ronto

Release 1.5.0

Mar 25, 2021

Contents

Ronto is a cli tool for building stuff using repotool, Yocto and Docker. It is intended to simplify Yocto build environments and processes.

- It can be used by developers who just want to build a single recipe.
- It can also be used for headless CI builds or release builds covering a set of machines and images.
- All build activities can be transparently performed within a docker container or on bare metal.

Ronto is just the **ronto** command and it's sub-commands plus **ronto.yml** control file located in the yocto project directory.

Ronto is the proposed prefix for 10^-27 of something. It is like Yocto which is the prefix for 10^-24 of something.

CHAPTER 1

User Guide

1.1 Installation and Dependencies

`ronto` requires python3, specifically python3.5+.

The simplest way to install ronto is using Pip.

```
$ pip3 install ronto
```

This will install `ronto` and all of its dependencies.

1.2 Usage

The usage is very simple. There is just the command **ronto** with sub-commands for the different specific tasks.

The **ronto** command reveals what it is capable of and specifically what sub-commands exist by running:

```
$ ronto
```

without any parameter.

or if you need help to one of the sub command call:

```
$ ronto <subcommand> --help
```

Be aware that there are global options like *–verbose* or *–dryrun* that must be given right after the **ronto** command whereby the sub-commands can have sub-command specific options.

The **ronto** command is expected to be called in the yocto project root directory. It operates based on configuration in a **ronto.yml**. That **ronto.yml** is supposed to be located in the very same directory. An alternative **ronto.yml** can be given by *-f* or *–file* global option.

1.3 Ronto Commands

1.3.1 Bootstrap

ronto simplifies the process of starting from scratch by bootstrapping initial *ronto.yml* file and also a *site.conf* configuration.

This bootstrapping ends up in essential consistent settings. **ronto** can manage different sources of configuration and recipes.

- do not manage at all - *ignore* option
- use a set of git repository specification - *git* option
- use repotool and manifest repository specification - *repo* option

Also, the build process runs

- either in a docker container
- or bare metal.

While bootstrapping those two orthogonal parameter ranges are considered. Such that the set of user questions are minimized and tailored according to those two parameters.

e.g. by invoking:

```
# get an idea
ronto bootstrap --help
# use manifest repository run in docker container
ronto bootstrap --source repo --container
# do not manage sources and do not run in a container
ronto bootstrap -s ignore
```

1.3.2 Docker

ronto offers the ability to run all Yocto build activities in a docker container. This happens fully transparently if docker is configured in the *ronto.yml*.

```
docker:
```

If the top level entry *docker* exists, all activities are performed in a docker container.

There are many docker related things to configure in the *ronto.yml* described at *ronto-file-docker*.

As a prerequisite,

- docker engine and docker command must be installed and
- the user who runs *ronto* must have docker privileges.

Ronto deals with docker in the following way:

1. Ronto pulls a docker general Yocto tools image
2. Ronto creates a privatized docker image for building.
3. Ronto creates a container with injected directories for project home, yocto cache and optionally, ssh keys and publishing
4. Ronto start the container and keep it running, exec into the container and run the required commands and tasks. After the work has been done stop the container.

The privatized image is derived from the general yocto image and is characterized as follow:

- It has a default user named *yocto* including a home directory at the usual path */home/yocto* available. Most importantly, the UID and GId of the *yocto* user is the same as the calling user on the host. This is because yocto build jobs do not run as root (and must not). It allows to nicely inject ssh keys for pulling private source repositories while building, it allows reading, and housekeeping (removal) of produced build artifacts outside of the container if injected.
- It has the ronto tool installed. This allows invoking ronto commands inside the image as without docker, also fully transparently.

There a two ways to use build approaches with docker containers.

1. Build the the container on the fly from a capable image, mount in the build environment, build and remove the container.
2. Build a container (including mounting the build environment). Give the container a name and reuse the container. I.e. do not dispose the container.

Second approach is perceived as bad practice since additional container housekeeping is required.

Still, due to the fact, that mounting the build environment might require up to four volumes (complex). Since it is desireable to also use the docker build easily without *ronto*, second approach has been chosen.

Each yocto build environment has to have its own container, because of different build environment mounts.

