Stack: Build Tool & Package Manager

Introduction

Stack is a tool to build Haskell projects and manage their dependencies. It uses the Cabal library but with a curated version of the Hackage repository called *Stackage*

Stackage is a distribution of a subset of packages from *Hackage*, each package chosen at a version to make the set self-consistent. Note that Stackage does not patch any packages.

Hackage is the Haskell community's central package archive of open source software. Hackage has been online since January 2007 and is constantly growing. You can publish libraries and programs or download and install packages with tools like cabal-install (or via your distro's package manager).

Stack's functions

Stack handles the management of your toolchain (including GHC — the Glasgow Haskell Compiler — and, for Windows users, MSYS2), building and registering libraries, building build tool dependencies, and more. While it can use existing tools on your system, Stack has the capacity to be your one-stop shop for all Haskell tooling you need. This guide will follow that Stack-centric approach.

What makes Stack special?

The primary Stack design point is reproducible builds. If you run stack build today, you should get the same result running stack build tomorrow. There are some cases that can break that rule (changes in your operating system configuration, for example), but, overall, Stack follows this design philosophy closely. To make this a simple process, Stack uses curated package sets called ***snapshots***.

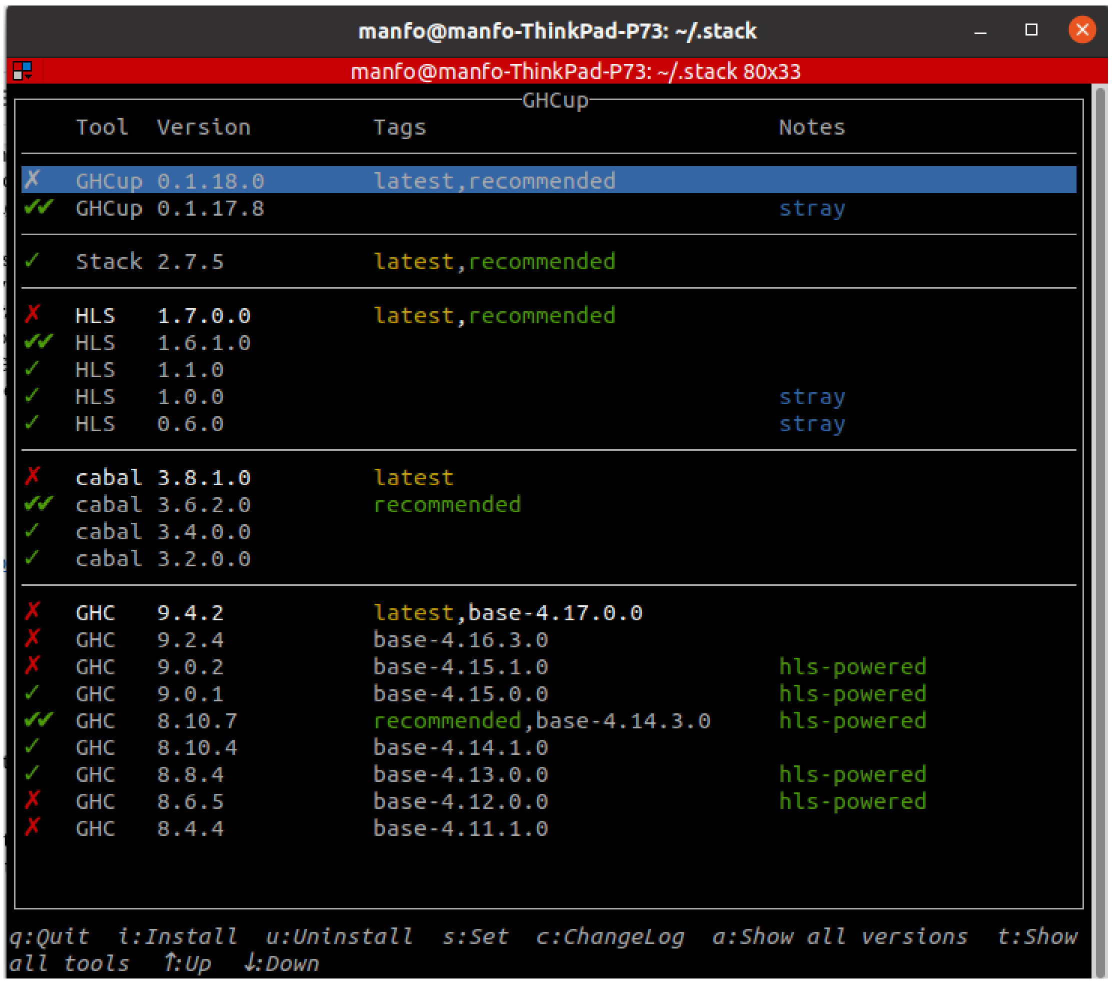
Stack has also been designed from the ground up to be user friendly, with an intuitive, discoverable command line interface. For many users, simply downloading Stack and reading stack --help will be enough to get up and running. This guide provides a more gradual tour for users who prefer that learning style.

