repo2docker Documentation

Release 0.1

Project Jupyter

Feb 21, 2019

Contents

jupyter-repo2docker is a tool to build, run, and push Docker images from source code repositories. repo2docker fetches a repository (e.g., from GitHub or other locations) and builds a container image based on the configuration files found in the repository. It can be used to explore a repository locally by building and executing the constructed image of the repository.

Please report Bugs, ask questions or contribute to the project.

CHAPTER 1

Site Contents

1.1 Installing `repo2docker`

repo2docker requires Python 3.4 and above on Linux and macOS. See *below* for more information about Windows support.

1.1.1 Prerequisite: docker

Install Docker as it is required to build Docker images. The Community Edition, is available for free.

Recent versions of Docker are recommended. The latest version of Docker, `18.03`, successfully builds repositories from binder-examples. The BinderHub helm chart uses version `17.11.0-ce-dind`. See the helm chart for more details.

1.1.2 Installing with `pip`

We recommend installing `repo2docker` with the `pip` tool:

```
python3 -m pip install jupyter-repo2docker
```

For information on using `repo2docker`, see *Using repo2docker*.

1.1.3 Installing from source code

Alternatively, you can install repo2docker from source, i.e. if you are contributing back to this project:

```
git clone https://github.com/jupyter/repo2docker.git
cd repo2docker
pip install -e .
```

That's it! For information on using `repo2docker`, see *Using repo2docker*.

1.1.4 Windows support

Windows support for repo2docker is still in the experimental stage.

An article about using Windows and the WSL (Windows Subsytem for Linux or Bash on Windows) provides additional information about Windows and docker.

1.1.5 JupyterHub-ready images

JupyterHub allows multiple users to collaborate on a shared Jupyter server. repo2docker can build Docker images that can be shared within a JupyterHub deployment. For example, mybinder.org uses JupyterHub and repo2docker to allow anyone to build a Docker image of a git repository online and share an executable version of the repository with a URL to the built image.

To build JupyterHub-ready Docker images with repo2docker, the version of your JupterHub deployment must be included in the environment.yml or requirements.txt of the git repositories you build.

If your instance of JupyterHub uses DockerSpawner, you will need to set its command to run jupyterhub-singleuser by adding this line in your configuration file:

```
c.DockerSpawner.cmd = ['jupyterhub-singleuser']
```

1.2 Using repo2docker

Docker **must be running** in order to run repo2docker. For more information on installing repo2docker, see *Installing repo2docker*.

repo2docker performs two steps:

1. builds a Docker image from a git repo
2. runs a Jupyter server within the image to explore the repository

repo2docker is called with this command:

```
jupyter-repo2docker <URL-or-path to repository>
```

where <URL-or-path to repository> is a URL or path to the source repository.

For example, use the following to build an image of Peter Norvig's Pytudes:

```
jupyter-repo2docker https://github.com/norvig/pytudes
```

To build a particular branch and commit, use the argument --ref to specify the branch-name or commit-hash:

```
jupyter-repo2docker https://github.com/norvig/pytudes --ref 9ced85dd9a84859d0767369e58f33912a214a3cf
```

Tip: For reproducible research, we recommend specifying a commit-hash to deterministically build a fixed version of a repository. Not specifying a commit-hash will result in the latest commit of the repository being built.

Building the image may take a few minutes.

During building, repo2docker clones the repository to obtain its contents and inspects the repository for *configuration files*.

By default, `repo2docker` will assume you are using Python 3.6 unless you include the version of Python in your *configuration files*. `repo2docker` support is best with Python 2.7, 3.5, and 3.6. In the case of this repository, a Python version is not specified in their configuration files and Python 3.6 is installed.

Pytudes uses a requirements.txt file to specify its Python environment. `repo2docker` uses `pip` to install dependencies listed in the `requirement.txt` in the image. To learn more about configuration files in `repo2docker` visit *Configuration Files*.

When the image is built, a message will be output to your terminal:

```
Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
    http://0.0.0.0:36511/?token=f94f8fabb92e22f5bfab116c382b4707fc2cade56ad1ace0
```

Pasting the URL into your browser will open Jupyter Notebook with the dependencies and contents of the source repository in the built image.