Note: The naming scheme for the container is simple and as follow:

<privatized-docker-image-name>-<yocto project directory short name>.

E.g. if the privatized docker image is *my-yocto-bitbaker* (This is the default if not differently specified in *ronto.yml*) and the yocto project directory is */home/volker/yocto/ams* the container name is: *my-yocto-bitbaker-ams*.

What volumes are mounted and where is fully described in the *ronto-file-docker* section.

Note: Cache or download directories described e.g. in *site.conf* must address paths **inside** the container.

There is a dedicated *ronto docker* command that allows running own command in the build container.

```
ronto docker --help
ronto docker ls  # list content of project folder
ronto docker --interactive  # run interactive bash in container
```

As without docker it is possible to run an bitbake tasks interactively within a sourced Yocto environment like

```
ronto build --interactive
```

This is the same as:

```
ronto docker --interactive 'ronto build -i'
```

Note: Interactive session can be finished by typing *exit* command in bash. It might be possible that entering *exist* is required multiple times if docker exec bash calls e.g ronto command and ronto itself invokes a bash again for interactive building.

For convenience it is possible to cleanup docker by:

- Remove the build container:

```
ronto docker --rm-container pwd  # pwd is just a short arbitrary command
```

- Remove the build container, the privatized image

```
ronto docker --rm-priv-image pwd
```

- Remove the build container, the privatized image and also the pulled big image that contains the yocto prerequisite tools.

```
ronto docker --rm-priv-image pwd
```

1.3.3 Run Ronto Scripts

like *npm* does, *ronto* offers a script execution with the *run* command.

```
$ ronto run
```

looks for the default script named *all* (borrowed from make) and runs that script. if *all* is not defined in the *ronto.yml* file, it assumes the following default for all.

```
scripts:
  all:
    - fetch
    - init --clean-conf
    - build
```

Thus, all will execute three steps by calling*ronto fetch*, *ronto –clean-conf* and *build* as sub-processes sequencially.

It will stop after an error and not continue any further steps.

Environment variables will be passed on. and so would global settings like verbosity flag, dry-run flag or the set of command line variables. injections.

It is possible to use environment variables or to set variables. The following example shows how it works.

scripts do not have options. Any required variability can be easily addressed by injecting command line variables prior the name of the run <script> command.

```
defaults:
  INTEGRATION: default
  RELEASE: stable
  MANIFEST: default
repo:
  url: git@github.com:almedso/repo-yocto.git
  manifest: {{ MANIFEST }}.xml
build: sources/ams/conf/{{ TARGET }}.yml
scripts:
  release:
    - "--env MANIFEST={{ RELEASE }} fetch --force"
    - "init  --rm-build"
    - "--env TARGET={{ RELEASE }} build --publish"
  integration:
    - "--env MANIFEST={{ INTEGRATION }} fetch --force"
    - "init  --rm-conf"
    - "--env TARGET={{ INTEGRATION }} build --publish"
```

E.g the release script can be run like

```
$ ronto -env RELEASE=2020-04 run release
```

It runs release script and uses *2020-04* as *RELEASE* environment variable. The variable is used to pick appropriate manifest files to pull the sources and would select 2020-04 targets as being subject for a release.

1.4 Rontofile Reference

The ronto file named *ronto.yml* is the home of ronto settings. This name *ronto.yml* plus the location in the root directory of the (Yocto) project is established as convention. It enforces two things:

- Projects are operated from this directory and as long as the those conventions are maintained experienced developers and integrators find themselves "at home" quickly.
- One save time in providing additional parameters on the command line. Thus, less tiny typing errors can be made.

The following full blown *ronto.yml* including documentation shows the maximum capabilities and explains the meaning of those single values.

Yaml format is used to arrange and express the settings. See Wikipedia for introduction and Yaml home for formal specification.

If an obvious content line is commented out, this means the given value is taken as a default. There is no need to have this setting part of the *ronto.yml*, it will assemble a compact presentation.

1.4.1 Rontofile protocol version

Since the *ronto.yml* is subject to modification a strict mode reading is supported if a version tag is set. The version is an unsigned integer.

```
## providing a version forces ronto to check against the list of
## supported versions. If no version is given, ronto just tries
## its best.
version: 1
```

In case version a version is set processing is stopped if

- either the version cannot be converted to an unsigned integer
- or the version is higher than the currently supported version.

