Contributing to FreeBSD Ports with Git

BY JOSEPH MINGRONE

The FreeBSD ports tree was created in 1994 and tracked using CVS until July 15, 2012 when Subversion took over. A second repository conversion occurred on April 6, 2021 when the *source of truth* was migrated from Subversion to Git. As both CVS and Subversion are centralized version control systems, the required workflow changes associated with the first conversion were not as complex as with the conversion to Git, a distributed version control system.

This is not a comprehensive guide to using Git. The goal of this article is to guide those new to either Git or FreeBSD ports through *a* Git workflow that can be used to contribute to FreeBSD ports. Topics covered include:

- a brief overview of important Git concepts
- staying up to date with remote repositories
- working with branches
- committing
- modifying history
- working with Phabricator reviews
- testing changes with poudriere
- keeping track of upstream releases.

For a thorough introduction, refer to the Pro Git book and the Git Primer Chapter of the FreeBSD Committer's Guide. Also not covered is how to work with the `make` specifications that ports and the ports infrastructure are written in. This is covered in detail in the FreeBSD Porter's Handbook.

What makes Git fundamentally different from centralized version control systems like Subversion is its support for distributed workflows. Git does not require a central server that contains *blessed* copies of the versioned files because 1. copies of the repository are full clones that include meta-data and full history and 2. Git commits are snapshots of the repository rather than delta-based changes to files. Snapshots are described using hash algorithms that take as input the state of the repository and produce a deterministic hash value in the form of a hexadecimal string. If two copies of the repository are in the same state, the hash values describing the cop-

ies will be the same, whereas two repositories that differ by a single bit flip will produce different hash values. The history of a Git repository is a collection of these snapshots joined together so that each commit points to its parent commit(s).

A Git workflow may include 1. creating a local branch to develop a new feature, 2. merging the work in the feature branch with the main branch, and 3. pushing the changes to another Git repository. For a FreeBSD ports contributor, a new feature might mean creating or updating a port, or even something as simple as fixing a typo. When work in the feature branch is ready, it can be reviewed and merged with the official FreeBSD ports repository. Git branches are well suited for keeping the development of new features organized and isolated and their creation is very lightweight, as it simply involves creating a new pointer to a snapshot.

Because much of the work with Git occurs locally, there is no single workflow that all contributors must subscribe to. Work the way that best suits you. The official FreeBSD ports repository does enforce certain conventions though. For example, we require a simple, linear history of commits, so that the history of the main branch under Git looks similar to how it looked under Subversion. To do this, certain constraints are required, which we will discuss. Other projects use different workflows that results in parallel paths in the main branch of the repository. In short, Git is flexible and there is no single workflow that suits all people or projects. Indeed, as of early 2022 there is a FreeBSD working group exploring how we can optimize the way we work with Git, so refinements may be forthcoming. With these caveats out of the way, let's explore a Git workflow that is suitable for contributing to the FreeBSD ports tree.

Installing Git

The simplest way to get Git installed on your FreeBSD system is to use the official FreeBSD package.

```
pkg install git
```

For more information about installing third-party software on FreeBSD, refer to the FreeBSD Handbook Chapter on installing applications.

Cloning the Ports Tree

If you would like to contribute a new port to the tree, but do not already have something in mind, you can start by scanning the list of requested ports on the FreeBSD Wiki. Suppose we wish to create a new port for an application currently on the list, the Nyxt browser. The first step is to clone the FreeBSD ports repository. If you are using ZFS, you may wish to create a dedicated dataset for your development ports tree.

```
zfs create zroot/usr/home/ashish/freebsd/ports
```

Of course, substitute `zroot/usr/home/ashish/freebsd/ports` for your dataset layout. Now clone the repository. You are downloading the entire repository, which includes over 40,000 ports and a 28-year history, so this will take some time.

```
git clone -o freebsd --config remote.freebsd.fetch='+refs/notes/*:refs/notes/*'
https://git.freebsd.org/ports.git ~/freebsd/ports
```

