Pi

Jun 15, 2023

Contents

Command-line tool for managing containerized environments. Helps you build nice and unique CLI for your project, manage containers, images and services.

Project CLI means that you can create nested structure of commands, which will use containers (Docker) to run and services (e.g. PostgreSQL) to perform some complex tasks.

Managing images means that you can define hierarchical images structure and Pi will build them on demand, for example when you call a command, which require some images to run, which are not built yet. You don't have to assign versions (tags) for these images yourself, Pi will use hashing algorithm to automatically create them.

Managing services means that you can specify, that some services should be started before running a command. Or you can manually start and stop services.

CHAPTER 1

Installation

Pi requires Python 3.5 or higher. You can install Pi directly into your system packages, it has zero dependencies, so it can be uninstalled without leaving any traces in your system:

```
$ pip3 install pi-env
```

CHAPTER 2

Example

Example `pi.yaml` file:

```
- !Meta
  namespace: foo
  description: |
    Project command-line interface

- !Service
  name: pg
  network-name: postgres
  image: !DockerImage postgres:10-alpine

- !Image
  name: test
  from: !DockerImage python:3.6-alpine
  repository: localhost/foo/test
  tasks:
  - run: pip3 install --no-deps --no-cache-dir -r {{reqs}}
    reqs: !File requirements.txt

- !Command
  name: test
  image: test
  requires: [pg]
  description: Run py.test
  params:
  - !Argument {name: tests, default: ''}
  run: py.test {{tests}}
```

If you call `pi test`, Pi will build `test` image if needed and will make sure that `pg` service is running, which is required to run tests.

This `pg` service will be available for the tests at `postgres:5432` address. Both command `pi test` and `pg` service will be running inside containers, which will be in the same unique network, automatically created for specified `foo` namespace.

Here is how your project will be looking in the shell:

```
$ pi
Usage: pi [OPTIONS] COMMAND [ARGS]...

  Project command-line interface

Options:
  --debug  Run in debug mode
  --help   Show this message and exit.

Core commands:
  + image    Images creation and delivery
  + service  Services status and management

Custom commands:
  test  Run py.test
```

You can see list of all defined images:

```
$ pi image -l
  Image name    Docker image                       Size        Versions
--------------  ---------------------------------  --------  ----------
✓ test          localhost/foo/test:4efe5a0454a9  88.58 MB           1
```

You also can see status of all defined services:

```
$ pi service -s
Service name    Status    Docker image
--------------  --------  ------------------
pg              running   postgres:10-alpine
```

And of cause you can run your commands:

```
$ pi test
.............................
31 passed in 0.35 seconds
```

CHAPTER 3

Contributing

Run `python -m pi test` and `python -m pi lint` in order to test and lint your changes before submitting your pull requests.

CHAPTER 4

The User Guide

4.1 Types

Pi uses YAML format and it's tagged values feature to assemble types defined here into complex structure, which will describe your project's CLI and environment.

`pi.yaml` - is a list of these top-level types: *Meta*, *Service*, *Image* and *Command*. Their order is not significant.

class Meta
Project-specific settings

```
- !Meta:
  namespace: example
  description: |
    This is an example project
```

Parameters

- **namespace** – Name, used to namespace such things like network, to make them unique and isolated for every project
- **description** – Description for a project, which will be seen when users will run `pi --help` command

class DockerImage
Reference to a name of the Docker image

Takes single argument - image name. Image name should include repository name and tag:

```
!DockerImage "python:3.6-alpine"
```

class Image
Defines how to build and distribute an image

```
- !Image
  name: env
  repository: my.registry/project/name
  from: base
  description: "Project environment"
  tasks:
  - run: cp {{config}} /etc
    config: !File "config.py"
```

Parameters

- **name** – short name of the image, used to reference it within this config/project
- **repository** – full name of the image. This name is used to distribute image using registries
- **from** – base image, to build this one from. It is a name of the other image defined in this config, or a regular external Docker image
- **description** – description of this image
- **tasks** – list of tasks, used to build this image

Each task represents a shell command to run. This command can be a simple string:

```
tasks:
- run: mkdir /etc/app
```

