

# Hydra User's Guide



**Draft**

---

Copyright © 2009, 2010 Eelco Dolstra

---

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	About Hydra . . . . .	1
1.2	About Us . . . . .	2
1.3	About this Manual . . . . .	2
1.4	License . . . . .	2
1.5	Hydra at <code>nixos.org</code> . . . . .	3
1.6	Hydra on your own buildfarm . . . . .	3
<b>2</b>	<b>Installation</b>	<b>4</b>
2.1	Prerequisites . . . . .	4
2.2	Getting Nix . . . . .	4
2.3	Installation . . . . .	4
2.4	Creating the database . . . . .	5
2.5	Upgrading . . . . .	5
2.6	Getting Started . . . . .	6
<b>3</b>	<b>Creating and Managing Projects</b>	<b>7</b>
3.1	Project Information . . . . .	7
3.2	Job Sets . . . . .	8
3.3	Release Set . . . . .	8
3.4	Building Jobs . . . . .	8
3.5	Build Recipes . . . . .	8
3.6	Building from the Command Line . . . . .	10
3.7	Adding More Jobs . . . . .	11

---

# Chapter 1

## Introduction

### 1.1 About Hydra

Hydra is a tool for continuous integration testing and software release that uses a purely functional language to describe build jobs and their dependencies. Continuous integration is a simple technique to improve the quality of the software development process. An automated system continuously or periodically checks out the source code of a project, builds it, runs tests, and produces reports for the developers. Thus, various errors that might accidentally be committed into the code base are automatically caught. Such a system allows more in-depth testing than what developers could feasibly do manually:

- *Portability testing*: The software may need to be built and tested on many different platforms. It is infeasible for each developer to do this before every commit.
- Likewise, many projects have very large test sets (e.g., regression tests in a compiler, or stress tests in a DBMS) that can take hours or days to run to completion.
- Many kinds of static and dynamic analyses can be performed as part of the tests, such as code coverage runs and static analyses.
- It may also be necessary to build many different *variants* of the software. For instance, it may be necessary to verify that the component builds with various versions of a compiler.
- Developers typically use incremental building to test their changes (since a full build may take too long), but this is unreliable with many build management tools (such as Make), i.e., the result of the incremental build might differ from a full build.
- It ensures that the software can be built from the sources under revision control. Users of version management systems such as CVS and Subversion often forget to place source files under revision control.
- The machines on which the continuous integration system runs ideally provides a clean, well-defined build environment. If this environment is administered through proper SCM techniques, then builds produced by the system can be reproduced. In contrast, developer work environments are typically not under any kind of SCM control.
- In large projects, developers often work on a particular component of the project, and do not build and test the composition of those components (again since this is likely to take too long). To prevent the phenomenon of “big bang integration”, where components are only tested together near the end of the development process, it is important to test components together as soon as possible (hence *continuous integration*).
- It allows software to be *released* by automatically creating packages that users can download and install. To do this manually represents an often prohibitive amount of work, as one may want to produce releases for many different platforms: e.g., installers for Windows and Mac OS X, RPM or Debian packages for certain Linux distributions, and so on.

In its simplest form, a continuous integration tool sits in a loop building and releasing software components from a version management system. For each component, it performs the following tasks:

- It obtains the latest version of the component's source code from the version management system.
-

- It runs the component's build process (which presumably includes the execution of the component's test set).
- It presents the results of the build (such as error logs and releases) to the developers, e.g., by producing a web page.

Examples of continuous integration tools include Jenkins, CruiseControl Tinderbox, Sisyphus, Anthill and BuildBot. These tools have various limitations.

- They do not manage the *build environment*. The build environment consists of the dependencies necessary to perform a build action, e.g., compilers, libraries, etc. Setting up the environment is typically done manually, and without proper SCM control (so it may be hard to reproduce a build at a later time). Manual management of the environment scales poorly in the number of configurations that must be supported. For instance, suppose that we want to build a component that requires a certain compiler X. We then have to go to each machine and install X. If we later need a newer version of X, the process must be repeated all over again. An ever worse problem occurs if there are conflicting, mutually exclusive versions of the dependencies. Thus, simply installing the latest version is not an option. Of course, we can install these components in different directories and manually pass the appropriate paths to the build processes of the various components. But this is a rather tiresome and error-prone process.
- They do not easily support *variability in software systems*. A system may have a great deal of build-time variability: optional functionality, whether to build a debug or production version, different versions of dependencies, and so on. (For instance, the Linux kernel now has over 2,600 build-time configuration switches.) It is therefore important that a continuous integration tool can easily select and test different instances from the configuration space of the system to reveal problems, such as erroneous interactions between features. In a continuous integration setting, it is also useful to test different combinations of versions of subsystems, e.g., the head revision of a component against stable releases of its dependencies, and vice versa, as this can reveal various integration problems.