To build your project, Stack uses a project-level configuration file, named ***stack.yaml***, in the root directory of your project as a sort of blueprint. That file contains a reference, called a ***resolver***, to the ***snapshot*** which your package will be built against.

Finally, Stack is isolated: it will not make changes outside of specific Stack directories. Stack-built files generally go in either the Stack root directory (default: ***~/.stack on Unix-like operating systems***, or, ***%LOCALAPPDATA%\Programs\stack on Windows***) or ***./.stack-work*** directories local to each project. The Stack root directory holds packages belonging to snapshots and any Stack-installed versions of GHC. Stack will not tamper with any system version of GHC or interfere with packages installed by other build tools (such as Cabal (the tool)).

Downloading and Installation

We recommend using the ghcup platform (https://www.haskell.org/ghcup/). It comes with several useful tools. Once installed, run the following command to check that all the tools (GHCup, stack, Haskell Language Server (HLS), Cabal and GHC) are installed.



```
manfo@manfo-ThinkPad-P73: ~/.stack
manfo@manfo-ThinkPad-P73: ~/.stack 80x33
                                   GHCup
     Tool   Version             Tags                          Notes
 ✗   GHCup  0.1.18.0            latest,recommended
 ✔✔  GHCup  0.1.17.8                                          stray

 ✓   Stack  2.7.5               latest,recommended

 ✗   HLS    1.7.0.0             latest,recommended
 ✔✔  HLS    1.6.1.0
 ✓   HLS    1.1.0
 ✓   HLS    1.0.0                                             stray
 ✓   HLS    0.6.0                                             stray

 ✗   cabal  3.8.1.0             latest
 ✔✔  cabal  3.6.2.0             recommended
 ✓   cabal  3.4.0.0
 ✓   cabal  3.2.0.0

 ✗   GHC    9.4.2               latest,base-4.17.0.0
 ✗   GHC    9.2.4               base-4.16.3.0
 ✗   GHC    9.0.2               base-4.15.1.0                 hls-powered
 ✓   GHC    9.0.1               base-4.15.0.0                 hls-powered
 ✔✔  GHC    8.10.7              recommended,base-4.14.3.0     hls-powered
 ✓   GHC    8.10.4              base-4.14.1.0
 ✓   GHC    8.8.4               base-4.13.0.0                 hls-powered
 ✗   GHC    8.6.5               base-4.12.0.0                 hls-powered
 ✗   GHC    8.4.4               base-4.11.1.0

q:Quit  i:Install  u:Uninstall  s:Set  c:ChangeLog  a:Show all versions  t:Show
all tools  ↑:Up  ↓:Down
```

The stack new command

We'll start off with the stack new command to create a new project that will contain a Haskell package of the same name. So let's pick a valid package name first:

A package is identified by a globally-unique package name, which consists of one or more alphanumeric words separated by hyphens. To avoid ambiguity, each of these words should contain at least one letter.

From the root directory for all our Haskell projects, we command:

```
stack new helloworld new-template
```

- ***stack*** is the command
- ***new*** is the first parameter indicating we are creating a new project
- ***helloworld*** is the name of our project
- ***new-template*** the default template and can be omitted

The stack build command

Next, we'll run the most important Stack command, stack build:

```
stack build

# installing ... building ...
```

Stack needs a version of GHC in order to build your project. Stack will discover that you are missing it and will install it for you. You can do this manually by using the `stack setup` command.

Note

GHC will be installed to your Stack programs directory, so calling ghc on the command line won't work. See the stack exec, stack ghc, and stack runghc commands below for more information.

Once a version of GHC is installed, Stack will then build your project.