Because JupyterLab is a server extension of the classic Jupyter Notebook server, you can launch JupyterLab by opening Jupyter Notebook and visiting the `` ` ``/lab`` to the end of the URL:

```
http(s)://<server:port>/<lab-location>/lab
```

To switch back to the classic notebook, add `/tree` to the URL:

```
http(s)://<server:port>/<lab-location>/tree
```

To learn more about URLs in JupyterLab and Jupyter Notebook, visit starting JupyterLab.

1.2.1 `--debug` and `--no-build`

To debug the docker image being built, pass the `--debug` parameter:

```
jupyter-repo2docker --debug https://github.com/norvig/pytudes
```

This will print the generated `Dockerfile`, build it, and run it.

To see the generated `Dockerfile` without actually building it, pass `--no-build` to the commandline. This `Dockerfile` output is for **debugging purposes** of `repo2docker` only - it can not be used by docker directly.

```
jupyter-repo2docker --no-build --debug https://github.com/norvig/pytudes
```

1.3 Configuration Files

`repo2docker` looks for configuration files in the repository being built to determine how to build it. In general, `repo2docker` uses the same configuration files as other software installation tools, rather than creating new custom configuration files.

A number of `repo2docker` configuration files can be combined to compose more complex setups.

`repo2docker` will look for configuration files in either:

- A folder named `binder/` in the root of the repository.
- The root directory of the repository.

If the folder `binder/` is located at the top level of the repository, only configuration files in the `binder/` **folder will be considered**.

The binder examples organization on GitHub contains a list of sample repositories for common configurations that `repo2docker` can build with various configuration files such as Python and R installation in a repository.

Below is a list of supported configuration files (roughly in the order of build priority):

- *Dockerfile*
- *environment.yml*
- *requirements.txt*
- *REQUIRE*
- *install.R*
- *apt.txt*
- *setup.py*
- *postBuild*
- *runtime.txt*

1.3.1 `Dockerfile`

In the majority of cases, providing your own Dockerfile is not necessary as the base images provide core functionality, compact image sizes, and efficient builds. We recommend trying the other configuration files before deciding to use your own Dockerfile.

With Dockerfiles, a regular Docker build will be performed. **If a Dockerfile is present, all other configuration files will be ignored.**

See the Binder Documentation for best-practices with Dockerfiles.

1.3.2 `environment.yml`

`environment.yml` is the standard configuration file used by Anaconda, conda, and miniconda that lets you install Python packages. You can also install files from pip in your `environment.yml` as well. Our example environment.yml shows how one can specify a conda environment for repo2docker.

You can also specify which Python version to install in your built environment with `environment.yml`. By default, `repo2docker` **installs Python 3.6** with your `environment.yml` unless you include the version of Python in the file. `conda` supports Python versions 3.6, 3.5, 3.4, and 2.7. `repo2docker` support is best with Python 3.6, 3.5, and 2.7. If you include a Python version in a `runtime.txt` file in addition to your `environment.yml`, your `runtime.txt` **will be ignored**.

1.3.3 `requirements.txt`

This specifies a list of Python packages that should be installed in your environment. Our requirements.txt example on GitHub shows a typical requirements file.

1.3.4 `REQUIRE`

This specifies a list of Julia packages. Repositories with a `REQUIRE` file **must also contain an** `environment.yml` **file**. To see an example of a Julia repository with `REQUIRE` and `environment.yml`, visit binder-examples/julia-python.

1.3.5 install.R

This is used to install R libraries pinned to a specific snapshot on MRAN. To set the date of the snapshot add a *runtime.txt*. For an example install.R file, visit our example install.R file.

1.3.6 apt.txt

A list of Debian packages that should be installed. The base image used is usually the latest released version of Ubuntu.

We use apt.txt, for example, to install LaTeX in our example apt.txt for LaTeX.

1.3.7 setup.py

To install your repository like a Python package, you may include a setup.py file. repo2docker installs setup.py files by running pip install -e ..