*Hydra*, is a continuous integration tool that solves these problems. It is built on top of the [Nix package manager](#), which has a purely functional language for describing package build actions and their dependencies. This allows the build environment for projects to be produced automatically and deterministically, and variability in components to be expressed naturally using functions; and as such is an ideal fit for a continuous build system.

## 1.2 About Us

Hydra is the successor of the Nix Buildfarm, which was developed in tandem with the Nix software deployment system. Nix was originally developed at the Department of Information and Computing Sciences, Utrecht University by the TraCE project (2003-2008). The project was funded by the Software Engineering Research Program Jacquard to improve the support for variability in software systems. Funding for the development of Nix and Hydra is now provided by the NIRICT LaQuSo Build Farm project.

## 1.3 About this Manual

This manual tells you how to install the Hydra buildfarm software on your own server and how to operate that server using its web interface.

## 1.4 License

Hydra is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Hydra is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the [GNU General Public License](#) for more details.

---

## 1.5 Hydra at `nixos.org`

The `nixos.org` installation of Hydra runs at <http://hydra.nixos.org/>. That installation is used to build software components from the [Nix](#), [NixOS](#), [GNU](#), [Stratego/XT](#), and related projects.

If you are one of the developers on those projects, it is likely that you will be using the NixOS Hydra server in some way. If you need to administer automatic builds for your project, you should pull the right strings to get an account on the server. This manual will tell you how to set up new projects and build jobs within those projects and write a `release.nix` file to describe the build process of your project to Hydra. You can skip the next chapter.

If your project does not yet have automatic builds within the NixOS Hydra server, it may actually be eligible. We are in the process of setting up a large buildfarm that should be able to support open source and academic software projects. Get in touch.

## 1.6 Hydra on your own buildfarm

If you need to run your own Hydra installation, [Chapter 2](#) explains how to download and install the system on your own server.

---

## Chapter 2

# Installation

This chapter explains how to install Hydra on your own build farm server.

### 2.1 Prerequisites

To install and use Hydra you need to have installed the following dependencies:

- Nix
- either PostgreSQL or SQLite
- many Perl packages, notably Catalyst, EmailSender, and NixPerl (see the [Hydra expression in Nixpkgs](#) for the complete list)

At the moment, Hydra runs only on GNU/Linux (*i686-linux* and *x86\_64\_linux*).

For small projects, Hydra can be run on any reasonably modern machine. For individual projects you can even run Hydra on a laptop. However, the charm of a buildfarm server is usually that it operates without disturbing the developer's working environment and can serve releases over the internet. In conjunction you should typically have your source code administered in a version management system, such as subversion. Therefore, you will probably want to install a server that is connected to the internet. To scale up to large and/or many projects, you will need at least a considerable amount of diskspace to store builds. Since Hydra can schedule multiple simultaneous build jobs, it can be useful to have a multi-core machine, and/or attach multiple build machines in a network to the central Hydra server.

Of course we think it is a good idea to use the [NixOS](#) GNU/Linux distribution for your buildfarm server. But this is not a requirement. The Nix software deployment system can be installed on any GNU/Linux distribution in parallel to the regular package management system. Thus, you can use Hydra on a Debian, Fedora, SuSE, or Ubuntu system.

### 2.2 Getting Nix

If your server runs NixOS you are all set to continue with installation of Hydra. Otherwise you first need to install Nix. The latest stable version can be found on [the Nix web site](#), along with a manual, which includes installation instructions.

### 2.3 Installation

The latest development snapshot of Hydra can be installed by visiting the URL <http://hydra.nixos.org/view/hydra-unstable> and using the one-click install available at one of the build pages. You can also install Hydra through the channel by performing the following commands:

```
nix-channel --add http://hydra.nixos.org/jobset/hydra/trunk/channel/latest
nix-channel --update
nix-env -i hydra
```

Command completion should reveal a number of command-line tools from Hydra:

```
hydra-build      hydra-init      hydra-update-gc-roots
hydra-eval-jobs  hydra-queue-runner
hydra-evaluator  hydra-server
```

## 2.4 Creating the database

Hydra stores its results in a database, which can be a PostgreSQL or SQLite database. The latter is easier to setup, but the former scales better.