The stack exec command

Looking closely at the output of the previous command, you can see that it built both a library called helloworld and an executable called helloworld-exe (on Windows, helloworld-exe.exe). We'll explain more in the next section, but, for now, just notice that the executables are installed in a location in our project's .stack-work directory.

Now, Let's use the stack exec command to run our executable (which just outputs the result of the main function in the Main module)

```
stack exec helloworld-exe
```

The stack test command

Finally, like all good software, helloworld actually has a test suite that we can find in the file

Let's run it with the stack test command:

```
stack test
# build output ...
```

Stack first builds the test suite and then automatically runs it for us. For both the build and test command, already built components are not built again. You can see this by using the stack build and stack test commands a second time:

```
stack build
stack test
# build output ...
```

Inner Workings of Stack

In this subsection, we'll dissect the helloworld example in more detail.

Files in helloworld

Before studying Stack more, let's understand our project a bit better. The files in the directory include:

```
app/Main.hs
src/Lib.hs
test/Spec.hs
ChangeLog.md
README.md
LICENSE
Setup.hs
helloworld.cabal
package.yaml
stack.yaml
.gitignore
```

The

- `app/Main.hs`,
- `src/Lib.hs`,
- `test/Spec.hs`

files are all Haskell source files that compose the actual functionality of our project.

The

- `ChangeLog.md`,
- `README.md`,
- `LICENSE`
- `.gitignore`

files have no effect on the build.

The

- `helloworld.cabal`

file is updated automatically as part of the stack build process and ***should not be modified***.

The files of interest here are ***Setup.hs***, ***stack.yaml****,* and ***package.yaml***.

The Setup.hs file is a component of the Cabal build system which Stack uses. It's technically not needed by Stack, but it is still considered good practice in the Haskell world to include it. The file we're using is straight boilerplate:

```
import Distribution.Simple
main = defaultMain
```

Next, let's look at our stack.yaml file, which gives our project-level settings. Ignoring comments beginning #, the contents will look something like this:

```
resolver:
  url:
https://raw.githubusercontent.com/commercialhaskell/stackage-snapshots/m
aster/lts/19/17.yaml
```

```
packages:
    - .
```

The value of the resolver key tells Stack how to build your package: which GHC version to use, versions of package dependencies, and so on.

The value of the packages key tells Stack which local packages to build. In our simple example, we have only a single package in our project, located in the same directory, so '.' suffices. There is support for multi packages projects

Another file important to the build is package.yaml.

The package.yaml file describes the package in the *Hpack format*. Stack has in-built Hpack functionality and this is its preferred package format. The default behavior is to generate the Cabal file named ***myprojectname.cabal*** from this package.yaml file, and accordingly you should not modify the Cabal file.

The stack setup command

As we saw above, the build command installed GHC for us. Just for kicks, let's manually run the setup command:

stack setup
stack will use a sandboxed GHC it installed
For more information on paths, see 'stack path' and 'stack exec env'
To use this GHC and packages outside of a project, consider using:
stack ghc, stack ghci, stack runghc, or stack exec
Thankfully, the command is smart enough to know not to perform an installation twice. As the command output above indicates, you can use stack path for quite a bit of path information (which we'll play with more later).

For now, we'll just look at where GHC is installed:

On Unix-like operating systems, command:

```
stack exec -- which ghc
/home/<user_name>/.stack/programs/x86_64-linux/ghc-9.0.2/bin/ghc
On Windows (with PowerShell), command:


stack exec -- where.exe ghc
C:\Users\<user_name>\AppData\Local\Programs\stack\x86_64-windows\ghc-9.0
.2\bin\ghc.exe
```

As you can see from that path (and as emphasized earlier), the installation is placed to not interfere with any other GHC installation, whether system-wide or even different GHC versions installed by Stack.

Cleaning your project

You can clean up build artifacts for your project using the stack clean and stack purge commands.

The stack clean command

stack clean deletes the local working directories containing compiler output. By default, that means the contents of directories in .stack-work/dist, for all the .stack-work directories within a project.

Use stack clean <specific-package> to delete the output for the package specific-package only.

The stack purge command

stack purge deletes the local stack working directories, including extra-deps, git dependencies and the compiler output (including logs). It does not delete any snapshot packages, compilers or programs installed using stack install. This essentially reverts the project to a completely fresh state, as if it had never been built. `stack purge` is just a shortcut for `stack clean --full`

The stack build command

The build command is the heart and soul of Stack. It is the engine that powers building your code, testing it, getting dependencies, and more. Quite a bit of the remainder of this guide will cover more advanced build functions and features, such as building tests and Haddocks at the same time, or constantly rebuilding blocking on file changes.

