry add https://github.com/JuliaRegistries/General.git
```

> **Note**
>
> The environment variable `JULIA_PKG_SERVER` controls whether package servers are used. Setting it to an empty string (`JULIA_PKG_SERVER=""`) disables package server usage and forces git clones. To force unpacking even when using a package server, set `JULIA_PKG_UNPACK_REGISTRY=true`.

42.4 Registry flavors

The default Pkg Server (`pkg.julialang.org`) offers two different "flavors" of registry.

> **Julia 1.8**
>
> Registry flavors are only available starting with Julia 1.8.

- `conservative`: suitable for most users; all packages and artifacts in this registry flavor are available from the Pkg Server, with no need to download from other sources
- `eager`: this registry offers the latest versions of packages, even if the Pkg and Storage Servers have not finished processing them; thus, some packages and artifacts may not be available from the Pkg Server, and thus may need to be downloaded from other sources (such as GitHub)

The default registry flavor is `conservative`. We recommend that most users stick to the `conservative` flavor unless they know that they need to use the `eager` flavor.

To select the `eager` flavor:

```
ENV["JULIA_PKG_SERVER_REGISTRY_PREFERENCE"] = "eager"

import Pkg

Pkg.Registry.update()
```

To select the `conservative` flavor:

```
ENV["JULIA_PKG_SERVER_REGISTRY_PREFERENCE"] = "conservative"

import Pkg

Pkg.Registry.update()
```

42.5 Creating and maintaining registries

Pkg only provides client facilities for registries, rather than functionality to create or maintain them. However, Registrator.jl and LocalRegistry.jl provide ways to create and update registries, and RegistryCI.jl provides automated testing and merging functionality for maintaining a registry.

Part IX

9. 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 43

Basic Usage

Pkg artifacts are declared in an `Artifacts.toml` file, which can be placed in your current directory or in the root of your package. Currently, Pkg supports downloading of tarfiles (which can be compressed) from a URL. Following is a minimal `Artifacts.toml` file which will permit the downloading of a `socrates.tar.gz` file from `github.com`. In this example, a single artifact, given the name `socrates`, is defined.

```
# a simple Artifacts.toml file
[socrates]
git-tree-sha1 = "43563e7631a7eafae1f9f8d9d332e3de44ad7239"

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

If this `Artifacts.toml` file is placed in your current directory, then `socrates.tar.gz` can be downloaded, unpacked and used with `artifact"socrates"`. Since this tarball contains a folder `bin`, and a text file named `socrates` within that folder, we could access the content of that file as follows.

```
using Pkg.Artifacts

rootpath = artifact"socrates"
open(joinpath(rootpath, "bin", "socrates")) do file
    println(read(file, String))
end
```

If you have an existing tarball that is accessible via a `url`, it could also be accessed in this manner. To create the `Artifacts.toml` you must compute two hashes: the sha256 hash of the download file, and the `git-tree-sha1` of the unpacked content. These can be computed as follows.

```
using Tar, Inflate, SHA

filename = "socrates.tar.gz"
println("sha256: ", bytes2hex(open(sha256, filename)))
println("git-tree-sha1: ", Tar.tree_hash(IOBuffer(inflate_gzip(filename))))
```

To access this artifact from within a package you create, place the `Artifacts.toml` at the root of your package, adjacent to `Project.toml`. Then, make sure to add Pkg in your deps and set `julia = "1.3"` or higher in your `compat` section.

Chapter 44

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.x8
    ↪  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.x8
    ↪  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 45

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 46

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`
iris_dataset_path = artifact_path(iris_hash)
```

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.
iris_dataset_path = artifact"iris"
```

Chapter 47

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, 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 48

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 an 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 each other, 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 can 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.

Chapter 49

Extending Platform Selection

Julia 1.7

Pkg's extended platform selection requires at least Julia 1.7, and is considered experimental.

New in Julia 1.7, `Platform` objects can have extended attributes applied to them, allowing artifacts to be tagged with things such as CUDA driver version compatibility, microarchitectural compatibility, julia version compatibility and more! Note that this feature is considered experimental and may change in the future. If you as a package developer find yourself needing this feature, please get in contact with us so it can evolve for the benefit of the whole ecosystem. In order to support artifact selection at `Pkg.add()` time, Pkg will run the specially-named file `<project_root>/.pkg/select_artifacts.jl`, passing the current platform triplet as the first argument. This artifact selection script should print a `TOML`-serialized dictionary representing the artifacts that this package needs according to the given platform, and perform any inspection of the system as necessary to auto-detect platform capabilities if they are not explicitly provided by the given platform triplet. The format of the dictionary should match that returned from `Artifacts.select_downloadable_artifacts()`, and indeed most packages should simply call that function with an augmented `Platform` object. An example artifact selection hook definition might look like the following, split across two files:

```
# .pkg/platform_augmentation.jl
using Libdl, Base.BinaryPlatforms
function augment_platform!(p::Platform)
    # If this platform object already has a `cuda` tag set, don't augment
    if haskey(p, "cuda")
        return p
    end

    # Open libcuda explicitly, so it gets `dlclose()`'ed after we're done
    dlopen("libcuda") do lib
        # find symbol to ask for driver version; if we can't find it, just silently continue
        cuDriverGetVersion = dlsym(lib, "cuDriverGetVersion"; throw_error=false)
        if cuDriverGetVersion !== nothing
            # Interrogate CUDA driver for driver version:
            driverVersion = Ref{Cint}()
            ccall(cuDriverGetVersion, UInt32, (Ptr{Cint},), driverVersion)

            # Store only the major version
            p["cuda"] = div(driverVersion, 1000)
        end
    end
```

```
    # Return possibly-altered `Platform` object
    return p
end
```

```
using TOML, Artifacts, Base.BinaryPlatforms
include("./platform_augmentation.jl")
artifacts_toml = joinpath(dirname(@__DIR__), "Artifacts.toml")

# Get "target triplet" from ARGS, if given (defaulting to the host triplet otherwise)
target_triplet = get(ARGS, 1, Base.BinaryPlatforms.host_triplet())

# Augment this platform object with any special tags we require
platform = augment_platform!(HostPlatform(parse(Platform, target_triplet)))

# Select all downloadable artifacts that match that platform
artifacts = select_downloadable_artifacts(artifacts_toml; platform)

# Output the result to `stdout` as a TOML dictionary
TOML.print(stdout, artifacts)
```

In this hook definition, our platform augmentation routine opens a system library (`libcuda`), searches it for a symbol to give us the CUDA driver version, then embeds the major version of that version number into the cuda property of the `Platform` object we are augmenting. While it is not critical for this code to actually attempt to close the loaded library (as it will most likely be opened again by the CUDA package immediately after the package operations are completed) it is best practice to make hooks as lightweight and transparent as possible, as they may be used by other Pkg utilities in the future. In your own package, you should also use augmented platform objects when using the `@artifact_str` macro, as follows:

```
include("../.pkg/platform_augmentation.jl")

function __init__()
    p = augment_platform!(HostPlatform())
    global my_artifact_dir = @artifact_str("MyArtifact", p)
end
```

This ensures that the same artifact is used by your code as Pkg attempted to install.

Artifact selection hooks are only allowed to use `Base`, `Artifacts`, `Libdl`, and `TOML`. They are not allowed to use any other standard libraries, and they are not allowed to use any packages (including the package to which they belong).

Part X

10. 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. The file name may also be suffixed by `-v{major}.{minor}.toml` which Julia will prefer if the version matches `VERSION`, allowing multiple environments to be maintained for different Julia versions.

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.

> **Note**
>
> **Packages vs. Modules:** A package is a source tree with a `Project.toml` file and other components that Pkg can install and manage. A module is a Julia language construct (created with the `module` keyword) that provides a namespace for code. Typically, a package contains a module of the same name (e.g., the `DataFrames` package contains a `DataFrames` module), but they are distinct concepts: the package is the distributable unit that Pkg manages, while the module is the namespace that your code interacts with using `import` or `using`.

Application: a project which provides standalone functionality not intended to be reused by other Julia projects. For example a web application or a command-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 set and change the global configurations of packages it depends on. Packages, on the other hand, may not change the global state of their dependencies 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 UUIDs but don't need them.
> 3. **Applications** can provide global configuration, whereas packages cannot.

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

Materialize: the process of installing all packages and dependencies specified in a manifest file to recreate an exact environment state. When you `instantiate` a project, Pkg materializes its environment by downloading and installing all the exact package versions recorded in the `Manifest.toml` file. This ensures reproducibility across different machines and users.

Canonical: refers to a single, authoritative location for each specific version of a package. When the same package version is used by multiple environments, Pkg stores it in one canonical location and all environments reference that same location, rather than duplicating the package files. This saves disk space and ensures consistency.

Part XI

11. `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 determine e.g. what `using Example` should do. For more details see the section about Code Loading in the Julia manual.

Chapter 50

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.

50.1 The `authors` field

For a package, the optional `authors` field is a TOML array describing the package authors. Entries in the array can either be a string in the form `"NAME"` or `"NAME <EMAIL>"`, or a table keys following the Citation File Format schema for either a `person` or an `entity`.

For example:

```
authors = [
  "Some One <someone@email.com>",
  "Foo Bar <foo@bar.com>",
  {given-names = "Baz", family-names = "Qux", email = "bazqux@example.com", orcid =
  ↪  "https://orcid.org/0000-0000-0000-0000", website = "https://github.com/bazqux"},
]
```

If all authors are specified by tables, it is possible to use the TOML Array of Tables syntax

```
[[authors]]
given-names = "Some"
family-names = "One"
email = "someone@email.com"

[[authors]]
given-names = "Foo"
family-names = "Bar"
email = "foo@bar.com"

[[authors]]
given-names = "Baz"
family-names = "Qux"
email = "bazqux@example.com"
orcid = "https://orcid.org/0000-0000-0000-0000"
website = "https://github.com/bazqux"
```

50.2 The name field

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

```
name = "Example"
```

The name must be a valid identifier (a sequence of Unicode characters that does not start with a number and is neither `true` nor `false`). For packages, it is recommended to follow the package naming rules. The `name` field is mandatory for packages.

50.3 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.

> **Note**
>
> It is recommended that `UUIDs.uuid4()` is used to generate random UUIDs.

