Installing and Maintaining the Yocto Autobuilder 2

This guide will walk through how to install a stand-alone autobuilder controller and worker, and then reconfigure it with new builds, etc. The guide was written with the help of Richard Purdie under the context of a headless Ubuntu 18.04.02 Server installation.

1) Stand-alone Installation

The final outputs of this section are a controller and worker installed in the same server, ready for trimming back to an individual organization's needs.

> NOTE: The guide assumes that your host OS has the packages installed to support BitBake for the release(s) you are targeting. Please refer to the Yocto manual for those packages.

The latest version of BuildBot is written in Python 3, so installation via pip3:

```
apt install python-pip3 git build-essential python3-pip virtualenv enchang npm
sudo pip3 install buildbot buildbot-www buildbot-waterfall-view buildbot-console-view buildbot-grid-vie
```

It is recommended to also install `testtools` and `libccpunit-subunit-dev` (via apt, in this case) on the worker in order for certain image tests to work correctly (e.g., `core-image-sato-sdk:do_testimage`).

For a new installation you will need a system user that can run Autobuilder2 as well as a couple of repositories: Autobuilder2 itself and a helper plugin. As root:

```
useradd -m --system pokybuild3
cd /home/pokybuild3
buildbot create-master -r yocto-controller
buildbot-worker create-worker -r --umask=0o22 yocto-worker localhost example-worker pass
cd yocto-controller
git clone https://git.yoctoproject.org/git/yocto-autobuilder2 yoctoabb
ln -rs yoctoabb/master.cfg master.cfg
cd ~
git clone https://git.yoctoproject.org/git/yocto-autobuilder-helper
chown -R pokybuild3:nogroup /home/pokybuild3
```

> **IMPORTANT:** In the above command you created a controller and a worker, which will attempt to join the controller using `pass` as the password. Feel free to change this, knowing that if you do, you must change the controller's master configuration file to match.

At the end of this, your build user's home directory (e.g., `/home/pokybuild3`) should look like this:

```
yocto-autobuilder-helper
yocto-controller
yocto-controller/yoctoabb
yocto-worker
```

Next, we need to update the `yocto-controller/yoctoabb/master.cfg` towards the bottom where the `title`, `titleURL`, and `buildbotURL` are all set. This is also where you would specify a different password for binding workers to the master.

Then, we need to update the `yocto-controller/yoctoabb/config.py` to include our worker. In that file, find the line where `workers` is set and add: ["example-worker"]. *NOTE:* if your worker's name is different, use that here. Section 3.1 discusses how to further refine this list of workers.

> **IMPORTANT:** You should also take this time to edit the `sharedrepodir` and `publish_dest` variables to be in your build user's home as well. You will need to create these directories since the related code that will check for if they exist *will not* also attempt to create them (and the server will crash on start).

Next, if you do not want to edit the original `yocto-autobuilder-helper/config.json`, you can overlay your own by creating, for instance, `yocto-autobuilder-helper/config-local.json`. *NOTE:* there is no way for your overlay to *remove* builders or other attributes, so this route is really more about extenending the original set of builders.

Here are some suggestions for the sake of :

1. In the original `config.json`, find all instances of whatever `BASE_HOMEDIR` was set to, for example `/home/pokybuild3`. Copy those variables to your `config-local.json` replace `/home/pokybuild3` with `${BASE_HOMEDIR}`. These will be variables like `BUILDPERF_STATEDIR` and `EXTRAPLAINCMDS`. Set `BASE_HOMEDIR` should be your build user's home directory. (There are shell scripts where this is assumed.)
2. Add `BASE_SHAREDDIR` and `BASE_PUBLISHDIR` such that they are subtrees of your `BASE_HOMEDIR`, e.g., `${BASE_HOMEDIR}/srv/autobuilder.yoursite.com`.
3. Change your `WEBPUBLISH_URL` to match your `config.py` definition for `buildbotURL`.
4. In order for this to work, you must export `ABHELPER_JSON="config.json config-local.json"` into the environment of the controller and janitor services (the example service scripts included below already have this).

   NOTE: The way the build step is written, the worker will pull a fresh copy of the helper from the server. Therefore these configuration files must be committed to the `yocto-autobuilder-helper` repo location you have specified in `yoctoabb/config.py` because the worker is given a build step that pulls from that repo (see `yoctoabb/builders.py`).