While one can specify dependencies in setup.py, repo2docker **requires configuration files such as** environment.yml or requirements.txt to install dependencies during the build process.

1.3.8 postBuild

A script that can contain arbitrary commands to be run after the whole repository has been built. If you want this to be a shell script, make sure the first line is `#!/bin/bash.

An example use-case of postBuild file is JupyterLab's demo on mybinder.org. It uses a postBuild file in a folder called binder to prepare their demo for binder.

start

A script that can contain simple commands to be run at runtime (as an *ENTRYPOINT <https://docs.docker.com/engine/reference/builder/#entrypoint>* to the docker container). If you want this to be a shell script, make sure the first line is `#!/bin/bash. The last line must be `exec "$@"` equivalent.

Use this to set environment variables that software installed in your container expects to be set. This script is executed each time your binder is started and should at most take a few seconds to run.

If you only need to run things once during the build phase use *postBuild*.

1.3.9 runtime.txt

This allows you to control the runtime of Python or R.

To use python-2.7: add python-2.7 in runtime.txt file. The repository will run in a virtualenv with Python 2 installed. To see a full example repository, visit our Python2 example. **Python versions in ``runtime.txt`` are ignored when** environment.yml **is present in the same folder**.

repo2docker uses R libraries pinned to a specific snapshot on MRAN. You need to have a runtime.txt file that is formatted as r-<YYYY>-<MM>-<DD>, where YYYY-MM-DD is a snapshot at MRAN that will be used for installing libraries.

To see an example R repository, visit our R example in binder-examples.

1.4 Frequently Asked Questions (FAQ)

A collection of frequently asked questions with answers. If you have a question and have found an answer, send a PR to add it here!

1.4.1 How should I specify another version of Python 3?

One can specify a Python version in the `environment.yml` file of a repository.

1.4.2 Can I add executable files to the user's PATH?

Yes! With a :ref:`postBuild` file, you can place any files that should be called from the command line in the folder `~/.local/`. This folder will be available in a user's PATH, and can be run from the command line (or as a subsequent build step.)

1.4.3 How do I set environment variables?

Use the `-e` or `--env` flag for each variable that you want to define.

For example `jupyter-repo2docker -e VAR1=val1 -e VAR2=val2 ...`

1.4.4 Can I use repo2docker to bootstrap my own Dockerfile?

No, you can't.

If you pass the `--debug` flag to `repo2docker`, it outputs the intermediate Dockerfile that is used to build the docker image. While it is tempting to copy this as a base for your own Dockerfile, that is not supported & in most cases will not work. The `--debug` output is just our intermediate generated Dockerfile, and is meant to be built in a very specific way. Hence the output of `--debug` can not be built with a normal `docker build -t .` or similar traditional docker command.

Check out the binder-examples GitHub organization for example repositories you can copy & modify for your own use!

1.5 Using `repo2docker` as part of your Continuous Integration

We've created for you the continuous-build repository so that you can push a Docker container to Docker Hub directly from a Github repository that has a Jupyter notebook. Here are instructions to do this.

1.5.1 Getting Started

Today you will be doing the following:

1. Fork and clone the continuous-build Github repository to obtain the hidden `.circleci` folder.
2. creating an image repository on Docker Hub
3. connecting your repository to CircleCI
4. push, commit, or create a pull request to trigger a build.

You don't need to install any dependencies on your host to build the container, it will be done on a continuous integration server, and the container built and available to you to pull from Docker Hub.

Step 1. Clone the Repository

First, fork the continuous-build Github repository to your account, and clone the branch.

> git clone https://www.github.com/<username>/continuous-build # or git clone git@github.com:<username>/continuous-build.git

Step 2. Choose your Configuration

The hidden folder *.circleci/config.yml* has instructions for CircleCI to automatically discover and build your repo2docker jupyter notebook container. The default template provided in the repository in this folder will do the most basic steps, including:

1. clone of the repository with the notebook that you specify
2. build
3. push to Docker Hub

This repository aims to provide templates for your use. If you have a request for a new template, please let us know. We will add templates as they are requested to do additional tasks like test containers, run nbconvert, etc.

