Palantir.net presents

Git

Branching, Merging, Commits, and Tagging: Basics and Best Practices

palantir.net

Git Started

- This assumes you already have a repo.
- **Instructions for that are already on Github.**

Programming Branching

Naming Your New ~~Baby~~ Branch

There are four types of branches:

- **Master**
- **Release-[x]**
- **Feature-[x]**
- **[xx]-Support**

The Master

- **The Master branch should, at all times, reflect what is on the client's live site.**

- When a phase of development is to be released to the client, **the release-[x] branch is merged into the Master.**

Release-[x]

All development for project phase [x] happens in a branch named release-[x] *

- i.e., phase one of development happens on the **release-1** branch.
- Development on **release-1** will often continue post launch while work has also begun on **release-2.**

* (where [**x**] starts at 1 and increments with each new project cycle)

Features and Support

- These are branches that any dev can create.

- **Release-[x] and master are only touched by the team lead.**
- **Devs will create a feature or support branches as needed.**

- These are merged into **release-[x]** by the Lead when finished.

Feature Branches

- Development for pre-launch **features** are branched from **release-[x].**
- Prefix with the release branch to which they relate. Add ticket numbers where applicable.
  - e.g. **release-1-widget-of-doom** for doom_widget.module in phase 1.

When ready, it gets merged back into release-1 and then deleted.

Support Branches

Development for post-launch **support tickets** are branched from **master.**

Branch names should consist of a **ticket number** and a **short description.**

- Development for support ticket **#57: "Widget is full of bugs"** would be done on a branch named **57-widget-of-bugs.**

Naming your branch well is important. Almost all metadata about the branch lives in the name.

Your Very Own Branch

1. Create your own branch
2. Write some code
3. Stage it
4. Commit it
5. Push it

Your Very Own Branch

1. Create your own branch
2. Write some code
3. Stage it
4. Commit it
5. Push it

```
// Create your new branch.
$ git checkout -b new-feature-awesome-branch
// Write some awesome code, then stage it.
$ git add my-awesome-code.inc
// Look it over. (the 'cached' parameter shows you stuff you've already staged.)
$ git diff --cached
(Press 'q' when you're done looking at it.)
```

Your Very Own Branch

1. Create your own branch
2. Write some code
3. Stage it
4. **Commit it**
5. **Push it**

```
//Commit it.
$ git commit m- “This is my commit message”
//Push it.
$ git push -origin new-feature-awesome-branch*
```

* You only have to do the ‘-u’ and everything after that the first time you push the branch. If you don’t have a remote set up, check out the link to Github’s instructions on slide #2.

Merging

Get Ready

When your branch has reached perfection, it's ready to be merged.

Clean Merge

The downstream person is responsible for maintaining a branch that will merge back up cleanly.

Feature Branch

Start from the latest release branch version by merging it into your feature branch.

```
//If you don't have the branch yet, check it out.
$ git checkout feature-x
//Or if you do have it, pull the latest code.
$ git pull
//Update the feature branch* (with up-to-date release branch)
$ git merge origin/release-x
```

Merge Conflicts

- Fix any **merge conflicts.**
  - Consult with the author(s) of the changes
  - Consult with the Team Lead
  - Use a merge tool *
  - When in doubt, have a merge party
  - Communicate, **communicate, COMMUNICATE!**

Merge Tool

* **Wait, what merge tool?**

If you don’t yet have a favorite merge tool, **Gitx is a must**

- See what other people are doing
- Be aware of your project history
- **Investigate Git mysteries (!!!)**
- See when you haven’t pushed yet

Push It

Push your updated version of the feature branch:

```
$ git push
```

- Test.
- **If testing was unsuccessful, note that in the ticket.**
  - You're done.
- If testing was successful, continue.

Do the Merge

```
// Check out the release branch to do mergery on it
$ git checkout release-X
// Here it comes! The real merge!
$ git merge feature-X
```

Optionally add a more useful commit message for the merge with

```
$ git commit --amend
```

A Clean Send

Send it off:

```
$ git push
```

Clean up:

```
// Delete the remote feature branch after merge
$ git push origin :feature-X
// Delete the local feature branch after merge
$ git branch -d feature-X
```

Multi-Layered Approach

The tip of master is always stable

release-[x] is what Pro Git calls a *dev branch*

release-[x]-feature and **[xx]-support-branch** are subsidiary branches

Multi-Layered Approach

This keeps **production**, **staging**, and **development** separate.