Why UUIDs are important

UUIDs serve several critical purposes in the Julia package ecosystem:

- **Unique identification**: UUIDs uniquely identify packages across all registries and repositories, preventing naming conflicts. Two different packages can have the same name (e.g., in different registries), but their UUIDs will always be different.
- **Multiple registries**: UUIDs enable the use of multiple package registries (including private registries) without conflicts, as each package is uniquely identified by its UUID regardless of which registry it comes from.

50.4 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 the section on pre-1.0 behavior for more details.

50.5 The `readonly` field

The `readonly` field is a boolean that, when set to `true`, marks the environment as read-only. This prevents any modifications to the environment, including adding, removing, or updating packages. For example:

```
readonly = true
```

When an environment is marked as readonly, Pkg will throw an error if any operation that would modify the environment is attempted. If the `readonly` field is not present or set to `false` (the default), the environment can be modified normally.

You can also programmatically check and modify the readonly state using the `Pkg.readonly` function:

```
# Check if current environment is readonly
is_readonly = Pkg.readonly()

# Enable readonly mode
previous_state = Pkg.readonly(true)

# Disable readonly mode
Pkg.readonly(false)
```

When readonly mode is enabled, the status display will show `(readonly)` next to the project name to indicate the environment is protected from modifications.

50.6 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.

50.7 The `[sources]` section

Specifying a path or repo (+ branch) for a dependency is done in the `[sources]` section. These are especially useful for controlling unregistered dependencies without having to bundle a corresponding manifest file.

Each entry in the `[sources]` section supports the following keys:

- **`url`**: The URL of the Git repository. Cannot be used with `path`.
- **`rev`**: The Git revision (branch name, tag, or commit hash) to use. Only valid with `url`.
- **`subdir`**: A subdirectory within the repository containing the package.
- **`path`**: A local filesystem path to the package. Cannot be used with `url` or `rev`. This will dev the package.

This might in practice look something like:

```
[sources]
Example = {url = "https://github.com/JuliaLang/Example.jl", rev = "custom_branch"}
WithinMonorepo = {url = "https://github.org/author/BigProject", subdir = "SubPackage"}
SomeDependency = {path = "deps/SomeDependency.jl"}
```

When `[sources]` entries are used

Sources are read and applied in the following situations:

1. **Active environment**: When resolving dependencies for the currently active environment, sources from the environment's `Project.toml` override registry information for direct dependencies.
2. **Automatic addition**: When you add a package by URL (e.g., `pkg> add https://github.com/...`) or develop a package (e.g., `pkg> dev Example`), Pkg automatically adds an entry to `[sources]` for that package in your active environment's `Project.toml`.
3. **Recursive collection**: When a package is added by URL or path, Pkg recursively collects `[sources]` entries from that package's dependencies. This allows private dependency chains to resolve without registry metadata. For example:
   - If you add Package A by URL, and Package A has a `[sources]` entry for Package B
   - And Package B (also specified by URL in A's sources) has a `[sources]` entry for Package C
   - Then all three packages' source information will be collected and used during resolution

This recursive behavior is particularly useful for managing chains of unregistered or private packages.

> **Scope of sources**
>
> Sources are only used when the environment containing them is the active environment being resolved. If a package is used as a dependency in another project, its `[sources]` section is **not** consulted (except when that package itself was added by URL or path, in which case recursive collection applies as described above).

> **Test-specific dependencies**
>
> A use case for `[sources]` with `path` is in `test/Project.toml` to reference the parent package using `path = ".."`. This allows test dependencies to be managed independently with their own manifest file. See Test-specific dependencies for more details on this and other approaches.

> **Compat**
>
> Specifying sources requires Julia 1.11+.

50.8 The [weakdeps] section

Weak dependencies are optional dependencies that will not automatically install when the package is installed, but for which you can still specify compatibility constraints. Weak dependencies are typically used in conjunction with package extensions (see [extensions] below), which allow conditional loading of code when the weak dependency is available in the environment.

Example:

```
[weakdeps]
SomePackage = "b3785f31-9d33-4cdf-bc73-f646780f1739"

[compat]
SomePackage = "1.2"
```

For more details on using weak dependencies and extensions, see the Weak dependencies section in the Creating Packages guide.

> **Compat**
>
> Weak dependencies require Julia 1.9+.

50.9 The [extensions] section

Extensions allow packages to provide optional functionality that is only loaded when certain other packages (typically listed in [weakdeps]) are available. Each entry in the [extensions] section maps an extension name to one or more package dependencies required to load that extension.

Example:

```
[weakdeps]
Contour = "d38c429a-6771-53c6-b99e-75d170b6e991"

[extensions]
ContourExt = "Contour"
```

The extension code itself should be placed in an ext/ directory at the package root, with the file name matching the extension name (e.g., ext/ContourExt.jl). For more details on creating and using extensions, see the Conditional loading of code in packages (Extensions) section in the Creating Packages guide.

> **Compat**
>
> Extensions require Julia 1.9+.

50.10 The [compat] section

Compatibility constraints for dependencies can be listed in the [compat] section. This applies to packages listed under [deps], [weakdeps], and [extras].

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"
```

50.11 The `[workspace]` section

A project file can define a workspace by giving a set of projects that is part of that workspace. Each project in a workspace can include their own dependencies, compatibility information, and even function as full packages.

When the package manager resolves dependencies, it considers the requirements of all the projects in the workspace. The compatible versions identified during this process are recorded in a single manifest file located next to the base project file.

A workspace is defined in the base project by giving a list of the projects in it:

```
[workspace]
projects = ["test", "docs", "benchmarks", "PrivatePackage"]
```

This structure is particularly beneficial for developers using a monorepo approach, where a large number of unregistered packages may be involved. It's also useful for adding test-specific dependencies to a package by including a `test` project in the workspace (see Test-specific dependencies), or for adding documentation or benchmarks with their own dependencies.

Workspace can be nested: a project that itself defines a workspace can also be part of another workspace. In this case, the workspaces are "merged" with a single manifest being stored alongside the "root project" (the project that doesn't have another workspace including it).

50.12 The `[extras]` section (legacy)

> **Warning**
>
> The `[extras]` section is a legacy feature maintained for compatibility. For Julia 1.13+, using workspaces is the recommended approach for managing test-specific and other optional dependencies.

The `[extras]` section lists additional dependencies that are not regular dependencies of the package, but may be used in specific contexts like testing. These are typically used in conjunction with the `[targets]` section.

Example:

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

For more information, see the Test-specific dependencies section.

50.13 The [targets] section (legacy)

> **Warning**
>
> The [targets] section is a legacy feature maintained for compatibility. For Julia 1.13+, using workspaces is the recommended approach for managing test-specific and build dependencies.

The [targets] section specifies which packages from [extras] should be available in specific contexts. The only supported targets are test (for test dependencies) and build (for build-time dependencies used by deps/build.jl scripts).

Example:

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

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

For more information, see the Test-specific dependencies section.

Chapter 51

`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. Given a `Project.toml` + `Manifest.toml` pair, it is possible to instantiate the exact same package environment, which is very useful for reproducibility. For the details, see `Pkg.instantiate`.

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

51.1 Different Manifests for Different Julia versions

Starting from Julia v1.10.8, there is an option to name manifest files in the format `Manifest-v{major}.{minor}.toml`. Julia will then preferentially use the version-specific manifest file if available. For example, if both `Manifest-v1.11.toml` and `Manifest.toml` exist, Julia 1.11 will prioritize using `Manifest-v1.11.toml`. However, Julia versions 1.10, 1.12, and all others will default to using `Manifest.toml`. This feature allows for easier management of different instantiated versions of dependencies for various Julia versions. Note that there can only be one `Project.toml` file. While `Manifest-v{major}.{minor}.toml` files are not automatically created by Pkg, users can manually rename a `Manifest.toml` file to match the versioned format, and Pkg will subsequently maintain it through its operations.

51.2 `Manifest.toml` entries

There are three top-level entries in the manifest which could look like this:

```
julia_version = "1.8.2"
manifest_format = "2.0"
project_hash = "4d9d5b552a1236d3c1171abf88d59da3aaac328a"
```

This shows the Julia version the manifest was created on, the "format" of the manifest and a hash of the project file, so that it is possible to see when the manifest is stale compared to the project file.

Manifest format versions

The `manifest_format` field indicates the structure version of the manifest file:

- `"2.0"`: The standard format for Julia 1.7+
- `"2.1"`: The current format (requires Julia 1.13+). This format introduced registry tracking in the `[registries]` section.

51.3 The `[registries]` section

> **Compat**
>
> Registry tracking in manifests requires Julia 1.13+ and manifest format `"2.1"`.

Starting with manifest format `2.1`, the manifest can include a `[registries]` section that tracks metadata about the registries from which packages were obtained. This ensures that the exact source of each package version can be identified, which is particularly important when using multiple registries or private registries.

Each registry entry in the manifest looks like this:

```
[registries.General]
uuid = "23338594-aafe-5451-b93e-139f81909106"
url = "https://github.com/JuliaRegistries/General.git"
```

The registry entries include:

- **`uuid`** (required): The unique identifier for the registry.
- **`url`** (optional): The URL where the registry can be found. This enables automatic installation of registries when instantiating an environment on a new machine.

The section key (e.g., `General` in the example above) is the registry name.

51.4 Package 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. 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/Julia`
- `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`
- `registries`: a reference to the registry IDs from which this package version was obtained. This can be either a single string (e.g., `registries = "General"`) or a vector of strings if the package is available in multiple registries (e.g., `registries = ["General", "MyRegistry"]`). All registries containing this package version are recorded. This field is only present in manifest format `2.1` or later, and only for packages that were added from a registry (not for developed or git-tracked packages).

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:

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

Note, in particular, that no `repo-url` is present, since that information is included in the registry where this package was found. The `registries` field (present in manifest format `2.1+`) references an entry in the `[registries]` section that contains the registry metadata.

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:

```
[[deps.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` 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#cf6ba6cc0be0bb5f5684` looks like:

```
[[deps.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:

```
[[deps.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 e.g. `pkg> add Example; pin Example` looks like:

```
[[deps.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 packages 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 this:

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

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

[[deps.B]]
uuid = "f41f7b98-334e-11e9-1257-49272045fb24"
[[deps.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 to be explicit about which B package A depends on.

Part XII

11. REPL Mode Reference

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

Chapter 52

package commands

```
add [--preserve=<opt>] [-w|--weak] [-e|--extra] 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 the active environment is a package (the Project has both `name` and `uuid` fields) compat entries will be added automatically with a lower bound of the added version.