The `-o freebsd` sets the name for the default remote repository for collaboration (pulling and pushing changes). The `--config remote.freebsd.fetch='+refs/notes/*:refs/notes/*.` adds Subversion revision numbers to the notes field of commits that occurred prior to the conversion to Git. When the clone is finished, you can optionally create a child ZFS dataset where software distribution files will be stored when building ports.

```
zfs create zroot/usr/home/ashish/ports/distfiles
```

Unlike the ports themselves, which are mostly text files, the software distribution files are usually already compressed, so zfs compression can be turned off for the `zroot/usr/home/ashish/freebsd/ports/distfiles` dataset.

```
zfs set compression=off zroot/usr/home/ashish/freebsd/ports/distfiles
```

You have a few options for telling `make(1)` about the location of your ports tree. The first option is to add configuration to `/etc/make.conf`.

```
.if ${.CURDIR:M/usr/home/ashish/freebsd/ports/*}
PORTSDIR=/usr/home/ashish/freebsd/ports
.endif
```

An alternative method is to set the PORTSDIR environment variable. For example, if your shell is `zsh`, you can add the line below to `~/.zshrc`.

```
export PORTSDIR=/usr/home/ashish/freebsd/ports
```

If you plan on working with multiple ports trees, a tool like `sysutils/direnv` is useful for loading or unloading environment variables depending on the current directory.

Staying Up to Date

The ports tree is actively developed, so changes will be pushed frequently to git.freebsd.org/ports.git. To fetch the changes that occurred in the upstream FreeBSD repository, use

```
git -C ~/freebsd/ports fetch freebsd
```

Fetching gives you an opportunity to inspect what changes have been made before integrating those changes into a local branch. Here `-C ~/freebsd/ports` instructs Git to operate on the repository under `~/freebsd/ports`. If the current working directory is `~/freebsd/ports`, which from this point on is assumed, this flag can be omitted. The `freebsd` argument means fetch from that remote repository.

To list the commits that were pushed to `freebsd`'s main branch that are not part of the local main branch, run

```
git log --oneline main..freebsd/main
```

Beside the topmost hash, you will see two pointers, `freebsd/main` and `freebsd/HEAD`. `HEAD` is normally a pointer to the last commit in the branch and in this case, like `freebsd/main`, it points to the last commit in the main branch of the remote repository. If we run

```
git log --oneline freebsd/main
```

and continue down the list of commits, we will eventually see `HEAD` and `main` which both point to the last commit on the local main branch. To integrate the new commits from `freebsd/main` into our local main branch, run

```
git merge freebsd/main --ff-only
```

The `--ff-only` (fast-forward only) option means only integrate the work from `freebsd/main` into `main` if it can be done by moving the `main` branch pointer to point to the same commit as `freebsd/main`. This can only happen when the commits listed in the output of

```
git log --oneline main..freebsd/main
```

descend from the local main branch. If changes have been made to the local main branch that are not part of `freebsd/main`, `--ff-only` will cause the `merge` to fail. In the workflow described here, we will never make direct changes to the local main branch, so this should never be a problem, but to be safe, we can configure the merge command to always use `--ff-only` with

```
git config merge.ff only
```

As a convenience, there is a `pull` command that will do both the `fetch` and `merge`. Depending on the circumstances, using `pull` may not be wise, because you do not get the opportunity to inspect what will be integrated into your local branch. If the commits in the main branch of your ports repository are always a subset of the commits in `freebsd/main` (as recommended here), this is less of a concern. To reduce the chances of diverging from `freebsd/main` when using `git pull`, we can configure the command to only do fast-forward merges as well with

```
git config pull.ff only
```

Creating a Local Branch

Now that we can keep our repository copy up-to-date with `git.freebsd.org/ports.git`, let's *create* changes. This is where Git really shines with the use of local branches, which provide a clean and efficient way to keep work-in-progress organized. Start by creating a new feature branch to work on the new nyxt port.

```
git branch nyxt
```

Now switch to the nyxt branch using

```
git checkout nyxt
```

A shorthand for both creating and switching to a branch is

```
git checkout -b nyxt
```

To check which branch you have checked out, you can run

```
git branch --show-current
```

You may find it useful to display the current branch in your shell prompt. If your shell is zsh, you can use `shells/git-prompt.zsh` from the ports tree. A nice feature of `git-prompt-zsh` is that it updates the prompt asynchronously, so when `git status` or some other Git operation is taking time to complete, it doesn't block other work. If this appeals to you and you use a shell other than zsh, there are similar code snippets to get Git status information in your prompt if your shell is `bash`, `fish`, or `tcsh`.

