repo2docker Documentation

Release 0.1

Project Jupyter

Feb 21, 2019

Getting started with repo2docker

`jupyter-repo2docker` is a tool to **build, run, and push Docker images from source code repositories** that run via a Jupyter server.

`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, or as a means of building images that are pushed to a Docker registry.

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

CHAPTER 1

Installing `repo2docker`

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

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.

1.2 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*.

1.3 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*.

1.4 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.

CHAPTER 2

Using `repo2docker`

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

`repo2docker` is called with a URL/path to a git repository. It then performs these steps:

1. Inspects the repository for *configuration files*. These will be used to build the environment needed to run the repository.
2. Builds a Docker image with an environment specified in these *configuration files*.
3. Runs a Jupyter server within the image that lets you explore the repository interactively (optional)
4. Pushes the images to a Docker registry so that it may be accessed remotely (optional)

2.1 Calling repo2docker

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 which you'd like to build an image.

For example, the following command will build an image of Peter Norvig's Pytudes repository:

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

Building the image may take a few minutes.

Pytudes uses a requirements.txt file to specify its Python environment. Because of this, `repo2docker` will use `pip` to install dependencies listed in this `requirement.txt` file, and these will be present in the generated Docker 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.

2.2 Building a specific branch / commit / tag

To build a particular branch and commit, use the argument `--ref` and specify the `branch-name` or `commit-hash`. For example:

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

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.

2.3 Where to put configuration files

`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.

Note: `repo2docker` builds an environment with Python 3.6 by default. If you'd like a different version, you can specify this in your *configuration files*.

2.4 Debugging repo2docker with `--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
```

CHAPTER 3

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!

3.1 How should I specify another version of Python?

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

3.2 What versions of Python (or R or Julia...) are supported?

3.2.1 Python

Repo2docker officially supports the following versions of Python (specified in environment.yml or runtime.txt):

- 3.7 (added in 0.7)
- 3.6 (default)
- 3.5

Additional versions may work, as long as the base environment can be installed for your version of Python. The most likely source of incompatibility is if one of the packages in the base environment is not packaged for your Python, either because the version of the package is too new and your chosen Python is too old, or vice versa.

Additionally, if Python 2.7 is specified, a separate environment for the kernel will be installed with Python 2. The notebook server will run in the default Python 3.6 environment.

3.2.2 Julia

The following versions of Julia are supported (specified in REQUIRE):

- 1.0 (added in 0.7)

- 0.7 (added in 0.7)
- 0.6 (default)

3.2.3 R

Only R 3.4.4 is currently supported, which is installed via `apt` from the ubuntu bionic repository.

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

Yes! With a *postBuild - Run code after installing the environment* 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.)

3.4 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 ...`

3.5 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!

3.6 Can I use repo2docker to edit a local host repository within a Docker environment?

Yes: use the `--editable` or `-E` flag (don't confuse this with the `-e` flag for environment variables), and run repo2docker on a local repository:

```
repo2docker -E my-repository/.
```

This builds a Docker container from the files in that repository (using, for example, a `requirements.txt` or `install.R` file), then runs that container, while connecting the working directory inside the container to the local repository outside the container. For example, in case there is a notebook file (`.ipynb`), this will open in a local webbrowser, and one can edit it and save it. The resulting notebook is updated in both the Docker container and the local repository. Once the container is exited, the changed file will still be in the local repository.

This allows for easy testing of the container while debugging some items, as well as using a fully customizable container to edit notebooks (among others).

Note: Editable mode is a convenience option that will bind the repository to the container working directory (usually `$HOME`). If you need to mount to a different location in the container, use the `--volumes` option instead. Similarly, for a fully customized user Dockerfile, this option is not guaranteed to work.

CHAPTER 4

Configure the user interface

You can build several user interfaces into the resulting Docker image. This is controlled with various *configuration files*.

4.1 JupyterLab

You do not need any extra configuration in order to allow the use of the JupyterLab interface. You can launch JupyterLab from within a user session by opening the Jupyter Notebook and appending `/lab` to the end of the URL like so:

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

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

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

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

4.2 RStudio

The RStudio user interface is automatically enabled if a configuration file for R is detected (i.e. an R version specified in `runtime.txt`). If this is detected, RStudio will be accessible by appending `/rstudio` to the URL, like so:

```
http(s)://<server:port>/rstudio
```

4.3 Stencila

The Stencila user interface is automatically enabled if a Stencila document (i.e. a file *manifest.xml*) is detected. Stencila will be accessible by appending `/stencila` to the URL, like so:

```
http(s)://<server:port>/stencila
```

The editor will open the Stencila document corresponding to the last *manifest.xml* found in the file tree. If you want to open a different document, you can configure the path in the URL parameter *archive*:

```
http(s)://<server:port>/stencila/?archive=other-dir
```

CHAPTER 5

Choose languages for your environment

You can define many different languages in your configuration files. This page describes how to use some of the more common ones.

5.1 Python

Your environment will have Python (and specified dependencies) installed when you use one of the following configuration files:

- `requirements.txt`
- `environment.yml`

Note that by default, the environment will have **Python 3** installed.

5.1.1 Specifying a version of Python

To specify a specific version of Python, you have two options:

- Use `runtime.txt`. Include a line that specifies the Python version in this file. This line takes the following form:

  ```
  python=X.X
  ```

  For example,:

  ```
  python=2.7
  ```

- Use `environment.yml`. The Anaconda distribution also lets you define the Python environment within `environment.yml`. To do so, add `python=X.X` to your dependencies section, like so:

```
name: python 2.7
dependencies:
  - python=2.7
  - numpy
```

5.2 The R Language

To ensure that R is installed, you must specify a version of R in a `runtime.txt` file. This takes the following form:

```
r-YYYY-MM-DD
```

The date corresponds to the state of the MRAN repository at this day. Make sure that you choose a day with the desired version of your packages. For example, to use the MRAN repository on January 1st, 2018, add this line to `runtime.txt`:

```
r-2018-01-01
```

Note that to install specific packages with the R environment, you should use the `install.R` configuration file.

5.3 Julia

To build an environment with Julia, include a configuration file called `REQUIRE`. Each line of this file should include a package that you wish to have installed with Julia. For example, the following contents of `REQURE` would install the `PyPlot` package with your Julia environment.:

```
PyPlot
```

5.4 Languages not covered here

If a language is not "officially" supported by a build pack, it can often be installed with a `postBuild` script. This will run arbitrary `bash` commands, and can be used to download / install a language.

5.5 Using multiple languages at once

It may also be possible to combine multiple languages in a single environment. The details on how to accomplish this with all possible combinations are outside the scope of this guide. However we recommend that you take a look at the Multi-Language Demo repository for some inspiration.

CHAPTER 6

Build 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']
```

CHAPTER 7

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.

7.1 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.

