tweag / jupyterWith

<> Code | Issues 31 | Pull requests 5 | Actions | Projects

master ▾ | Go to file | Add file ▾ | Code ▾

guaraqe Merge pull request … ... ✓ on Oct 2 **269**

.github/w…	Test only standard kernels	last month
example	Bump yargs-parser in /ex…	2 months ago
kernels	Re-add ihaskell to ghc pa…	last month
lib	Build ihaskell_labextensio…	3 months ago
nix	bump ihaskell. No need fo…	2 months ago
tests	Some reformatting to re-l…	last month
.gitignore	Add .gitignore	2 years ago
LICENSE	add LICENSE	2 years ago
Makefile	add Makefile to project	12 months ago
READM…	bump ihaskell. No need fo…	2 months ago
default.nix	Build ihaskell_labextensio…	3 months ago
kernels.png	update kernels image	2 years ago

About

declarative and reproducible Jupyter environments - powered by Nix

#jupyterlab #nix #jupyter #jupyter-notebooks #reproducibility

Readme

MIT License

Releases

No releases published

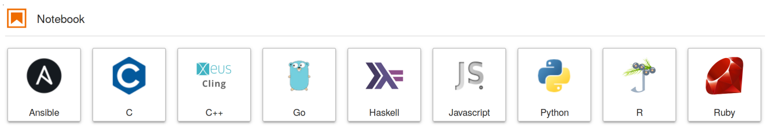
Packages

No packages published

Contributors 17

README.md

JupyterWith



This repository provides a Nix-based framework for the definition of declarative and reproducible Jupyter environments. These environments include JupyterLab - configurable with extensions - the classic notebook, and configurable Jupyter kernels.

In practice, a Jupyter environment is defined in a single `shell.nix` file which can be distributed together with a notebook as a self-contained reproducible package.

These kernels are currently included by default:

- IPython
- IHaskell (long build time)
- CKernel
- IRuby
- Juniper RKernel (limited jupyterlab support)
- IRkernel
- Ansible Kernel
- Xeus-Cling CPP (experimental, not yet configurable with packages, long build time)
- IJavascript (not yet configurable with packages)
- gophernotes (not yet configurable with packages)

Example notebooks are here.

Getting started

Nix must be installed in order to use JupyterWith. A simple JupyterLab environment with kernels can be defined in a `shell.nix` file such as:

```
let
  jupyter = import (builtins.fetchGit {
    url = https://github.com/tweag/jupyterWit
    rev = "";
```

+ 6 contributors

Languages

●	**Jupyter Notebook**	92.8%
●	**Nix**	6.8%
●	**Ruby**	0.2%
●	**Python**	0.1%
●	**Makefile**	0.1%
●	**Shell**	0.0%

```
  }) {};

  iPython = jupyter.kernels.iPythonWith {
    name = "python";
    packages = p: with p; [ numpy ];
  };

  iHaskell = jupyter.kernels.iHaskellWith {
    extraIHaskellFlags = "--codemirror Haskel
    name = "haskell";
    packages = p: with p; [ hvega formatting
  };

  jupyterEnvironment =
    jupyter.jupyterlabWith {
      kernels = [ iPython iHaskell ];
    };
in
  jupyterEnvironment.env
```

JupyterLab can then be started by running:

```
nix-shell --command "jupyter lab"
```

This can take a while, especially when it is run for the first time because all dependencies of JupyterLab have to be downloaded, built and installed. Subsequent runs are instantaneous for the same environment, or much faster even when some packages or kernels are changed, since a lot will already be cached.

This process can be largely accelerated by using cachix:

```
cachix use jupyterwith
```

Using JupyterLab extensions

Lab extensions can be used with JupyterWith by generating a JupyterLab frontend directory. This is so for two reasons:

- Jupyter expects this folder to be mutable, so extensions can be turned on and off. This makes it impossible for it to be in the Nix store and be completely useful.
- Jupyter manages its own packages, which is hard to do deterministically. It is easier to just manage extensions manually for the moment.