First Commit

After you have hacked on your new port, it is time to commit your changes. First, let's take a look at the status of the working tree with

```
git status
```

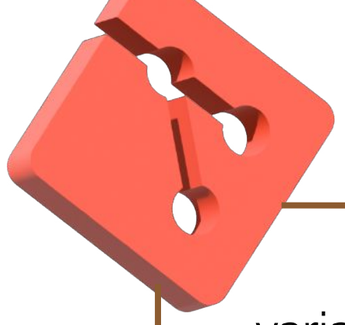
Depending on what work you did, this may tell you that the file `www/Makefile` was modified when you added `SUBDIR += nyxt` and you should also see `www/nyxt` as untracked. When interacting with the filesystem under the repository by adding, editing, or removing files, you are interacting with Git's working tree. Before you can commit changes to the repository, you have to stage which changes will be included in the next snapshot. In Git terminology, you add files from your working tree to the index. This extra step is useful, because it gives you precise control over what goes into a commit. To add all the changes to the index, run

```
git add www/Makefile www/nyxt
```

Now `git status` will list all the modified or added files as staged and ready to be committed. Before we commit though, there are a few more one-time tasks to complete. Git has a hook feature, which is a way to execute custom scripts when certain events like committing or merging occur. To configure Git to search the location where ports-specific hooks are stored in the ports repository, with the current working directory anywhere under the repository, run

```
git config --add core.hooksPath .hooks
```

That directory contains the `prepare-commit-msg hook`, which provides a helpful template for formatting commit messages. We also want to configure the editor that will be launched to create commit messages. Git chooses the editor to launch in this order: the value of the `GIT_EDITOR` environment variable, its `core.editor` configuration variable, the `VISUAL` environment



variable, and the `EDITOR` environment variable. For example, we can tell Git to use terminal Emacs to edit commit messages with

```
git config core.editor "emacs -nw"
```

If you would like to use this editor for all your Git repositories, add the `--global option` when setting `core.editor`.

```
git config --global core.editor "emacs -nw"
```

To commit your changes run

```
git commit
```

Your editor should now be displaying the commit template, which provides tips for creating a commit message. The subject line should take the form `<part of the ports tree that is changing>: <brief overview of the change>` and ideally be under 50 characters. A good subject line might be `www/nyxt: (WIP) First attempt to port Nyxt browser`. After a blank line, the body of the commit message provides more detail. An example might be

```
Makefile is still a skeleton.
TODO:
- Add _DEPENDS
- Add license information
- Fix QL_DEPS
- Add do-build target
```

After saving and exiting the editor your changes will be committed. So far, our changes progressed from the working tree, to the staging area (index), and finally to the local repository. To inspect your commit, use `git log`, which will also confirm that the `HEAD` and `nyxt` pointers have advanced one commit ahead of the main branch pointer.

Rewriting Local History

Whereas committing with Subversion meant sending your changes to the server, committing in Git simply means recording your changes locally in a new snapshot. Thus, with Git, it is wise to commit often. When it is time to share your work with others, you can refine your local history. There are a few different ways to rewrite history. For example, if you see a typo in your latest commit message, this is a good time to fix it, since your changes are still local. To modify the most recent commit, run

```
git commit --amend
```

and amend the commit message in your editor. If you accidentally did not stage and commit your changes to `www/Makefile` in the last commit, simply stage that file before running `git commit --amend` and it will be added to the last commit. Methods for rewriting history beyond the most recent commit will be discussed later.