Or a template with parameters. Jinja2 is used as a template language:

```
tasks:
- run: pip install {{packages|join(" ")}}
  packages:
  - flask
  - sqlalchemy
```

You can also use some special handy directives:

```
tasks:
- run: sh -c {{install_sh}}
  install_sh: !Download "https://some.host/install.sh"
```

Pi will download this file for you and it will be available inside container during build process. All you need it to describe what you want to do with already downloaded file. So you don't have to install curl with ca-certificates into container and remove it in the end.

class Download

Directive to transfer downloaded on the host machine file into container

Takes single argument - url:

```
tasks:
- run: sh -c {{install_sh}}
  install_sh: !Download "https://some.host/install.sh"
```

class File

Directive to transfer file from the host machine into container

Takes single argument - local file path:

```
tasks:
- run: cp {{config}} /etc/config.yaml
  config: !File "config.yaml"
```

class Bundle

Directive to transfer directory from the host machine into container

Takes single argument - local directory path:

```
tasks:
- run: cd {{src}} && python setup.py install
  src: !Bundle "src"
```

class Service

Defines a service

```
- !Service
  name: pg
  network-name: postgres
  image: !DockerImage postgres:10-alpine
```

Parameters

- **name** – name of this service
- **image** – image, used to run this service
- **volumes** – list of volumes to mount, defined using *LocalPath* or *NamedVolume* types
- **ports** – list of exposed ports, defined using *Expose* type
- **environ** – map of environment variables
- **requires** – list of service names; Pi will ensure that these services are running before starting this service
- **exec** – service's entry point
- **args** – args passed to the service's entry point
- **network-name** – host name of the container, by default `network-name` will be equal to the `name` of the service
- **description** – description, used to help users when they run `pi service --help` command, which will list all defined services and their descriptions

class Command

Defines a command with parameters, to run inside configured container and environment

```
- !Command
  name: test
  image: test
  requires: [pg]
  description: Run py.test
  params:
  - !Argument {name: tests, default: ''}
  run: py.test {{tests}}
```

Parameters

- **name** – name of this command
- **image** – image, used to run this command
- **run** – command to run inside container
- **params** – list of command-line arguments of type *Argument* and options of type *Option*
- **volumes** – list of volumes to mount, defined using *LocalPath* or *NamedVolume* types
- **ports** – list of exposed ports, defined using *Expose* type
- **environ** – map of environment variables
- **requires** – list of service names; Pi will ensure that these services are running
- **network-name** – make this container available to the other containers in current namespace under specified host name
- **description** – description, used to help users, when they run `pi [command] --help` command

class Argument

Defines command's argument

Parameters

- **name** – argument's name
- **type** – argument's type - `str` (default), `int` or `bool`
- **default** – argument's default value

class Option

Defines command's option

Parameters

- **name** – option's name
- **type** – option's type - `str` (default), `int` or `bool`
- **default** – option's default value

class LocalPath

Specifies file or directory from the local file system to mount

```
volumes:
- !LocalPath {from: "config.yaml", to: "/etc/config.yaml"}
```

Parameters

- **from** – Local path
- **to** – Path inside container
- **mode** – *RO* (default) or *RW*

class NamedVolume

Specifies existing named volume to mount

```
...
volumes:
- !NamedVolume {name: db, to: "/var/db/data", mode: !RW }
```

Parameters

- **name** – Volume's name
- **to** – Path inside container
- **mode** – *RO* (default) or *RW*

class RO
Defines read-only mode

class RW
Defines read/write mode

class Expose
Defines port mapping to expose

```
...
ports:
- !Expose {port: 5000, as: 5000, addr: 0.0.0.0}
```

Parameters

- **port** – port inside container
- **as** – port outside container
- **addr** – network interface for binding, `127.0.0.1` by default
- **proto** – protocol, `tcp` by default

4.2 Changelog

4.2.1 0.1.1

- Added "pi image info" command
- Refactored "pi image" and "pi service" UI
- Removed `dumb-init` usage in favor to Docker's own init process
- Fixed `!Bundle` task to use proper relative paths
- Added `!File` task
- Services can now require other services
- Added special naming convention for local-only images

4.2.2 0.1.0

- Initial release

Index