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. If the package is not located at the top of the git repository, a subdirectory can be specified with `path:subdir/path`.

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
`installed`	Like `all` except also only add versions that are already installed
`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_installed`	Like `tiered` except first try to add only installed versions
`tiered`	Use the tier that will preserve the most version information while allowing version resolution to succeed (this is the default)

Note: To make the default strategy `tiered_installed` set the env var `JULIA_PKG_PRESERVE_TIERED_INSTALLED` to true.

After the installation of new packages the project will be precompiled. For more information see `pkg> ?precompile`.

With the `installed` strategy the newly added packages will likely already be precompiled, but if not this may be because either the combination of package versions resolved in this environment has not been resolved and precompiled before, or the precompile cache has been deleted by the LRU cache storage (see `JULIA_MAX_NUM_PRECOMPILE_FILES`).

Examples

```
pkg> add Example
pkg> add --preserve=all Example
pkg> add --weak Example
pkg> add --extra 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 "git@github.com:JuliaLang/Example.jl.git"#master
pkg> add https://github.com/Company/MonoRepo:juliapkgs/Package.jl
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`.

```
compat [pkg] [compat_string]
compat
compat --current
compat <pkg> --current
```

Edit project [compat] entries directly, or via an interactive menu by not specifying any arguments. Use –current flag to automatically populate missing compat entries with currently resolved versions. When used alone, applies to all packages missing compat entries. When combined with a package name, applies only to that package. When directly editing use tab to complete the package name and any existing compat entry. Specifying a package with a blank compat entry will remove the entry. After changing compat entries a `resolve` will be attempted to check whether the current environment is compliant with the new compat rules.

```
[dev|develop] [--preserve=<opt>] [--shared|--local] pkg[=uuid] ...
[dev|develop] [--preserve=<opt>] path
```

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 is not supported for paths, only registered packages.

This operation is undone by `free`.

The preserve strategies offered by add are also available via the `preserve` argument. See add for more information.

Examples

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

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

Free pinned packages, 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. Specifying `--all` will free all dependencies (direct and indirect).

```
generate pkgname
```

Create a minimal project called pkgname in the current folder. For more featureful package creation, please see `PkgTemplates.jl`.

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

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.. Specifying `--all` will pin all dependencies (direct and indirect).

Examples

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

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

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. All packages can be removed by passing `--all`.

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

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. All packages can be removed by passing `--all`.

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

Run the tests for package pkg, or for the current project (which thus needs to be a package) if pkg is omitted. 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|update] [-p|--project]  [opts] pkg[=uuid] [@version] ...
[up|update] [-m|--manifest] [opts] pkg[=uuid] [@version] ...

opts: --major | --minor | --patch | --fixed
      --preserve=<all/direct/none>
```

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.

After any package updates the project will be precompiled. For more information see `pkg> ?precompile`.

Chapter 53

`registry` commands

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

Add package registries `reg...` to the user depot. Without arguments it adds known registries, i.e. the General registry and registries served by the configured package server.

Examples

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

```
registry [rm|remove] reg...
```

Remove package registries `reg...`.

Examples

```
pkg> registry [rm|remove] General
```

```
registry [st|status]
```

Display information about installed registries.

Examples

```
pkg> registry status
```

```
registry [up|update]
registry [up|update] reg...
```

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

Examples

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

Chapter 54

Other commands

```
activate
activate [--shared] path
activate --temp
activate - (activates the previously active environment)
```

Activate the environment at the given `path`, or return to the default environment if no `path` is specified. When called with no arguments, this returns you to the default shared environment (typically `@v#.#` in `~/.julia/environments/v#.#/`), which is the standard way to "deactivate" a project environment.

The active environment is the environment that is modified by executing package commands. Activating an environment only affects the current Julia session and does not persist when you restart Julia (unless you use the `--project` startup flag).

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.

Use the `--temp` option to create temporary environments which are removed when the julia process is exited.

Use a single `-` to activate the previously active environment.

```
gc [-v|--verbose] [--all]
```

Free disk space by garbage collecting packages not used for a significant time. The `--all` option will garbage collect all packages which can not be immediately reached from any existing project. Use verbose mode for detailed output.

```
[?|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] [--workspace] [--julia_version_strict]
instantiate [-v|--verbose] [--workspace] [--julia_version_strict] [-m|--manifest]
instantiate [-v|--verbose] [--workspace] [--julia_version_strict] [-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. If `--workspace` is given, all dependencies in the workspace will be downloaded. If `--julia_version_strict` is given, manifest version check failures will error instead of log warnings.

After packages have been installed the project will be precompiled. For more information see pkg> `?precompile`.

```
precompile [--workspace]
precompile [--workspace] pkgs...
```

Precompile all or specified dependencies of the project in parallel. The `startup.jl` file is disabled during precompilation unless julia is started with `--startup-file=yes`. The `workspace` option will precompile all packages in the workspace and not only the active project.

Errors will only throw when precompiling the top-level dependencies, given that not all manifest dependencies may be loaded by the top-level dependencies on the given system.

This method is called automatically after any Pkg action that changes the manifest. Any packages that have previously errored during precompilation won't be retried in auto mode until they have changed. To disable automatic precompilation set the environment variable `JULIA_PKG_PRECOMPILE_AUTO=0`. To manually control the number of tasks used set the environment variable `JULIA_NUM_PRECOMPILE_TASKS`.

```
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.

```
[st|status] [-d|--diff] [--workspace] [-o|--outdated] [--deprecated] [pkgs...]
[st|status] [-d|--diff] [--workspace] [-o|--outdated] [--deprecated] [-p|--project] [pkgs...]
[st|status] [-d|--diff] [--workspace] [-o|--outdated] [--deprecated] [-m|--manifest] [pkgs...]
[st|status] [-d|--diff] [--workspace] [-e|--extensions] [-p|--project] [pkgs...]
[st|status] [-d|--diff] [--workspace] [-e|--extensions] [-m|--manifest] [pkgs...]
[st|status] [-c|--compat] [pkgs...]
```

Show the status of the current environment. Packages marked with ^ have new versions that may be installed, e.g. via pkg> up. Those marked with ⌅ have new versions available, but cannot be installed due to compatibility constraints. To see why use pkg> `status --outdated` which shows any packages that are not at their latest version and if any packages are holding them back. Packages marked with `[yanked]` have been yanked from the registry and should be updated or removed. Packages marked with `[deprecated]` are no longer maintained.

Use pkg> `status --deprecated` to show only deprecated packages along with deprecation information such as the reason and alternative packages (if provided by the registry).

Use pkg> `status --extensions` to show dependencies with extensions and what extension dependencies of those that are currently loaded.

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. The `--compat` option alone shows project compat entries. The `--workspace` option shows the (merged) status of packages in the workspace.

> **Julia 1.8**
>
> The ^ and ⌅ indicators were added in Julia 1.8. The `--outdated` and `--compat` options require at least Julia 1.8.

Part XIII

13. API Reference

This section describes the functional API for interacting with Pkg.jl. It is recommended to use the functional API, rather than the Pkg REPL mode, for non-interactive usage, for example in scripts.

Chapter 55

General API Reference

Certain options are generally useful and can be specified in any API call. You can specify these options by setting keyword arguments.

55.1 Redirecting output

Use the `io::IOBuffer` keyword argument to redirect Pkg output. For example, `Pkg.add("Example"; io=devnull)` will discard any output produced by the add call.

Chapter 56

Package API Reference

In the Pkg 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 functional API, 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.add` – Function.

```
Pkg.add(pkg::Union{String, Vector{String}}; preserve=PRESERVE_TIERED, target::Symbol=:deps)
Pkg.add(pkg::Union{PackageSpec, Vector{PackageSpec}}; preserve=PRESERVE_TIERED,
↪  target::Symbol=:deps)
```

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.

If the active environment is a package (the Project has both `name` and `uuid` fields) compat entries will be added automatically with a lower bound of the added version.

To add as a weak dependency (in the `[weakdeps]` field) set the kwarg `target=:weakdeps`. To add as an extra dep (in the `[extras]` field) set `target=:extras`.

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):

Note

To change the default strategy to `PRESERVE_TIERED_INSTALLED` set the env var `JULIA_PKG_PRESERVE_TIERED_INSTALLED` to true.

After the installation of new packages the project will be precompiled. For more information see `pkg> ?precompile`.

With the `PRESERVE_ALL_INSTALLED` strategy the newly added packages will likely already be precompiled, but if not this may be because either the combination of package versions resolved in this environment has

Value	Description
PRESERVE_ALL_INSTALLED	Like PRESERVE_ALL and only add those already installed
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_INSTALLED	Like PRESERVE_TIERED except PRESERVE_ALL_INSTALLED is tried first
PRESERVE_TIERED	Use the tier that will preserve the most version information while allowing version resolution to succeed (this is the default)

not been resolved and precompiled before, or the precompile cache has been deleted by the LRU cache storage (see `JULIA_MAX_NUM_PRECOMPILE_FILES`).

Julia 1.9

The `PRESERVE_TIERED_INSTALLED` and `PRESERVE_ALL_INSTALLED` strategies requires at least Julia 1.9.

Julia 1.11

The `target` kwarg requires at least Julia 1.11.

Examples

```
Pkg.add("Example") # Add a package from registry
Pkg.add("Example", target=:weakdeps) # Add a package as a weak dependency
Pkg.add("Example", target=:extras) # Add a package to the `[extras]` list
Pkg.add("Example"; preserve=Pkg.PRESERVE_ALL) # Add the `Example` package and strictly preserve
↪  existing dependencies
Pkg.add(name="Example", version="0.3") # Specify version; latest release in the 0.3 series
Pkg.add(name="Example", version="0.3.1") # Specify version; exact release
Pkg.add(url="https://github.com/JuliaLang/Example.jl", rev="master") # From url to remote
↪  gitrepo
Pkg.add(url="/remote/mycompany/juliapackages/OurPackage") # From path to local gitrepo
Pkg.add(url="https://github.com/Company/MonoRepo", subdir="juliapkgs/Package.jl") # With subdir
```

After the installation of new packages the project will be precompiled. See more at Environment Precompilation.

See also PackageSpec, `Pkg.develop`.

source

`Pkg.develop` – Function.

```
Pkg.develop(pkg::Union{String, Vector{String}}; io::IO=stderr, preserve=PRESERVE_TIERED,
↪  installed=false)
Pkg.develop(pkgs::Union{PackageSpec, Vector{PackageSpec}}; io::IO=stderr,
↪  preserve=PRESERVE_TIERED, installed=false)
```

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 `joinpath(DEPOT_PATH[1],"dev")` being the default.