Testing

Before requesting a review, your new port must be tested. There are two *port linters* that can alert you about common violations. Install them with

```
pkg install portlint portfmt
```

To lint your port with portlint, from `~/freebsd/ports/www/nyxt`, run

```
portlint -AC
```

To lint your port with portclippy from the portfmt package, also from `~/freebsd/ports/www/nyxt`, run

```
portclippy Makefile
```

Be aware, while these tools are generally quite helpful, they do not catch all mistakes and they can occasionally make ill-advised suggestions. Another useful tool is `portfmt`. As the name suggests, it can help with formatting your port's Makefile.

```
portfmt -D Makefile
```

Testing with Poudriere

Section 3.4 of the Porter's Handbook describes steps to test your port. It also refers readers to Chapter 10, which includes a guide for setting up poudriere, FreeBSD's bulk package builder and port tester. That section describes the merits of testing with poudriere. "[Various] tests are done automatically when running poudriere testport. It is highly recommended that every ports contributor install and test their ports with it." That Chapter of the Porter's Handbook describes a few different ways to set up a ports tree for poudriere. When you reach that section, it makes sense to tell poudriere to use our existing ports tree with

```
poudriere ports -c -m null -M ~/freebsd/ports
```

The `-m` option tells poudriere to use the null method, i.e., use an existing ports tree found at the location specified as the argument to `-M`. Using the null method means that we will manually manage the tree, including keeping it up-to-date and checking out the appropriate branch when testing. Once you have poudriere set up, you can test your port. If you created a jail named 13amd64, you can test the new port in that jail with

```
poudriere testport -j 13amd64 www/nyxt
```

Ideally you should test your port on the various tier 1 platforms (currently 12i386, 12amd64, 13amd64, and 13arm64). To test your new port after building it, poudriere can build a package and leave the jail running with the package installed.

```
poudriere bulk -i -j 13amd64 <category>/<port>
```

It's `-i` that instructs poudriere to leave the jail running with the package installed. This is useful for testing terminal applications, but not graphical applications like nyxt.

If the port has OPTIONS, poudriere will test and build the package as the official package builder will, i.e., with the default OPTIONS chosen. If you want to test or build the package with non-default options, you can run

```
poudriere options -j 13amd64 www/nyxt
```

before `poudriere testport...` or `poudriere bulk....`

Poudriere also creates a repository that pkg can use to install packages. If you want to install the package on the same system as poudriere, you have to configure pkg to use it. From PKG.CONF(5), a local configuration can be placed under */usr/local/etc/pkg/repos/*. The name of the file is not important, but it must have a `.conf suffix`. To set a local repository configuration and disable the default official repository configured in `/etc/pkg/FreeBSD.conf`, `create /usr/local/etc/pkg/repos/local.conf` with

```
FreeBSD: {
 enabled: no
}
Poudriere: {
 url: "file:///usr/local/poudriere/data/packages/13amd64-default"
}
```

The path given above assumes poudriere's default repository location, the repository based on the 13amd64 jail, and the default ports tree.

If you want to serve packages to remote hosts, you will need to configure a web server. Poudriere also has a web interface that can display information about current and past builds. If your webserver is nginx, you can configure it to host poudriere's interface and repository with a server entry like this in `nginx.conf`.

```
server {
    listen 80 accept_filter=httpready;
    listen 443 ssl;

    server_name pkg.example.org;

    root /usr/local/share/poudriere/html;

    ssl_certificate /usr/local/etc/dehydrated/certs/example.org/fullchain.pem;
    ssl_certificate_key /usr/local/etc/dehydrated/certs/example.org/privkey.pem;

    # If you use dehydrated as a Lets Encrypt client
    location /.well-known/acme-challenge {
```

```
alias /usr/local/www/dehydrated;
}

location /data {
    alias /usr/local/poudriere/data/logs/bulk;

# Allow caching dynamic files but ensure they get rechecked
location ~* ^.+\.(log|txz|tbz|bz2|gz)$ {
    add_header Cache-Control "public, must-revalidate, proxy-revalidate";
}

# Don't log json requests as they come in frequently and ensure
# caching works as expected
location ~* ^.+\.(json)$ {
    add_header Cache-Control "public, must-revalidate, proxy-revalidate";
    access_log off;
    log_not_found off;
}

# Allow indexing only in log dirs
location ~ /data/?.*/(logs|latest-per-pkg)/ {
    autoindex on;
}

    break;
 }

 location /repo {
    alias /usr/local/poudriere/data/packages;
    autoindex on;
 }
}
```