This can be done by running `nix-shell` from the folder with the `shell.nix` file and then using the `generate-directory` executable that is available from inside the shell.

```
$ generate-directory [EXTENSIONS]
```

That is, if you want to install the `jupyterlab-ihaskell` and `jupyterlab_boken` extensions, you can do:

```
$ generate-directory jupyterlab-ihaskell
jupyterlab_bokeh
```

This will generate a folder called `jupyterlab` (this name is always the same, and it is not configurable for the moment). This folder can then be passed to `jupyterWith`. With extensions, the example above becomes:

```
let
  jupyter = import (builtins.fetchGit {
    url = https://github.com/tweag/jupyterWit
    rev = "";
  }) {};
  iPython = jupyter.kernels.iPythonWith {
    name = "python";
    packages = p: with p; [ numpy ];
  };

  iHaskell = jupyter.kernels.iHaskellWith {
    name = "haskell";
    packages = p: with p; [ hvega formatting
  };

  jupyterEnvironment =
    jupyter.jupyterlabWith {
      kernels = [ iPython iHaskell ];
      ## The generated directory goes here
      directory = ./jupyterlab;
    };
in
  jupyterEnvironment.env
```

After the folder is generated, it can be manipulated as a regular Jupyter folder. Take a look at the source of the `generateDirectory` function for more advanced usage.

Impure generator

WARNING: This is not guaranteed to work every time. Read this section thoroughly before trying.

Another option, which is useful for simple tests, is to use the impure `mkDirectoryWith` Nix function that comes with this repo:

```
{
  jupyterEnvironment =
    jupyter.jupyterlabWith {
      kernels = [ iPython iHaskell ];
      ## The directory is generated here
      directory = mkDirectoryWith {
        extensions = [
          "jupyterlab-ihaskell"
          "jupyterlab_bokeh"
        ];
      };
    };
  }
```

In this case, you must make sure that sandboxing is disabled in your Nix configuration. Newer Nix versions have it enabled by default. Sandboxing can be disabled:

- either by running `nix-shell --option sandbox false` ; or
- by setting `sandbox = false` in `/etc/nix/nix.conf` .

For this to work, your user must be in the `nix.trustedUsers` list in `configuration.nix` .

The JupyterLab docs say that the `extensions` list elements can be "the name of a valid JupyterLab extension npm package on npm," or "can be a local directory containing the extension, a gzipped tarball, or a URL to a gzipped tarball."

For a "local directory containing the extension" use the impure `mkBuildExtension` function, for example:

```
extensions = [
  jupyter.mkBuildExtension "${ihaskel
];
```

Changes to the default package sets

The kernel environments rely on the default package sets that are provided by the Nixpkgs repository that is defined in the nix folder. These package sets can be modified using overlays, for example to add a new Python package from PIP. You can see examples of this in the `./nix/python-overlay.nix` and `./nix/haskell-overlay.nix` files. You can also modify the package set directly in the `shell.nix` file, as demonstrated in this example that adds a new Haskell package to the package set.

Building the Docker images

One can easily generate Docker images from Jupyter environments defined with JupyterWith with a `docker.nix` file:

```
let
  jupyter = import (builtins.fetchGit {
    url = https://github.com/tweag/jupyterWit
    rev = "";
  }) {};

  jupyterEnvironment = jupyter.jupyterlabWith
in
  jupyter.mkDockerImage {
    name = "jupyter-image";
    jupyterlab = jupyterEnvironment;
  }
```

`nix-build docker.nix` builds the image and it can be passed to Docker with:

```
$ cat result | docker load
$ docker run -v $(pwd)/example:/data -p
8888:8888 jupyter-image:latest
```