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

The preserve strategies offered by `Pkg.add` are also available via the `preserve` kwarg. See `Pkg.add` for more information.

Examples

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

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

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

See also `PackageSpec`, `Pkg.add`.

source

`Pkg.activate` – Function.

```
Pkg.activate([s::String]; shared::Bool=false, io::IO=stderr)
Pkg.activate(; temp::Bool=false, shared::Bool=false, io::IO=stderr)
```

Activate the environment at `s`, or return to the default environment if no argument is given. The active environment is the environment that is modified by executing package commands. Activating an environment only affects the current Julia session and does not persist when you restart Julia (unless you use the `--project` startup flag).

Returning to the default environment

If no argument is given to `activate`, this returns you to the default shared environment (typically `@v#.#` in `~/.julia/environments/v#.#/`). This is the standard way to "deactivate" a project environment and return to your base package setup. There is no separate `deactivate` command—`Pkg.activate()` with no arguments serves this purpose.

Activating a path

When `s` is provided, 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 `temp` is `true` this will create and activate a temporary environment which will be deleted when the julia process is exited.
- 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.

Examples

```
# Return to default environment (deactivate current project)
Pkg.activate()

# Activate a project in a specific directory
Pkg.activate("local/path")

# Activate a developed package by name
Pkg.activate("MyDependency")

# Create and activate a temporary environment
Pkg.activate(; temp=true)
```

See also `LOAD_PATH`.

source

`Pkg.rm` – Function.

```
Pkg.rm(pkg::Union{String, Vector{String}}; mode::PackageMode = PKGMODE_PROJECT)
Pkg.rm(pkg::Union{PackageSpec, Vector{PackageSpec}}; mode::PackageMode = PKGMODE_PROJECT)
```

Remove a package from the current project. If `mode` is equal to `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,
↪  preserve::PreserveLevel)
Pkg.update(pkg::Union{String, Vector{String}})
Pkg.update(pkg::Union{PackageSpec, Vector{PackageSpec}})
```

If no positional 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).

If packages are given as positional arguments, the `preserve` argument can be used to control what other packages are allowed to update:

- `PRESERVE_ALL` (default): Only allow `pkg` to update.
- `PRESERVE_DIRECT`: Only allow `pkg` and indirect dependencies that are not a direct dependency in the project to update.
- `PRESERVE_NONE`: Allow `pkg` and all its indirect dependencies to update.

After any package updates the project will be precompiled. See more at Environment Precompilation.

See also `PackageSpec`, `PackageMode`, `UpgradeLevel`.

source

`Pkg.test` – Function.

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

Keyword arguments:

- `coverage::Union{Bool,String}=false`: enable or disable generation of coverage statistics for the tested package. If a string is passed it is passed directly to `--code-coverage` in the test process so e.g. "user" will test all user code.
- `allow_reresolve::Bool=true`: allow Pkg to reresolve the package versions in the test environment
- `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 1.9

`allow_reresolve` requires at least Julia 1.9.

Julia 1.9

Passing a string to `coverage` requires at least Julia 1.9.

Run the tests for the given package(s), or for the current project if no positional argument is given to `Pkg.test` (the current project would need to be a package). The package is tested by running its `test/runtests.jl` file.

The tests are run in a temporary environment that also includes the test specific dependencies of the package. The versions of dependencies in the current project are used for the test environment unless there is a compatibility conflict between the version of the dependencies and the test-specific dependencies. In that case, if `allow_reresolve` is `false` an error is thrown and if `allow_reresolve` is `true` a feasible set of versions of the dependencies is resolved and used.

Test-specific dependnecies are declared in the project file as:

```
[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`. The tests can be run as if different command line arguments were passed to julia by passing the arguments instead to the `julia_args` keyword argument, e.g.

```
Pkg.test("foo"; julia_args=["--inline"])
```

To pass some command line arguments to be used in the tests themselves, pass the arguments to the `test_args` keyword argument. These could be used to control the code being tested, or to control the tests in some way. For example, the tests could have optional additional tests:

```
if "--extended" in ARGS
    @test some_function()
end
```

which could be enabled by testing with

```
Pkg.test("foo"; test_args=["--extended"])
```

source

`Pkg.build` – Function.

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

Keyword arguments:

- `verbose::Bool=false`: print the build output to `stdout`/`stderr` instead of redirecting to the `build.log` file.
- `allow_reresolve::Bool=true`: allow Pkg to reresolve the package versions in the build environment

Julia 1.13

`allow_reresolve` requires at least Julia 1.13.

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.

The build takes place in a new process matching the current process with default of `startup-file=no`. If using the startup file (`~/.julia/config/startup.jl`) is desired, start julia with an explicit `--startup-file=yes`.

source

`Pkg.pin` – Function.

```
Pkg.pin(pkg::Union{String, Vector{String}}; io::IO=stderr, all_pkgs::Bool=false)
Pkg.pin(pkgs::Union{PackageSpec, Vector{PackageSpec}}; io::IO=stderr, all_pkgs::Bool=false)
```

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 automatically updated: if pkg is tracking a path, or a repository, those remain tracked but will not update. To get updates from the origin path or remote repository the package must first be freed.

> **Julia 1.7**
>
> The `all_pkgs` kwarg was introduced in julia 1.7.

Examples

```
# Pin a package to its current version
Pkg.pin("Example")

# Pin a package to a specific version
Pkg.pin(name="Example", version="0.3.1")

# Pin all packages in the project
Pkg.pin(all_pkgs = true)
```

source

`Pkg.free` – Function.

```
Pkg.free(pkg::Union{String, Vector{String}}; io::IO=stderr, all_pkgs::Bool=false)
Pkg.free(pkgs::Union{PackageSpec, Vector{PackageSpec}}; io::IO=stderr, all_pkgs::Bool=false)
```

If pkg is pinned, remove the pin. If pkg is tracking a path, e.g. after `Pkg.develop`, go back to tracking registered versions. To free all dependencies set `all_pkgs=true`.

> **Julia 1.7**
>
> The `all_pkgs` kwarg was introduced in julia 1.7.

Examples

```
# Free a single package (remove pin or stop tracking path)
Pkg.free("Package")

# Free multiple packages
Pkg.free(["PackageA", "PackageB"])

# Free all packages in the project
Pkg.free(all_pkgs = true)
```

source

`Pkg.instantiate` – Function.

```
Pkg.instantiate(; verbose = false, workspace=false, io::IO=stderr, julia_version_strict=false)
```

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. `workspace=true` will also instantiate all projects in the workspace. If no `Project.toml` exist in the current active project, create one with all the dependencies in the manifest and instantiate the resulting project. `julia_version_strict=true` will turn manifest version check failures into errors instead of logging warnings.

After packages have been installed the project will be precompiled. See more and how to disable auto-precompilation at Environment Precompilation.

> **Julia 1.12**
>
> The `julia_version_strict` keyword argument requires at least Julia 1.12.

source

`Pkg.resolve` – Function.

```
Pkg.resolve(; io::IO=stderr)
```

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

source

`Pkg.gc` – Function.

```
Pkg.gc(; collect_delay::Period=Day(7), io::IO=stderr)
```

Garbage-collect package and artifact installations by sweeping over all known `Manifest.toml` and `Artifacts.toml` files, noting those that have been deleted, and then finding artifacts and packages that are thereafter not used by any other projects, marking them as "orphaned". This method will only remove orphaned objects (package versions, artifacts, and scratch spaces) that have been continually un-used for a period of `collect_delay`; which defaults to seven days.

To disable automatic garbage collection, you can set the environment variable `JULIA_PKG_GC_AUTO` to `"false"` before starting Julia or call `API.auto_gc(false)`.

source

`Pkg.status` – Function.

```
Pkg.status([pkgs...]; outdated::Bool=false, mode::PackageMode=PKGMODE_PROJECT, diff::Bool=false,
          compat::Bool=false, extensions::Bool=false, workspace::Bool=false, io::IO=stdout)
```

Print out the status of the project/manifest.

Packages marked with ^ have new versions that can be installed, e.g. via `Pkg.update`. Those marked with ⌅ have new versions available, but cannot be installed due to compatibility conflicts with other packages. To see why, set the keyword argument `outdated=true`. Packages marked with `[yanked]` are yanked versions that should be updated or replaced as they may contain bugs or security vulnerabilities.

Setting `outdated=true` will only show packages that are not on the latest version, their maximum version and why they are not on the latest version (either due to other packages holding them back due to compatibility constraints, or due to compatibility in the project file). As an example, a status output like:

```
julia> Pkg.status(; outdated=true)
Status `Manifest.toml`
^ [a8cc5b0e] Crayons v2.0.0 [<v3.0.0], (<v4.0.4)
⌅ [b8a86587] NearestNeighbors v0.4.8 (<v0.4.9) [compat]
⌅ [2ab3a3ac] LogExpFunctions v0.2.5 (<v0.3.0): SpecialFunctions
```

means that the latest version of Crayons is 4.0.4 but the latest version compatible with the `[compat]` section in the current project is 3.0.0. The latest version of NearestNeighbors is 0.4.9 but due to compat constrains in the project it is held back to 0.4.8. The latest version of LogExpFunctions is 0.3.0 but SpecialFunctions is holding it back to 0.2.5.

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 `ext=true` will show dependencies with extensions and what extension dependencies of those that are currently loaded.

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.

Setting `workspace=true` will show the (merged) status of packages in the workspace.

See `Pkg.project` and `Pkg.dependencies` to get the project/manifest status as a Julia object instead of printing it.

Julia 1.8

The ^ and ⌅ indicators were added in Julia 1.8. The `outdated` keyword argument requires at least Julia 1.8.

source

`Pkg.compat` – Function.

```
Pkg.compat()
```

Interactively edit the [compat] entries within the current Project.

```
Pkg.compat(pkg::String, compat::String)
```

Set the [compat] string for the given package within the current Project.

See Compatibility for more information on the project [compat] section.

source

Pkg.precompile – Function.

```
Pkg.precompile(; strict::Bool=false, timing::Bool=false)
Pkg.precompile(pkg; strict::Bool=false, timing::Bool=false)
Pkg.precompile(pkgs; strict::Bool=false, timing::Bool=false)
Pkg.precompile(f, args...; kwargs...)
```

Precompile all or specific dependencies of the project in parallel.