This documentation references to *ronto.yml* protocol *version 1*. This related **ronto** applications supports to *ronto.yml* protocol *version 1*.

1.4.2 Build Source Specification

Build sources (or configuration sources or just sources) are all Yocto layers, classes, recipes and configurations needed to build something using Yocto. The *ronto* tool allows to specify

- nothing - it is up to manual configuration out of scope from *ronto*.
- a set of git repositories - *ronto* supports initial cloning only.
- a manifest repository - *ronto* invokes google repo tool to init and sync.

A mix is possible although not really recommended.

```
## if repo is set the google repo tool is used to pull sources from
## upstream and locally. It requires the repo tool installed.
## the url parameter is mandatory.
repo:
  url: git@github.com:group/manifest-repo.git  ## replace by your url
  ## optional manifest default is default.xml
  manifest: default.xml
  ## optional branch default is master
  branch: master

## If repo is not used: Alternative source definition is via repo.
## Only if not locally available yet the sources are pulled
## If a source directory is available no update is performed
git:
  ## in case the list of git repositories is empty the following (poky)
  ## is picked as a default
  - source_dir: sources/poky  # this entry is used if the list is empty
    git_url: git://git.yoctoproject.org/poky  # same here
  ## more than one repository are possible like below
  - source_dir: sources/meta-openembedded
    git_url: https://github.com/openembedded/meta-openembedded
```

1.4.3 Build Processing Specification

This section clarifies how configuration files (local.con, bblayers.conf, site.conf) are created/updated, how the build directory structure looks like as well as what kind of clean is applied before building.

```
## The build section
build:
  ## The init script needs to be sourced to prepare the environment to
  ## run bitbake. The default poky script is used if nothing is given
  init_script: sources/poky/oe-init-build-env

  ## if not set no template dir is injected and the defaults from poky are
  ## used. This is the place to inject custom local.conf(.sample) and
  ## custom bblayers.conf(.sample)
  template_dir: sources/poky/meta-poky/conf

  ## If not set the default "build" as specified in the poky init script is
  ## used. Most likely it is not subject to change.
  build_dir: build

  ## <build_dir>/conf/site.conf is used to establish site specific settings
  ## Use an alternative file to establish <build_dir>/conf/site.conf
  ## Default is site.conf in project root directory
  site:
    file: site.conf ## path is relative to project root directory
```

1.4.4 Build Targets

Build targets are best defined in the meta layer where machines and images are defined. This is where they belong to.

```
## Part of the build section
build:
  ## Build targets are best defined by referencing a remote yaml formatted
  ## file containing a list of target specification.
  ## The file should be in the repository where respective machines/ images
  ## are defined and therefore are known.
  targets_file: sources/my-repo/conf/build-targets.yml

  ## If the targets_file item is not given, alternatively the targets are
  ## given by the targets items directly. The sub-element is a list of
  ## targets. If not given the yocto/poky getting started default is
  ## assumed.
  targets:
    - image: core-image-sato  ## yocto default
      machine: qemux86  ## of getting started
```

The *targets_file* yaml format is a list of dictionaries that must have *machine* and *image* keys. Other keys are possible like publish that indicates that further processing, like publishing the build artifact.

```
- image: ams-image
  machine: roderigo
  publish: yes
- image: ams-image
  machine: roderigo
  publish: no
```

1.4.5 Publishing

Different targets are possible and useful. Publishing happens to certain web URL's that are provided by a web server. on the backend site those urls are mapped to a publishing base directory.

- *image artifacts* are needed for initial installation.
- *package artifacts* are needed for individual package update via package management.

Furthermore, publishing of root filesystems via nfs as well as and kernels and device trees via tftp boot protocol. is usefully during development.

```
## Publishing of packages and
## if set 'publish_package is defined packageing actions are performed
publish:
  ## directory where packages will be sent to.
  ## must be an absolute path
  ## if in docker part of the publish volume
  ## if not set - and docker is configured this is equal
  ## to docker: -> publish_dir: -> container
  webserver_host_dir: xxx
  ## relative path extension to webserver_host_dir
  ## pointing to package feed root
  ## if set -> build output of packages packages are "rsynced" to that folder
  package_dir: feeds  ## default feeds
  ## relative path extension to webserver_host_dir
  ## pointing to image folder root
  ## if set targets with publish flag are copied over there
  image_dir: images  ## default images
```

Note: Package index has to be computed during build. If not configured by

```
build:
  packageindex:
```

publishing of packages will be suppressed.