Thus, if I have a repository named `myrepo` and I want to use the default configuration on circleCI, I would copy it there from the `continuous-build` folder. In the example below, I'm creating a new folder called "myrepo" and then copying the entire folder there.

> mkdir -p myrepo cp -R continuous-build/.circleci myrepo/

You would then logically create a Github repository in the "myrepo" folder, add the circleci configuration folder, and continue on to the next steps.

> cd myrepo git init git add .circleci

Step 3. Docker Hub

Go to Docker Hub, log in, and click the big blue button that says "create repository" (not an automated build). Choose an organization and name that you like (in the traditional format `<ORG>/<NAME>`), and remember it! We will be adding it, along with your Docker credentials, to be encrypted CircleCI environment variables.

Step 4. Connect to CircleCI

If you navigate to the main app page you should be able to click "Add Projects" and then select your repository. If you don't see it on the list, then select a different organization in the top left. Once you find the repository, you can click the button to "Start Building" and accept the defaults.

Before you push or trigger a build, let's set up the following environment variables. Also in the project interface on CirleCi, click the gears icon next to the project name to get to your project settings. Under settings, click on the "Environment Variables" tab. In this section, you want to define the following:

1. `CONTAINER_NAME` should be the name of the Docker Hub repository you just created.
2. `DOCKER_TAG` is the tag you want to use. If not defined, will use first 10 characters of commit.
3. `DOCKER_USER` and `DOCKER_PASS` should be your credentials (to allowing pushing)

4. `REPO_NAME` should be the full Github url (or other) of the repository with the notebook. This doesn't have to coincide with the repository you are using to do the build (e.g., "myrepo" in our example).

If you don't define the `CONTAINER_NAME` it will default to be the repository where it is building from, which you should only do if the Docker Hub repository is named equivalently. If you don't define either of the variables from step 3. for the Docker credentials, your image will build but not be pushed to Docker Hub. Finally, if you don't define the `REPO_NAME` it will again use the name of the repository defined for the `CONTAINER_NAME`.

Step 5. Push Away, Merrill!

Once the environment variables are set up, you can push or issue a pull request to see circle build the workflow. Remember that you only need the `.circleci/config.yml` and not any other files in the repository. If your notebook is hosted in the same repository, you might want to add these, along with your requirements.txt, etc.

Tip: By default, new builds on CircleCI will not build for pull requests and you can change this default in the settings. You can easily add filters (or other criteria and actions) to be performed during or after the build by editing the `.circleci/config.yml` file in your repository.

Step 5. Use Your Container!

You should then be able to pull your new container, and run it! Here is an example:

> docker pull <ORG>/<NAME> docker run -it –name repo2docker -p 8888:8888 <ORG>/<NAME> jupyter notebook –ip 0.0.0.0

For a pre-built working example, try the following:

> docker pull vanessa/repo2docker docker run -it –name repo2docker -p 8888:8888 vanessa/repo2docker jupyter notebook –ip 0.0.0.0

You can then enter the url and token provided in the browser to access your notebook. When you are done and need to stop and remove the container:

> docker stop repo2docker docker rm repo2docker

1.6 Design

The repo2docker buildpacks are inspired by Heroku's Build Packs. The philosophy for the repo2docker buildpacks includes:

- using common configuration files for familiar installation and packaging tools
- allowing configuration files to be combined to compose more complex setups
- specifying default locations for configuration files (the repository's root directory or .binder directory)

When designing `repo2docker` and adding to it in the future, the developers are influenced by two primary use cases. The use cases for `repo2docker` which drive most design decisions are:

1. Automated image building used by projects like BinderHub
2. Manual image building and running the image from the command line client, `jupyter-repo2docker`, by users interactively on their workstations

1.6.1 Deterministic output

The core of `repo2docker` can be considered a deterministic algorithm. When given an input directory which has a particular repository checked out, it deterministically produces a Dockerfile based on the contents of the directory. So if we run `repo2docker` on the same directory multiple times, we get the exact same Dockerfile output.