Set `timing=true` to show the duration of the precompilation of each dependency.

To delay autoprecompilation of multiple Pkg actions until the end use. This may be most efficient while manipulating the environment in various ways.

```
Pkg.precompile() do
    # Pkg actions here
end
```

Note

Errors will only throw when precompiling the top-level dependencies, given that not all manifest dependencies may be loaded by the top-level dependencies on the given system. This can be overridden to make errors in all dependencies throw by setting the kwarg `strict` to `true`

Note

This method is called automatically after any Pkg action that changes the manifest. Any packages that have previously errored during precompilation won't be retried in auto mode until they have changed. To disable automatic precompilation set `ENV["JULIA_PKG_PRECOMPILE_AUTO"]=0`. To manually control the number of tasks used set `ENV["JULIA_NUM_PRECOMPILE_TASKS"]`.

Julia 1.8

Specifying packages to precompile requires at least Julia 1.8.

Julia 1.9

Timing mode requires at least Julia 1.9.

Julia 1.13

The `Pkg.precompile(f, args...; kwargs...)` do-block syntax requires at least Julia 1.13.

Examples

```
Pkg.precompile()
Pkg.precompile("Foo")
Pkg.precompile(["Foo", "Bar"])
```

source

Pkg.autoprecompilation_enabled – Function.

```
Pkg.autoprecompilation_enabled(state::Bool)
```

Enable or disable automatic precompilation for Pkg operations.

When `state` is `true` (default), Pkg operations that modify the project environment will automatically trigger precompilation of affected packages. When `state` is `false`, automatic precompilation is disabled and packages will only be precompiled when explicitly requested via `Pkg.precompile`.

This setting affects the global state and persists across Pkg operations in the same Julia session. It can be used in combination with `Pkg.precompile` do-syntax for more fine-grained control over when precompilation occurs.

Julia 1.13

This function requires at least Julia 1.13.

Examples

```
# Disable automatic precompilation
Pkg.autoprecompilation_enabled(false)
Pkg.add("Example")  # Will not trigger auto-precompilation
Pkg.precompile()    # Manual precompilation

# Re-enable automatic precompilation
Pkg.autoprecompilation_enabled(true)
Pkg.add("AnotherPackage")  # Will trigger auto-precompilation
```

See also `Pkg.precompile`.

source

Pkg.offline – Function.

```
Pkg.offline(b::Bool=true)
```

Enable (b=`true`) or disable (b=`false`) offline mode.

In offline mode Pkg tries to do as much as possible without connecting to internet. For example, when adding a package Pkg only considers versions that are already downloaded in version resolution.

To work in offline mode across Julia sessions you can set the environment variable `JULIA_PKG_OFFLINE` to `"true"` before starting Julia.

source

`Pkg.why` – Function.

```
Pkg.why(pkg::Union{String, Vector{String}}; workspace::Bool=false)
Pkg.why(pkg::Union{PackageSpec, Vector{PackageSpec}}; workspace::Bool=false)
```

Show the reason why this package is in the manifest. The output is all the different ways to reach the package through the dependency graph starting from the dependencies. If `workspace` is true, this will consider all projects in the workspace and not just the active one.

Julia 1.9

This function requires at least Julia 1.9.

source

`Pkg.dependencies` – Function.

```
Pkg.dependencies()::Dict{UUID, PackageInfo}
```

This feature is considered experimental.

Query the dependency graph of the active project. 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)
`tree_hash`	A file hash of the package directory tree
`is_direct_dep`	The package is a direct dependency
`is_pinned`	Whether a package is pinned
`is_tracking_path`	Whether a package is tracking a path
`is_tracking_repo`	Whether a package is tracking a repository
`is_tracking_registry`	Whether a package is being tracked by registry i.e. not by path nor by repository
`git_revision`	The git revision when tracking by repository
`git_source`	The git source when tracking by repository
`source`	The directory containing the source code for that package
`dependencies`	The dependencies of that package as a vector of UUIDs

source

`Pkg.respect_sysimage_versions` – Function.

```
Pkg.respect_sysimage_versions(b::Bool=true)
```

Enable (`b=true`) or disable (`b=false`) respecting versions that are in the sysimage (enabled by default).

If this option is enabled, Pkg will only install packages that have been put into the sysimage (e.g. via PackageCompiler) at the version of the package in the sysimage. Also, trying to add a package at a URL or `develop` a package that is in the sysimage will error.

source

`Pkg.project` – Function.

```
Pkg.project()::ProjectInfo
```

This feature 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
`ispackage`	Whether the project is a package (has a name and uuid)
`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.

```
undo()
```

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.

```
redo()
```

Redoes the changes from the latest undo.

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")

# Use HTTPS for GitHub (default, good for most users)
julia> Pkg.setprotocol!(domain = "github.com", protocol = "https")

# Reset to default (let package developer decide)
julia> Pkg.setprotocol!(domain = "github.com", protocol = nothing)

# Set protocol for custom domain without specifying protocol
julia> Pkg.setprotocol!(domain = "gitlab.mycompany.com")

# Use Git protocol for a custom domain
julia> Pkg.setprotocol!(domain = "gitlab.mycompany.com", protocol = "git")
```

source

`Pkg.readonly` – Function.

```
readonly([state::Bool], [ctx::Context])
```

Get or set the readonly state of the current environment.

Examples

```
julia> Pkg.readonly()  # check current readonly state
false

julia> Pkg.readonly(true)  # enable readonly mode
false  # returns previous state

julia> Pkg.readonly()
true

julia> Pkg.readonly(false)  # disable readonly mode
true
```

source

`Pkg.PackageSpec` – Type.

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

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`. If the version is given as a `String` this means that unspecified versions are "free", for example `version="0.5"` allows any version `0.5.x` to be installed. If given as a `VersionNumber`, the exact version is used, for example `version=v"0.5.3"`.
- 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 `subdir` which can be used when adding a package that is not in the root of a repository.

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

Many functions that take a `PackageSpec` or a `Vector{PackageSpec}` can be called with a more concise notation with `NamedTuple`s. For example, `Pkg.add` can be called either as the explicit or concise versions as:

Explicit	Concise
`Pkg.add(PackageSpec(name="Package"))`	`Pkg.add(name = "Package")`
`Pkg.add(PackageSpec(url="www.myhost.com/MyPkg")))`	`Pkg.add(url="www.myhost.com/MyPkg")`
`Pkg.add([PackageSpec(name="Package"), PackageSpec(path="/MyPkg"])`	`Pkg.add([(;name="Package"), (;path="/MyPkg")])`

Below is a comparison between the REPL mode and the functional API:

REPL	API
`Package`	`PackageSpec("Package")`
`Package@0.2`	`PackageSpec(name="Package", version="0.2")`
`-`	`PackageSpec(name="Package", version=v"0.2.1")`
`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")`
`--major Package`	`PackageSpec(name="Package", version=UPLEVEL_MAJOR)`

source

`Pkg.PackageMode` – Type.

```
PackageMode
```

An enum with the instances

- `PKGMODE_MANIFEST`
- `PKGMODE_PROJECT`

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

source

`Pkg.UpgradeLevel` – Type.

```
UpgradeLevel
```

An enum with the instances

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

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

source

Chapter 57

Registry API Reference

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

`Pkg.RegistrySpec` – Type.

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

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

- The `name` of the registry.
- The registry's unique `uuid`.
- The `url` to the registry.
- A local `path`.

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

Many functions that take a `RegistrySpec` can be called with a more concise notation with keyword arguments. For example, `Pkg.Registry.add` can be called either as the explicit or concise versions as:

Explicit	Concise
`Pkg.Registry.add(RegistrySpec(name="General"))`	`Pkg.Registry.add(name = "General")`
`Pkg.Registry.add(RegistrySpec(url="https://github.com/JuliaRegistries/General.git")))`	`Pkg.Registry.add(url = "https://github.com/JuliaRegistries/General.git")`

Below is a comparison between the REPL mode and the functional API::

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

source

`Pkg.Registry.add` – Function.

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

Add new package registries.

The no-argument `Pkg.Registry.add()` will install the default registries.

Examples

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

source

`Pkg.Registry.rm` – Function.

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

Remove registries.

Examples

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

source

`Pkg.Registry.update` – Function.

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

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

Examples

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

source

`Pkg.Registry.status` – Function.

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

Display information about available registries.

Examples

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

source

Chapter 58

Artifacts API Reference

`Pkg.PkgArtifacts.create_artifact` – Function.

```
create_artifact(f::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.PkgArtifacts.remove_artifact` – Function.

```
remove_artifact(hash::SHA1; honor_overrides::Bool=false)
```

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.PkgArtifacts.verify_artifact` – Function.

```
verify_artifact(hash::SHA1; honor_overrides::Bool=false)
```

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.PkgArtifacts.bind_artifact!` – Function.

```
bind_artifact!(artifacts_toml::String, name::String, hash::SHA1;
               platform::Union{AbstractPlatform,Nothing} = nothing,
               download_info::Union{Vector{Tuple},Nothing} = nothing,
               lazy::Bool = false,
               force::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 vector that contains tuples 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.PkgArtifacts.unbind_artifact!` – Function.

```
unbind_artifact!(artifacts_toml::String, name::String; platform = nothing)
```

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

source

`Pkg.PkgArtifacts.download_artifact` – Function.

```
download_artifact(tree_hash::SHA1, tarball_url::String, tarball_hash::String;
                  verbose::Bool = false, io::IO=stderr)
```

Download/install an artifact into the artifact store. Returns `true` on success, returns an error object on failure.

> **Julia 1.8**
>
> As of Julia 1.8 this function returns the error object rather than `false` when failure occurs

source

`Pkg.PkgArtifacts.ensure_artifact_installed` – Function.

```
ensure_artifact_installed(name::String, artifacts_toml::String;
                          platform::AbstractPlatform = HostPlatform(),
                          pkg_uuid::Union{Base.UUID,Nothing}=nothing,
                          verbose::Bool = false,
                          quiet_download::Bool = false,
                          io::IO=stderr)
```

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.PkgArtifacts.ensure_all_artifacts_installed` – Function.