If you want to display poudriere's package building logs in the browser, tell nginx about text files with a `.log` suffix by editing the `text/plain` line in Nginx's `mime.types` to contain

```
text/plain log txt;
```

After restarting nginx with `service nginx restart`, point your browser to `http://pkg.example.org` to see poudriere's web interface.

Rewriting History to Prepare for Review

Before sharing your work, the commit history should be well organized, including the commit logs and the number of commits. Suppose the history on your nyxt branch contains seven WIP (work in progress) commits.

```
% git log --oneline
061be9ca5d98 (HEAD -> nyxt) www/nyxt: (WIP) ready for testing
cddad2b5886b www/nyxt: (WIP) Add missing www/Makefile entry
e42f79383312 www/nyxt: (WIP) Add build and install targets
807099e08e33 www/nyxt: (WIP) Fix QL_DEPENDS
3cc5f266b434 www/nyxt: (WIP) Complete _DEPENDS
80d098cd8367 www/nyxt: (WIP) Add license information
9ec91c5fb244 www/nyxt: (WIP) First attempt to port Nyxt browser
9f77e9601564 (freebsd/main, freebsd/HEAD, main) net-im/toxic: upgrade to v0.11.2
```

The commits above the `freebsd/main, freebsd/HEAD`, and `main` pointers are those in your `nyxt` branch that you want to clean up.

```
git rebase -i main
```

will show a log of the commits in your local nyxt branch. The `-i` option means the rebase will be interactive. We specify the commit preceding the subset of commits we wish to modify. In this case it is easiest to specify that commit with the `main` pointer. We could have also used tilde syntax, i.e., `HEAD~7` which means seven commits before HEAD, but it's tedious to count the seven commits.

This is what you should see in your editor.

```
pick 9ec91c5fb244 www/nyxt: (WIP) First attempt to port Nyxt browser
pick 80d098cd8367 www/nyxt: (WIP) Add license information
pick 3cc5f266b434 www/nyxt: (WIP) Complete _DEPENDS
pick 807099e08e33 www/nyxt: (WIP) Fix QL_DEPENDS
pick e42f79383312 www/nyxt: (WIP) Add build and install targets
pick cddad2b5886b www/nyxt: (WIP) Add missing www/Makefile entry
pick 061be9ca5d98 www/nyxt: (WIP) Ready for testing

# Rebase 9f77e9601564..061be9ca5d98 onto 9f77e9601564 (7 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
```

```
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified); use -c <commit> to reword the commit message
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
```

The history is written so that older commits are at the top. The comments below list all the commands we can use. We instruct Git on to how to modify history by placing these commands next to the commits. The default command beside each commit is `pick`, i.e., keep the commit as is. Here, we want to squash these WIP commits into a single commit for review. To squash the six latest commits into the first commit, change the `pick` command to squash in these bottom six commits.

```
pick 9ec91c5fb244 www/nyxt: (WIP) First attempt to port Nyxt browser
squash 80d098cd8367 www/nyxt: (WIP) Add license information
squash 3cc5f266b434 www/nyxt: (WIP) Complete _DEPENDS
squash 807099e08e33 www/nyxt: (WIP) Fix QL_DEPENDS
squash e42f79383312 www/nyxt: (WIP) Add build and install targets
squash cddad2b5886b www/nyxt: (WIP) Add missing www/Makefile entry
squash 061be9ca5d98 www/nyxt: (WIP) Ready for testing
```