1.4.6 Using Docker

ronto is capable to delegate all builds to a docker container, running a docker image with Yocto prerequisites installed. *ronto* takes over container management (image download, creation), container startup and volume injection and build execution transparently.

```
## docker is a toplevel item. if present, building is delegated
## to a docker container, otherwise the local machine is used to
## build.
docker:

  ## Docker image that contains the Yocto requirements for building plus
  ## ronto tool (this tool) and optionally if desired the google repo tool.
  image: almedso/yocto-bitbaker:latest

  ## Privatized_image item indicates that a privatized image is to be used
  ## if it is present. If additionally an image name is given, this image
  ## name is used instead of the default.
  ## privatized images are needed if sources need to be pulled where access
  ## credentials (ssh key pairs) are required. Only in privatized build
  ## containers ssh key pairs and ssh configuration can be injected.
  ## privatized means: a user 'yocto' exists that has the same uid:gid like
  ## the invoking user. The users home directory is '/home/yocto'.
  ## Yocto builds cannot be executed as root.
  privatized_image: # my-yocto-bitbaker

  ## The docker container requires several volumes to be injected.
  ## Per volume mapping there is the directory name/volume name on
  ## the _host_ side and the directory name on the _container_ side.
  ## The respective names are arranged along those keys.

  ## A project root directory must be injected as volume to the container.
  ## On the host side the directory is always the project directory (as
  ## the name suggests. It cannot be configured differently.
  project_dir: /yocto/root

  ## The cache directory is the optional.
  ## If not given, all caching is done inside the container and thrown
  ## away when the container is destroyed.
  ## The site.conf script should set download cache (DL_DIR) and
  ## Shared state cache (SSTATE_DIR) to directories below this directory
  cache_dir:
    host: $(pwd)/../cache  ## one level up the project directory
    container: /yocto/cache  ## interacts with side.conf settings

  ## If a publishing dir is given publishing of results (images or packages)
  ## is possible. This means images or packages are copied/rsynced
```

(continues on next page)

(continued from previous page)

```
  ## to the respective container path. and would show up on the host path.
  publish_dir:
    host: volume or path
    ## Used as default by this script
    container: /yocto/publish
```

Variables

Definitions can be overwritten by shell environment variables or variables injected on the command line via *-e* or *–env* global option.

- Injection via command line parameter overwrites injection via environment variables.
- Injection via command line comes along with site effects but shows up in shell history
- Injection via shell environment variables might be important if secrets need to be passed on.
- Injection via shell environment might be complicated when used in a docker environment

There are two constraints:

- Each used environment variable must be listed in the default section.
- A default value must be given for every environment variable. In case a certain environment variable is not set, this default is used.

Variables without a default that are not provided cause an processing error at runtime when they are evaluated. Variables are evaluated at the moment they are needed (late evaluation).

It is possible to have up to two variable evaluation per yml element.

Assuming on the shell the SSTATE_DIR environment variable is set:

```
export SSTATE_DIR=/yocto/foobar
```

and the content of the *ronto.yml* is:

```
# Environment variable defaults
defaults:
  DL_DIR: "/foo"
  YOCTO_BASE: "/yocto"
  SSTATE_DIR: "/yocto/bar"
build:
  download: "{{ YOCTO_BASE }}{{ DL_DIR }}"
  shared_state: "{{ SSTATE_DIR }}"
```

download will be set to */yocto/foo* (the default) and *shared_state* will be set to */yocto/foobar* (obtained from the process environment.

Alternatively the SSTATE_DIR can be set on the command line like

```
SSTATE_DIR=/yocto/foobar ronto --env SSTATE_DIR=/yocto/foo init
```

The result would be that *shared_state* will be set to */yocto/foo* (obtained from command line parameter (because of it's higher priority)).