This provides a few advantages:

1. Reuse of cached built artifacts based on a repository's identity increases efficiency and reliability. For example, if we had already run `repo2docker` on a git repository at a particular commit hash, we know we can just re-use the old output, since we know it is going to be the same. This provides massive performance & architectural advantages when building additional tools (like BinderHub) on top of `repo2docker`.
2. We produce Dockerfiles that have as much in common as possible across multiple repositories, enabling better use of the Docker build cache. This also provides massive performance advantages.

1.6.2 Reproducibility and version stability

Many ingredients go into making an image from a repository:

1. version of the base docker image
2. version of `repo2docker` itself
3. versions of the libraries installed by the repository

`repo2docker` controls the first two, the user controls the third one. The current policy for the version of the base image is that we will keep pace with Ubuntu releases until we reach the next release with Long Term Support (LTS). We currently use Artful Aardvark (17.10) and the next LTS version will be Bionic Beaver (18.04).

The version of `repo2docker` used to build an image can influence which packages are installed by default and which features are supported during the build process. We will periodically update those packages to keep step with releases of Jupyter Notebook, JupyterLab, etc. For packages that are installed by default but where you want to control the version we recommend you specify them explicitly in your dependencies.

1.6.3 Unix principles "do one thing well"

`repo2docker` should do one thing, and do it well. This one thing is:

> Given a repository, deterministically build a docker image from it.

There's also some convenience code (to run the built image) for users, but that's separated out cleanly. This allows easy use by other projects (like BinderHub).

There is additional (and very useful) design advice on this in the Art of Unix Programming which is a highly recommended quick read.

1.6.4 Composability

Although other projects, like s2i, exist to convert source to Docker images, `repo2docker` provides the additional functionality to support *composable* environments. We want to easily have an image with Python3+Julia+R-3.2 environments, rather than just one single language environment. While generally one language environment per container works well, in many scientific / datascience computing environments you need multiple languages working together to get anything done. So all buildpacks are composable, and need to be able to work well with other languages.

1.6.5 Pareto principle (The 80-20 Rule)

Roughly speaking, we want to support 80% of use cases, and provide an escape hatch (raw Dockerfiles) for the other 20%. We explicitly want to provide support only for the most common use cases - covering every possible use case never ends well.

An easy process for getting support for more languages here is to demonstrate their value with Dockerfiles that other people can use, and then show that this pattern is popular enough to be included inside `repo2docker`. Remember that 'yes' is forever (very hard to remove features!), but 'no' is only temporary!

1.7 Architecture

This is a living document talking about the architecture of repo2docker from various perspectives.

1.7.1 Buildpack

The **buildpack** concept comes from Heroku and Ruby on Rails' Convention over Configuration doctrine.

Instead of the user specifying a complete specification of exactly how they want their environment to be, they can focus only on how their environment differs from a conventional environment. This means instead of deciding 'should I get Python from Apt or pyenv or ?', user can just specify 'I want python-3.6'. Usually, specifying a **runtime** and list of **libraries** with explicit **versions** is all that is needed.

In repo2docker, a Buildpack does the following things:

1. **Detect** if it can handle a given repository
2. **Build** a base language environment in the docker image
3. **Copy** the contents of the repository into the docker image
4. **Assemble** a specific environment in the docker image based on repository contents
5. **Push** the built docker image to a specific docker registry (optional)
6. **Run** the build docker image as a docker container (optional)

Detect

When given a repository, repo2docker first has to determine which buildpack to use. It takes the following steps to determine this:

1. Look at the ordered list of `BuildPack` objects listed in `Repo2Docker.buildpacks` traitlet. This is populated with a default set of buildpacks in most-specific-to-least-specific order. Other applications using this can add / change this using traditional traitlet configuration mechanisms.
2. Calls the `detect` method of each `BuildPack` object. This method assumes that the repository is present in the current working directory, and should return `True` if the repository is something that it should be used for. For example, a `BuildPack` that uses `conda` to install libraries can check for presence of an `environment.yml` file and say 'yes, I can handle this repository' by returning `True`. Usually buildpacks look for presence of specific files (`requirements.txt`, `environment.yml`, `install.R`, etc) to determine if they can handle a repository or not.
3. If no `BuildPack` returns true, then repo2docker will use the default `BuildPack` (defined in `Repo2Docker.default_buildpack` traitlet).

