ixian-docker

Peter Krenesky

May 23, 2020

GETTING STARTED

CHAPTER ONE

INSTALLATION

```
pip install ixian_docker
```

CHAPTER TWO

SETUP

2.1 1. Create `ixian.py`

Ixian apps must be initialize in `ixian.py`. Here is a basic setup for a django app.

```
# ixian.py
from ixian.config import CONFIG
from ixian.module import load_module

def init():
    # Load ixian core + docker core.
    CONFIG.PROJECT_NAME = 'my_project'
    load_module('ixian.modules.core')
    load_module('ixian_docker.modules.docker')

    # Minimal setup for Django backend + Webpack compiled front end
    load_module('ixian_docker.modules.python')
    load_module('ixian_docker.modules.django')
    load_module('ixian_docker.modules.npm')
    load_module('ixian_docker.modules.webpack')
```

2.2 2. Configure Docker Registries

Configure docker registries for pulling and pushing images.

```
# Specify the registry for your images. The image's name will be generated
# from this url and path.
#
#   e.g. my.registries.domain.name.com/my_project
#
CONFIG.DOCKER.REGISTRY = 'my.registry.domain.name.com'
CONFIG.DOCKER.REGISTRY_PATH = 'my_project'
CONFIG.DOCKER.REGISTRIES = {
    'my.registry.domain.name.com': {
        'client': DockerClient,

        # addtional options may be passed in
        'options': {
            'username': "my_registry_user"
            'password': "my_registry_password"
        }
```

(continues on next page)

(continued from previous page)

```
        }
}
```

See the section on *docker registries* for more information.

2.3 3. Module config

Modules each have their own requirements for configuration. Built-in modules have sane defaults where possible. See specific module docs for details.

CHAPTER

THREE

USAGE

3.1 Basics

Ixian apps are executed using the `ix` runner. This is the entry point for `ixian` apps. The general help page lists the available tasks.

```
$ ix build_image

usage: ixian [--help] [--log LOG] [--force] [--force-all] [--clean]
     [--clean-all]
     ...

Run an ixian task.

positional arguments:
  remainder    arguments for task.

optional arguments:
  --help       show this help message and exit
  --log LOG    Log level (DEBUG|INFO|WARN|ERROR|NONE)
  --force      force task execution
  --force-all  force execution including task dependencies
  --clean      clean before running task
  --clean-all  clean all dependencies before running task

Type 'ix help <subcommand>' for help on a specific subcommand.

Available subcommands:

[ Build ]
  build_image                Build app image
```

Internal flags should be placed before the `task`.

```
ix --force build_image
```

Any args after the task name are passed to task's execute method.

For example, many tasks are wrappers around other command line tools. Pass `--help` after the command to get that tool's internal help.

```
ix pytest --help
```

3.2 Tasks

Once configured you will have access to a number of tasks for building and interacting with the app in your image. These will vary depending on what modules you've enabled. Here are a couple of examples.

- Build a docker image.

  ```
  ix build_image
  ```

- Run the django test server.

  ```
  ix runserver
  ```

- Run automated tests.

  ```
  ix test
  ```

3.3 Task checks

Many tasks have state checks that determine if they are already complete. This can be viewed in ixian task help. Completed dependencies are indicated by a check.

```
STATUS
 build_image
    ✓ build_base_image
     build_npm_image
     build_webpack_image
     build_python_image
```

If the task or any of it's dependency are incomplete then the task and it's incomplete dependencies will be run.

```
STATUS
 build_image
    ✓ build_base_image
    ✓ build_npm_image
     build_webpack_image
     build_python_image
```

When all are complete then the task can be skipped. If checkers detect changes, such as modified config files, the checkers will indicate a build.

```
STATUS
✓ build_image
    ✓ build_base_image
    ✓ build_npm_image
    ✓ build_webpack_image
    ✓ build_python_image
```

Note: If there are no checkers then a task runs every time it is called.

3.4 Forcing tasks

Task checks may be bypassed with `--force`. Pass `--force-all` to bypass checks for all dependencies.

3.5 Clean build

Some tasks have a clean function that removes build artifacts. Pass `--clean` to call the clean function prior to building. Pass `--clean-all` to trigger clean for all dependencies. If a task doesn't define a clean method then `--clean` does nothing.

Passing `--clean` also implies `--force`.

3.6 Built-in help

All tasks have built in help generated from task docstrings and metadata. The help page should explain how to configure and use the task. It also displays the state of tasks and any dependencies.