When you save and quit your editor, Git will complete the rebase, then show you the log messages in your editor, so that you can write a new log message for the new, single commit. Here is an example commit message that we might want to use when sharing our work with others for review.

```
www/nyxt: New port for the Nyxt browser

Nyxt is a keyboard-driven web browser designed for power users.
Inspired by Emacs and Vim, it has familiar key-bindings and is
infinitely extensible in Lisp.

WWW: https://nyxt.atlas.engineer/
```

Refer to the November 2020 Journal article for a deeper discussion on Writing Good FreeBSD Commit Messages. Now `git log --oneline` will show a single commit in our `nyxt` branch.

```
7392483f6147 (HEAD -> nyxt) www/nyxt: New port for the Nyxt browser
9f77e9601564 (freebsd/main, freebsd/HEAD, main) net-im/toxic: upgrade to v0.11.2
```

Another way we will want to rewrite the history is by rebasing our work in the `nyxt` branch on top of an up-to-date main branch. First update the main branch.

```
git checkout main
git pull
```

Then switch back to the `nyxt` branch and tell Git to do the rebase.

```
git checkout nyxt
git rebase main
```

If all goes well, `git log` will show your commits in the `nyxt` branch descending from the latest commits from the main branch. If conflicting changes were made in `freebsd/main` and your `nyxt` branch, Git will inform you which files have conflicts and give you the opportunity to manually resolve them.

```
~/freebsd/ports [nyxt|✔] % git rebase main
Auto-merging www/Makefile
CONFLICT (content): Merge conflict in www/Makefile
error: could not apply 531d9081dfb1... Add new entry for nyxt browser
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase
--abort".
Could not apply 531d9081dfb1... Add new entry for nyxt browser
```

We can see the conflict is in `www/Makefile` and Git tells us what options we have to resolve the conflict manually. Here is an example of what we might see in `www/Makefile`

```
<<<<<<< HEAD
SUBDIR += nyan
||||||| parent of 531d9081dfb1 (Add new entry for nyxt browser)
=======
SUBDIR += nyxt
>>>>>>> 531d9081dfb1 (Add new entry for nyxt browser)
```

In this case, it is straightforward to manually fix the conflict. We want to add our entry for `nyxt` below the new entry for `nyan`. After editing the file so it looks like

```
SUBDIR += nyan
SUBDIR += nyxt
```

tell Git that we are ready to continue with

```
git add www/Makefile
git rebase --continue
```

Rebasing your feature branch onto an updated main branch is something you will do often enough that you may want to use a convenience script to do it in one step. Here is a simple example. Run `rum` from the feature branch to do the rebase in one step.

```
#!/bin/sh

# rum, r_ebase onto u_pdated m_ain
#
# Usage: rum
#
# globals expected in ${HOME}/.ports.conf with sample values
# No leading '/' on directory names means they are relative to $HOME
# portsd='/usr/home/ashish/ports' # ports directory

. "$HOME/.ports.conf"

usage () {
 cat <<EOF 1>&2
Usage: ${0##*/}
EOF
}
############################################## main
[ $# != 0 ] && { usage; exit 1; }

[ -n "${portsd##/*}" ] && portsd="${HOME}/$portsd"

# current branch
cb="$(git -C "$portsd" branch --show-current)"

if [ -z "$cb" ]; then
  printf "Could not determine the current branch.\\"
  exit 1
elif [ "$cb" = "main" ]; then
  printf "The main branch is checked out.\\n"
  exit 1
fi

git -C "$portsd" checkout main && \
  pull && \
  git -C "$portsd" checkout "$cb" && \
  git rebase main
```

Submitting Work for Review

Now we are ready to submit our work for review. FreeBSD currently has two ways to do this. Bugzilla is used for submitting bugs and Phabricator is used for reviewing source code changes. Both accept patches, but Phabricator has helpful features that are missing from Bugzilla, such as allowing reviewers to add comments specific to one or more lines of the patch. To cover both methods, let's create a review in Phabricator, then a new bug in Bugzilla that points to the Phabricator review.