```
ensure_all_artifacts_installed(artifacts_toml::String;
                                platform = HostPlatform(),
                                pkg_uuid = nothing,
                                include_lazy = false,
                                verbose = false,
                                quiet_download = false,
                                io::IO=stderr)
```

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.

This function is deprecated and should be replaced with the following snippet:

```
artifacts = select_downloadable_artifacts(artifacts_toml; platform, include_lazy)
for name in keys(artifacts)
    ensure_artifact_installed(name, artifacts[name], artifacts_toml; platform=platform)
end
```

> **Warning**
>
> This function is deprecated in Julia 1.6 and will be removed in a future version. Use `select_downloadable_artifacts()` and `ensure_artifact_installed()` instead.

source

`Pkg.PkgArtifacts.archive_artifact` – Function.

```
archive_artifact(hash::SHA1, tarball_path::String; honor_overrides::Bool=false)
```

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

source

Chapter 59

Package Server Authentication Hooks

`Pkg.PlatformEngines.register_auth_error_handler` – Function.

```
register_auth_error_handler(urlscheme::Union{AbstractString, Regex}, f)
```

Registers `f` as the topmost handler for failures in package server authentication.

A handler is only invoked if `occursin(urlscheme, url)` is true (where `url` is the URL Pkg is currently trying to download.)

`f` must be a function that takes three input arguments `(url, pkgserver, err)`, where `url` is the URL currently being downloaded, `pkgserver = Pkg.pkg_server()` the current package server, and `err` is one of `no-auth-file`, `insecure-connection`, `malformed-file`, `no-access-token`, `no-refresh-key` or `insecure-refresh-url`.

The handler `f` needs to return a tuple of `Bool`s `(handled, should_retry)`. If handled is `false`, the next handler in the stack will be called, otherwise handling terminates; `get_auth_header` is called again if `should_retry` is `true`.

`register_auth_error_handler` returns a zero-arg function that can be called to deregister the handler.

source

`Pkg.PlatformEngines.deregister_auth_error_handler` – Function.

```
deregister_auth_error_handler(urlscheme::Union{AbstractString, Regex}, f)
```

Removes `f` from the stack of authentication error handlers.

source

Part XIV

14. Package and Storage Server Protocol Reference

The Julia Package Server Protocol (Pkg Protocol) and the Package Storage Server Protocol (Storage Protocol) define how Julia's package manager, Pkg, obtains and manages packages and their associated resources. They aim to enhance the Julia package ecosystem, making it more efficient, reliable, and user-friendly, avoiding potential points of failure, and ensuring the permanent availability of package versions and artifacts, which is paramount for the stability and reproducibility of Julia projects.

The Pkg client, by default, gets all resources over HTTPS from a single open source service run by the Julia community. This service for serving packages is additionally backed by multiple independent storage services which interface with proprietary origin services (GitHub, etc.) and guarantee persistent availability of resources into the future.

The protocols also aim to address some of the limitations that existed prior to its introduction.

- **Vanishing Resources.** It is possible for authors to delete code repositories of registered Julia packages. Without some kind of package server, no one can install a package which has been deleted. If someone happens to have a current fork of a deleted package, that can be made the new official repository for the package, but the chances of them having no or outdated forks are high. An even worse situation could happen for artifacts since they tend not to be kept in version control and are much more likely to be served from "random" web servers at a fixed URL with content changing over time. Artifact publishers are unlikely to retain all past versions of artifacts, so old versions of packages that depend on specific artifact content will not be reproducible in the future unless we do something to ensure that they are kept around after the publisher has stopped hosting them. By storing all package versions and artifacts in a single place, we can ensure that they are available forever.

- **Usage Insights.** It is valuable for the Julia community to know how many people are using Julia or what the relative popularity of different packages and operating systems is. Julia uses GitHub to host its ecosystem. GitHub - a commercial, proprietary service - has this information but does not make it available to the Julia community. We are of course using GitHub for free, so we can't complain, but it seems unfortunate that a commercial entity has this valuable information while the open source community remains in the dark. The Julia community really could use insight into who is using Julia and how, so that we can prioritize packages and platforms, and give real numbers when people ask "how many people are using Julia?"

- **Decoupling from Git and GitHub.** Prior to this, Julia package ecosystem was very deeply coupled to git and was even specialized on GitHub specifically in many ways. The Pkg and Storage Protocols allowed us to decouple ourselves from git as the primary mechanism for getting packages. Now Julia continues to support using git, but does not require it just to install packages from the default public registry anymore. This decoupling also paves the way for supporting other version control systems in the future, making git no longer so special. Special treatment of GitHub will also go away since we get the benefits of specializing for GitHub (fast tarball downloads) directly from the Pkg protocols.

- **Firewall problems.** Prior to this, Pkg's need to connect to arbitrary servers using a miscellany of protocols caused several problems with firewalls. A large set of protocols and an unbounded list of servers needed to be whitelisted just to support default Pkg operation. If Pkg only needed to talk to a single service over a single, secure protocol (i.e. HTTPS), then whitelisting Pkg for standard use would be dead simple.

Chapter 60

Protocols & Services

1. **Pkg Protocol:** what Julia Pkg Clients speak to Pkg Servers. The Pkg Server serves all resources that Pkg Clients need to install and use registered packages, including registry data, packages and artifacts. It is designed to be easily horizontally scalable and not to have any hard operational requirements: if service is slow, just start more servers; if a Pkg Server crashes, forget it and boot up a new one.

2. **Storage Protocol:** what Pkg Servers speak to get resources from Storage Services. Julia clients do not interact with Storage services directly and multiple independent Storage Services can symmetrically (all are treated equally) provide their service to a given Pkg Server. Since Pkg Servers cache what they serve to Clients and handle convenient content presentation, Storage Services can expose a much simpler protocol: all they do is serve up complete versions of registries, packages and artifacts, while guaranteeing persistence and completeness. Persistence means: once a version of a resource has been served, that version can be served forever. Completeness means: if the service serves a registry, it can serve all package versions referenced by that registry; if it serves a package version, it can serve all artifacts used by that package.

Both protocols work over HTTPS, using only GET and HEAD requests. As is normal for HTTP, HEAD requests are used to get information about a resource, including whether it would be served, without actually downloading it. As described in what follows, the Pkg Protocol is client-to-server and may be unauthenticated, use basic auth, or OpenID; the Storage Protocol is server-to-server only and uses mutual authentication with TLS certificates.

The following diagram shows how these services interact with each other and with external services such as GitHub, GitLab and BitBucket for source control, and S3 and HDFS for long-term persistence:

```
                   ┌───────────┐
                   │ Amazon S3 │
                   │  Storage  │
                   └───────────┘
                         ▲
                         ║
                         ▼
       Storage     ╔═══════════╗       ┌───────────┐
```