When in doubt, check `help`.

```
$ ix help build_image

NAME
    build_image -- Build app image

DESCRIPTION
Builds a docker image using CONFIG.DOCKER_FILE

STATUS
 build_image
     build_base_image
     build_npm_image
     build_webpack_image
     build_python_image
```

CHAPTER

FOUR

BUILDING IMAGES

Ixian-docker can help you build images. More specifically, it orchestrates multi-stage builds that produce heirarchies of docker images. It enable projects to stand up application stacks without worrying (as much) how to configure it all. The goal is that you spend less time on the platform tooling and more building your application.

Ixian-docker projects combine a set of modules to form an application stack. Ixian modules provide tasks that build intermediate images for platform features and provide development tools such as test runners and linters.

4.1 Setup

The set of modules that make up your stack is configured in `ixian.py`.

```
# ixian.py

def init():
    # load ixian core
    load_module('ixian.modules.core')

    # load core docker module - provides core framework for building docker apps
    load_module('ixian_docker.modules.docker')

    # load modules to build your stack to your needs
    load_module('ixian_docker.modules.python')

    # Most modules provide config to customize their usage.
    # Update config as needed after loading modules.
    CONFIG.PYTHON.REQUIREMENTS_FILES += [
        "{PYTHON.ETC}/requirements-dev.txt"
    ]
```

4.2 Choosing Stages

4.2.1 Built-ins

Ixian-docker comes with built-in modules that provide support for common build tools. They're pre-wired to work with each other making it the easiest way to stand up a project.

All modules are built on the core ixian module and the docker module. For all other modules see their pages for setup instructions.

Python

- *Python* - pip python packaging
- *Black* - black code formatter
- *Pytest* - pytest test runner
- *Django* - django web framework

NodeJS

- *Python* - NPM package manager
- *Black* - Prettier code formatter
- *Pytest* - Jest test runner
- *Django* - Webpack javascript bundler
- *Django* - ESLint javascript linter

4.2.2 Custom Build Stages

Ixian is a modular system that's easily extended to add additional build stages. Read more about that *here*

CHAPTER

FIVE

DOCKER REGISTRIES

Ixian has support baked in to push and pull images from your docker registry.

5.1 Build Cache

Caching is baked into image building tasks and utilities. The docker registry is used as a cache. Images, including intermediate images, may be pushed to the registry. Subsequent builds will pull those images when available.

This is built into existing image building tasks (e.g. `build_image`) and can be extended to other build layers.

5.2 Setup

5.2.1 Image Registry

1. Setup the image registry.

```
# Specify the registry for your images. The image's name will be generated
# from this url and path.
#
#   e.g. my.registries.domain.name.com/my_project
#
CONFIG.DOCKER.REGISTRY = 'my.registry.domain.name.com'
CONFIG.DOCKER.REGISTRY_PATH = 'my_project'

# Registries for built-in modules default to DOCKER.REGISTRY but
# you may override if needed.
CONFIG.PYTHON.REGISTRY = 'my.other.registry.domain.name.com'
```

2. Configure the registry

   All registries that are configured for images must be configured in `DOCKER.REGISTRIES` to be able to push/pull

```
CONFIG.DOCKER.REGISTRIES = {
    'my.registry.domain.name.com': {
        # registry specific config goes here.
    }
}
```

5.2.2 Docker Registry / Docker.io

The default docker registry requires authentication to push images.

```
from ixian_docker.modules.docker.utils.client.DockerClient

def init():
    # ... load modules ...

    CONFIG.DOCKER.REGISTRIES = {
        'my.registry.domain.name.com': {
            'client': DockerClient,

            # addtional options may be passed in
            'options': {
                'username': "my_registry_user"
                'password': "my_registry_password"
            }
        }
    }
```

Warning: This hasn't been tested, but it may work.

Warning: Don't store your password in `ixian.py` use vault or similar to load it at runtime.

5.2.3 Openshift

Not supported yet.

5.2.4 Amazon ECR

Elastic Container Registry (ECR) is Amazon's docker registry.

1. Setup AWS CLI

   ECR integration uses boto3 to authenticate via the AWS API. You must configure the AWS CLI in your host environment. Ixian-docker will use whichever authentication method is configured for the CLI.