1.5 Change Log

1.5.0

- bootstrap sub-commands
- Documentation update

1.4.0

- run sub-commands
- Documentation update

1.3.0

- update docker sub-commands
- Documentation update

1.2.0

- publish sub-commands
- Documentation update

1.1.0

- build sub-commands
- Documentation update

1.0.0

- Fetch sub-commands
- Documentation update
- Read variables from command line
- Component tests (bdd) introduced
- Unittest replaced by py.test
- Coverage removed
- Rontofile version strict reading

0.1.2

- Move to pipy registry
- Publish to read the docs

0.1.1

- Make it releasable

0.1.0

- Rontofile version 1 (not checked)
- init, build and arbitrary docker sub-commands

0.0.1

- Project created.

1.6 Report Bugs

Report bugs at the issue tracker.

Please include:

- Operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

CHAPTER 2

Architecture

This architecture guide is the home for vision, requirements (scenarios, use cases, functional and non-functional requirements) as well as of architecture stuff.

Furthermore, the structure is rather arbitrary, established on the fly rather than a "formal" requirement specification or architecture documentation

Nevertheless, it is the home to understand some backgrounds, ideas better.

Note: This section does not claim to be up to date.

2.1 Preample

Yocto builds require a complex environment to run on.

Application developers, BSP developers, integration engineers, release engineers have different needs to build something with Yocto.

A lot of upstream input has to be maintained in terms of

- bblayers
- sources

Different quality attributes are applied like

- fast turn around time for app developers
- reproducibility for release engineers
- fast feedback for integrators

The output of Yocto builds is complex as well and is handle differently.

2.1.1 Build Process

0. Access the build host (login to bare metal machine or pick docker image)
1. Gather the build specification (layers, recipes, configuration)
2. Initialize the build environment (source an init script)
3. Pick a machine and an image/recipe and build
4. Use the build result (publish, run)

Gather build specification

Quite often upstream projects suggest using the Google repotool to access bblayers available and maintained in git repositories.

Alternatively kas tool. - very good but a technology lock.

Gather build specification

2.1.2 Terminology

Developer build

- Performed by a developer on a developers computer
- Purpose is pure development, one build "small" target only

Verification build

- Performed by a ci service on a headless computer
- Fast feedback if the change/pull request might break master branch/trunk

Integration build

- Performed by a ci service on a headless computer
- Fast feedback if current state of integrated software still works
- Provides latest integrated software in all variants

Integration release build

- Performed by a ci service on a headless computer
- Input is a pinned configuration
- Provides reproducible and identifiable (by version) integrated software in all variants

2.2 Requirements

2.2.1 Use Cases

Developer bootstraps Yocto build environment

```
ronto bootstrap
```

Developer has to answer a couple of questions. Result is a created project directory plus a *ronto.yml* inside the project directory.

Developer builds something on his machine

Developer builds locally on his machine a *one specific* package and deploys it to his *private development bare metal* target.

```
$ ronto build --fetch --init --interactive
(yocto)> bitbake
```

Integrator builds locally

Integrator builds locally on his machine an image for two targets and deploy and tests to to integration targets.

```
ronto --env TARGETS=mytargets.yml build --fetch --init
```

CI server builds and publishes

Continuous Integration Server builds a set of images for a set of machines and publishes some images as well as all packages

```
ronto fetch
ronto init --cleanconf
ronto --env TARGETS=sources/recipe-repo/conf/integration-targets.yml build
ronto --env TARGETS=sources/recipe-repo/conf/integration-targets.yml publish
```

Release Engineer releases

A release engineer pins/releases a certain successful CI build. afterwards he/she build the pinned release.

> # do some pinning stuff ronto –env MANIFEST=release_xyz.yml fetch ronto init –cleanbuild ronto –env TARGETS=sources/recipe-repo/conf/release-targets.yml build ronto –env TARGETS=sources/recipe-repo/conf/release-targets.yml publish

2.2.2 Requirements

- This tool shall work with or without docker
- This tool shall work with and without repo tool.

This tool is a convenient wrapper: Yocto builds with or without this tool shall work the same way.