```
                 Pkg           Protocol  ║  Storage   ║   ┌──▶│  GitHub   │
               Protocol              ┌──▶║ Service A ║──┤   └───────────┘
┏━━━━━━━━━━━━┓     ┏━━━━━━━━━━━━┓    │   ╚═══════════╝   │   ┌───────────┐
┃ Pkg Client ┃────▶┃ Pkg Server ┃────┤   ╔═══════════╗   ├──▶│  GitLab   │
┗━━━━━━━━━━━━┛     ┗━━━━━━━━━━━━┛    │   ║  Storage   ║   │   └───────────┘
                                     └──▶║ Service B ║──┤   ┌───────────┐
                                         ╚═══════════╝   └──▶│ BitBucket │
                                               ▲              └───────────┘
                                               ║
                                               ▼
                                         ┌───────────┐
                                         │   HDFS    │
                                         │  Cluster  │
                                         └───────────┘
```

Each Julia Pkg Client is configured to talk to a Pkg Server. By default, they talk to `pkg.julialang.org`, a public, unauthenticated Pkg Server. If the environment variable `JULIA_PKG_SERVER` is set, the Pkg Client connects to that host instead. For example, if `JULIA_PKG_SERVER` is set to `pkg.company.com` then the Pkg Client will connect to `https://pkg.company.com`. So in typical operation, a Pkg Client will no longer rely on `libgit2` or a git command-line client, both of which have been an ongoing headache, especially behind firewalls and on Windows. If fact, git will only be necessary when working with git-hosted registries and unregistered packages - those will continue to work as they have previously, fetched using git.

While the default Pkg Server at `pkg.julialang.org` is unauthenticated, other parties may host Pkg Server instances elsewhere, authenticated or unauthenticated, public or private, as they wish. People can connect to those servers by setting the `JULIA_PKG_SERVER` variable. There will be a configuration file for providing authentication information to Pkg Servers using either basic auth or OpenID. The Pkg Server implementation will be open source and have minimal operational requirements. Specifically, it needs:

1. The ability to accept incoming connections on port 443;
2. The ability to connect to a configurable set of Storage Services;
3. Temporary disk storage for caching resources (registries, packages, artifacts).

A Pkg Service may be backed by more than one actual server, as is typical for web services. The Pkg Service is stateless, so this kind of horizontal scaling is straightforward. Each Pkg Server serves registry, package and artifact resources to Pkg Clients and caches whatever it serves. Each Pkg Server, in turn, gets those resources from one or more Storage Services. Storage services are responsible for fetching resources from code hosting sites like GitHub, GitLab and BitBucket, and for persisting everything that they have ever served to long-term storage systems like Amazon S3, hosted HDFS clusters - or whatever an implementor wants to use. If the original copies of resources vanish, Pkg Servers must always serve up all previously served versions of resources.

The Storage Protocol is designed to be extremely simple so that multiple independent implementations can coexist, and each Pkg Server may be symmetrically backed by multiple different Storage Services, providing both redundant backup and ensuring that no single implementation has a "choke hold" on the ecosystem - anyone can implement a new Storage Service and add it to the set of services backing the default Pkg Server at `pkg.julialang.org`. The simplest possible version of a Storage Service is a static HTTPS site serving files generated from a snapshot of a registry. Although this does not provide adequate long-term backup capabilities, and would need to be regenerated whenever a registry changes, it may be sufficient for some private uses. Having multiple independently operated Storage Services helps ensure that even if one Storage Service becomes unavailable or unreliable - for technical, financial, or political reasons - others will keep operating and so will the Pkg ecosystem.

Chapter 61

The Pkg Protocol

This section describes the protocol used by Pkg Clients to get resources from Pkg Servers, including the latest versions of registries, package source trees, and artifacts. There is also a standard system for asking for diffs of all of these from previous versions, to minimize how much data the client needs to download in order to update itself. There is additionally a bundle mechanism for requesting and receiving a set of resources in a single request.

61.1 Authentication

The authentication scheme between a Pkg client and server will be HTTP authorization with bearer tokens, as standardized in RFC6750. This means that authenticated access is accomplished by the client by making an HTTPS request including a `Authorization: Bearer $access_token` header.

The format of the token, its contents and validation mechanism are not specified by the Pkg Protocol. They are left to the server to define. The server is expected to validate the token and determine whether the client is authorized to access the requested resource. Similarly at the client side, the implementation of the token acquisition is not specified by the Pkg Protocol. However Pkg provides hooks that can be implemented at the client side to trigger the token acquisition process. Tokens thus acquired are expected to be stored in a local file, the format of which is specified by the Pkg Protocol. Pkg will be able to read the token from this file and include it in the request to the server. Pkg can also, optionally, detect when the token is about to expire and trigger a refresh. The Pkg client also supports automatic token refresh, since bearer tokens are recommended to be short-lived (no more than a day).

The authorization information is saved locally in `$(DEPOT_PATH[1])/servers/$server/auth.toml` which is a TOML file with the following fields:

- `access_token` (REQUIRED): the bearer token used to authorize normal requests
- `expires_at` (OPTIONAL): an absolute expiration time
- `expires_in` (OPTIONAL): a relative expiration time
- `refresh_token` (OPTIONAL): bearer token used to authorize refresh requests
- `refresh_url` (OPTIONAL): URL to fetch a new token from

The `auth.toml` file may contain other fields (e.g. user name, user email), but they are ignored by Pkg. The two other fields mentioned in RFC6750 are `token_type` and `scope`: these are omitted since only tokens of type `Bearer` are supported currently and the scope is always implicitly to provide access to Pkg protocol URLs. Pkg servers should, however, not send `auth.toml` files with `token_type` or `scope` fields, as these names may be used in the future, e.g. to support other kinds of tokens or to limit the scope of an authorization to a subset of Pkg protocol URLs.

Initially, the user or user agent (IDE) must acquire a `auth.toml` file and save it to the correct location. After that, Pkg will determine whether the access token needs to be refreshed by examining the `expires_at` and/or `expires_in` fields of the auth file. The expiration time is the minimum of `expires_at` and `mtime(auth_file) + expires_in`. When the Pkg client downloads a new `auth.toml` file, if there is a relative `expires_in` field, an absolute `expires_at` value is computed based on the client's current clock time. This combination of policies allows expiration to work gracefully even in the presence of clock skew between the server and the client.

If the access token is expired and there are `refresh_token` and `refresh_url` fields in `auth.toml`, a new auth file is requested by making a request to `refresh_url` with an `Authorization: Bearer $refresh_token` header. Pkg will refuse to make a refresh request unless `refresh_url` is an HTTPS URL. Note that `refresh_url` need not be a URL on the Pkg server: token refresh can be handled by a separate server. If the request is successful and the returned `auth.toml` file is a well-formed TOML file with at least an `access_token` field, it is saved to `$(DEPOT_PATH[1])/servers/$server/auth.toml`.

Checking for access token expiry and refreshing `auth.toml` is done before each Pkg client request to a Pkg server, and if the auth file is updated the new access token is used, so the token should in theory always be up to date. Practice is different from theory, of course, and if the Pkg server considers the access token expired, it may return an HTTP 401 Unauthorized response, and the Pkg client should attempt to refresh the auth token. If, after attempting to refresh the access token, the server still returns HTTP 401 Unauthorized, the Pkg client will present the body of the error response to the user or user agent (IDE).

Chapter 62

Authentication Hooks

A mechanism to register a hook at the client is provided to allow the user agent to handle an auth failure. It can, for example, present a login page and take the user through the necessary authentication flow to get a new auth token and store it in `auth.toml`.

- A handler can also be registered using `register_auth_error_handler`. It returns a function that can be called to deregister the handler.
- A handler can also be deregistered using `deregister_auth_error_handler`.

Example:

```
# register a handler
dispose = Pkg.PlatformEngines.register_auth_error_handler((url, svr, err) -> begin
    PkgAuth.authenticate(svr*"/auth")
    return true, true
end)

# ... client code ...

# deregister the handler
dispose()
# or
Pkg.PlatformEngines.deregister_auth_error_handler(url, svr)
```

62.1 Resources

The client can make GET or HEAD requests to the following resources:

- `/registries`: map of registry uuids at this server to their current tree hashes, each line of the response data is of the form `/registry/$uuid/$hash` representing a resource pointing to particular version of a registry
- `/registry/$uuid/$hash`: tarball of registry uuid at the given tree hash
- `/package/$uuid/$hash`: tarball of package uuid at the given tree hash
- `/artifact/$hash`: tarball of an artifact with the given tree hash

Only the `/registries` changes - all other resources can be cached forever and the server will indicate this with the appropriate HTTP headers.

62.2 Compression Negotiation

The Pkg protocol supports multiple compression formats.

- **Zstd compression** (current): Modern clients send `Accept-Encoding: zstd, gzip` to request Zstandard-compressed resources with gzip as a fallback.
- **Gzip compression** (legacy): Older clients that only support gzip send `Accept-Encoding: gzip` or omit the header entirely.

Clients verify the actual compression format by reading file magic bytes after download:

- **Zstd format**: Magic bytes `0x28 0xB5 0x2F 0xFD` (4 bytes) - decompressed with `zstd` (significantly faster)
- **Gzip format**: Magic bytes `0x1F 0x8B` (2 bytes) - decompressed with 7z

62.3 Reference Implementation

A reference implementation of the Pkg Server protocol is available at PkgServer.jl.

Chapter 63

The Storage Protocol

This section describes the protocol used by Pkg Servers to get resources from Storage Servers, including the latest versions of registries, package source trees, and artifacts. The Pkg Server requests each type of resource when it needs it and caches it for as long as it can, so Storage Services should not have to serve the same resources to the same Pkg Server instance many times.

63.1 Authentication

Since the Storage protocol is a server-to-server protocol, it uses certificate-based mutual authentication: each side of the connection presents certificates of identity to the other. The operator of a Storage Service must issue a client certificate to the operator of a Pkg Service certifying that it is authorized to use the Storage Service.

63.2 Resources

The Storage Protocol is similar to the Pkg Protocol:

- `/registries`: map of registry uuids at this server to their current tree hashes
- `/registry/$uuid/$hash`: tarball of registry uuid at the given tree hash
- `/package/$uuid/$hash`: tarball of package uuid at the given tree hash
- `/artifact/$hash`: tarball of an artifact with the given tree hash

As is the case with the Pkg Server protocol, only the `/registries` resource changes over time—all other resources are permanently cacheable and Pkg Servers are expected to cache resources indefinitely, only deleting them if they need to reclaim storage space.

63.3 Interaction

Fetching resources from a single Storage Server is straightforward: the Pkg Server asks for a version of a registry by UUID and hash and the Storage Server returns a tarball of that registry tree if it knows about that registry and version, or an HTTP 404 error if it doesn't.

Each Pkg Server may use multiple Storage Services for availability and depth of backup. For a given resource, the Pkg Server makes a HEAD request to each Storage Service requesting the resource, and then makes a GET request for the resource to the first Storage Server that replies to the HEAD request with a 200 OK. If no

Storage Service responds with a 200 OK in enough time, the Pkg Server should respond to the request for the corresponding resource with a 404 error. Each Storage Service which responds with a 200 OK must behave as if it had served the resource, regardless of whether it does so or not - i.e. persist the resource to long-term storage.

One subtlety is how the Pkg Server determines what the latest version of each registry is. It can get a map from registry UUIDs to version hashes from each Storage Server, but hashes are unordered - if multiple Storage Servers reply with different hashes, which one should the Pkg Server use? When Storage Servers disagree on the latest hash of a registry, the Pkg Server should ask each Storage Server about the hashes that the other servers returned: if Service A knows about Service B's hash but B doesn't know about A's hash, then A's hash is more recent and should be used. If each server doesn't know about the other's hash, then neither hash is strictly newer than the other one and either could be used. The Pkg Server can break the tie any way it wants, e.g. randomly or by using the lexicographically earlier hash.

63.4 Guarantees

The primary guarantee that a Storage Server makes is that if it has ever successfully served a resource—registry tree, package source tree, artifact tree — it must be able to serve that same resource version forever.

It's tempting to also require it to guarantee that if a Storage Server serves a registry tree, it can also serve every package source tree referred to within that registry tree. Similarly, it is tempting to require that if a Storage Server can serve a package source tree that it should be able to serve any artifacts referenced by that version of the package. However, this could fail for reasons entirely beyond the control of the server: what if the registry is published with wrong package hashes? What if someone registers a package version, doesn't git tag it, then force pushes the branch that the version was on? In both of these cases, the Storage Server may not be able to fetch a version of a package through no fault of its own. Similarly, artifact hashes in packages might be incorrect or vanish before the Storage Server can retrieve them.

Therefore, we don't strictly require that Storage Servers guarantee this kind of closure under resource references. We do, however, recommend that Storage Servers proactively fetch resources referred to by other resources as soon as possible. When a new version of a registry is available, the Storage Server should fetch all the new package versions in the registry immediately. When a package version is fetched—for any reason, whether because it was included in a new registry snapshot or because an upstream Pkg Server requested it by hash—all artifacts that it references should be fetched immediately.

Chapter 64

Verification

Since all resources are content addressed, the Pkg Clients and Pkg Server can and should verify that resources that they receive from upstream have the correct content hash. If a resource does not have the right hash, it should not be used and not be served further downstream. Pkg Servers should try to fetch the resource from other Storage Services and serve one that has the correct content. Pkg Clients should error if they get a resource with an incorrect content hash.

Git uses SHA1 for content hashing. There is a pure Julia implementation of git's content hashing algorithm, which is being used to verify artifacts in Julia 1.3 (among other things). The SHA1 hashing algorithm is considered to be cryptographically compromised at this point, and while it's not completely broken, git is already starting to plan how to move away from using SHA1 hashes. To that end, we should consider getting ahead of this problem by using a stronger hash like SHA3-256 in these protocols. Having control over these protocols actually makes this considerably easier than if we were continuing to rely on git for resource acquisition.

The first step to using SHA3-256 instead of SHA1 is to populate registries with additional hashes for package versions. Currently each package version is identified by a git-tree-sha1 entry. We would add git-tree-sha3-256 entries that give the SHA3-256 hashes computed using the same git tree hashing logic. From this origin, the Pkg Client, Pkg Server and Storage Servers all just need to use SHA3-256 hashes rather than SHA1 hashes.

Chapter 65

References

1. Pkg & Storage Protocols https://github.com/JuliaLang/Pkg.jl/issues/1377
2. Authenticated Pkg Client Support: https://github.com/JuliaLang/Pkg.jl/pull/1538
3. Authentication Hooks: https://github.com/JuliaLang/Pkg.jl/pull/1630

Part XV

15. Depots

The packages installed for a particular environment, defined in the files `Project.toml` and `Manifest.toml` within the directory structure, are not actually installed within that directory but into a "depot". The location of the depots are set by the variable `DEPOT_PATH`.

For details on the default depot locations and how they vary by installation method, see the `DEPOT_PATH` documentation.

Packages which are installed by a user go into the first depot and the Julia standard library is in the last depot.

You should not need to manage the user depot directly. Pkg will automatically clean up the depots when packages are removed after a delay. However you may want to manually remove old `.julia/compiled/` subdirectories if you have any that reside for older Julia versions that you no longer use (hence have not been run to tidy themselves up).

Chapter 66

Configuring the depot path with JULIA_DEPOT_PATH

The depot path can be configured using the JULIA_DEPOT_PATH environment variable, which is used to populate the global Julia DEPOT_PATH variable at startup. For complete details on the behavior of this environment variable, see the environment variables documentation.

66.1 When to customize the depot path

You may want to change your depot location in several scenarios:

- **Corporate environments**: When your user folder synchronizes with a server (such as with Active Directory roaming profiles), storing thousands of package files in the default depot can cause significant slowdowns during login/logout.
- **Storage constraints**: When your user directory has limited quota or is on a slow network drive.
- **Shared computing**: When multiple users need access to the same packages on a shared system.
- **Custom organization**: When you prefer to organize Julia packages separately from your user directory.

66.2 Platform-specific configuration

JULIA_DEPOT_PATH is an **operating system environment variable**, not a Julia REPL command. The method for setting it varies by platform:

Unix/Linux/macOS

For temporary configuration (current shell session only):