2. Configure registry

```
from ixian_docker.modules.docker.utils.client.ECRDockerClient

def init():
    # ... load modules ...

    CONFIG.DOCKER.REGISTRIES = {
        'my.registry.domain.name.com': {
            'client': ECRDockerClient,

            # addtional options may be passed in
            'options': {
```

(continues on next page)

(continued from previous page)

```
                'region_name': "us-west-2"
            }
        }
    }
```

Error: `~/.docker/config.json` must be cleared manually for ECR authentication. Tokens aren't removed when they expire. Once a token expires it will cause login failures until it's manually cleared.

CHAPTER

SIX

WRITING MODULES

6.1 Basics

Ixian provides a module system. See their documentation for the basics on how to build a module: https://ixian.readthedocs.io/en/latest/modules.html

Modules for Ixian-docker may provide a few things:

- Build stages - A stage that produces an image.
- Build fragments - A fragment that contributes to another build stage.
- Runtime tools - Anything needed for runtime, including development tools.
- Config - Configuration settings to make all of the above configurable.

6.2 Designing Build Stages

Check out the documentation for *multi-stage builds* to learn more about how build stages work and how to construct a custom build stage.

6.3 Image Layout

Ixian modules use a Dockerfile layout designed to support modular *multi stage builds*. This requires a common image layout scheme so modules play nice together.

Checkout the documentation on the *common image layout* for more information.

CHAPTER

SEVEN

DESIGNING BUILD STAGES

7.1 Multi stage builds

Ixian-docker provides tools to create and arrange build images in stages. Each stage produces an image which may be cached in the registry. Each stage builds on the stages before it.

```
# For example, a python web app with a javascript front-end might have these␣
↪build steps.

Base -> Python -> NodeJS -> Pip -> NPM -> Webpack -> Runtime
```

By splitting the build into stages earlier steps can be cached and skipped, reducing the length of rebuilds. The ideal stage to cache is one that is lengthy to build but doesn't change too often.

Stages can be linked together dynamically using args in the Dockerfile for the image name and tag.

```
# The base image can be configured dynamically using build args
ARG $BASE_IMAGE
FROM $BASE_IMAGE
```

7.2 Nonlinear Builds

Multi stage builds need not be linear. They may be arranged in a tree structure to decouple lengthy build steps that aren't interdependent.

```
# For example, NPM and Python can be decoupled.

Base -> Python --------------> Runtime
     |                    /
     -> NPM -> Webpack -/
```

Once all intermediate images are built, they must be merged into a final runtime.

1. Pick one of your images to be the main branch. This should probably be the largest image.
2. `COPY` files in from other intermediate images

```
# build can be configured at runtime with tagged images.
ARG $PYTHON_IMAGE
ARG $WEBPACK_IMAGE

# merge compiled static from webpack into runtime
```

(continues on next page)

(continued from previous page)

```
FROM $WEBPACK_IMAGE AS webpack
FROM $PYTHON_IMAGE
COPY --from=webpack compiled_static $APP_ENV/
```

7.3 Registry Caching

Docker-ix supports using your docker registry as a cache for building images. Task state hashes can be used as identifiers for builds. When building the registry is checked for a matching identifier. If an image is present it's pulled instead of built.

Hint: if a stage is built, all descendant stages will be built too. Order your stages so slower and least frequently updated stages come first.

Registries are configured in your `ixian.py` see Registry Setup for details.

CHAPTER

EIGHT

IMAGE LAYOUT

Ixian modules use a Dockerfile layout designed to support modular *multi stage builds*. The layout is structured minimize COPY commands and simplify task checks when working with multiple stages.

8.1 Base Image

Image builds must specify a base image. This may be a stock image such as Phusion/base-image or your own base image.

A customized base image is a good place to put lengthy installs or configuration that doesn't change very often. Note that all other images extend the base image. Changes here trigger a rebuild of all other images.

Examples of files that belong in the base: * Package updates and installs (e.g. apt, yum) * Certificates * Any other common tooling

Note: Ixian tries to be platform agnostic but when needed it's build stages will be designed for Phusion/base-image, a docker optimized Ubuntu variant.

http://phusion.github.io/baseimage-docker/

8.2 Modules

Modules build a heirarchy of stages on top of the base image.