7.1.1 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
```

7.1.2 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

7.1.3 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.

7.1.4 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`.

7.1.5 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.

7.1.6 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

CHAPTER 8

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.

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):

- `environment.yml` - *Install a Python environment*
- `requirements.txt` - *Install a Python environment*
- `setup.py` - *Install Python packages*
- `REQUIRE` - *Install a Julia environment*
- `install.R` - *Install an R/RStudio environment*
- `apt.txt` - *Install packages with apt-get*
- `DESCRIPTION` - *Install an R package*
- `manifest.xml` - *Install Stencila*
- `postBuild` - *Run code after installing the environment*
- `start` - *Run code before the user sessions starts*
- `runtime.txt` - *Specifying runtimes*
- `default.nix` - *the nix package manager*
- `Dockerfile` - *Advanced environments*

8.1 `environment.yml` - Install a Python environment

`environment.yml` is the standard configuration file used by Anaconda, conda, and miniconda that lets you install packages in the data analytics stack (it primarily installs Python packages, though can be used to install a range of non-Python packages as well).

Note: You can install files from pip in your `environment.yml` as well. For example, see the binder-examples environment.yml file.

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.

Warning: If you include a Python version in a `runtime.txt` file in addition to your `environment.yml`, your `runtime.txt` will be ignored.

8.2 `requirements.txt` - Install a Python environment

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.

8.3 `setup.py` - Install Python packages

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 .`.

8.4 `REQUIRE` - Install a Julia environment

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

8.5 `install.R` - Install an R/RStudio environment

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.

8.6 `apt.txt` - Install packages with apt-get

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.

8.7 DESCRIPTION - Install an R package

To install your repository like an R package, you may include a DESCRIPTION file. repo2docker installs the package and dependencies from the DESCRIPTION by running devtools:install_git(".").

You also need to have a runtime.txt file that is formatted as r-<YYYY>-<MM>-<DD>, where YYYY-MM-DD is a snapshot of MRAN that will be used for your R installation.

8.8 manifest.xml - Install Stencila

Stencila is an open source office suite for reproducible research. It is powered by the open file format Dar.

If your repository contains a Stencila document, repo2docker detects it based on the file manifest.xml. The required *execution contexts <https://stenci.la/learn/intro.html>* are extracted from a Dar article (i.e. files named *.jats.xml).

You may also have a runtime.txt and/or an install.R to manually configure your R installation.

To see example repositories, visit our Stencila with R and Stencila with Python examples.

8.9 postBuild - Run code after installing the environment

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.

8.10 start - Run code before the user sessions starts

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 - Run code after installing the environment*.

8.11 runtime.txt - Specifying runtimes

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.

8.12 `default.nix` - the nix package manager

Specify packages to be installed by the nix package manager. When you use this config file all other configuration files (like `requirements.txt`) that specify packages are ignored. When using `nix` you have to specify all packages and dependencies explicitly, including the Jupyter notebook package that repo2docker expects to be installed. If you do not install Jupyter explicitly repo2docker will no be able to start your container.

nix-shell is used to evaluate a `nix` expression written in a `default.nix` file. Make sure to pin your nixpkgs to produce a reproducible environment.

To see an example repository visit nix binder example.

8.13 `Dockerfile` - Advanced environments

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.

Note: If a Dockerfile is present, all other configuration files will be ignored.

See the Advanced Binder Documentation for best-practices with Dockerfiles.

CHAPTER 9

Contributing to repo2docker development

9.1 Process for making a code contribution

This outlines the process for getting changes to the code of repo2docker merged.

- If your change is relatively significant, **open an issue to discuss** before spending a lot of time writing code. Getting consensus with the community is a great way to save time later.
- Make edits in your fork of the repo2docker repository
- Submit a pull request (this is how all changes are made)
- Edit the changelog by appending your feature / bug fix to the development version.
- Wait for a community member to merge your changes
- (optional) Deploy a new version of repo2docker to mybinder.org by following these steps

9.2 Guidelines to getting a Pull Request merged

These are not hard rules to be enforced by :police_car: but instead guidelines to help you make the most effictive / efficient contribution.

- prefix the title of your pull request with `[MRG]` if the contribution is complete and should be subjected to a detailed review;
- create a PR as early as possible, marking it with `[WIP]` while you work on it (good to avoid duplicated work, get broad review of functionality or API, or seek collaborators);
- a PR solves one problem (do not mix problems together in one PR) with the minimal set of changes;
- describe why you are proposing the changes you are proposing;
- try to not rush changes (the definition of rush depends on how big your changes are);
- someone else has to merge your PR;

- new code needs to come with a test;
- apply PEP8 as much as possible, but not too much;
- no merging if travis is red;
- do use merge commits instead of merge-by-squashing/-rebasing. This makes it easier to find all changes since the last deployment `git log --merges --pretty=format:"%h %<(10,trunc)%an %<(15)%ar %s" <deployed-revision>..`

9.3 Setting up for Local Development

To develop & test repo2docker locally, you need:

1. Familiarity with using a command line terminal
2. A computer running macOS / Linux
3. Some knowledge of git
4. At least python 3.6
5. Your favorite text editor
6. A recent version of Docker Community Edition

9.3.1 Clone the repository

First, you need to get a copy of the repo2docker git repository on your local disk. Fork the repository on GitHub, then clone it to your computer:

```
git clone https://github.com/<your-username>/repo2docker
```

This will clone repo2docker into a directory called `repo2docker`. You can make that your current directory with `cd repo2docker`.

9.3.2 Set up a local virtual environment

After cloning the repository (or your fork of the repository), you should set up an isolated environment to install libraries required for running / developing repo2docker.

There are many ways to do this but here we present you with two approaches: `virtual environment` or `pipenv`.

- Using `virtual environment`

```
python3 -m venv .
source bin/activate
pip3 install -e .
pip3 install -r dev-requirements.txt
pip3 install -r docs/doc-requirements.txt
```

This should install all the libraries required for testing & running repo2docker!

- Using `pipenv`

Note that you will need to install pipenv first using `pip3 install pipenv`. Then from the root directory of this project you can use the following commands:

```
pipenv install --dev
```

This should install both the dev and docs requirements at once!

9.3.3 Set up

9.3.4 Verify that docker is installed and running

If you do not already have Docker, you should be able to download and install it for your operating system using the links from the official website. After you have installed it, you can verify that it is working by running the following commands:

```
docker version
```

It should output something like:

```
Client:
 Version:      17.09.0-ce
 API version:  1.32
 Go version:   go1.8.3
 Git commit:   afdb6d4
 Built:        Tue Sep 26 22:42:45 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.09.0-ce
 API version:  1.32 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   afdb6d4
 Built:        Tue Sep 26 22:41:24 2017
 OS/Arch:      linux/amd64
 Experimental: false
```

Then you are good to go!

CHAPTER 10

The repo2docker roadmap

This roadmap collects "next steps" for repo2docker. It is about creating a shared understanding of the project's vision and direction amongst the community of users, contributors, and maintainers. The goal is to communicate priorities and upcoming release plans. It is not a aimed at limiting contributions to what is listed here.

10.1 Using the roadmap

10.1.1 Sharing Feedback on the Roadmap

All of the community is encouraged to provide feedback as well as share new ideas with the community. Please do so by submitting an issue. If you want to have an informal conversation first use one of the other communication channels. After submitting the issue, others from the community will probably respond with questions or comments they have to clarify the issue. The maintainers will help identify what a good next step is for the issue.

10.1.2 What do we mean by "next step"?

When submitting an issue, think about what "next step" category best describes your issue:

- **now**, concrete/actionable step that is ready for someone to start work on. These might be items that have a link to an issue or more abstract like "decrease typos and dead links in the documentation"
- **soon**, less concrete/actionable step that is going to happen soon, discussions around the topic are coming close to an end at which point it can move into the "now" category
- **later**, abstract ideas or tasks, need a lot of discussion or experimentation to shape the idea so that it can be executed. Can also contain concrete/actionable steps that have been postponed on purpose (these are steps that could be in "now" but the decision was taken to work on them later)

10.1.3 Reviewing and Updating the Roadmap

The roadmap will get updated as time passes (next review by 31st January 2019) based on discussions and ideas captured as issues. This means this list should not be exhaustive, it should only represent the "top of the stack" of ideas. It should not function as a wish list, collection of feature requests or todo list. For those please create a new issue.

The roadmap should give the reader an idea of what is happening next, what needs input and discussion before it can happen and what has been postponed.

10.2 The roadmap proper

10.2.1 Project vision

Repo2docker is a dependable tool used by humans that reduces the complexity of creating the environment in which a piece of software can be executed.

10.2.2 Now

The "Now" items are being actively worked on by the project:

- reduce documentation typos and syntax errors
- increase test coverage to 80% (see https://codecov.io/gh/jupyter/repo2docker/tree/master/repo2docker for low coverage files)
- mounting repository contents in locations that is not `/home/jovyan`
- investigate options for pinning repo2docker versions (#490)

10.2.3 Soon

The "Soon" items are being discussed/a plan of action is being made. Once an item reaches the point of an actionable plan and person who wants to work on it, the item will be moved to the "Now" section. Typically, these will be moved at a future review of the roadmap.

- create the contributor highway, define the route from newcomer to project lead
- add Julia Manifest support (https://docs.julialang.org/en/v1/stdlib/Pkg/index.html, #486)
- support different base images/build pack stacks (#487)

10.2.4 Later

The "Later" items are things that are at the back of the project's mind. At this time there is no active plan for an item. The project would like to find the resources and time to discuss and then execute these ideas.

- support execution on a remote host (with more resources than available locally) via the command-line
- add support for using ZIP files as the repo (`repo2docker https://example.com/an-archive.zip`) this will give us access to several archives (like Zenodo) that expose things as ZIP files.
- add support for Zenodo (`repo2docker 10.5281/zenodo.1476680`) so Zenodo software archives can be used as the source in addition to a git repository

CHAPTER 11

Architecture of repo2docker

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

11.1 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)

11.1.1 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`, `manifest.xml` etc) to determine if they can handle a repository or not. Buildpacks may also look into specific files to determine specifics of the required environment, such as the Stencila integration which extracts the required language-specific executions contexts from an XML file (see base `BuildPack`). More than one buildpack may use such information, as properties can be inherited (e.g. the R buildpack uses the list of required Stencila contexts to see if R must be installed).
3. If no `BuildPack` returns true, then repo2docker will use the default `BuildPack` (defined in `Repo2Docker.default_buildpack` traitlet).

11.2 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.

11.3 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 `BuildPacks`, and the code is in the `build` method of the `BuildPack` base class.

11.4 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!

11.5 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.

11.6 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.

CHAPTER 12

Design of repo2docker

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

12.1 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 reuse 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.

12.2 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.

12.3 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.

12.4 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.

12.5 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!

CHAPTER 13

Common tasks

These are some common tasks to be done as a part of developing and maintaining repo2docker. If you'd like more guidance for how to do these things, reach out in the JupyterHub Gitter channel.

13.1 Running tests

We have a lot of tests for various cases supported by repo2docker in the `tests/` subdirectory. If you fix a bug or add new functionality consider adding a new test to prevent the bug from coming back. These use py.test.

You can run all the tests with:

```
py.test -s tests/*
```

If you want to run a specific test, you can do so with:

```
py.test -s tests/<path-to-test>
```

13.2 Update and Freeze BuildPack Dependencies

This section covers the process by which repo2docker defines and updates the dependencies that are installed by default for several buildpacks.

For both the `conda` and `virtualenv` (`pip`) base environments in the **Conda BuildPack** and **Python BuildPack**, we install specific pinned versions of all dependencies. We explicitly list the dependencies we want, then *freeze* them at commit time to explicitly list all the transitive dependencies at current versions. This way, we know that all dependencies will have the exact same version installed at all times.

To update one of the dependencies shared across all `repo2docker` builds, you must follow these steps (with more detailed information in the sections below):

1. Make sure you have Docker running on your computer

2. Bump the version numbers of the dependencies you want to update in the `conda` environment (link)
3. Make a pull request with your changes (link)

See the subsections below for more detailed instructions.

13.2.1 Conda dependencies

1. There are two files related to conda dependencies. Edit as needed.
   - `repo2docker/buildpacks/conda/environment.yml`

     Contains list of packages to install in Python3 conda environments, which are the default. **This is where all Notebook versions & notebook extensions (such as JupyterLab / nteract) go**.
   - `repo2docker/buildpacks/conda/environment.py-2.7.yml`

     Contains list of packages to install in Python2 conda environments, which can be specifically requested by users. **This only needs `IPyKernel` and kernel related libraries**. Notebook / Notebook Extension need not be installed here.
2. Once you edit either of these files to add a new package / bump version on an existing package, you should then run:

   ```
   cd ./repo2docker/buildpacks/conda/
   python freeze.py
   ```

   This script will resolve dependencies and write them to the respective `.frozen.yml` files. You will need `docker` installed to run this script.
3. After the freeze script finishes, a number of files will have been created. Commit the following subset of files to git:

   ```
   repo2docker/buildpacks/conda/environment.yml
   repo2docker/buildpacks/conda/environment.frozen.yml
   repo2docker/buildpacks/conda/environment.py-2.7.yml
   repo2docker/buildpacks/conda/environment.py-2.7.frozen.yml
   repo2docker/buildpacks/conda/environment.py-3.5.frozen.yml
   repo2docker/buildpacks/conda/environment.py-3.6.frozen.yml
   ```
4. Make a pull request; see details below.
5. Once the pull request is approved (but not yet merged), Update the change log (details below) and commit the change log, then update the pull request.

13.2.2 Make a Pull Request

Once you've made the commit, please make a Pull Request to the `jupyterhub/repo2docker` repository, with a description of what versions were bumped / what new packages were added and why. If you fix a bug or add new functionality consider adding a new test to prevent the bug from coming back/the feature breaking in the future.

13.3 Creating a Release

We try to make a release of repo2docker every few months if possible.

We follow semantic versioning.

Check that the Change log is ready and then tag a new release on GitHub.

When the travis run completes check that the new release is available on PyPI.

13.3.1 Update the change log

To add your change to the change log, find the relevant Feature/Bug fix/API change section for the next release near the top of the file; then add one or two sentences as a new bullet point about your changes. Include the pull request or issue number between square brackets at the end.

Some details:

- versioning follows the x.y.z, major.minor.bugfix numbering
- bug fixes go into the next bugfix release. If there isn't any, you can create a new section (see point below). Don't worry if you're not sure about that, and think it should go into a next major or minor release: an admin will let you know, or move the change later to the appropriate section
- API changes should preferably go into the next major release, unless they are backward compatible (for example, a deprecated function keyword): then they can go into the next minor release. For release with major release 0, non-backward compatible breaking changes are also fine for the next minor release.
- new features should go into the next minor release.
- if there is no section for the appropriate release, you can add one:

  follow the versioning scheme, by simply increasing the relevant number for one of the major /minor/bugfix numbers, appropriate for your change (see the above bullet points); add the release section. Then add three subsections: new features, api changes, and bug fixes. Leave out the sections that are not appropriate for the newlye added release section.

Release candidate versions in the change log are only temporary, and should be superseded by either a next release candidate, or the final release for that version (bugfix version 0).

13.3.2 Keeping the Pipfile and requirements files up to date

We now have both a `dev-requirements.txt` and a `Pifile` for repo2docker, as such it is important to keep these in sync/up-to-date.

Both files use `pip identifiers` so if you are updating for example the Sphinx version in the `doc-requirements.txt` (currently `Sphinx = ">=1.4,!=1.5.4"`) you can use the same syntax to update the Pipfile and viceversa.

At the moment this has to be done manually so please make sure to update both files accordingly.

CHAPTER 14

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.

14.1 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".

14.2 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.