```
export JULIA_DEPOT_PATH="/custom/depot:"
```

For permanent configuration, add the export command to your shell configuration file (e.g., `~/.bashrc`, `~/.zshrc`, or `~/.profile`).

Windows

For temporary configuration in **PowerShell** (current session only):

```
$env:JULIA_DEPOT_PATH = "C:\custom\depot;"
```

For temporary configuration in **Command Prompt** (current session only):

```
set JULIA_DEPOT_PATH=C:\custom\depot;
```

For permanent system-wide or user-level configuration:

1. Press `Win+R` to open the Run dialog
2. Type `sysdm.cpl` and press Enter
3. Go to the "Advanced" tab
4. Click "Environment Variables"
5. Add a new user or system variable named `JULIA_DEPOT_PATH` with your desired path (e.g., `C:\custom\depot;`)

Note

The trailing path separator (`:` on Unix, `;` on Windows) is crucial for including the default system depots, which contain the standard library and other bundled resources. Without it, Julia will only use the specified depot and will have to precompile standard library packages, which can be time-consuming and inefficient.

66.3 Alternative configuration methods

Instead of setting an operating system environment variable, you can configure the depot path using Julia's `startup.jl` file, which runs automatically when Julia starts:

```
# In ~/.julia/config/startup.jl (Unix) or C:\Users\USERNAME\.julia\config\startup.jl (Windows)
empty!(DEPOT_PATH)
push!(DEPOT_PATH, "/custom/depot")
push!(DEPOT_PATH, joinpath(homedir(), ".julia"))  # Include default depot as fallback
```

This approach provides per-user permanent configuration without requiring operating system environment variable changes. However, setting `JULIA_DEPOT_PATH` is generally preferred as it takes effect before Julia loads any code.

Warning

Modifying `DEPOT_PATH` at runtime (in the REPL or in scripts) after Julia has started is generally not recommended, as Julia may have already loaded packages from the original depot locations.

Chapter 67

Shared depots for distributed computing

When using Julia in distributed computing environments, such as high-performance computing (HPC) clusters, it's recommended to use a shared depot via `JULIA_DEPOT_PATH`. This allows multiple Julia processes to share precompiled packages and reduces redundant compilation.

Since Julia v1.10, multiple processes using the same depot coordinate via pidfile locks to ensure only one process precompiles a package while others wait. However, due to the caching of native code in pkgimages since v1.9, you may need to set the `JULIA_CPU_TARGET` environment variable appropriately to ensure cache compatibility across different worker nodes with varying CPU capabilities.

For more details, see the FAQ section on distributed computing and the environment variables documentation.

Chapter 68

Setting up shared depots for multi-user systems

In multi-user environments such as JupyterHub deployments, university computing labs, or shared servers, system administrators often want to provide a set of commonly-used packages that are available to all users while still allowing individual users to install their own packages. This can be achieved by setting up a layered depot structure with a read-only shared depot and user-specific writable depots.

68.1 Overview of the approach

The key concept is to use `JULIA_DEPOT_PATH` to create a layered depot structure where:

1. **User depot** (first in path): User-specific packages and modifications
2. **Shared depot** (middle in path): Common packages installed by administrators
3. **System depot** (last in path): Julia standard library and bundled resources

When Julia searches for packages, it looks through depots in order. This allows users to:

- Access pre-installed packages from the shared depot
- Install additional packages into their own depot
- Override shared packages if needed by installing different versions in their user depot

68.2 Administrator setup

Step 1: Create the shared depot

As a system administrator, create a shared depot location accessible to all users:

```
# Create shared depot directory
sudo mkdir -p /opt/julia/shared_depot

# Create a shared user for managing the depot (optional but recommended)
sudo useradd -r -s /bin/bash -d /opt/julia/shared_depot julia-shared

# Set ownership
sudo chown -R julia-shared:julia-shared /opt/julia/shared_depot
```

Step 2: Install shared packages

Switch to the shared user account and configure Julia to use the shared depot:

```
sudo su - julia-shared
export JULIA_DEPOT_PATH="/opt/julia/shared_depot:"
```

Then install commonly-used packages. You can do this interactively or by instantiating from a Project.toml:

```
# Interactive installation
julia -e 'using Pkg; Pkg.add(["Plots", "DataFrames", "CSV", "LinearAlgebra"])'

# Or from a Project.toml file
cd /opt/julia/shared_depot
# Create or copy your Project.toml and Manifest.toml files here
julia --project=. -e 'using Pkg; Pkg.instantiate()'
```

> **Tip**
>
> Using a `Project.toml` and `Manifest.toml` file to define the shared environment is recommended as it provides reproducibility and version control. You can maintain these files in a git repository for tracking changes.

Step 3: Clean the shared depot (optional)

To minimize the shared depot size, you can remove registries from the shared depot:

```
rm -rf /opt/julia/shared_depot/registries
```

Since Pkg only writes to the first depot in `JULIA_DEPOT_PATH`, users will maintain their own registries in their user depots anyway. Removing registries from the shared depot simply avoids storing duplicate registry data.

Step 4: Set appropriate permissions

Make the shared depot read-only for regular users:

```
# Make shared depot readable by all users
sudo chmod -R a+rX /opt/julia/shared_depot

# Ensure it's not writable by others
sudo chmod -R go-w /opt/julia/shared_depot
```

68.3 User configuration

Each user should configure their `JULIA_DEPOT_PATH` to include both their personal depot and the shared depot. The exact syntax depends on where you want the user depot:

Using default user depot location

To use the default `~/.julia` as the user depot with the shared depot as a fallback:

```
export JULIA_DEPOT_PATH="~/.julia:/opt/julia/shared_depot:"
```

The trailing `:` ensures the system depot (with standard library) is still included.

Using a custom user depot location

If you want users to have their depot in a different location (e.g., to avoid home directory quotas):

```
export JULIA_DEPOT_PATH="/scratch/$USER/julia_depot:/opt/julia/shared_depot:"
```

System-wide configuration

To configure this for all users automatically, add the export command to system-wide shell configuration files:

On Linux:

```
# In /etc/profile.d/julia.sh
export JULIA_DEPOT_PATH="~/.julia:/opt/julia/shared_depot:"
```

On macOS:

```
# In /etc/zshrc or /etc/bashrc
export JULIA_DEPOT_PATH="~/.julia:/opt/julia/shared_depot:"
```

Users can then further customize their individual depot paths if needed.

68.4 Pre-seeding user environments

In some scenarios (e.g., for student lab computers or container images), you may want to pre-seed individual user environments. This can be done by:

1. Creating a template environment with a `Project.toml` and `Manifest.toml`
2. Copying these files to each user's Julia project directory
3. Having users (or a startup script) run `Pkg.instantiate()` on first use

Since packages in the shared depot will be found automatically, `instantiate()` will only download packages that aren't already available in the shared depot.

```
# As administrator, create template
mkdir -p /opt/julia/template_project
# Create Project.toml with desired packages
julia --project=/opt/julia/template_project -e 'using Pkg; Pkg.add("Example"); Pkg.add("Plots")'

# Users copy the template and instantiate
```

```
cp -r /opt/julia/template_project ~/my_project
cd ~/my_project
julia --project=. -e 'using Pkg; Pkg.instantiate()'
```

68.5 Updating shared packages

To update packages in the shared depot:

1. Switch to the shared user account
2. Set `JULIA_DEPOT_PATH` to point only to the shared depot
3. Update packages as needed
4. Optionally, clean up old package versions to save space

```
sudo su - julia-shared
export JULIA_DEPOT_PATH="/opt/julia/shared_depot:"
julia -e 'using Pkg; Pkg.update()'
```

Note

Updating packages in the shared depot adds new versions alongside existing ones. Users with Manifest.toml files remain pinned to their specific versions and won't be affected. If you explicitly clean up old package versions to save disk space, users who need those versions can run `Pkg.instantiate()` to download them to their local depot.

68.6 Troubleshooting

Packages not found despite being in shared depot: Verify that `JULIA_DEPOT_PATH` is set correctly and includes the shared depot. Check that the trailing separator is present to include system depots. Use `DEPOT_PATH` in the Julia REPL to verify the depot search path.

```
julia> DEPOT_PATH
3-element Vector{String}:
 "/home/user/.julia"
 "/opt/julia/shared_depot"
 "/usr/local/share/julia"
```