Finally, as root, add the `yocto-*.service` files to `/lib/systemd/system` (See Appendix A). Run: `systemctl daemon-reload`. You should now be able to successfully start these services (e.g., `sudo systemctl start yocto-*`). The controller may take up to 15 seconds to start.

1.1) Special Notes for the Worker Environment

The QEMU tap interfaces also need to be generated and owned by the worker's user (created above). One way to this is to compile the `meta/recipes-devtools/qemu/qemu-helper/tunctl.c` file and run it *N* times on the worker's host. See the related `qemu-helper-native` recipe for instructions. The resulting executable would be run *N* times (e.g., 8), so for example: `sudo tunctl -u $(id -u pokybuild3) -g $(id -g pokybuild3) -t tun0`.

Another way to create these interface is to let the build fail once, then issue a command like this from a user with sudo permissions on the worker:

```
sudo /home/pokybuild3/yocto-worker/qemuarm/build/scripts/runqemu-gen-tapdevs \
    $(id -u pokybuild3) $(id -g pokybuild3) \
    8 \
    /home/pokybuild3/yocto-worker/qemuarm/build/build/tmp/sysroots-components/x86_64/qemu-helper-native
```

In the above command, we assume the a build named qemuarm failed. The value of 8 is the number of tap interfaces to create on the worker.

2) Basics

This section is an overview of operation and a few basic configuration file relationships. See Section 3 for more detailed instructions.

2.1) Start / Stop the Master and Worker Services

Per the installation in Section 1, both the Master and Worker are running on the same host and the service files (Appendix A) maintain that assumption. To start all services:

```
sudo systemctl start yocto-controller yocto-worker yocto-janitor
```

Depending on your web front-end setup (reverse proxy, etc.), you should now be able to access the BuildBot web UI at the address you specified in `yoctoabb/master.cfg` during the installation (e.g., https://localhost:8010).

2.2) Build Schedule Types

The `yoctoabb/schedulers.py` defines three main types of build schedules: triggerable, force, and nightly. The `wait-quick` and `wait-full` schedules are triggerable, and each has a specific list of builders that are defined in the `yoctoabb/config.py` variables: `trigger_builders_wait_quick` and `trigger_builders_wait_full`, respectively. Each of the builders in those lists also have a force schedule for manual runs. To run all of them, the `a-quick` and `a-full` schedules exist; those names are also the names of the builders. Finally, there is one nightly build defined which runs the a-quick builder against the `HEAD` of the various master branches to help maintain freshness of the shared state and downloads caches.

2.3) Running a Triggered Build Manually

Assuming you have the controller and worker running, log into the BuildBot web server on port 8010. Click on the Builds, Builders left menu. It should populate the main panel (right) with a table showing all builders defined between the `yoctoabb/config.py` and the combination of any config JSON files loaded for `yocto-autobuilder-helper` (e.g., `config.json`). The status of associated workers is shown with a bubble graph on the far right of this panel. One should be green, yours, because it's associated in `yoctoabb/config.py`, `builder_to_workers` map to that build by being one of the default builders.

Select one of the builders, for example `beaglebone`. At the top right, you will see *Force Build* (this is a force because all builders in the `yoctoabb/config.py` `subbuilders` list get a *Force Scheduler*, per `yoctoabb/schedulers.py`). Click that button and fill in your name (as *you your@email.com*) and a reason for forcing a build. Scroll to the bottom, setting branches and revisions you want along the way, and press *Start Build*.

Your browser will automatically navigate to the builder's status as the worker begins executing the build.

2.4) Shared State and Downloads Mirrors

One of the main useful features of having this build server is to speed up builds at development workstations. Both of these are defined by the `yocto-autobuilder-json/config.json` (or your extra configuration if following the standalone installation instructions above). The main variables are `BASE_SHAREDIR` and `BASE_PUBLISHDIR`. These two are re-used in the bitbake configuration variables DLDIR and SSTATEDIR to help individual builds re-use prior work. The default is to append `BASE_SHAREDDIR`

with `current_sources` and `pub/sstate`, respectively. These paths can be shared by NFS, HTTPS, etc. In the example below, we're using https and assuming that a static file server has been configured to share the former under `yocto_downloads` and the latter as `yocto_sstate`. To use these at a developer's station, set the following in the `build/conf/local.conf`:

```
SSTATE_MIRRORS ?= "file://.* http://your.site.com/yocto_sstate/PATH;downloadfilename=PATH \n"
PREMIRRORS_prepend = "\
    git://.*/.* http://your.site.com/yocto_downloads/ \n \
    ftp://.*/.* http://your.site.com/yocto_downloads/ \n \
   http://.*/.* http://your.site.com/yocto_downloads/ \n \
   https://.*/.* http://your.site.com/yocto_downloads/ \n \
"
```

The developers have stated that this single shared state and downloads cache can be shared across multiple tagged versions of Poky, so there is no need to maintain separate paths for different releases.