Modules files are stored in DOCKER.WORK_DIR. Files are arranged by the stages they are added in.

```
/opt/project
     +- bin            // All module executables may go in this directory.
     |   exe_1         // These do not trigger rebuilds.
     |   exe_2
     |
     \- etc
        +- module_1    // Module config belongs in etc. Each module uses it's own
        |    file_a    // directory
        |    file_b
        \- module_2
             file_c
```

Executables share a directory. Changes to executables don't trigger rebuilds so they share a directory.

```
/opt/project
     +- bin
         exe_1
         exe_2
```

Config files are split up into separate directories to simplify completeness checks and building images. Built-in modules each only use their own directory. Only this directory needs to be checked by completeness checks.

```
class MyImageBuildTask(Task)
    check = [
        # Checking the entire directory without enumerating specific files
        FileHash("/opt/project/etc/my_module")
    ]
```

When the image is built only these two directories need to be copied in.

```
# In the modules Dockerfile, copy in the related files.
COPY root/project/bin/ /opt/project/bin
COPY root/project/etc/my_module /opt/project/etc/
```

8.3 Runtime Image

The runtime image is used to combine files from the intermediate images. It also adds runtime config files that weren't needed by build stages. This is the final step to building an image.

Examples of runtime files:

- Test runner and lint configs
- Web server configs
- `.env` files

```
\- etc
   +- runtime    // The runtime has a config directory like everything other module
   |    file_a
   |    file_b
```

If they can be, built-in tools are configured to expect the config file in the modules `etc`. Some build tools require symlinks to the config file to be added to the `DOCKER.WORKING_DIR`. Built-ins that require this will indicate so in their setup instructions. These symlinks can be created in either base image or runtime image.

```
# for example, package.json must be in the working directory ``NPM install`` is␣
↪called from
RUN ln -s /opt/project/etc/npm/package.json
```

8.4 Development Environment

Development enviroment uses docker-compose to buid a runtime rather than the runtime itself. Within docker-compose volumes may be used for live-code editing. This avoids rebuilding the runtime image whenever it's dependencies change. If your build-stage stages are designed well, the runtime image only has operations to merge intermediate images making it simple to replicate with docker-compose volumes.

`bin` and `etc` only require a single volume each. Modules don't need to do anything special as they store their files under this directory.

```
docker-compose run \
-v root/project/bin/:/opt/project/bin \
-v root/project/etc/:/opt/project/etc
app
```

CHAPTER

NINE

DOCKER

Core docker layout and other utility tasks. Features for creating images and running containers are included.

9.1 Config

9.2 Tasks

9.2.1 clean_docker

Kill and remove all docker containers

9.2.2 build_base_image

Builds the docker app using `CONFIG.DOCKER_FILE_BASE`. This image is built prior to all intermediate layers.

9.2.3 build_image

Builds the final docker image using `CONFIG.DOCKER_FILE`

9.2.4 compose

Run a docker-compose command in app container.

9.2.5 bash

Bash shell in app container.

9.2.6 up

Start app container.

9.2.7 down

Stop app container.

CHAPTER

TEN

PYTHON

The Python module

https://www.npmjs.com/

This module builds an image containing installed python packages

Warning: This module is requires Python be installed in your image. Ixian does not *yet* provide a module that installs python for you but one is on the drawing board. For now install a system python. Likely the new module will use pyenv to install python versions.

10.1 Setup

1. Load the Python module within your `ixian.py`

```
# ixian.py

def init():
    load_module('ixian_docker.modules.python')
```

10.2 Config

10.3 Tasks

10.3.1 build_python_image

Build image with packages installed from requirements.txt

10.3.2 pip

The Pip package manager.

This task is is a wrapper around the `pip` command line utility. It runs within the container started by `compose`. You may use it to manage packages in a development environment.

Other arguments and flags are passed through to `pip`. For example, this returns `pip` internal help.

```
ix pip --help
```

CHAPTER

ELEVEN

PYTEST

The Pytest python test runner. This module provides the tasks and configs for using Pytest within your project.

```
pytest is a mature full-featured Python testing tool that helps you write better␣
↪programs.

The pytest framework makes it easy to write small tests, yet scales to support complex
functional testing for applications and libraries.
```

https://docs.pytest.org/en/latest/

Note: This module requires python is installed in your image. Ixian does not *yet* provide a Python module that does this for you but one is coming soon based on `pyenv`. Until then it is recommended you install a version of python using `pyenv`.

11.1 Setup

1. Load the Pytest module within your `ixian.py`

```
# ixian.py

def init():
    load_module('ixian_docker.modules.pytest')
```

2. Install pytest

Ixian doesn't install `pytest` for you. There are too many versions so it's up to you to install the version compatibile with your code.

If you're using the `PYTHON` module then just add `pytest` to your `requirements.txt`.

```
pip install pytest
```

3. Configure pytest

Pytest may be configured by creating a `pytest.ini` file. This is where you configure your app specific settings.

