User guide

stack is a modern, cross-platform build tool for Haskell code.

This guide takes a new stack user through the typical workflows. This guide will not teach Haskell or involve much code, and it requires no prior experience with the Haskell packaging system or other build tools.

NOTE This document is probably out of date in some places and deserves a refresh. If you find this document helpful, please drop a note on *issue #4252*.

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 `stack.yaml` file 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` or, on Windows, `%LOCALAPPDATA%\Programs\stack`) 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 `cabal` or any other build tools.

NOTE In this guide, we'll use commands as run on a GNU/Linux system (specifically Ubuntu 14.04, 64-bit) and share output from that. Output on other systems — or with different versions of stack — will be slightly different, but all commands work cross-platform, unless explicitly stated otherwise.

Downloading and Installation

The *documentation dedicated to downloading stack* has the most up-to-date information for a variety of operating systems, including multiple GNU/Linux flavors. Instead of repeating that content here, please go check out that page and come back here when you can successfully run `stack --version`. The rest of this session will demonstrate the installation procedure on a vanilla Ubuntu 14.04 machine.

```
michael@d30748af6d3d:~$ sudo apt-get install wget
# installing ...
michael@d30748af6d3d:~$ wget -qO- https://get.haskellstack.org/ | sh
# downloading ...
michael@d30748af6d3d:~$ stack --help
# help output ...
```

With stack now up and running, you're good to go. Though not required, we recommend setting your PATH environment variable to include `$HOME/.local/bin`:

```
michael@d30748af6d3d:~$ echo 'export PATH=$HOME/.local/bin:$PATH' >>
~/.bashrc
```

Hello World Example

With stack installed, let's create a new project from a template and walk through the most common stack commands.

stack new

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 *Cabal users guide*)

We'll call our project `helloworld`, and we'll use the `new-template` project template:

```
michael@d30748af6d3d:~$ stack new helloworld new-template
```

For this first stack command, there's quite a bit of initial setup it needs to do (such as downloading the list of packages available upstream), so you'll see a lot of output. Over the course of this guide a lot of the content will begin to make more sense.

We now have a project in the `helloworld` directory!

stack build

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

```
michael@d30748af6d3d:~$ cd helloworld
michael@d30748af6d3d:~/helloworld$ stack build
# installing ... building ...
```

stack needs a 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.

You'll get intermediate download percentage statistics while the download is occurring. This command may take some time, depending on download speeds.

NOTE: GHC will be installed to your global stack root 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 GHC is installed, stack will then build your project.

stack exec

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". We'll explain more in the next section, but, for now, just notice that the executables are installed in our project's `./.stack-work` directory.

Now, Let's use `stack exec` to run our executable (which just outputs the string "someFunc"):

```
michael@d30748af6d3d:~/helloworld$ stack exec helloworld-exe
someFunc
```

`stack exec` works by providing the same reproducible environment that was used to build your project to the command that you are running. Thus, it knew where to find `helloworld-exe` even though it is hidden in the `./.stack-work` directory.

stack test

Finally, like all good software, helloworld actually has a test suite.

Let's run it with `stack test`:

```
michael@d30748af6d3d:~/helloworld$ stack test
# build output ...
```

Reading the output, you'll see that 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 running `stack build` and `stack test` a second time:

```
michael@d30748af6d3d:~/helloworld$ stack build
michael@d30748af6d3d:~/helloworld$ 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.

```
michael@d30748af6d3d:~/helloworld$ find * -type f
LICENSE
README.md
Setup.hs
app/Main.hs
helloworld.cabal
package.yaml
src/Lib.hs
stack.yaml
test/Spec.hs
```

The `app/Main.hs`, `src/Lib.hs`, and `test/Spec.hs` files are all Haskell source files that compose the actual functionality of our project (we won't dwell on them here).

The `LICENSE` file and `README.md` have no impact 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.

If you're familiar with YAML, you may recognize that the `flags` and `extra-deps` keys have empty values. We'll see more interesting usages for these fields later. Let's focus on the other two fields. `packages` 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. However, stack has powerful support for multi-package projects, which we'll elaborate on as this guide progresses.

The final field is `resolver`. This tells stack *how* to build your package: which GHC version to use, versions of package dependencies, and so on. Our value here says to use *LTS Haskell version 3.2*, which implies GHC 7.10.2 (which is why `stack setup` installs that version of GHC). There are a number of values you can use for `resolver`, which we'll cover later.

Another file important to the build is `package.yaml`.

Since Stack 1.6.1, the `package.yaml` is the preferred package format that is provided built-in by stack through *the hpack tool*. The default behaviour is to generate the `.cabal` file from this `package.yaml`, and accordingly you should **not** modify the `.cabal` file.

It is also important to remember that stack is built on top of the Cabal build system. Therefore, an understanding of the moving parts in Cabal are necessary. In Cabal, we have individual *packages*, each of which contains a single `.cabal` file. The `.cabal` file can define 1 or more *components*: a library, executables, test suites, and benchmarks. It also specifies additional information such as library dependencies, default language pragmas, and so on.

In this guide, we'll discuss the bare minimum necessary to understand how to modify a `package.yaml` file. You can see a full list of the available options at the *hpack documentation*. Haskell.org has the definitive *reference for the `.cabal` file format*.

The setup command

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

```
michael@d30748af6d3d:~/helloworld$ 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:

```
michael@d30748af6d3d:~/helloworld$ stack exec -- which ghc
/home/michael/.stack/programs/x86_64-linux/ghc-7.10.2/bin/ghc
```

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.

`stack clean`

`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.

`stack purge`

`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 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 test and Haddocks at the same time, or constantly rebuilding blocking on file changes.