Adding packages to the environment

It is possible to add extra packages to the JupyterWith environment. For example, if you want to add `pandoc` to the environment in order to convert notebooks to PDF, you can do the following.

```
let
  jupyter = import (builtins.fetchGit {
    url = https://github.com/tweag/jupyterWit
    rev = "";
  }) {};

  jupyterEnvironment = jupyter.jupyterlabWith
    extraPackages = p: [p.pandoc];
  };
```

Here, the `p` argument corresponds to Nixpkgs checkout being used. Note that this can easily be made to use packages from outside `jupyterWith` 's scope, by providing a function that ignores its argument:

```
  extraPackages = _ : [ pkgs.pandoc ];
```

You may also bring all inputs from a package in scope using the "extraInputsFrom" argument:

```
let
  jupyter = import (builtins.fetchGit {
    url = https://github.com/tweag/jupyterWit
    rev = "";
  }) {};

  jupyterEnvironment = jupyter.jupyterlabWith
    extraInputsFrom = p: [p.pythonPackages.nu
  };
```

Adding packages to the Jupyter PATH

Jupyter is ran within its own environment, meaning that packages added with the method above will not be visible by Jupyter. The `extraJupyterPath` argument can be used to add extra packages to the scope of Jupyter itself (i.e. not the kernel). A possible use case of this is to make accessible libraries to the Python executable that is used by Jupyter itself, which can be necessary when installing server extensions (see, for example, this issue).

Using the example in the linked issue, one can add the `jupytext` package to the Jupyter scope with:

```
let
  jupyter = import (builtins.fetchGit {
    url = https://github.com/tweag/jupyterWit
    rev = "";
  }) {};

  jupyterEnvironment = jupyter.jupyterlabWith
    extraJupyterPath = pkgs:
      "${pkgs.python3Packages.jupytext}/lib/p
  };
```

Using as an overlay

JupyterWith can be used as an overlay. That is, you can add JupyterWith's packages to your own Nixpkgs snapshot, with some caveats:

- Some overrides need to be added so that kernels work correctly. That's why we import the overlays in the file below.
- There are chances that the overrides defined here will not be compatible with your snapshot. Your mileage may vary.

In order to use it as an overlay, add the following (replacing `commit-hash` and `<nixpkgs>` with suitable values) to a `shell.nix` file:

```
let
  # Path to the JupyterWith folder.
  jupyterWithPath = builtins.fetchGit {
    url = https://github.com/tweag/jupyterWit
    rev = "commit-hash";
  };

  # Importing overlays from that path.
  overlays = [
    # Only necessary for Haskell kernel
    (import "${jupyterWithPath}/nix/haskell-c
    # Necessary for Jupyter
    (import "${jupyterWithPath}/nix/python-ov
    (import "${jupyterWithPath}/nix/overlay.r
  ];

  # Your Nixpkgs snapshot, with JupyterWith p
  pkgs = import <nixpkgs> { inherit overlays;

  # From here, everything happens as in other
  jupyter = pkgs.jupyterWith;

  jupyterEnvironment =
    jupyter.jupyterlabWith {
    };
in
  jupyterEnvironment.env
```

Contributing

Kernels

New kernels are easy to add to `jupyterWith`. Kernels are derivations that expose a `kernel.json` file with all information that is required to run a kernel to the main Jupyter derivation. Examples can be found in the kernels folder.

About extensions

In order to install extensions, JupyterLab runs `yarn` to resolve the precise compatible versions of the JupyterLab core modules, extensions, and all of their dependencies. This resolution process is difficult to replicate with Nix. We therefore decided to use the JupyterLab build system for now to prebuild a custom JupyterLab version with extensions.

If you have ideas on how to make this process more declarative, feel free to create an issue or PR.

Nixpkgs

The final goal of this project is to be completely integrated into Nixpkgs eventually. However, the migration path, in part due to extensions, is not completely clear.

If you have ideas, feel free to create an issue so that we can discuss.

License

This project is licensed under the MIT License. See the LICENSE file for details.