FreeBSD Phabricator Reviews

To begin using FreeBSD's Phabricator instance for code review at https://reviews.freebsd.org, you must first create an account, then install the arcanist command line tool.

```
pkg install arcanist-php80
```

Set up `~/.arcrc` with the required certificates by running

```
arc install-certificate https://reviews.freebsd.org
```

and follow the instructions. Next, configure Arcanist to use https://reviews.freebsd.org as the default URI.

```
arc set-config default https://reviews.freebsd.org/
```

To submit your review, from the nyxt branch run

```
arc diff --create main
```

This will create a new review with all the commits in the nyxt branch. In this example, we squashed our commits into a single commit, so the revision will be created with that single commit. When your editor opens, you will have the opportunity to edit the fields that are part of the revision. The top line will be the subject of your commit log, `www/nyxt: New port for the Nyxt browser` and the summary will contain the rest of the commit log. Under test plan, you can list what you did to test the port. For example, if you did `poudriere testport` for each of the supported versions on the tier 1 architectures, you could write

```
poudriere testport 12/13 amd64/aarch64
```

You must also add at least one reviewer. If you have one or more ports committers that you have been working with, you can add their usernames here. For example

```
Reviewers: ashish rene
```

You can also specify group reviewers, which are of the form `#group_name` such as `#ports_committers`. The Subscribers: field, like `Reviewers:` takes a list of users, but these users do not reject or approve your work. When reviewers request changes, you can update the revision with

```
arc diff --update <revision>
```

where <revision> is the revision ID and takes the form DXXXXX. It can be found in the email sent to your address when you created the revision. For example, if your revision is found at https://reviews.freebsd.org/D33314, then use D33314 as <revision>.

Submitting Bugzilla Bug Reports

To create a new Bugzilla bug, point your browser to https://bugs.freebsd.org and click the `New` link at the top of the page. If you are not logged in to the FreeBSD Bugzilla instance, you will be prompted to do so. If you do not have a FreeBSD Bugzilla account, you can use the link on the login page to create a new one.

From here, you choose the `Ports & Packages` link since we are creating a new port and choose `Individual Port(s)` for the `Component`. For ports-specific bugs, the bug's subject line can be the commit subject prefixed with [NEW PORT], i.e., [NEW PORT] `www/nyxt: New port for the Nyxt browser`. If the port isn't new, the `category/port` prefix will automatically assign the bug to the maintainer of the port. In the description you can add the rest of the commit message and any other information helpful for others reading the bug. If you created a Phabricator review, add it to `See also`.

When your new port is accepted and pushed to `git.freebsd.org/ports.git`, your new job as the maintainer of the port begins. For an outline of the responsibilities of port maintainers, refer to the The challenge for port maintainers article. To keep up-to-date with upstream, portscout is a helpful service to alert you when there is a new release, so you can submit a port update. If upstream uses GitHub, you can also be alerted to new releases by following the `Watch` and `Custom` links, then check `Releases` on the project's page. When it's time to update your port and the changes are simple (e.g., only DISTVERSION/distinfo changes), submitting a Phabricator review may not be necessary. From a Git feature branch, you can create a patch using `git format-patch main` and attach it to a new Bugzilla bug. With Git, we now have more flexibility when crediting contributors for their work. When you submit a patch this way and a committer pushes it to `git.freebsd.org/ports.git`, `git log` will give you credit for your work. Even if you submit a traditional diff, committers have the option to set you as the author.

Opinionated Conclusions

Change can be hard. Many FreeBSD developers and contributors who dedicated significant time to becoming productive using Subversion were reluctant to change to a new version control system, especially one so fundamentally different. We lost some practical features like simple, monotonically increasing commit revisions and deterministic history retention when directories and files are moved within the repository. However, after three quarters of year, most indications suggest developers and the wider community are pleased and productive with the change. It is difficult to isolate the cause of certain outcomes, but the number of commits to the ports tree from the conversion date until the time of writing, 2021-04-06 to 2021-12-31 is 29,238. This is 1,748 more than the number for the same time last year. Let's hope this is a continuing trend in contributions to the ports tree.

JOE MINGRONE is a FreeBSD ports developer and works for the FreeBSD Foundation. He lives with his wife and two cats in Dartmouth, Nova Scotia, Canada.