Why? To **protect the codebase** and to give adequate **opportunity for review** before pushing to production.

What Should You Be Doing?

You should be developing

- in **branches**
- that you create **as needed**
- **Feature branches** branch off of **release-[x]**
  - and are named for the feature and release
- **Support branches** off of master
  - and are issue-number-keyed

Commits

What Should You Be Doing?

Making commits that are

- Small
- Frequent
- Well-commented
- To **support** of **feature** branches

Commit Early. Commit Often.

A Note About Compiled CSS

SASS compiles to CSS. If you edit a SASS file, those changes go in a commit that excludes the compiled CSS.

1. Edit the .scss file.
2. Commit the .scss file with a concise, but descriptive message.
3. Commit the .css file in a separate commit. A generic message is ok here.
4. If/when a merge conflict ever occurs, the mergeon* will resolve the conflict and then generate new compiled CSS**

* "**Mergeon**" n. 1. person doing the mergery.

** Mergery is performed on compiled CSS by running `$ compass clean` and then `$ compass watch` (assuming you're using Compass.)

Commits

Make small commits.

Why? To mitigate merge conflicts.

Well-Commented Commits

Commits should have concise **description in commit messages.**

Why? Many well-commented commits **form a roadmap** of how you got to the final release. Also helps you **prepare for meetings.**

Trackable Commits

Support commits should have the **ticket number*** **in the commit.**

Why? To enable **integration with issue tracker,*** and to provide **background info.**

*Assuming you're using an issue tracker such as those provided by Redmine, Jira, Github, etc.

Tagging

Making Tags

```
$ git tag
```

lists tags.

```
$ git tag -a v#.#.# -m “Comment.”
```

creates a tag with a release number and a comment.

Read more about tagging in git:

- Git Ready
- Pro Git

Pushing Tags

```
$ git push --tags
```

or

```
$ git push [tag-name]
```

If you want to push your tags somewhere other than origin, specify.

Make sure to push tags AND code.

TAGGING

Releasing Tags

Release tags are always on the master.

v1.1.2

“This release introduces support for foo and bar.”

Eventually, someone will screw up the tag number sequence.

Annotating Tags

Tagging with -m

records the following with the tag

- a message
- a timestamp
- a user

```
$ git tag -m v#.#.# “Comment.”
```

Note: If you type it without a comment in quotes, it will pop up a vim window for you to put your comments in.

Annotating Tags

Tagging with -a

records the following with the tag

- a timestamp
- a user

```
$ git tag -a v#.#.#
```

*Tagging with -a doesn't let you leave a message with the tag. **It's better to include a message,** but at least this way you can find out who to hunt down and ask about this release.

Annotating Tags

Tagging with -m gives **message**, **timestamp**, and a **user** (pops up message)

```
$ git tag -m v1.0.0 “Added support for foo bar baz.”
```

Tagging with -a gives you a **timestamp** and a **user**.

```
$ git tag -a v#.#.#
```

Use these to avoid starting a war with your future self.

Releases

Releases are roughly aligned by name with project milestones.

The stages of a release cycle are:

- Project begins
- Project is ready for Internal QA (**alpha** tags)
- Project is ready for client review (**beta** tags)
- Project is ready for **release**

Releases

NOTE:

Project with **multiple phases** with separate launch dates

= separate release cycles

Client with **multiple separate engagement contracts**

= separate release cycles

SUMMARY

tl;dr

- Branches get **descriptive names.**
- Development happens on **feature/support** branches.
- Keep **dev/stage/prod** branches separate
- Tag master releases with **annotated tags.**
- Commit early, often, and with **concise messages.**
- When in doubt, **communicate, communicate, communicate.**

SUMMARY

Resources

- Pro Git (public)
- Git Ready (public)

SUMMARY

Credits

- **Bec White** said the thing about starting a war with yourself.
- **Beth Binkovitz** wrote the words and put in the funny pictures.
- **Palantir design team** helped with the template.

Thanks!

Beth Binkovitz
binkovitz@palantir.net

palantir.net