To setup a PostgreSQL database with *hydra* as database name and user name, issue the following commands on the PostgreSQL server:

```
createuser -S -D -R -P hydra
createdb -O hydra hydra
```

Note that *\$prefix* is the location of Hydra in the nix store.

Hydra uses an environment variable to know which database should be used, and a variable which point to a location that holds some state. To set these variables for a PostgreSQL database, add the following to the file `~/.profile` of the user running the Hydra services.

```
export HYDRA_DBI="dbi:Pg:dbname=hydra;host=dbserver.example.org;user=hydra;"
export HYDRA_DATA=/var/lib/hydra
```

You can provide the username and password in the file `~/.pgpass`, e.g.

```
dbserver.example.org:*:hydra:hydra:password
```

Make sure that the *HYDRA\_DATA* directory exists and is writable for the user which will run the Hydra services. For a SQLite database, the *HYDRA\_DBI* should be set to something like `dbi:SQLite:/path/to/hydra.sqlite`

Having set these environment variables, you can now initialise the database by doing:

```
hydra-init
```

To add a user *root* with *admin* privileges, execute:

```
echo "INSERT INTO Users(userName, emailAddress, password) VALUES ('root', 'some@email. ↵
  adress.com', '$(echo -n foobar | shasum | cut -c1-40)');" | psql hydra
echo "INSERT INTO UserRoles(userName, role) values('root', 'admin');" | psql hydra
```

For SQLite the same commands can be used, with `psql hydra` replaced by `sqlite3 /path/to/hydra.sqlite`.

## 2.5 Upgrading

If you're upgrading Hydra from a previous version, you should do the following to perform any necessary database schema migrations:

```
hydra-init
```

## 2.6 Getting Started

To start the Hydra web server, execute:

```
hydra-server
```

When the server is started, you can browse to <http://localhost:3000/> to start configuring your Hydra instance.

The **hydra-server** command launches the web server. There are two other processes that come into play:

- The *evaluator* is responsible for periodically evaluating job sets, checking out their dependencies off their version control systems (VCS), and queueing new builds if the result of the evaluation changed. It is launched by the **hydra-evaluator** command.
- The *queue runner* launches builds (using Nix) as they are queued by the evaluator, scheduling them onto the configured Nix hosts. It is launched using the **hydra-queue-runner** command.

All three processes must be running for Hydra to be fully functional, though it's possible to temporarily stop any one of them for maintenance purposes, for instance.

---

## Chapter 3

# Creating and Managing Projects

Once Hydra is installed and running, the next step is to add projects to the build farm. We follow the example of the [Patchelf project](#), a software tool written in C and using the GNU Build System (GNU Autoconf and GNU Automake).

Log in to the web interface of your Hydra installation using the user name and password you inserted in the database (by default, Hydra's web server listens on `localhost:3000`). Then follow the "Create Project" link to create a new project.

### 3.1 Project Information

A project definition consists of some general information and a set of job sets. The general information identifies a project, its owner, and current state of activity. Here's what we fill in for the patchelf project:

```
Identifier: patchelf
```

The *identifier* is the identity of the project. It is used in URLs and in the names of build results.

The identifier should be a unique name (it is the primary database key for the project table in the database). If you try to create a project with an already existing identifier you'd get an error message such as:

```
I'm very sorry, but an error occurred:  
DBIx::Class::ResultSet::create(): DBI Exception: DBD::SQLite::st execute failed: column ↵  
name is not unique(19) at dbdimp.c line 402
```

So try to create the project after entering just the general information to figure out if you have chosen a unique name. Job sets can be added once the project has been created.

```
Display name: Patchelf
```

The *display name* is used in menus.

```
Description: A tool for modifying ELF binaries
```

The *description* is used as short documentation of the nature of the project.

```
Owner: eelco
```

The *owner* of a project can create and edit job sets.

```
Enabled: Yes
```

Only if the project is *enabled* are builds performed.

Once created there should be an entry for the project in the sidebar. Go to the project page for the [Patchelf project](#).