1.7.2 Build base environment

Once a buildpack is chosen, it builds a **base environment** that is mostly the same for various repositories built with the same buildpack.

For example, in `CondaBuildPack`, the base environment consists of installing miniconda and basic notebook packages (from `repo2docker/buildpacks/conda/environment.yml`). This is going to be the same for most repositories built with `CondaBuildPack`, so we want to use docker layer caching as much as possible for performance reasons. Next time a repository is built with `CondaBuildPack`, we can skip straight to the **copy** step (since the base environment docker image *layers* have already been built and cached).

The `get_build_scripts` and `get_build_script_files` methods are primarily used for this. `get_build_scripts` can return arbitrary bash script lines that can be run as different users, and `get_build_script_files` is used to copy specific scripts (such as a conda installer) into the image to be run as pat of `get_build_scripts`. Code in either has following constraints:

1. You can *not* use the contents of repository in them, since this happens before the repository is copied into the image. For example, `pip install -r requirements.txt` will not work, since there's no `requirements.txt` inside the image at this point. This is an explicit design decision, to enable better layer caching.
2. You *may*, however, read the contents of the repository and modify the scripts emitted based on that! For example, in `CondaBuildPack`, if there's Python 2 specified in `environment.yml`, a different kind of environment is set up. The reading of the `environment.yml` is performed in the BuildPack itself, and not in the scripts returned by `get_build_scripts`. This is fine. BuildPack authors should still try to minimize the variants created in this fashion, to optimize the build cache.

1.7.3 Copy repository contents

The contents of the repository are copied unconditionally into the Docker image, and made available for all further commands. This is common to most `BuildPack`s, and the code is in the `build` method of the `BuildPack` base class.

1.7.4 Assemble repository environment

The **assemble** stage builds the specific environment that is requested by the repository. This usually means installing required libraries specified in a format native to the language (`requirements.txt`, `environment.yml`, `REQUIRE`, `install.R`, etc).

Most of this work is done in `get_assemble_scripts` method. It can return arbitrary bash script lines that can be run as different users, and has access to the repository contents (unlike `get_build_scripts`). The docker image layers produced by this usually can not be cached, so less restrictions apply to this than to `get_build_scripts`.

At the end of the assemble step, the docker image is ready to be used in various ways!

1.7.5 Push

Optionally, repo2docker can **push** a built image to a docker registry. This is done as a convenience only (since you can do the same with a `docker push` after using repo2docker only to build), and implemented in `Repo2Docker.push` method. It is only activated if using the `--push` commandline flag.

1.7.6 Run

Optionally, repo2docker can **run** the built image and allow the user to access the Jupyter Notebook running inside by default. This is also done as a convenience only (since you can do the same with `docker run` after using repo2docker only to build), and implemented in `Repo2Docker.run`. It is activated by default unless the `--no-run` commandline flag is passed.

1.8 Adding a new buildpack to repo2docker

A new buildpack is needed when a new language or a new package manager should be supported. Existing buildpacks are a good model for how new buildpacks should be structured.

1.8.1 Criteria to balance and consider

Criteria to balance are:

1. Maintenance burden on repo2docker.
2. How easy it is to use a given setup without support from repo2docker natively. There are two escape hatches here - `postBuild` and `Dockerfile`.
3. How widely used is this language / package manager? This is the primary tradeoff with point (1). We (the Binder / Jupyter team) want to make new formats as little as possible, so ideally we can just say "X repositories on binder already use this using one of the escape hatches in (2), so let us make it easy and add native support".

1.8.2 Adding libraries or UI to existing buildpacks

Note that this doesn't apply to adding additional libraries / UI to existing buildpacks. For example, if we had an R buildpack and it supported IRKernel, it is much easier to just support RStudio / Shiny with it, since those are library additions instead of entirely new buildpacks.