Note

Using the build command twice with the same options and arguments should generally do nothing (besides things like rerunning test suites), and should, in general, produce a reproducible result between different runs.

Adding dependencies

Let's say we decide to modify our helloworld source a bit to use a new library, perhaps the ubiquitous text package. In src/Lib.hs, we can, for example add:

```haskell
{-# LANGUAGE OverloadedStrings #-}
module Lib
    ( someFunc
    ) where

import qualified Data.Text.IO as T

someFunc :: IO ()
someFunc = T.putStrLn "someFunc"
```

When we try to build this, things don't go as expected:

```
stack build
# build failure output (abridged for clarity) ...
src\Lib.hs:6:1: error:
    Could not load module 'Data.Text.IO'
    It is a member of the hidden package 'text-1.2.5.0'.
    Perhaps you need to add 'text' to the build-depends in your .cabal
file.
    Use -v (or `:set -v` in ghci) to see a list of the files searched
for.
  |
6 | import qualified Data.Text.IO as T
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

This means that the package containing the module in question is not available. ***To tell Stack to use text, you need to add it to your package.yaml file*** — specifically in your dependencies section, like this:

```
dependencies:
- base >= 4.7 && < 5
- text # added here
```

Now if we rerun stack build, we should get a successful result. Command:

```
stack build
# build output ...
```

This output means that the text package was downloaded, configured, built, and locally installed. Once that was done, we moved on to building our local package (helloworld). At no point did we need to ask Stack to build dependencies — it does so automatically.

Listing Dependencies

Let's have Stack add a few more dependencies to our project. First, we'll include two new packages in the dependencies section for our library in our package.yaml:

```
dependencies:
- base >= 4.7 && < 5
- text
- filepath
- containers
```

After adding these two dependencies, we can again run `stack build` to have them installed. Command:

```
stack build
# build output ...
```

Finally, to find out which versions of these libraries Stack installed, we can ask Stack to ls dependencies. Command:

```
stack ls dependencies
# dependency output ...
```

extra-deps

Let's try a more off-the-beaten-track package: the joke acme-missiles package. Our source code is simple:

```haskell
module Lib
    ( someFunc
    ) where

import Acme.Missiles

someFunc :: IO ()
someFunc = launchMissiles
```

Again, we add this new dependency to the package.yaml file like this:

```
dependencies:
- base >= 4.7 && < 5
- text
- filepath
- containers
- acme-missiles # added
```

However, rerunning stack build shows us the following error message. Command:

```
stack build
# build failure output ...
```

It says that it was unable to construct the build plan.

This brings us to the next major topic in using Stack.

Curated package sets

Remember above when stack new selected some ***LTS resolver for us? That defined our build plan*** and available packages. ***When we tried using the text package, it just worked, because it was part of the LTS package set***.

We've specified the acme-missiles package in the package.yaml file (see above), ***but acme-missiles is not part of that LTS package set***, so building failed.

To add acme-missiles to the available packages, we'll use the extra-deps key in the stack.yaml file. That key defines extra packages, not present in the resolver, that will be needed as dependencies. You can add this like so:

```
extra-deps:
- acme-missiles-0.3 # not in the LTS resolver
```

Now stack build will succeed.

Resolvers and changing your compiler version

Let's explore package sets a bit further. Instead of lts-19.17, let's change our stack.yaml file to use the latest nightly. Right now, this is currently 2022-07-31 - please see the resolve from the link above to get the latest.

In the stack.yaml file, enter:

```
#---

resolver: lts-18.19

#---
```

Then, commanding stack build again will produce:

```
stack build
# Downloaded ... plan.
# build output ...
```

The stack path command

Generally, you don't need to worry about where Stack stores various files. But some people like to know this stuff. That's when the stack path command is useful. For example, command:

```
stack path

snapshot-doc-root: ...
local-doc-root: ...
local-hoogle-root: ...
stack-root: ...
project-root: ...
```