> Full Disclosure: In practice, the author has not seen a single successful cache query from the shared state mirror, as seen through the web server logs, despite using the same revision of branches at the server and desktop. YMMV. The downloads mirror however worked as expected.

2.5) Clearing Build History, Old Workers

> *TODO:* This is really heavy-handed; it removes all history. You could try your luck at directly editing the state.sqlite file.

All of the build history and worker related information is stored in a database, `yocto-controller/state.sqlite`. From the `yocto-controller` directory, while it isn't runnign, delete the database and recreate it: `buildbot upgrade-master`. Then restart the controller.

3) Configuration

As mentioned before, BuildBot is the underlying tool that Autobuilder2 uses. It's a python3-based framework that consists of a master and multiple workers that may be remotely connected. Per this guide, both are installed on the same host under `/home/pokybuild3`.

The configuration for the controller is in `/home/pokybuild3/yocto-controller`. This directory also contains the Yocto Project's Autobuilder source code (in `yoctoabb`) and master configuration file, `master.cfg`. The `master.cfg` is a python file that pulls in other configuration data from the Autobuilder source code directory. Based on comments in the `yoctoabb/master.cfg`, provisions have been made so that you can run `buildbot sighup` from this directory, which would cause the configuration to reload without taking down the controller. This configuration data file primarily controls the web server and what port(s) are open for workers. It also controls authentication and notification services (by way of importing `services.py` and `www.py`, see BuildBot's own documentation for answers).

The main `yoctoabb/config.py` pairs up with the `yocto-autobuilder-helper/config.json` to define the relationship between what builds exist and what those builds *do* in terms of steps to run. There is a lot of duplication between these two scripts that must be managed manually, especially as it pertains to builders, the layer repositories each needs, and the locations of those layer repositories.

Another interesting thing about this configuration is that only one branch of the `yocto-`

autobuilder-helper is ever pulled even if you manually specify a different branch for a non-triggered build. For example, manually Force Building beaglebone does not give you a chance to change yocto-autobuilder-helper branches but doing the same for a-quick would. So if you have a repository that contains multiple layers and for rocko you need one of them, but for thud you need 2, your builder will fail if you run the build on rocko because the worker will try to use the thud build instructions...the only ones it knows unless you are using multiple branches of yocto-autobuilder-helper (thud, warrior, etc.) and running a-quick forced builds or configured additional nightly builds for different branchs. There is also no way, from a single configuration JSON stack, to specify a builder being only compatible with a specific layer branch other than having multiple branches and omitting that builder from the incompatible branches, then forcing a trigger build. This is controlled by schedulers.py, so it's not a limitation; instead consider this a *flag* of something you probably want to change if you have to maintain multiple branches in regression.

The remaining portion of this section will focus on the changes required to the various configuration files to accomplish specific tasks, which will hopefully provide some guardrails and sign posts along the way for when you do something wrong.

3.1) Repositories

There are two main areas where repositories are defined. The first is in yoctoabb/config.py, which provides builders with a default set of repositories stored locally at the worker. The second place is more nuanced, in your config.json stack of the yocto-autobuilder-helper under the repo-defaults. This map defines a duplicate of that information, which is consumed by the layer-config and run-config scripts by way of the NEEDREPOS and ADDLAYER lists on a given template or override.

The NEEDREPOS behavior ensures that the copies of your meta layers are organized in the BUILDDIR/../ correctly, and then if no-layer-add is set to false (or omitted), will automatically call bitbake-layers add-layer... to update your build's bblayers.conf file. This process goes in order of build dependencies. The content of NEEDREPOS can be either a repo that is a layer, or a repo that contains multiple layers. In the latter case, specifying meta-openembedded/meta-oe will copy the whole repo meta-openembedded and then call bitbake-layers add-layer... for only the sub-layer, meta-oe (assuming you've set no-layer-add to false for that repo; the default is true).