## 3.2 Job Sets

A project can consist of multiple *job sets* (hereafter *jobsets*), separate tasks that can be built separately, but may depend on each other (without cyclic dependencies, of course). Go to the [Edit](#) page of the Patchelf project and "Add a new jobset" by providing the following "Information":

```
Identifier:      trunk
Description:    Trunk
Nix expression: release.nix in input patchelfSrc
```

This states that in order to build the `trunk` jobset, the Nix expression in the file `release.nix`, which can be obtained from input `patchelfSrc`, should be evaluated. (We'll have a look at `release.nix` later.)

To realize a job we probably need a number of inputs, which can be declared in the table below. As many inputs as required can be added. For `patchelf` we declare the following inputs.

```
patchelfSrc
  'Subversion checkout' https://svn.nixos.org/repos/nix/patchelf/trunk

nixpkgs 'Subversion checkout' https://svn.nixos.org/repos/nix/nixpkgs/trunk

officialRelease Boolean false

system String value "i686-linux"
```

## 3.3 Release Set

there must be one primary job check the radio button of exactly one job <https://svn.nixos.org/repos/nix/nixpkgs/trunk>

## 3.4 Building Jobs

## 3.5 Build Recipes

Build jobs and *build recipes* for a jobset are specified in a text file written in the [Nix language](#). The recipe is actually called a *Nix expression* in Nix parlance. By convention this file is often called `release.nix`.

The `release.nix` file is typically kept under version control, and the repository that contains it one of the build inputs of the corresponding—often called `hydraConfig` by convention. The repository for that file and the actual file name are specified on the web interface of Hydra under the `Setup` tab of the jobset's overview page, under the `Nix expression` heading. See, for example, the [jobset overview page](#) of the PatchELF project, and [the corresponding Nix file](#).

Knowledge of the Nix language is recommended, but the example below should already give a good idea of how it works:

**Example 3.1** `release.nix` file for GNU Hello

```

{ nixpkgs }: ❶

let
  pkgs = import nixpkgs {}; ❷

  jobs = rec { ❸

    tarball = ❹
      { helloSrc }: ❺

      pkgs.releaseTools.sourceTarball { ❻
        name = "hello-tarball";
        src = helloSrc;
        buildInputs = (with pkgs; [ gettext texLive texinfo ]);
      };

    build = ❼
      { tarball ? jobs.tarball {} ❽
      , system ? builtins.currentSystem
      }:

      let pkgs = import nixpkgs { inherit system; }; in
      pkgs.releaseTools.nixBuild { ❾
        name = "hello";
        src = tarball;
        configureFlags = [ "--disable-silent-rules" ];
      };

  };
in
  jobs ❿

```

Example 3.1 shows what a `release.nix` file for **GNU Hello** would you like. GNU Hello is representative of many GNU and non-GNU free software projects:

- it uses the GNU Build System, namely GNU Autoconf, and GNU Automake; for users, it means it can be installed using the usual `./configure && make install` procedure;
- it uses Gettext for internationalization;
- it has a Texinfo manual, which can be rendered as PDF with TeX.

The file defines a jobset consisting of two jobs: `tarball`, and `build`. It contains the following elements (referenced from the figure by numbers):

- ❶ This specifies a function of one named arguments, `nixpkgs`. This function and those defined below is called by Hydra. Here the `nixpkgs` argument is meant to be a checkout of the **Nixpkgs** software distribution. Hydra inspects the formal argument list of the function (here, the `nixpkgs` argument) and passes it the corresponding parameter specified as a build input on Hydra's web interface. In this case, the web interface should show a `nixpkgs` build input, which is a checkout of the Nixpkgs source code repository.
- ❷ This defines a variable `pkgs` holding the set of packages provided by Nixpkgs.
- ❸ This defines a variable holding the two Hydra jobs—an *attribute set* in Nix.
- ❹ This is the definition of the first job, named `tarball`. The purpose of this job is to produce a usable source code tarball.
- ❺ The `tarball` takes an additional argument called `helloSrc`. Again, this argument is passed by Hydra and is meant to be a checkout of GNU Hello's source code repository.

- 6 The `tarball` job calls the `sourceTarball` function, which (roughly) runs **`autoreconf && ./configure && make dist`** on the checkout. The `buildInputs` attribute specifies additional software dependencies for the job<sup>1</sup>.
- 7 This is the definition of the `build` job, whose purpose is to build Hello from the tarball produced above.
- 8 The `build` function takes two additional parameter: `tarball`, which is meant to be the result of the `tarball` job, and `system`, which should be a string defining the Nix system type—e.g., `"x86_64-linux"`.  
Again, these parameters are passed by Hydra when it calls `build`. Thus, they must be defined as build inputs in Hydra: `tarball` should have type `Build Output`, its value being the latest output of the `tarball` job, and `system` should be a string.  
The question mark after `tarball` and `system` defines default values for these arguments, and is only useful for debugging.
- 9 The `build` job calls the `nixBuild` function, which unpacks the tarball, then runs **`./configure && make && make check && make install`**.
- 10 Finally, the set of jobs is returned to Hydra, as a Nix attribute set.