```
config-location: ...
bin-path: ...
programs: ...
compiler-exe: ...
compiler-bin: ...
compiler-tools-bin: ...
local-bin: ...
extra-include-dirs: ...
extra-library-dirs: ...
snapshot-pkg-db: ...
local-pkg-db: ...
global-pkg-db: ...
ghc-package-path: ...
snapshot-install-root: ...
local-install-root: ...
dist-dir: ...
local-hpc-root: ...
local-bin-path: ...
ghc-paths: ...
```

The stack ghci or stack repl command

GHCi is the interactive GHC environment, a.k.a. the REPL. You could access it with command:

```
stack exec ghci
```

Note

But that won't load up locally written modules for access. For that, use the

```
stack ghci

-- or

stack repl
```

commands, which are equivalent. To then load modules from your project in GHCi, use the :module command (:m for short) followed by the module name.

stack.yaml versus Cabal files

Now that we've covered a lot of Stack use cases, this quick summary of stack.yaml versus Cabal files will hopefully make sense and be a good reminder for future uses of Stack:

A project can have multiple packages.
Each project has a stack.yaml.
Each package has a Cabal file, named <package_name>.cabal.
The Cabal file specifies which packages are dependencies.
The stack.yaml file specifies which packages are available to be used.
The Cabal file specifies the components, modules, and build flags provided by a package
stack.yaml can override the flag settings for individual packages
stack.yaml specifies which packages to include

Comparison to other tools

Stack is not the only tool around for building Haskell code. Stack came into existence due to limitations with some of the existing tools.

If you're a new user who has no experience with other tools, we recommend going with Stack. The defaults match modern best practices in Haskell development, and there are less corner cases you need to be aware of. You can develop Haskell code with other tools, but you probably want to spend your time writing code, not convincing a tool to do what you want.

Before jumping into the differences, let me clarify an important similarity:

Same package format. Stack, Cabal (the tool), and presumably all other tools share the same underlying Cabal package format, consisting of a Cabal file, modules, etc. This is a Good Thing: we can share the same set of upstream libraries, and collaboratively work on the same project with Stack, Cabal (the tool), and NixOS. In that sense, we're sharing the same ecosystem.

Now the differences:

Curation vs dependency solving as a default.
Stack defaults to using curation (Stackage snapshots, LTS Haskell, Nightly, etc) as a default instead of defaulting to dependency solving, as Cabal (the tool) does*.* This is just a default: as described above, Stack can use dependency solving if desired, and Cabal (the tool) can use curation. However, most users will stick to the defaults. The Stack team firmly believes that the majority of users want to simply ignore dependency resolution nightmares and get a valid build plan from day one, which is why we've made this selection of default behavior.
Reproducible.

Stack goes to great lengths to ensure that stack build today does the same thing tomorrow. Cabal (the tool) does not: build plans can be affected by the presence of pre-installed packages, and running cabal update can cause a previously successful build to fail. With Stack, changing the build plan is always an explicit decision.
Automatically building dependencies.

With Cabal (the tool), you need to use cabal install to trigger dependency building. This is somewhat necessary due to the previous point, since building dependencies can, in some cases, break existing installed packages. So for example, in Stack, stack test does the same job as cabal install --run-tests, though the latter additionally performs an installation that you may not want. The closest equivalent command sequence is: cabal install --enable-tests --only-dependencies, cabal configure --enable-tests, cabal build && cabal test (newer versions of Cabal (the tool) may make this command sequence shorter).
Isolated by default.

This has been a pain point for new Stack users. In Cabal, the default behavior is a non-isolated build where working on two projects can cause the user package database to become corrupted. The Cabal solution to this is sandboxes. Stack, however, provides this behavior by default via its databases. In other words: when you use Stack, there's no need for sandboxes, everything is (essentially) sandboxed by default.
Other tools for comparison (including active and historical)

stackage-cli was an initial attempt to make Cabal (the tool) work more easily with curated snapshots, but due to a slight impedance mismatch between cabal.config constraints and snapshots, it did not work as well as hoped. It is deprecated in favor of Stack.

More resources

There are lots of resources available for learning more about Stack:

stack --help
stack --version — identify the version and Git hash of the Stack executable
--verbose (or -v) — much more info about internal operations (useful for bug reports)
The home page
The Stack mailing list
The FAQ
The haskell-stack tag on Stack Overflow
Here Another getting started with Stack tutorial
Understand Why is Stack not Cabal?

Ref: https://docs.haskellstack.org/en/stable/GUIDE/