The ADDLAYER behavior is similar but is processed during the run-config step that executes all of your steps. You can add this variable at the *step* level (it does not work at the builder level; run-config doesn't pick it up). Each list item in this variable takes the form: ${BUILDDIR}/../path/to/layer.

> **IMPORTANT:** The order of these two lists matter! If you add a layer that has unmet dependencies on other layers (i.e., they're not in bblayers.conf yet), the next repo/layer in the list will fail to add because you've technically broken your layer configuration (bitbake cannot parse it because dependencies are missing).
>
> **IMPORTANT:** If you allowed NEEDREPOS to update your bblayers.conf file, then you do not need to use ADDLAYER as it'll be redundant.

3.2) Workers

As stated previously, this is exclusively defined by the yoctoabb/config.py file. In it, there is a section of workers that culminate into a final map that defines build names vs.

compatible workers, with default carrying the generic meaning that it should be applicable to any build. Ultimately the list of workers defined in this configuration can be also thought of as *users* of the system because a worker will fail to join the master if it's not in this list or provides an incorrect password (variable: `worker_password`).

You will also notice in the standard configuration file that there are workers in the cluster for CentOS and Ubuntu, etc. since they're testing build host OS -specific things.

If you would like to trim this list down to just the workers you have at your site:

1. You can safely remove any of the `workers_*` lists since they're only used locally to the `config.py`.
2. Retain the following related variables: `workers`, `all_workers`, `builder_to_workers`. They're used elsewhere.

3.3) Builds and Builders

This section details how to configure new builds, and thus, new builders. There are two main files involved in this relationship: `yoctoabb/config.py` and the configuration JSON files you specified for `yocto-autobuilder-helper`. Only builds that exist in both places will actually be visible and usable at the UI. The remaining subsections below are general guidelines for how these various files interact.

Removing Builders:

1. You must keep the `buildertorepos` and `repos` maps. The former provides a top-level set of builders and must include a key for default. This map indicates which source code repositories to install for a given builder by way of a repository name. The compliment to this map is `repos`, the list of repository locations and revisions.
2. You must keep the `subbuilders` list because it is used by `yoctoabb/builders.py` and `yoctoabb/schedulers.py`.
3. You must keep the `a-quick` and `a-full` builds unless you are also modifying `yoctoabb/builders.py`, `generate-test-result-index.py`, and `schedulers.py` to remove those references as well.
4. For any build you remove from `yoctoabb/config.py`, you should also remove in your `yocto-autobuilder-helper/config.json` file (if not purely for the sake of being tidy).

Adding Builders:

1. If the build will only be manually run: add it to the `yoctoabb/config.py` subbuilders list.
2. If the build will be run by either of the default `a-quick` or `a-full` builds, add the name instead to the `trigger_builders_wait_[quick|full]` list of your choice. If you want both, add it to `the trigger_builders_wait_shared` list.
3. If you have added a build that has requirements differing from the `yocto-autobuilder-helper/config.json defaults` map, create an entry for the builder in `the overrides` map fill in the details (see below for more suggestions).
4. If the build has a set of layer repositories that differs from the `default` list in the `yoctoabb/config.py buildertorepos` map, you need to add a reference to its needs in that map.
5. If the the repositories the builder requires are not listed in the `yoctoabb/config.py repos` map, add it under the same name with its default branch.
6. If you have added repositories, you should also add it to the `yocto-autobuilder-helper/config.json repo-defaults` map.
7. You should create `an overrides` for the builder that specifies `NEEDSREPOS` to identify those layers and `ADDLAYER` for any layers that have the `no-layer-add` flag set to `false` in the `repo-defaults` map.

8. If these needs are shared among multiple builders, consider adding these changes instead to a new, named template in the `templates` map and then for each affected builder, set the value of its `overrides` `TEMPLATE` to that named template.

Anatomy: *overrides* and *templates*

All of these guidelines pertain to the `yocto-autobuilder-helper/config.json` file (and any overlay configurations you have). The main difference between these two items is that an `overrides` entry can specify the `TEMPLATE` variable, the value of which must exist in the `templates` map. There is no provision for template stacking (i.e., adding `TEMPLATE` to a template has no effect). Otherwise as the names imply the `overrides` values will take precedence over any named template and the `defaults`.

NOTE: This is not a comprehensive list.