### 3.6 Building from the Command Line

It is often useful to test a build recipe, for instance before it is actually used by Hydra, when testing changes, or when debugging a build issue. Since build recipes for Hydra jobsets are just plain Nix expressions, they can be evaluated using the standard Nix tools.

To evaluate the `tarball` jobset of Example 3.1, just run:

```
$ nix-build release.nix -A tarball
```

However, doing this with Example 3.1 as is will probably yield an error like this:

```
error: cannot auto-call a function that has an argument without a default value ('nixpkgs')
```

This is because no value was specified for the `nixpkgs` argument of the Nix expression.

This is fixed by providing a default value for that argument in the Nix expression, which will allow **nix-build** to auto-call the function: instead of writing `{ nixpkgs }:`, we now write `{ nixpkgs ? <nixpkgs> }:`. What it means is that, by default, the `nixpkgs` variable will be bound to the absolute path of any `nixpkgs` file found in the Nix search path. Similarly, a default value for `helloSrc` needs to be provided.

Thus, assuming a checkout of `Nixpkgs` is available under `$HOME/src/nixpkgs`, the `tarball` jobset can now be evaluated by running:

```
$ nix-build -I ~/src release.nix -A tarball
```

Similarly, the `build` jobset can be evaluated:

```
$ nix-build -I ~/src release.nix -A build
```

The `build` job reuses the result of the `tarball` job, rebuilding it only if it needs to.

<sup>1</sup>The package names used in `buildInputs`—e.g., `texLive`—are the names of the *attributes* corresponding to these packages in `Nixpkgs`, specifically in the `all-packages.nix` file. See the section entitled “Package Naming” in the `Nixpkgs` manual for more information.

## 3.7 Adding More Jobs

Example 3.1 illustrates how to write the most basic jobs, `tarball` and `build`. In practice, much more can be done by using features readily provided by `Nixpkgs` or by creating new jobs as customizations of existing jobs.

For instance, test coverage report for projects compiled with GCC can be automatically generated using the `coverageAnalysis` function provided by `Nixpkgs` instead of `nixBuild`. Back to our GNU Hello example, we can define a `coverage` job that produces an HTML code coverage report directly readable from the corresponding Hydra build page:

```
coverage =
  { tarball ? jobs.tarball {}
  , system ? builtins.currentSystem
  }:

  let pkgs = import nixpkgs { inherit system; }; in
  pkgs.releaseTools.coverageAnalysis {
    name = "hello";
    src = tarball;
    configureFlags = [ "--disable-silent-rules" ];
  };
```

As can be seen, the only difference compared to `build` is the use of `coverageAnalysis`.

`Nixpkgs` provides many more build tools, including the ability to run `build` in virtual machines, which can themselves run another GNU/Linux distribution, which allows for the creation of packages for these distributions. Please see [the `pkgs/build-support/release` directory](#) of `Nixpkgs` for more. The NixOS manual also contains information about whole-system testing in virtual machine.

Now, assume we want to build Hello with an old version of GCC, and with different **configure** flags. A new `build_exotic` job can be written that simply *overrides* the relevant arguments passed to `nixBuild`:

```
build_exotic =
  { tarball ? jobs.tarball {}
  , system ? builtins.currentSystem
  }:

  let
    pkgs = import nixpkgs { inherit system; };
    build = jobs.build { inherit tarball system; };
  in
  pkgs.lib.overrideDerivation build (attrs: {
    buildInputs = [ pkgs.gcc33 ];
    preConfigure = "gcc --version";
    configureFlags =
      attrs.configureFlags ++ [ "--disable-nls" ];
  });
```

The `build_exotic` job reuses `build` and overrides some of its arguments: it adds a dependency on GCC 3.3, a pre-configure phase that runs `gcc --version`, and adds the `--disable-nls` configure flags.

This customization mechanism is very powerful. For instance, it can be used to change the way Hello and *all* its dependencies—including the C library and compiler used to build it—are built. See the `Nixpkgs` manual for more.