Quality Attributes

- The project config file is the central play that should operate with references to details that are rather bound to recipes and configuration repositories.
- CLI and config file shall be self-explaining and intuitive
- Different pront colors (interactive if on docker/bare metal
- Different promt color if bitbake environment is active

2.3 Concept

Problems to be solved:

- support of docker environments | bare metal environments
- private repositories of sources (credential handling in docker)
- handling site.conf
- version pinning of integration releases
- Consistent CI/Releasing commit messages and tags (established conventions)

2.3.1 Include of sub-rontofiles

Quality requirement: Maintainability

The build targets specification has to be located in the custom recipes repository. Only in that repository are machines and images known. Following the "strong coupling principle" the build targets specification must be maintained in that repository as well.

Note: Late loading must be implemented, since it might be possible that this specification is only or updated available after the fetching step

---

2.3.2 Cleanup flags

E.g. in rontofile:

```
build:
  flags:
    - cleanconf
    - cleanbuild
    - cleansstate
```

Solution:

Are part of init sub-command can be customized into *ronto*scripts.

2.3.3 Command line variables

Like reading from environment it is possibible to read from command line (as global parameter)

2.3.4 Compose of own Commands

Allow custom commands (including options + injections) like git config alias.xxx "blablub"

```
scripts:
  do-special:
    - '--env REPO=foo fetch'
    - '-f integrate.yml build'
    - '-f integrate.yml publish'
```

Pro:

- more flexibility to handle e.g. process specifics of integration builds or release builds
- support of customized developer workflows / development cycles

Con:

- more complexity

Solution:

- is implemented as proposed redesigned as run command and scripts section.

2.3.5 Multiple prioritized config files

- Must be given at command line. Later on the command line implies higher priority.
- Items at higher priority overwrite items with lower priority. (same mechanism like css)

Pro:

- flexibility
- homogeneous handling
- little effort

Con:

- Overwrite rules are not super intuitive
- Concepts are not easily readable
- (like salt: Problem solving if you only have a hammer, everything turns a nail)

Solution:

- same thing is achived via extra "scripts" section, variable substituion, and include e.g. of target file.

2.4 Open Issues

2.4.1 Generate site.conf

Pro:

- intelligent mapping between docker and none docker directories
- no additional file to store

Con:

- one more concept

```
### -- Not implemented yet - still subject to evaluation --
## Generate build/conf/site.conf from values
## to do
## either with semantics for distro, upstream, download, sstate_cache
## or from list of define strings
generate:
    download: "download"
    shared_state: "shared-state"
    distro: "{{ ams }}"
```

```
ronto --env FOO=bar fetch
ronto -eBAR=foo build
```

Pro:

- compose of own commands is better supported
- scripting is better since amount of site effects is minimized

Con:

- little more effort

CHAPTER 3

Developers Guide

3.1 Contributing Guide

Contributions are welcome and greatly appreciated!

3.1.1 Workflow

A bug-fix or enhancement is delivered using a pull request. A good pull request should cover one bug-fix or enhancement feature. This ensures the change set is easier to review and less likely to need major re-work or even be rejected.

The workflow that developers typically use to fix a bug or add enhancements is as follows.

- Fork the `ronto` repo into your account.
- Obtain the source by cloning it onto your development machine.

```
$ git clone git@github.com:your_name_here/ronto.git
$ cd ronto
```

- Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

  Now you can make your changes locally.

- Familiarize yourself with the developer convenience rules in the Makefile.

```
$ make help
```

- Create and activate a Python virtual environment for local development.

```
$ make venv
$ source path/to/<venv-name>/bin/activate
(venv) $
```

The rule creates the virtual environment outside the project directory so that it never accidentally gets added to the change set.

Note: (venv) is used to indicate when the commands should be run within the virtual environment containing the development dependencies.

- Prepare your virtual environment for development:

```
(venv) $ python setup.py develop
(venv) $ pip install -r requirements.dev.txt
```

The *requirements.dev.txt* contains tools like coverage that are needed to test, docs, etc.

- Develop fix or enhancement:
  - Make a fix or enhancement (e.g. modify a class, method, function, module, etc).
  - Update an existing unit test or create a new unit test module to verify the change works as expected.
  - Run the test suite.

    ```
    (venv) $ make test
    ```

    See the *Testing* section for more information on testing.
  - Check code coverage of the area of code being modified.