Here is a very basic config file

```
[pytest]
python_files = tests.py test_*.py *_tests.py
testpaths = src/my_app
```

Warning: Pytest normally may be configured by other means such as `pyproject.toml` but those are not officially supported. It may be possible to alter `PYTEST.ARGS` to use these other means.

11.2 Config

11.3 Tasks

11.3.1 pytest

Run the pytest python test runner.

This task is a proxy to the Pytest python test runner. It uses *compose* to execute `pytest` within the context of the app container.

Other arguments and flags are passed through to Pytest. For example, this returns `pytest` internal help.

```
ix pytest --help
```

CHAPTER

TWELVE

BLACK

The Black python formatter. This module provides the tasks and configs for using Black within your project.

```
Black is the uncompromising Python code formatter. By using it, you agree to cede␣
↪control over
minutiae of hand-formatting. In return, Black gives you speed, determinism, and␣
↪freedom from
pycodestyle nagging about formatting. You will save time and mental energy for more␣
↪important
matters.

Blackened code looks the same regardless of the project you're reading. Formatting␣
↪becomes
transparent after a while and you can focus on the content instead.
```

https://black.readthedocs.io/en/stable/

Note: This module requires python is installed in your image. Ixian does not *yet* provide a Python module that does this for you but one is coming soon based on `pyenv`. Until then it is recommended you install a version of python using `pyenv`.

12.1 Setup

1. Load the Black module within your `ixian.py`

```
# ixian.py

def init():
    load_module('ixian_docker.modules.black')
```

2. Configure Black

Black uses `pyproject.toml` for configuration. Here is an example config:

12.2 Config

12.3 Tasks

12.3.1 black

Run the **black** formatter.

This is a wrapper around **black** that runs it within the docker image using `compose`. Args are passed through to **black**.

For example, this returns **black** internal help.

```
$ ix black --help
```

12.3.2 black_check

Run the black formatter with `--check`. This task will return non-zero if any files require formatting but won't update them.

CHAPTER

THIRTEEN

DJANGO

This module provides the tasks and configs for using the Django web framework within your app. It provides high level functions for interacting with django while it's running inside a docker container.

```
Django is a high-level Python Web framework that encourages rapid development and␣
↪clean,
pragmatic design. Built by experienced developers, it takes care of much of the␣
↪hassle of Web
development, so you can focus on writing your app without needing to reinvent the␣
↪wheel. It's
free and open source.
```

https://www.djangoproject.com/

Note: This module requires python is installed in your image. Ixian does not *yet* provide a Python module that does this for you but one is coming soon based on `pyenv`. Until then it is recommended you install a version of python using `pyenv`.

13.1 Setup

1. Load the Django module within your `ixian.py`

```python
# ixian.py

def init():
    load_module('ixian_docker.modules.django')
```

2. Install Django

Ixian doesn't install `django` for you. There are too many versions so it's up to you to install the version compatibile with your code.

If you're using the `PYTHON` module then just add `django` to your `requirements.txt`.

```
pip install django
```

3. Create your django app

TODO: need to describe how to setup file structure and configure along with the python module

13.2 Config

13.3 Tasks

13.3.1 manage

Run the django manage.py management script.

This task is a proxy to the Django management script. It uses *compose* to execute `manage.py` within the context of the app container.

Other arguments and flags are passed through to Pytest. For example, this returns `manage.py` internal help.

```
ix manage --help
```

13.3.2 shell

Shortcut to `manage.py shell` within `compose` environment.

13.3.3 shell_plus

Shortcut to `manage.py shell_plus` within `compose` environment.

13.3.4 django_test

Shortcut to Django test runner

This shortcut runs within the context of the app container. Volumes and environment variables for loaded modules are loaded automatically via docker-compose.

The command automatically sets these settings: –settings={DJANGO.SETTINGS_TEST} –exclude-dir={DJANGO.SETTINGS_MODULE}

Arguments are passed through to the command.

13.3.5 migrate

Shortcut to `manage.py migrate` within `compose` environment.

13.3.6 makemigrations

Shortcut to `manage.py makemigrations` within `compose` environment.

13.3.7 dbshell

Shortcut to `manage.py dbshell` within `compose` environment.

13.3.8 runserver

Shortcut to `manage.py runserver 0.0.0.0:8000` within `compose` environment.

`runserver` automatically sets `--service-ports`.