On a philosophical note: Running the build command twice with the same options and arguments should generally be a no-op (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:

```
{-# 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:

```
michael@d30748af6d3d:~/helloworld$ stack build
# build failure output (abridged for clarity) ...
/helloworld/src/Lib.hs:5:1: error:
    Could not find module `Data.Text.IO'
    Use -v to see a list of the files searched for.
   |
5 | 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:

```
michael@d30748af6d3d:~/helloworld$ 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:
- filepath
- containers
```

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

```
michael@d30748af6d3d:~/helloworld$ stack build
# build output ...
```

Finally, to find out which versions of these libraries stack installed, we can ask stack to `ls dependencies`:

```
michael@d30748af6d3d:~/helloworld$ 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:

```
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:

```
michael@d30748af6d3d:~/helloworld$ 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*.

But `acme-missiles` is not part of that package set, so building failed.

To add this new dependency, we'll use the `extra-deps` field in `stack.yaml` to define extra dependencies not present in the resolver. You can add this like so:

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

Now `stack build` will succeed.

With that out of the way, let's dig a little bit more into these package sets, also known as *snapshots*. We mentioned the LTS resolvers, and you can get quite a bit of information about it at *https://www.stackage.org/lts*, including:

- The appropriate resolver value (`resolver: lts-18.3`, as is currently the latest LTS)
- The GHC version used
- A full list of all packages available in this snapshot
- The ability to perform a Hoogle search on the packages in this snapshot
- A *list of all modules* in a snapshot, which can be useful when trying to determine which package to add to your `package.yaml` file.

You can also see a *list of all available snapshots*. You'll notice two flavors: LTS (for "Long Term Support") and Nightly. You can read more about them on the *LTS Haskell Github page*. If you're not sure which to use, start with LTS Haskell (which stack will lean towards by default as well).

Resolvers and changing your compiler version

Let's explore package sets a bit further. Instead of lts-18.3, let's change our `stack.yaml` file to use *the latest nightly*. Right now, this is currently 2020-03-24 - please see the resolve from the link above to get the latest.

Then, Rerunning `stack build` will produce:

```
michael@d30748af6d3d:~/helloworld$ stack build
Downloaded nightly-2020-03-24 build plan.
# build output ...
```

We can also change resolvers on the command line, which can be useful in a Continuous Integration (CI) setting, like on Travis. For example:

```
michael@d30748af6d3d:~/helloworld$ stack --resolver lts-18.3 build
Downloaded lts-18.3 build plan.
# build output ...
```

When passed on the command line, you also get some additional "short-cut" versions of resolvers: `--resolver nightly` will use the newest Nightly resolver available, `--resolver lts` will use the newest LTS, and `--resolver lts-2` will use the newest LTS in the 2.X series. The reason these are only available on the command line and not in your `stack.yaml` file is that using them:

1. Will slow down your build (since stack then needs to download information on the latest available LTS each time it builds)
2. Produces unreliable results (since a build run today may proceed differently tomorrow because of changes outside of your control)

Changing GHC versions

Finally, let's try using an older LTS snapshot. We'll use the newest 2.X snapshot:

```
michael@d30748af6d3d:~/helloworld$ stack --resolver lts-2 build
# build output ...
```

This succeeds, automatically installing the necessary GHC along the way. So, we see that different LTS versions use different GHC versions and stack can handle that.

Other resolver values

We've mentioned `nightly-YYYY-MM-DD` and `lts-X.Y` values for the resolver. There are actually other options available, and the list will grow over time. At the time of writing:

- `ghc-X.Y.Z`, for requiring a specific GHC version but no additional packages
- Experimental custom snapshot support

The most up-to-date information can always be found in the *stack.yaml documentation*.

Existing projects

Alright, enough playing around with simple projects. Let's take an open source package and try to build it. We'll be ambitious and use *yackage*, a local package server using *Yesod*. To get the code, we'll use the `stack unpack` command:

```
cueball:~$ stack unpack yackage-0.8.0
Unpacked yackage-0.8.0 to /var/home/harendra/yackage-0.8.0/
cueball:~$ cd yackage-0.8.0/
```

Note that you can also unpack to the directory of your liking instead of the current one by issuing:

```
cueball:~$ stack unpack yackage-0.8.0 --to ~/work
```

This will create a `yackage-0.8.0` directory inside `~/work`

stack init

This new directory does not have a `stack.yaml` file, so we need to make one first. We could do it by hand, but let's be lazy instead with the `stack init` command:

```
cueball:~/yackage-0.8.0$ stack init
# init output ...
```

stack init does quite a few things for you behind the scenes:

- Finds all of the `.cabal` files in your current directory and subdirectories (unless you use `--ignore-subdirs`) and determines the packages and versions they require
- Finds the best combination of snapshot and package flags that allows everything to compile with minimum external dependencies

- It tries to look for the best matching snapshot from latest LTS, latest nightly, other LTS versions in that order

Assuming it finds a match, it will write your `stack.yaml` file, and everything will work.

(Note: yackage does not currently support hpack, but you can also hpack-convert should you need to generate a package.yaml).

Excluded Packages

Sometimes multiple packages in your project may have conflicting requirements. In that case `stack init` will fail, so what do you do?

You could manually create `stack.yaml` by omitting some packages to resolve the conflict. Alternatively you can ask `stack init` to do that for you by specifying `--omit-packages` flag on the command line. Let's see how that works.

To simulate a conflict we will use acme-missiles-0.3 in yackage and we will also copy `yackage.cabal` to another directory and change the name of the file and package to yackage-test. In this new package we will use acme-missiles-0.2 instead. Let's see what happens when we re-run stack init:

```
cueball:~/yackage-0.8.0$ stack init --force --omit-packages
# init failure output ...
```

Looking at `stack.yaml`, you will see that the excluded packages have been commented out under the `packages` field. In case wrong packages are excluded you can uncomment the right one and comment the other one.

Packages may get excluded due to conflicting requirements among user packages or due to conflicting requirements between a user package and the resolver compiler. If all of the packages have a conflict with the compiler then all of them may get commented out.

When packages are commented out you will see a warning every time you run a command which needs the configuration file. The warning can be disabled by editing the configuration file and removing it.

Using a specific resolver

Sometimes you may want to use a specific resolver for your project instead of `stack init` picking one for you. You can do that by using `stack init --resolver <resolver>`.

You can also init with a compiler resolver if you do not want to use a snapshot. That will result in all of your project's dependencies being put under the `extra-deps` section.

Installing the compiler

stack will automatically install the compiler when you run `stack build` but you can manually specify the compiler by running `stack setup <GHC-VERSION>`.

Miscellaneous and diagnostics

Add selected packages: If you want to use only selected packages from your project directory you can do so by explicitly specifying the package directories on the command line.

Duplicate package names: If multiple packages under the directory tree have same name, stack init will report those and automatically ignore one of them.

Ignore subdirectories: By default stack init searches all the subdirectories for `.cabal` files. If you do not want that then you can use `--ignore-subdirs` command line switch.

Cabal warnings: stack init will show warnings if there were issues in reading a cabal package file. You may want to pay attention to the warnings as sometimes they may result in incomprehensible errors later on during dependency solving.

Package naming: If the `Name` field defined in a cabal file does not match with the cabal file name then `stack init` will refuse to continue.

Cabal install errors: stack init uses `cabal-install` to determine external dependencies. When cabal-install encounters errors, cabal errors are displayed as is by stack init for diagnostics.

User warnings: When packages are excluded or external dependencies added stack will show warnings every time configuration file is loaded. You can suppress the warnings by editing the config file and removing the warnings from it. You may see something like this:

```
cueball:~/yackage-0.8.0$ stack build
Warning: Some packages were found to be incompatible with the resolver
and have been left commented out in the packages section.
Warning: Specified resolver could not satisfy all dependencies. Some
external packages have been added as dependencies.
You can suppress this message by removing it from stack.yaml
```

Different databases

Time to take a short break from hands-on examples and discuss a little architecture. stack has the concept of multiple *databases*. A database consists of a GHC package database (which contains the compiled version of a library), executables, and a few other things as well. To give you an idea:

```
michael@d30748af6d3d:~/helloworld$ ls .stack-work/install/x86_64-
linux/lts-3.2/7.10.2/
bin  doc  flag-cache  lib  pkgdb
```

Databases in stack are *layered*. For example, the database listing we just gave is called a *local* database. That is layered on top of a *snapshot* database, which contains the libraries and executables specified in the snapshot itself. Finally, GHC itself ships with a number of libraries and executables, which forms the *global* database. To get a quick idea of this, we can look at the output of the `stack exec -- ghc-pkg list` command in our helloworld project:

```
/home/michael/.stack/programs/x86_64-linux/ghc-7.10.2/lib/ghc-
7.10.2/package.conf.d
   Cabal-1.22.4.0
   array-0.5.1.0
   base-4.8.1.0
   bin-package-db-0.0.0.0
   binary-0.7.5.0
   bytestring-0.10.6.0
   containers-0.5.6.2
   deepseq-1.4.1.1
   directory-1.2.2.0
   filepath-1.4.0.0
   ghc-7.10.2
   ghc-prim-0.4.0.0
   haskeline-0.7.2.1
   hoopl-3.10.0.2
   hpc-0.6.0.2
   integer-gmp-1.0.0.0
   pretty-1.1.2.0
   process-1.2.3.0
   rts-1.0
   template-haskell-2.10.0.0
   terminfo-0.4.0.1
   time-1.5.0.1
   transformers-0.4.2.0
   unix-2.7.1.0
   xhtml-3000.2.1
/home/michael/.stack/snapshots/x86_64-linux/nightly-2015-08-
26/7.10.2/pkgdb
```

```
   stm-2.4.4
/home/michael/helloworld/.stack-work/install/x86_64-linux/nightly-2015-
08-26/7.10.2/pkgdb
   acme-missiles-0.3
   helloworld-0.1.0.0
```

Notice that acme-missiles ends up in the *local* database. Anything which is not installed from a snapshot ends up in the local database. This includes: your own code, extra-deps, and in some cases even snapshot packages, if you modify them in some way. The reason we have this structure is that:

- it lets multiple projects reuse the same binary builds of many snapshot packages,
- but doesn't allow different projects to "contaminate" each other by putting non-standard content into the shared snapshot database

Typically, the process by which a snapshot package is marked as modified is referred to as "promoting to an extra-dep," meaning we treat it just like a package in the extra-deps section. This happens for a variety of reasons, including:

- changing the version of the snapshot package
- changing build flags
- one of the packages that the package depends on has been promoted to an extra-dep

As you probably guessed, there are multiple snapshot databases available, e.g.:

```
michael@d30748af6d3d:~/helloworld$ ls ~/.stack/snapshots/x86_64-linux/
lts-2.22  lts-3.1  lts-3.2  nightly-2015-08-26
```

These databases don't get layered on top of each other; they are each used separately.

In reality, you'll rarely — if ever — interact directly with these databases, but it's good to have a basic understanding of how they work so you can understand why rebuilding may occur at different points.

The build synonyms

Let's look at a subset of the `stack --help` output:

```
build    Build the package(s) in this directory/configuration
install  Shortcut for 'build --copy-bins'
test     Shortcut for 'build --test'
bench    Shortcut for 'build --bench'
haddock  Shortcut for 'build --haddock'
```

Note that four of these commands are just synonyms for the `build` command. They are provided for convenience for common cases (e.g., `stack test` instead of `stack build --test`) and so that commonly expected commands just work.

What's so special about these commands being synonyms? It allows us to make much more composable command lines. For example, we can have a command that builds executables, generates Haddock documentation (Haskell API-level docs), and builds and runs your test suites, with:

```
stack build --haddock --test
```

You can even get more inventive as you learn about other flags. For example, take the following:

```
stack build --pedantic --haddock --test --exec "echo Yay, it succeeded" --file-watch
```

This will:

- turn on all warnings and errors
- build your library and executables

- generate Haddocks
- build and run your test suite
- run the command `echo Yay, it succeeded` when that completes
- after building, watch for changes in the files used to build the project, and kick off a new build when done

install and copy-bins

It's worth calling out the behavior of the install command and `--copy-bins` option, since this has confused a number of users (especially when compared to behavior of other tools like cabal-install). The `install` command does precisely one thing in addition to the build command: it copies any generated executables to the local bin path. You may recognize the default value for that path:

```
michael@d30748af6d3d:~/helloworld$ stack path --local-bin
/home/michael/.local/bin
```

That's why the download page recommends adding that directory to your `PATH` environment variable. This feature is convenient, because now you can simply run `executable-name` in your shell instead of having to run `stack exec executable-name` from inside your project directory.

Since it's such a point of confusion, let me list a number of things stack does *not* do specially for the install command:

- stack will always build any necessary dependencies for your code. The install command is not necessary to trigger this behavior. If you just want to build a project, run `stack build`.
- stack will *not* track which files it's copied to your local bin path nor provide a way to automatically delete them. There are many great tools out there for managing

installation of binaries, and stack does not attempt to replace those.

- stack will not necessarily be creating a relocatable executable. If your executables hard-codes paths, copying the executable will not change those hard-coded paths.
  > - At the time of writing, there's no way to change those kinds of paths with stack, but see *issue #848 about --prefix* for future plans.

That's really all there is to the install command: for the simplicity of what it does, it occupies a much larger mental space than is warranted.

Targets, locals, and extra-deps

We haven't discussed this too much yet, but, in addition to having a number of synonyms *and* taking a number of options on the command line, the build command *also* takes many arguments. These are parsed in different ways, and can be used to achieve a high level of flexibility in telling stack exactly what you want to build.

We're not going to cover the full generality of these arguments here; instead, there's *documentation covering the full build command syntax*. Here, we'll just point out a few different types of arguments:

- You can specify a *package name*, e.g. `stack build vector`.
  > - This will attempt to build the vector package, whether it's a local package, in your extra-deps, in your snapshot, or just available upstream. If it's just available upstream but not included in your locals, extra-deps, or snapshot, the newest version is automatically promoted to an extra-dep.

- You can also give a *package identifier*, which is a package name plus version, e.g. `stack build yesod-bin-1.4.14`.
  > - This is almost identical to specifying a package name, except it will (1) choose the given version instead of latest, and (2) error out if the given version conflicts with the version of a local package.
- The most flexibility comes from specifying individual *components*, e.g. `stack build helloworld:test:helloworld-test` says "build the test suite component named helloworld-test from the helloworld package."
  > - In addition to this long form, you can also shorten it by skipping what type of component it is, e.g. `stack build helloworld:helloworld-test`, or even skip the package name entirely, e.g. `stack build :helloworld-test`.
- Finally, you can specify individual *directories* to build to trigger building of any local packages included in those directories or subdirectories.

When you give no specific arguments on the command line (e.g., `stack build`), it's the same as specifying the names of all of your local packages. If you just want to build the package for the directory you're currently in, you can use `stack build .`.

Components, --test, and --bench

Here's one final important yet subtle point. Consider our helloworld package: it has a library component, an executable helloworld-exe, and a test suite helloworld-test. When you run `stack build helloworld`, how does it know which ones to build? By default, it will build the library (if any) and all of the executables but ignore the test suites and benchmarks.

This is where the `--test` and `--bench` flags come into play. If you use them, those components will also be included. So `stack build --test helloworld` will end up including the helloworld-test component as well.

You can bypass this implicit adding of components by being much more explicit, and stating the components directly. For example, the following will not build the helloworld-exe executable:

```
michael@d30748af6d3d:~/helloworld$ stack clean
michael@d30748af6d3d:~/helloworld$ stack build :helloworld-test
helloworld-0.1.0.0: configure (test)
Configuring helloworld-0.1.0.0...
helloworld-0.1.0.0: build (test)
Preprocessing library helloworld-0.1.0.0...
[1 of 1] Compiling Lib              ( src/Lib.hs, .stack-
work/dist/x86_64-linux/Cabal-1.22.4.0/build/Lib.o )
In-place registering helloworld-0.1.0.0...
Preprocessing test suite 'helloworld-test' for helloworld-0.1.0.0...
[1 of 1] Compiling Main             ( test/Spec.hs, .stack-
work/dist/x86_64-linux/Cabal-1.22.4.0/build/helloworld-test/helloworld-
test-tmp/Main.o )
Linking .stack-work/dist/x86_64-linux/Cabal-1.22.4.0/build/helloworld-
test/helloworld-test ...
helloworld-0.1.0.0: test (suite: helloworld-test)
Test suite not yet implemented
```

We first cleaned our project to clear old results so we know exactly what stack is trying to do. Notice that it builds the helloworld-test test suite, and the helloworld library (since it's used by the test suite), but it does not build the helloworld-exe executable.

And now the final point: the last line shows that our command also *runs* the test suite it just built. This may surprise some people who would expect tests to only be run when using `stack test`, but this design decision is what allows the `stack build` command to be as composable as it is (as described previously). The same rule applies to benchmarks. To spell it out completely:

- The --test and --bench flags simply state which components of a package should be built, if no explicit set of components is given
- The default behavior for any test suite or benchmark component which has been built is to also run it

You can use the `--no-run-tests` and `--no-run-benchmarks` (from stack-0.1.4.0 and on) flags to disable running of these components. You can also use `--no-rerun-tests` to prevent running a test suite which has already passed and has not changed.

NOTE: stack doesn't build or run test suites and benchmarks for non-local packages. This is done so that running a command like `stack test` doesn't need to run 200 test suites!

Multi-package projects

Until now, everything we've done with stack has used a single-package project. However, stack's power truly shines when you're working on multi-package projects. All the functionality you'd expect to work just does: dependencies between packages are detected and respected, dependencies of all packages are just as one cohesive whole, and if anything fails to build, the build commands exits appropriately.

Let's demonstrate this with the wai-app-static and yackage packages:

```
michael@d30748af6d3d:~$ mkdir multi
michael@d30748af6d3d:~$ cd multi/
michael@d30748af6d3d:~/multi$ stack unpack wai-app-static-3.1.1 yackage-
0.8.0
wai-app-static-3.1.1: download
Unpacked wai-app-static-3.1.1 to /home/michael/multi/wai-app-static-
3.1.1/
```

```
Unpacked yackage-0.8.0 to /home/michael/multi/yackage-0.8.0/
michael@d30748af6d3d:~/multi$ stack init
Writing default config file to: /home/michael/multi/stack.yaml
Basing on cabal files:
- /home/michael/multi/yackage-0.8.0/yackage.cabal
- /home/michael/multi/wai-app-static-3.1.1/wai-app-static.cabal

Checking against build plan lts-3.2
Selected resolver: lts-3.2
Wrote project config to: /home/michael/multi/stack.yaml
michael@d30748af6d3d:~/multi$ stack build --haddock --test
# Goes off to build a whole bunch of packages
```

If you look at the `stack.yaml`, you'll see exactly what you'd expect:

```
flags:
  yackage:
    upload: true
  wai-app-static:
    print: false
packages:
- yackage-0.8.0/
- wai-app-static-3.1.1/
extra-deps: []
resolver: lts-3.2
```

Notice that multiple directories are listed in the `packages` key.

In addition to local directories, you can also refer to packages available in a Git repository or in a tarball over HTTP/HTTPS. This can be useful for using a modified version of a dependency that hasn't yet been released upstream.

Please note that when adding upstream packages directly to your project it is important to distinguish *local packages* from the upstream *dependency packages*. Otherwise you may have trouble running `stack ghci`. See *stack.yaml documentation* for more details.

Flags and GHC options

There are two common ways to alter how a package will install: with Cabal flags and with GHC options.

Cabal flag management

In the `stack.yaml` file above, you can see that `stack init` has detected that — for the yackage package — the upload flag can be set to true, and for wai-app-static, the print flag to false (it's chosen those values because they're the default flag values, and their dependencies are compatible with the snapshot we're using.) To change a flag setting, we can use the command line `--flag` option:

```
stack build --flag yackage:-upload
```

This means: when compiling the yackage package, turn off the upload flag (thus the `-`). Unlike other tools, stack is explicit about which package's flag you want to change. It does this for two reasons:

1. There's no global meaning for Cabal flags, and therefore two packages can use the same flag name for completely different things.
2. By following this approach, we can avoid unnecessarily recompiling snapshot packages that happen to use a flag that we're using.

You can also change flag values on the command line for extra-dep and snapshot packages. If you do this, that package will automatically be promoted to an extra-dep, since the build plan is different than what the plan snapshot definition would entail.

GHC options

GHC options follow a similar logic as in managing Cabal flags, with a few nuances to adjust for common use cases. Let's consider:

```
stack build --ghc-options="-Wall -Werror"
```

This will set the `-Wall -Werror` options for all *local targets*. Note that this will not affect extra-dep and snapshot packages at all. This design provides us with reproducible and fast builds.

(By the way: the above GHC options have a special convenience flag: `--pedantic`.)

There's one extra nuance about command line GHC options: Since they only apply to local targets, if you change your local targets, they will no longer apply to other packages. Let's play around with an example from the wai repository, which includes the wai and warp packages, the latter depending on the former. If we run:

```
stack build --ghc-options=-O0 wai
```

It will build all of the dependencies of wai, and then build wai with all optimizations disabled. Now let's add in warp as well:

```
stack build --ghc-options=-O0 wai warp
```

This builds the additional dependencies for warp, and then builds warp with optimizations disabled. Importantly: it does not rebuild wai, since wai's configuration has not been altered. Now the surprising case:

```
michael@d30748af6d3d:~/wai$ stack build --ghc-options=-O0 warp
wai-3.0.3.0-5a49351d03cba6cbaf906972d788e65d: unregistering (flags
changed from ["--ghc-options","-O0"] to [])
warp-3.1.3-a91c7c3108f63376877cb3cd5dbe8a7a: unregistering (missing
dependencies: wai)
wai-3.0.3.0: configure
```

You may expect this to be a no-op: neither wai nor warp has changed. However, stack will instead recompile wai with optimizations enabled again, and then rebuild warp (with optimizations disabled) against this newly built wai. The reason: reproducible builds. If we'd never built wai or warp before, trying to build warp would necessitate building all of its dependencies, and it would do so with default GHC options (optimizations enabled). This dependency would include wai. So when we run:

```
stack build --ghc-options=-O0 warp
```

We want its behavior to be unaffected by any previous build steps we took. While this specific corner case does catch people by surprise, the overall goal of reproducible builds is- in the stack maintainers' views- worth the confusion.

Final point: if you have GHC options that you'll be regularly passing to your packages, you can add them to your `stack.yaml` file (starting with stack-0.1.4.0). See *the documentation section on ghc-options* for more information.

path

NOTE: That's it, the heavy content of this guide is done! Everything from here on out is simple explanations of commands. Congratulations!

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.

```
michael@d30748af6d3d:~/wai$ stack path
global-stack-root: /home/michael/.stack
```

```
stack-root: /home/michael/.stack
project-root: /home/michael/wai
config-location: /home/michael/wai/stack.yaml
bin-path: /home/michael/.stack/snapshots/x86_64-linux/lts-
2.17/7.8.4/bin:/home/michael/.stack/programs/x86_64-linux/ghc-
7.8.4/bin:/home/michael/.stack/programs/x86_64-linux/ghc-
7.10.2/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
programs: /home/michael/.stack/programs/x86_64-linux
compiler: /home/michael/.stack/programs/x86_64-linux/ghc-7.8.4/bin/ghc
compiler-bin: /home/michael/.stack/programs/x86_64-linux/ghc-7.8.4/bin
local-bin-path: /home/michael/.local/bin
extra-include-dirs:
extra-library-dirs:
snapshot-pkg-db: /home/michael/.stack/snapshots/x86_64-linux/lts-
2.17/7.8.4/pkgdb
local-pkg-db: /home/michael/wai/.stack-work/install/x86_64-linux/lts-
2.17/7.8.4/pkgdb
global-pkg-db: /home/michael/.stack/programs/x86_64-linux/ghc-
7.8.4/lib/ghc-7.8.4/package.conf.d
ghc-package-path: /home/michael/wai/.stack-work/install/x86_64-linux/lts-
2.17/7.8.4/pkgdb:/home/michael/.stack/snapshots/x86_64-linux/lts-
2.17/7.8.4/pkgdb:/home/michael/.stack/programs/x86_64-linux/ghc-
7.8.4/lib/ghc-7.8.4/package.conf.d
snapshot-install-root: /home/michael/.stack/snapshots/x86_64-linux/lts-
2.17/7.8.4
local-install-root: /home/michael/wai/.stack-work/install/x86_64-
linux/lts-2.17/7.8.4
snapshot-doc-root: /home/michael/.stack/snapshots/x86_64-linux/lts-
2.17/7.8.4/doc
local-doc-root: /home/michael/wai/.stack-work/install/x86_64-linux/lts-
2.17/7.8.4/doc
dist-dir: .stack-work/dist/x86_64-linux/Cabal-1.18.1.5
local-hpc-root: /home/michael/wai/.stack-work/install/x86_64-linux/lts-
2.17/7.8.4/hpc
```

In addition, `stack path` accepts command line arguments to state which of these keys you're interested in, which can be convenient for scripting. As a simple example, let's find out the sandboxed versions of GHC that stack installed:

```
michael@d30748af6d3d:~/wai$ ls $(stack path --programs)/*.installed
/home/michael/.stack/programs/x86_64-linux/ghc-7.10.2.installed
/home/michael/.stack/programs/x86_64-linux/ghc-7.8.4.installed
```

(Yes, that command requires a *nix shell, and likely won't run on Windows.)

While we're talking about paths, to wipe our stack install completely, here's what needs to be removed:

1. The stack executable itself
2. The stack root, e.g. `$HOME/.stack` on non-Windows systems or, on Windows, `%LOCALAPPDATA%\Programs\stack`.
    - See `stack path --stack-root`
    - On Windows, you will also need to delete `stack path --programs`
3. Any local `.stack-work` directories inside a project

exec

We've already used `stack exec` multiple times in this guide. As you've likely already guessed, it allows you to run executables, but with a slightly modified environment. In particular: `stack exec` looks for executables on stack's bin paths, and sets a few additional environment variables (like adding those paths to `PATH`, and setting `GHC_PACKAGE_PATH`, which tells GHC which package databases to use).

If you want to see exactly what the modified environment looks like, try:

```
stack exec env
```

The only issue is how to distinguish flags to be passed to stack versus those for the underlying program. Thanks to the optparse-applicative library, stack follows the Unix convention of `--` to separate these, e.g.:

```
michael@d30748af6d3d:~$ stack exec --package stm -- echo I installed the
stm package via --package stm
Run from outside a project, using implicit global project config
Using latest snapshot resolver: lts-18.3
Writing global (non-project-specific) config file to:
/home/michael/.stack/global/stack.yaml
```

```
Note: You can change the snapshot via the resolver field there.
I installed the stm package via --package stm
```

Flags worth mentioning:

- `--package foo` can be used to force a package to be installed before running the given command.
- `--no-ghc-package-path` can be used to stop the `GHC_PACKAGE_PATH` environment variable from being set. Some tools — notably cabal-install — do not behave well with that variable set.

You may also find it convenient to use `stack exec` to launch a subshell (substitute `bash` with your preferred shell) where your compiled executable is available at the front of your `PATH`:

```
stack exec bash
```

ghci (the repl)

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

```
stack exec ghci
```

But that won't load up locally written modules for access. For that, use the `stack ghci` command. To then load modules from your project, use the `:m` command (for "module") followed by the module name.

IMPORTANT NOTE: If you have added upstream packages to your project please make sure to mark them as *dependency package_s for faster and reliable usage of `stack ghci`. Otherwise GHCi may have trouble due to conflicts of compilation flags or having to unnecessarily interpret too many modules. See stack.yaml documentation to learn how to mark a package as a _dependency package*.

ghc/runghc

You'll sometimes want to just compile (or run) a single Haskell source file, instead of creating an entire Cabal package for it. You can use `stack exec ghc` or `stack exec runghc` for that. As simple helpers, we also provide the `stack ghc` and `stack runghc` commands, for these common cases.

script interpreter

stack also offers a very useful feature for running files: a script interpreter. For too long have Haskellers felt shackled to bash or Python because it's just too hard to create reusable source-only Haskell scripts. stack attempts to solve that.

You can use `stack <file name>` to execute a Haskell source file or specify `stack` as the interpreter using a shebang line on a Unix like operating systems. Additional stack options can be specified using a special Haskell comment in the source file to specify dependencies and automatically install them before running the file.

An example will be easiest to understand:

```
michael@d30748af6d3d:~$ cat turtle-example.hs
#!/usr/bin/env stack
-- stack --resolver lts-6.25 script --package turtle
{-# LANGUAGE OverloadedStrings #-}
import Turtle
main = echo "Hello World!"
michael@d30748af6d3d:~$ chmod +x turtle-example.hs
michael@d30748af6d3d:~$ ./turtle-example.hs
Completed 5 action(s).
Hello World!
michael@d30748af6d3d:~$ ./turtle-example.hs
Hello World!
```

The first run can take a while (as it has to download GHC if necessary and build dependencies), but subsequent runs are able to reuse everything already built, and are therefore quite fast.

The first line in the source file is the usual "shebang" to use stack as a script interpreter. The second line, is a Haskell comment providing additional options to stack (due to the common limitation of the "shebang" line only being allowed a single argument). In this case, the options tell stack to use the lts-3.2 resolver, automatically install GHC if it is not already installed, and ensure the turtle package is available.

If you're on Windows: you can run `stack turtle.hs` instead of `./turtle.hs`. The shebang line is not required in that case.

Just-in-time compilation

You can add the `--compile` flag to make stack compile the script, and then run the compiled executable. Compilation is done quickly, without optimization. To compile with optimization, use the `--optimize` flag instead. Compilation is done only if needed; if the executable already exists, and is newer than the script, stack just runs the executable directly.

This feature can be good for speed (your script runs faster) and also for durability (the executable remains runnable even if the script is disturbed, eg due to changes in your installed ghc/snapshots, changes to source files during git bisect, etc.)

Using multiple packages

You can also specify multiple packages, either with multiple `--package` arguments, or by providing a comma or space separated list. For example:

```
#!/usr/bin/env stack
{- stack
  script
  --resolver lts-6.25
  --package turtle
  --package "stm async"
  --package http-client,http-conduit
-}
```

Stack configuration for scripts

With the `script` command, all Stack configuration files are ignored to provide a completely reliable script running experience. However, see the example below with `runghc` for an approach to scripts which will respect your configuration files. When using `runghc`, if the current working directory is inside a project then that project's stack configuration is effective when running the script. Otherwise the script uses the global project configuration specified in `~/.stack/global-project/stack.yaml`.

Specifying interpreter options

The stack interpreter options comment must specify a single valid stack command line, starting with `stack` as the command followed by the stack options to use for executing this file. The comment must always be on the line immediately following the shebang line when the shebang line is present otherwise it must be the first line in the file. The comment must always start in the first column of the line.

When many options are needed a block style comment may be more convenient to split the command on multiple lines for better readability. You can also specify ghc options the same way as you would on command line i.e. by separating the stack options and ghc options with a `--`. Here is an example of a multi line block comment with ghc options:

```
#!/usr/bin/env stack
{- stack
  script
  --resolver lts-6.25
  --package turtle
  --
  +RTS -s -RTS
-}
```

Writing independent and reliable scripts

With the release of Stack 1.4.0, there is a new command, `script`, which will automatically:

- Install GHC and libraries if missing
- Require that all packages used be explicitly stated on the command line

This ensures that your scripts are *independent* of any prior deployment specific configuration, and are *reliable* by using exactly the same version of all packages every time it runs so that the script does not break by accidentally using incompatible package versions.

In previous versions of Stack, the `runghc` command was used for scripts instead. In order to achieve the same effect with the `runghc` command, you can do the following:

1. Use the `--install-ghc` option to install the compiler automatically
2. Explicitly specify all packages required by the script using the `--package` option. Use `-hide-all-packages` ghc option to force explicit specification of all packages.
3. Use the `--resolver` Stack option to ensure a specific GHC version and package set is used.

Even with this configuration, it is still possible for configuration files to impact `stack runghc`, which is why `stack script` is strongly recommended in general. For those curious, here is an example with `runghc`:

```
#!/usr/bin/env stack
{- stack
  --resolver lts-6.25
  --install-ghc
  runghc
  --package base
  --package turtle
  --
  -hide-all-packages
  -}
```

The `runghc` command is still very useful, especially when you're working on a project and want to access the package databases and configurations used by that project. See the next section for more information on configuration files.

Platform-specific script issues

On Mac OSX:

- Avoid `{-# LANGUAGE CPP #-}` in stack scripts; it breaks the hashbang line (*GHC #6132*)
- Use a compiled executable, not another script, in the hashbang line. Eg `#!/usr/bin/env runhaskell` will work but `#!/usr/local/bin/runhaskell` would not.

Loading scripts in ghci

Sometimes you want to load your script in ghci REPL to play around with your program. In those cases, you can use `exec ghci` option in the script to achieve it. Here is an example:

```
#!/usr/bin/env stack
{- stack
     --resolver lts-8.2
     --install-ghc
     exec ghci
     --package text
-}
```

Finding project configs, and the implicit global project

Whenever you run something with stack, it needs a `stack.yaml` project file. The algorithm stack uses to find this is:

1. Check for a `--stack-yaml` option on the command line
2. Check for a `STACK_YAML` environment variable
3. Check the current directory and all ancestor directories for a `stack.yaml`

The first two provide a convenient method for using an alternate configuration. For example: `stack build --stack-yaml stack-7.8.yaml` can be used by your CI system to check your code against GHC 7.8. Setting the `STACK_YAML` environment variable can be convenient if you're going to be running commands like `stack ghc` in other directories, but you want to use the configuration you defined in a specific project.

If stack does not find a `stack.yaml` in any of the three specified locations, the *implicit global* logic kicks in. You've probably noticed that phrase a few times in the output from commands above. Implicit global is essentially a hack to allow stack to be useful in a non-project setting. When no implicit global config file exists, stack creates one for you with the latest LTS snapshot as the resolver. This allows you to do things like:

- compile individual files easily with `stack ghc`

- build executables without starting a project, e.g. `stack install pandoc`

Keep in mind that there's nothing magical about this implicit global configuration. It has no impact on projects at all. Every package you install with it is put into isolated databases just like everywhere else. The only magic is that it's the catch-all project whenever you're running stack somewhere else.

Setting stack root location

`stack path --stack-root` will tell you the location of the 'stack root'. This is where stack stores snapshot packages, among other things. On operating systems other than Windows, it is also where stack stores tools such as ghc and MSYS2 by default, in a `programs` folder. (On Windows, the default location for such tools is `%LOCALAPPDATA%\Programs\stack`.)

The location of the stack root can be configured by setting the `STACK_ROOT` environment variable or using stack's `--stack-root` option on the command line. It is particularly useful to do this on Windows, where the length of filepaths may be limited (to *MAX_PATH*), and things can break when this limit is exceeded.

`stack.yaml` vs `.cabal` files

Now that we've covered a lot of stack use cases, this quick summary of `stack.yaml` vs `.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.

- The `.cabal` file specifies which packages are dependencies.
- The `stack.yaml` file specifies which packages are available to be used.
- `.cabal` 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 unaffected by those limitations and are happily building Haskell code, you may not need stack. If you're suffering from some of the common problems in other tools, give stack a try instead.

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-install, 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-install, 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-install does. This is just a default: as described above, stack can use dependency solving if desired, and cabal-install 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 1, 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-install does not: build plans can be affected by the presence of preinstalled 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**.
  - In cabal-install, 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 closer command equivalent is `cabal install --enable-tests --only-dependencies && cabal configure --enable-tests && cabal build && cabal test` (newer versions of cabal-install may make this command 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)

- *cabal-dev* (deprecated in favor of cabal-install)
- *cabal-meta* inspired a lot of the multi-package functionality of stack. If you're still using cabal-install, cabal-meta is relevant. For stack work, the feature set is fully subsumed by stack.
- *cabal-src* is mostly irrelevant in the presence of both stack and cabal sandboxes, both of which make it easier to add additional package sources easily. The mega-sdist executable that ships with cabal-src is, however, still relevant. Its functionality may some day be folded into stack
- *stackage-cli* was an initial attempt to make cabal-install 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.

Fun features

This is just a quick collection of fun and useful feature stack supports.

Templates

We started off using the `new` command to create a project. stack provides multiple templates to start a new project from:

```
michael@d30748af6d3d:~$ stack templates
# Stack Templates

The `stack new` command will create a new project based on a project
template.
Templates can be located on the local filesystem, on Github, or arbitrary
URLs.
For more information, please see the user guide:

https://docs.haskellstack.org/en/stable/GUIDE/#templates

There are many templates available, some simple examples:

    stack new myproj # uses the default template
    stack new myproj2 rio # uses the rio template
    stack new website yesodweb/sqlite # Yesod server with SQLite DB

For more information and other templates, please see the `stack-
templates`
Wiki:

https://github.com/commercialhaskell/stack-templates/wiki

Please feel free to add your own templates to the Wiki for
discoverability.

Want to improve this text? Send us a PR!

https://github.com/commercialhaskell/stack-
templates/edit/master/STACK_HELP.md
```

You can specify one of these templates following your project name in the `stack new` command:

```
michael@d30748af6d3d:~$ stack new my-yesod-project yesodweb/simple
Downloading template "yesod-simple" to create project "my-yesod-project"
in my-yesod-project/ ...
Using the following authorship configuration:
author-email: example@example.com
author-name: Example Author Name
Copy these to /home/michael/.stack/config.yaml and edit to use different
values.
```

```
Writing default config file to: /home/michael/my-yesod-project/stack.yaml
Basing on cabal files:
- /home/michael/my-yesod-project/my-yesod-project.cabal

Checking against build plan lts-3.2
Selected resolver: lts-3.2
Wrote project config to: /home/michael/my-yesod-project/stack.yaml
```

The default `stack-templates` repository is on *Github*, under the user account `commercialstack`. You can download templates from a different Github user by prefixing the username and a slash:

```
stack new my-yesod-project yesodweb/simple
```

Then it would be downloaded from Github, user account `yesodweb`, repo `stack-templates`, and file `yesod-simple.hsfiles`.

You can even download templates from a service other that Github, such as *Gitlab* or *Bitbucket*:

```
stack new my-project gitlab:user29/foo
```

That template would be downloaded from Gitlab, user account `user29`, repo `stack-templates`, and file `foo.hsfiles`.

If you need more flexibility, you can specify the full URL of the template:

```
stack new my-project https://my-site.com/content/template9.hsfiles
```

(The `.hsfiles` extension is optional; it will be added if it's not specified.)

Alternatively you can use a local template by specifying the path:

```
stack new project ~/location/of/your/template.hsfiles
```

As a starting point for creating your own templates, you can use *the "simple" template*. An introduction into template-writing and a place for submitting official templates, you will find at *the stack-templates repository*.

Editor integration

For editor integration, stack has a related project called *intero*. It is particularly well supported by emacs, but some other editors have integration for it as well.

Visualizing dependencies

If you'd like to get some insight into the dependency tree of your packages, you can use the `stack dot` command and Graphviz. More information is *available in the Dependency visualization documentation*.

Travis with caching

This content has been moved to a dedicated *Travis CI document*.

Shell auto-completion

Love tab-completion of commands? You're not alone. If you're on bash, just run the following (or add it to `.bashrc`):

```
eval "$(stack --bash-completion-script stack)"
```

For more information and other shells, see *the Shell auto-completion wiki page*

Docker

Stack is able to build your code inside a Docker image, which means even more reproducibility to your builds, since you and the rest of your team will always have the same system libraries.

Nix

stack provides an integration with *Nix*, providing you with the same two benefits as the first Docker integration discussed above:

- more reproducible builds, since fixed versions of any system libraries and commands required to build the project are automatically built using Nix and managed locally per-project. These system packages never conflict with any existing versions of these libraries on your system. That they are managed locally to the project means that you don't need to alter your system in any way to build any odd project pulled from the Internet.
- implicit sharing of system packages between projects, so you don't have more copies on-disk than you need to.

When using the Nix integration, Stack downloads and builds Haskell dependencies as usual, but resorts on Nix to provide non-Haskell dependencies that exist in the Nixpkgs.

Both Docker and Nix are methods to *isolate* builds and thereby make them more reproducible. They just differ in the means of achieving this isolation. Nix provides slightly weaker isolation guarantees than Docker, but is more lightweight and more portable (Linux and OS X mainly, but also Windows). For more on Nix, its command-line interface and its package description language, read the *Nix manual*. But keep in mind that the point of stack's support is to obviate the need to write any Nix code in the common case or even to learn how to use the Nix tools (they're called under the hood).

For more information, see *the Nix-integration documentation*.

Power user commands

The following commands are a little more powerful, and won't be needed by all users. Here's a quick rundown:

- `stack update` will download the most recent set of packages from your package indices (e.g. Hackage). Generally, stack runs this for you automatically when necessary, but it can be useful to do this manually sometimes.

- `stack unpack` is a command we've already used quite a bit for examples, but most users won't use it regularly. It does what you'd expect: downloads a tarball and unpacks it. It accept optional `--to` argument to specify the destination directory.

- `stack sdist` generates an uploading tarball containing your package code

- `stack upload` uploads an sdist to Hackage. As of version *1.1.0* stack will also attempt to GPG sign your packages as per *our blog post*.

> - `--no-signature` disables signing of packages
> - `--candidate` upload a *package candidate*
> - Hackage API key can be used instead of username and password. Usage example:

  `bash HACKAGE_KEY=<api_key> stack upload .`

> - `username` and `password` can be read by environment

  `bash export $HACKAGE_USERNAME="<username>" export $HACKAGE_PASSWORD="<password>"`

- `stack upgrade` will build a new version of stack from source.

> - `--git` is a convenient way to get the most recent version from master for those testing and living on the

> bleeding edge.

- `stack ls snapshots` will list all the local snapshots by default. You can also view the remote snapshots using `stack ls snapshots remote`. It also supports option for viewing only lts (`-l`) and nightly (`-n`) snapshots.

- `stack ls dependencies` lists all of the packages and versions used for a project

- `stack list [PACKAGE]...` list the version of the specified package(s) in a snapshot, or without an argument list all the snapshot's package versions. If no resolver is specified the latest package version from Hackage is given.

- `stack sig` subcommand can help you with GPG signing & verification

> - `sign` will sign an sdist tarball and submit the signature to sig.commercialhaskell.org for storage in the sig-archive git repo. (Signatures will be used later to verify package integrity.)

Debugging

To profile a component of the current project, simply pass the `--profile` flag to `stack`. The `--profile` flag turns on the `--enable-library-profiling` and `--enable-executable-profiling` Cabal options *and* passes the `+RTS -p` runtime options to any testsuites and benchmarks.

For example the following command will build the `my-tests` testsuite with profiling options and create a `my-tests.prof` file in the current directory as a result of the test run.

```
stack test --profile my-tests
```

The `my-tests.prof` file now contains time and allocation info for the test run.

To create a profiling report for an executable, e.g. `my-exe`, you can run

```
stack exec --profile -- my-exe +RTS -p
```

For more fine-grained control of compilation options there are the `--library-profiling` and `--executable-profiling` flags which will turn on the `--enable-library-profiling` and `--enable-executable-profiling` Cabal options respectively. Custom GHC options can be passed in with `--ghc-options "more options here"`.

To enable compilation with profiling options by default you can add the following snippet to your `stack.yaml` or `~/.stack/config.yaml`:

```
build:
  library-profiling: true
  executable-profiling: true
```

Further reading

For more commands and uses, see *the official GHC chapter on profiling*, *the Haskell wiki*, and *the chapter on profiling in Real World Haskell*.

Tracing

To generate a backtrace in case of exceptions during a test or benchmarks run, use the `--trace` flag. Like `--profile` this compiles with profiling options, but adds the `+RTS -xc` runtime option.

Debugging symbols

Building with debugging symbols in the *DWARF information* is supported by `stack`. This can be done by passing the flag `--ghc-options="-g"` and also to override the default behaviour of stripping executables of debugging symbols by passing either one of the following flags: `--no-strip`, `--no-library-stripping` or `--no-executable-stripping`.

In Windows GDB can be installed to debug an executable with `stack exec -- pacman -S gdb`. Windows visual studio compiler's debugging format PDB is not supported at the moment. This might be possible by *separating* debugging symbols and *converting* their format. Or as an option when *using the LLVM backend*.

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 *stack wiki*
- The *haskell-stack tag on Stack Overflow*
- *Another getting started with stack tutorial*
- *Why is stack not cabal?*