    ```
    (venv) $ make check-coverage
    ```

    Review the output produced in `docs/source/coverage/coverage.html`. Add additional test steps, where practical, to improve coverage.
  - Fix any errors or regressions.
- The docs and the change log should be updated for anything but trivial bug fixes. Perform docs check.

  ```
  (venv) $ make docs
  ```

  See the *Documentation* section for more information.
- Commit and push changes to your fork.

```
$ git add .
$ git commit -m "A detailed description of the changes."
$ git push origin name-of-your-bugfix-or-feature
```

A pull request should preferably only have one commit upon the current master HEAD, (via rebases and squash).

- Submit a pull request through the service website (e.g. Github, Gitlab).
- Check automated continuous integration steps all pass. Fix any problems if necessary and update the pull request.

3.2 Testing

The ronto project implements a regression test suite that improves developer productivity by identifying capability regressions early.

Developers implementing fixes or enhancements must ensure that they have not broken existing functionality. The ronto project provides some convenience tools so this testing step can be quickly performed.

Developer **unit tests** are supported with *py.test* as well as **component tests** are supported with *behave*.

The **unit tests** are specified in the *tests* folder. The **component tests** are specified in the *features* folder.

Use the Makefile convenience rules to run the tests.

```
(venv) $ make test
```

To run tests verbosely use:

```
(venv) $ make test-verbose
```

Alternatively, you may want to run the tests suites directly. The following steps assume you are running in a virtual environment in which the `ronto` package has been installed. If this is not the case then you will likely need to set the `PYTHONPATH` environment variable so that the `ronto` package can be found.

```
(venv) $ py.test  # unit tests
(venv) $ behave  # component tests
```

3.3 Code Style

Adopting a consistent code style assists with maintenance. This project uses the code style formatter called Black. A Makefile convenience rule to enforce code style compliance is available.

```
(venv) $ make style
```

3.4 Type Annotations

The code base contains type annotations to provide helpful type information that can improve code maintenance.

Use the Makefile convenience rule to check no issues are reported.

```
(venv) $ make check-types
```

3.5 Documentation

To rebuild this project's documentation, developers should use the Makefile in the top level directory. It performs a number of steps to create a new set of sphinx html content.

```
(venv) $ make docs
```

To quickly check consistency of ReStructuredText files use the dummy run which does not actually generate HTML content.

```
(venv) $ make check-docs
```

To quickly view the HTML rendered docs, start a simple web server and open a browser to http://127.0.0.1:8000/.

```
(venv) $ make serve-docs
```

3.6 Versioning

This project is a command line tool. Semantic versioning is applied and interpreted as follow:

- **major change: New commands, new ronto.yml version - or just something** important
- **minor change: Additional options, additional fields in ronto.yml** no behavior change.
- patch change: Same behavior, bug-fixes applied

Versions with major equal to zero imply an expectation of sematic instability in cli and ronto file.

3.7 Release Process

The following steps are used to make a new software release.

The steps assume they are executed from within a development virtual environment.

- Check that the package version label in `__init__.py` is correct.
- Create and push a repo tag to Github.

```
$ git checkout master
$ git tag vMajor.Minor.Patch -m "A meaningful release tag comment"
$ git tag  # check release tag is in list
$ git push --tags origin master
```

  - This will trigger Github to create a release at:

```
https://github.com/{username}/ronto/releases/{tag}
```

- Create the release distribution. This project produces an artefact called a pure Python wheel. The wheel file will be created in the `dist` directory.

```
(venv) $ make dist
```

- Test the release distribution. This involves creating a virtual environment, installing the distribution into it and running project tests against the installed distribution. These steps have been captured for convenience in a Makefile rule.

```
(venv) $ make dist-test
```

- Upload the release to PyPI using

```
(venv) $ make dist-upload
```

  The package should now be available at https://pypi.org/project/ronto/

CHAPTER 4

Quick Start

ronto is available on PyPI and can be installed with pip. Ronto requires python at version 3.5 or higher.

```
$ pip3 install ronto
```

After installing ronto the ronto command is available to you.

The build specification is maintained in a *ronto.yml*.

Start and explore with

```
# bootstrap a new build ronto.yml
# -- not implemented yet --
ronto bootstrap

# fine grained step by step build
ronto fetch
ronto init
ronto build
# -- not implemented yet --
ronto publish

# or a custom command
# -- not implemented yet --
ronto run all
```