By default runserver will start on `0.0.0.0:8000`. If any args are passed the first arg must be the `host:port`. For example this changes the port.

```
ix runserver 0.0.0.0:8001
```

13.4 Utils

13.4.1 manage

`manage` is a shortcut to calling manage.py with `run`

CHAPTER

FOURTEEN

NPM

The NPM module provides tasks for installing and managing javascript packages using the NPM package manager.

```
Relied upon by more than 11 million developers worldwide, npm is committed␣
↪to making JavaScript
development elegant, productive, and safe. The free npm Registry has become␣
↪the center of
JavaScript code sharing, and with more than one million packages, the␣
↪largest software registry
in the world. Our other tools and services take the Registry, and the work␣
↪you do around it, to
the next level.
```

https://www.npmjs.com/

This module has these features:

- It builds an image with NPM packages installed.
- NPM and NCU tasks for managing packages

Note: This module requires NodeJS be installed in your image. Ixian does not *yet* provide a NodeJS module but one based on NVM is in the works. Until then it is recommended you install a node version using NVM.

14.1 Setup

1. Load the NPM module within your `ixian.py`

```
# ixian.py

def init():
    load_module('ixian_docker.modules.npm')
```

2. Configure NPM

NPM uses `package.json` for configuration which your project must provide. This is configured with `NPM.PACKAGE_JSON` which defaults to `DOCKER.APP_ENV/package.json`

NPM requires `package.json` be in your working directory.

14.2 Config

14.3 Tasks

14.3.1 build_npm_image

Build the NPM image.

This is an intermediate image built using `DOCKER.BASE_IMAGE` as it's base. The resulting image will contain all packages as defined by `NPM.PACKAGE_JSON`.

This task will reuse existing images if possible. It will only build if there is no image available locally or in the registry. If `--force` is received the image will build even if an image already exists.

`--force` implies skip-cache for docker build.

14.3.2 ncu

Update packages using Node Check Update (ncu).

This task is used to update package versions defined in `NPM.PACKAGE_JSON`. By default this will only update the config file without updating the installed versions.

This task is is a wrapper around the `ncu` command line utility. It runs within the container started by `compose`. You may use it to manage packages in a development environment.

Other arguments and flags are passed through to `ncu`. For example, this returns `ncu` internal help.

```
ix ncu --help
```

14.3.3 npm

The NPM package manager.

This task is is a wrapper around the `npm` command line utility. It runs within the container started by `compose`. You may use it to manage packages in a development environment.

Other arguments and flags are passed through to `npm`. For example, this returns `npm` internal help.

```
ix npm --help
```

CHAPTER

FIFTEEN

JEST

The Jest javascript test runner. This module provides the tasks and configs for using Jest within your project.

```
Jest is a delightful JavaScript Testing Framework with a focus on simplicity.

It works with projects using: Babel, TypeScript, Node, React, Angular, Vue and more!
```

https://jestjs.io/

15.1 Setup

1. Load the Jest module within your `ixian.py`

```
# ixian.py

def init():
    load_module('ixian_docker.modules.jest')
```

2. Install Jest

Ixian doesn't install `jest` for you. There are too many versions so it's up to you to install the version compatible with your code.

If you're using the *NPM* module then just add `prettier` to your `package.json`.

3. Configure Jest

Jest is configured by `CONFIG.JEST.CONFIG_FILE` which defaults to `jest.config.json`. Your project must provide this config file.

15.2 Config

15.3 Tasks

15.3.1 jest

Run the Jest javascript test runner.

This task is a proxy to the Jest javascript test runner. It uses `compose` to execute jest within the context of the app container.

Configuration is configured by default as:

```
--config={JEST.CONFIG_FILE_PATH}
```

Other arguments and flags are passed through to `jest`.

For example, this returns `jest` internal help.

```
$ ix jest --help
```

CHAPTER

SIXTEEN

PRETTIER

The Prettier javascript formatter. This module provides the tasks and configs for using Prettier within your project.

```
# What is Prettier?

- An opinionated code formatter
- Supports many languages
- Integrates with most editors
- Has few options
```

https://prettier.io/

16.1 Setup

1. Load the Prettier module within your `ixian.py`

```
# ixian.py

def init():
    load_module('ixian_docker.modules.prettier')
```

2. Install Prettier

Ixian doesn't install `prettier` for you. There are too many versions so it's up to you to install the version compatibile with your code.

If you're using the `NPM` module then just add `prettier` to your `package.json`.

```
npm install --save prettier
```

3. Customize config if needed

Prettier works without them but if you want to customize config as needed:

- .prettierrc - config file
- .prettierignore - ignore files

These files should be present or symlinked in the working directory of the app (`DOCKER.APP_ENV`).