1. Top-level variables:
2. `BUILDINFO` – boolean – If enabled, the values of `BUILDINFOVARS` are added to the list of variables in the build environment.
3. `BUILDHISTORY` – boolean – Enables `INHERIT += 'buildhistory'` BitBake behavior.
4. `BUILDINFOVARS` – list – For inheriting from `image-buildinfo` to save off additional build variables.
5. `DISTRO` – string – Distro conf to use from the layer in the build.
6. `DLDIR` – string – Set the bitbake `DL_DIR` to where downloads should be stored (e.g., `"DL_DIR = 'some_path'"`).
7. `PACKAGE_CLASSES` – string – Set to the value of the Yocto `PACKAGE_CLASSES` variable
8. `SDKEXTRAS` – list – Unknown. To some end, the example appends `SSTATE_MIRRORS` with the Yocto Project's shared state release mirror.
9. `SDKMACHINE` – string – Examples are `x86_64`, `i686`, etc.
10. `SENDERRORS` – boolean – Executes the `upload-error-reports` script which ultimately runs the `send-error-report` script (from `poky/scripts`) to upload the results to a log server.
11. `SSTATEDIR` – list – Presumably a list of additions to the `SSTATE_DIR` variable where each item in the list appends or removes that variable.
12. `SSTATEDIR_RELEASE` – list – Presumably a list of additions to the `SSTATE_DIR` variable during release builds
13. `WRITECONFIG` – boolean – If enabled, the `setup-config` script is run. **Required** if specifying `extravars`.
14. `extravars` – list – Contains additional BitBake varaibles that will be added to the tail of `build/conf/auto.conf`.
15. *stepN* -level variables:
16. `ADDLAYER` – list – These named layers will be added to the `bblayers.conf` file. At the end of the step, the layers will be removed in reverse order. This is useful if your `repo-defaults` -defined repository has `no-layer-add` set to `true`. This will log as *stepN**a***.
17. `BBTARGETS` – string – BitBake arguments passed to the bitbake, e.g., core-image-minimal. These will be appended with -k (continue) so that all targets will be attempted rather than stopping at the first error. This will log as *stepN**b***.
18. `SANITYTARGETS` – string – BitBake targets that will be run in an emulated environment for testing. This will log as *stepN**c***.
19. `EXTRACMDS` – list – List of commands to run within the BitBake environment (e.g., wic). This will log as *stepN**d***.
20. `EXTRAPLAINCMDS` – list – List of commands to run without sourcing the oe-init-build-env script. This will log as *stepN**d***.

Appendix A - Systemd Services

yocto-controller.service

```
[Unit]
Description=Yocto Autobuilder2 Master
Documentation=man:buildbot(1) https://docs.buildbot.net/
Wants=yocto-janitor.service

[Service]
PIDFile=/home/pokybuild3/yocto-controller/twistd.pid
Type=forking
WorkingDirectory=/home/pokybuild3/yocto-controller
User=pokybuild3
Group=nogroup
TimeoutStartSec=15
Environment=ABHELPER_JSON="config.json config-local.json"
ExecStart=/usr/bin/env buildbot start
ExecStop=/usr/bin/env buildbot stop
ExecReload=/usr/bin/env buildbot reconfig

[Install]
WantedBy=multi-user.target
```

yocto-worker.service

```
[Unit]
Description=Buildbot Worker
Wants=network.target
After=network.target
Wants=yocto-controller.service

[Service]
Type=forking
PIDFile=/home/pokybuild3/yocto-worker/twistd.pid
WorkingDirectory=/home/pokybuild3
ExecStart=/usr/bin/env buildbot-worker start yocto-worker
ExecReload=/usr/bin/env buildbot-worker restart yocto-worker
ExecStop=/usr/bin/env buildbot-worker stop yocto-worker
Restart=always
User=pokybuild3
Group=nogroup

[Install]
WantedBy=multi-user.target
```

yocto-janitor.service

```
[Unit]
Description=Buildbot Janitor
Wants=network.target
After=network.target

[Service]
Type=simple
PIDFile=/home/pokybuild3/yocto-autobuilder-helper/janitor.pid
WorkingDirectory=/home/pokybuild3/yocto-autobuilder-helper
Environment=ABHELPER_JSON="config.json config-local.json"
ExecStart=/home/pokybuild3/yocto-autobuilder-helper/janitor/ab-janitor
User=pokybuild3
Group=nogroup

[Install]
WantedBy=multi-user.target
```