16.2 Config

16.3 Tasks

16.3.1 prettier

Run the Prettier javascript formatter.

This task is a proxy to the Prettier python formatter. It uses `compose` to execute `prettier` within the context of the app container.

Other arguments and flags are passed through to prettier.

For example, this returns `prettier` internal help.

```
ix prettier --help
```

16.3.2 prettier_check

Run the prettier formatter with `--check`. This task will return non-zero if any files require formatting but won't update them.

CHAPTER

SEVENTEEN

WEBPACK

The Webpack javascript bundler. This module provides the tasks and configs for using Webpack within your project.

```
webpack is a module bundler. Its main purpose is to bundle JavaScript files for usage␣
↪in a
browser, yet it is also capable of transforming, bundling, or packaging just about␣
↪any resource
or asset.
```

https://github.com/webpack/webpack

17.1 Setup

1. Load the Webpack module within your `ixian.py`

```
# ixian.py

def init():
    load_module('ixian_docker.modules.webpack')
```

2. Install webpack

Ixian doesn't install `webpack` for you. There are too many versions so it's up to you to install the version compatibile with your code.

If you're using the `NPM` module then just add `webpack` to your `package.json`.

```
npm install --save webpack
```

3. Configure webpack

Webpack config is stored in `webpack.config.js`.

17.2 Config

17.3 Tasks

17.3.1 build_webpack_image

Build image with javascript, css, etc. compiled by Webpack.

This is an intermediate image that extends `DOCKER.BASE_IMAGE`.

Ixian includes a template for this image. The dockerfile is configured by `WEBPACK.DOCKERFILE`. By default it's a jinja template that renders to `WEBPACK.RENDERED_DOCKERFILE`.

The image will store compiled output in `WEPBACK.COMPILED_STATIC_DIR` by default.

This task will reuse existing images if possible. It will only build if there is no image available locally or in the registry. If `--force` is received the image will build even if an image already exists.

`--force` implies skip-cache for docker build.

17.3.2 webpack

Run the webpack javascript/css compiler.

This is a wrapper around `webpack` that runs it within the docker image using `compose`. Args are passed through to `webpack`.

For example, this returns `webpack` internal help.

```
$ ix webpack --help
```

CHAPTER

EIGHTEEN

ESLINT

The ESLint javascript linter. This module provides the tasks and configs for using ESLint within your project.

```
ESLint is a tool for identifying and reporting on patterns found in ECMAScript/
↪JavaScript code.
```

https://eslint.org/

18.1 Setup

1. Load the ESLint module within your `ixian.py`

```
# ixian.py

def init():
    load_module('ixian_docker.modules.eslint')
```

2. Install ESLint

Ixian doesn't install `eslint` for you. There are too many versions so it's up to you to install the version compatibile with your code.

If you're using the `NPM` module then just add `eslint` to your `package.json`.

```
npm install --save eslint
```

3. Customize config if needed

ESLint works without it but you may customize settings with `.eslintrc`

18.2 Config

18.3 Tasks

18.3.1 eslint

Run the ESLint javascript linter.

This task is a proxy to the Prettier python formatter. It uses `compose` to execute `eslint` within the context of the app container. This task returns non-zero if linting fails.

Other arguments and flags are passed through to prettier. For example, this returns `eslint` internal help.

```
ix eslint --help
```

Ixian-Docker is a tool that manages docker builds and provides development tooling for interacting with your docker app. Prebuilt modules are included to construct an application stack quickly. Ixian's goal is to build applications with sane defaults, but not stand in your way if you'd like to configure or extend it to better suit your needs.

There are several things Ixian-docker will help you with:

- Building a heirarchy of docker images.
- Pluggable platform features like Python, NodeJS, Django, and more.
- Provides a command line interface to your application running within a local container.

CHAPTER NINETEEN

INSTALLATION

```
pip install ixian_docker
```

CHAPTER

TWENTY

SETUP

20.1 1. Create `ixian.py`

Ixian apps must be initialize in `ixian.py`. Here is a basic setup for a django app.

```
# ixian.py
from ixian.config import CONFIG
from ixian.module import load_module

def init():
    # Load ixian core + docker core.
    CONFIG.PROJECT_NAME = 'my_project'
    load_module('ixian.modules.core')
    load_module('ixian_docker.modules.docker')

    # Minimal setup for Django backend + Webpack compiled front end
    load_module('ixian_docker.modules.python')
    load_module('ixian_docker.modules.django')
    load_module('ixian_docker.modules.npm')
    load_module('ixian_docker.modules.webpack')
```

20.2 2. Configure Docker Registries

Configure docker registries for pulling and pushing images.

```
# Specify the registry for your images. The image's name will be generated
# from this url and path.
#
#   e.g. my.registries.domain.name.com/my_project
#
CONFIG.DOCKER.REGISTRY = 'my.registry.domain.name.com'
CONFIG.DOCKER.REGISTRY_PATH = 'my_project'
CONFIG.DOCKER.REGISTRIES = {
    'my.registry.domain.name.com': {
        'client': DockerClient,

        # addtional options may be passed in
        'options': {
            'username': "my_registry_user"
            'password': "my_registry_password"
        }
```

(continues on next page)

(continued from previous page)

```
        }
}
```

See the section on *docker registries* for more information.

20.3 3. Module config

Modules each have their own requirements for configuration. Built-in modules have sane defaults where possible. See specific module docs for details.

CHAPTER TWENTYONE

USAGE

21.1 Basics

Ixian apps are executed using the `ix` runner. This is the entry point for `ixian` apps. The general help page lists the available tasks.

```
$ ix build_image

usage: ixian [--help] [--log LOG] [--force] [--force-all] [--clean]
     [--clean-all]
     ...

Run an ixian task.

positional arguments:
  remainder    arguments for task.

optional arguments:
  --help       show this help message and exit
  --log LOG    Log level (DEBUG|INFO|WARN|ERROR|NONE)
  --force      force task execution
  --force-all  force execution including task dependencies
  --clean      clean before running task
  --clean-all  clean all dependencies before running task

Type 'ix help <subcommand>' for help on a specific subcommand.

Available subcommands:

[ Build ]
  build_image                Build app image
```

Internal flags should be placed before the `task`.

```
ix --force build_image
```

Any args after the task name are passed to task's execute method.

For example, many tasks are wrappers around other command line tools. Pass `--help` after the command to get that tool's internal help.

```
ix pytest --help
```

21.2 Tasks

Once configured you will have access to a number of tasks for building and interacting with the app in your image. These will vary depending on what modules you've enabled. Here are a couple of examples.

- Build a docker image.

  ```
  ix build_image
  ```

- Run the django test server.

  ```
  ix runserver
  ```

- Run automated tests.

  ```
  ix test
  ```

21.3 Task checks

Many tasks have state checks that determine if they are already complete. This can be viewed in ixian task help. Completed dependencies are indicated by a check.

```
STATUS
 build_image
    ✓ build_base_image
      build_npm_image
      build_webpack_image
      build_python_image
```

If the task or any of it's dependency are incomplete then the task and it's incomplete dependencies will be run.

```
STATUS
 build_image
    ✓ build_base_image
    ✓ build_npm_image
      build_webpack_image
      build_python_image
```

When all are complete then the task can be skipped. If checkers detect changes, such as modified config files, the checkers will indicate a build.

```
STATUS
✓ build_image
    ✓ build_base_image
    ✓ build_npm_image
    ✓ build_webpack_image
    ✓ build_python_image
```

Note: If there are no checkers then a task runs every time it is called.

21.4 Forcing tasks

Task checks may be bypassed with `--force`. Pass `--force-all` to bypass checks for all dependencies.

21.5 Clean build

Some tasks have a clean function that removes build artifacts. Pass `--clean` to call the clean function prior to building. Pass `--clean-all` to trigger clean for all dependencies. If a task doesn't define a clean method then `--clean` does nothing.

Passing `--clean` also implies `--force`.

21.6 Built-in help

All tasks have built in help generated from task docstrings and metadata. The help page should explain how to configure and use the task. It also displays the state of tasks and any dependencies.

When in doubt, check `help`.

```
$ ix help build_image

NAME
    build_image -- Build app image

DESCRIPTION
Builds a docker image using CONFIG.DOCKER_FILE

STATUS
 build_image
     build_base_image
     build_npm_image
     build_webpack_image
     build_python_image
```

CHAPTER

TWENTYTWO

WHAT'S AN IXIAN?

Ixian is a flexible build tool that this project is built with. Ixian provides the platform to define and arrange a heirarchy of interrelated tasks into a command line app.

CHAPTER TWENTYTHREE

INDICES AND TABLES

- genindex
- modindex
- search