Git Branching Strategy for Web Development

Posted At : November 9, 2015 4:00 PM | Posted By : Steve
Related Categories: Git

There is no doubt at this point that Git is a successful version control system. GitHub is, so far as I can tell, the most popular place to host open source code. There are several popular software programs for managing Git and two very popular Git branching strategies.

Each of the two most popular branching strategies that have I seen does a great job of solving the problem for which it was created. **GitHub Flow** is great for open source projects. The basic idea is that work is done via cloning and merge requests.

This works great for open source projects. If, however, all of your developers are part of the same team then the GitHub flow can really add unneeded friction to slow the development process.

The "**Git Flow**" strategy, initially popularized by the a great blog entry titled "**A Successful Git branching model**" (if you haven't read it, go read that and then come back here) is another successful branching strategy. When I first started using Git, this was the branching model I chose and I never regretted it. It is really great stuff.

My only complaint about the strategy is that it is really written for versioned software (1.X, 2.X, etcetera) and most of my work is on web sites. The two are similar, but not quite the same.

In Git flow, there are two permanent branches called "master" (which represents the production of state of the code) and "develop" which is the main development branch, which will eventually be merged into "master". Git flow also has various feature branches that are branched off of - and merged back into - the "develop" branch.

Git flow also has release branches and bugfix branches, but hopefully I can circumvent a discussion of those as the model I propose here (that has been in use by our team for about the last three years) is a bit simpler.

While Git flow is a great branching model, the underlying assumption is that at some point there will be a new release with all of the new features. This is not my experience of web development.

In my experience, clients ask for several independent features. Any one of these features may need to go live at any time, independent of any other feature. From that standpoint, everything that we do could fit into the "hot fix" model of Git Flow.

The first problem that we ran into with Git Flow was that we couldn't have a "develop" branch that would ever get merged to master, as each feature needed to be able to go live independently. An all-too-common scenario is that we would start on feature "A" when someone asked for feature "B" (which would need to go live as soon as it was ready, even if feature "A" wasn't yet) and then discover a more urgent feature "C".

Feature branches, however, are not enough. We often work with several different stakeholders on any one web site and therefore (or for other reasons) often have multiple independent features in progress for any one web site at any one time.

Our Model

Our Git branching strategy has only two kinds of branches: "Server Branches" and "Feature Branches". A server branch doesn't necessarily reflect a physical server, but rather a state of the site. If your team has a "Dev site" and a "Live site" or a "Dev Site", a "Staging Site", and a "Production Site" then you would have a server branch for each of these.

The "master" branch always represents the state of the live, production site. For most of our sites, we only have one more state for each site which is used mostly to show features to our clients. We call this our "Dev Site", but our development isn't actually done there. It is really to demonstrate features to our clients. Consequently, we call the branch for this our "demo" branch.

All new feature branches should always be created off of the master branch. Feature branches will be merged into the "demo" branch as soon as they are ready for review and merged into "master" to take them to production. Your process may require more server branches, but this is a simple example of the idea.

Server branches can never be merged directly into each other. Branches should never be created from any server branch except "master".

This allows us to have our development site demonstrate multiple features that are under development at any one time but still take each feature live as soon as it is ready.

Local Development

All of our actual development is done on repositories for each developer one their own computers. We keep the feature branch that we are working on as the active branch on our individual computers. We only switch to the server branches locally to confirm that merges have worked correctly.

Initial Set Up

Although the process should hopefully be clear, this seems like a good time to review it.

To set up, start with your master branch and create a new branch for it for each "server" used in your site. Keep in mind, that this has nothing to do with physical servers, but instead states or stages of your site. At this point, each server branch should be identical to the master branch.

The origin repository will be cloned onto each server, with the different servers checking out their respective branches.

Each developer will have a clone of the origin repository.

Process

When a developer starts to work on a new feature, they will create a feature from the master branch (always only from the master branch). Each developer will work on the feature until they either need involvement from another developer or they are ready to show it to the client, in which case they will push the branch up to the main repository.

If the feature should go onto one of the servers, then it will be merged into that branch and the respective server will pull the branch down. Each merge to a server branch represents an ideal time to have a continuous integration server run a suite of tests.

When the feature is ready to go live, merge it to the master branch and pull the master branch live.

Subfeatures and related features

In practice, some features are sufficiently complicated to need more than one branch.

For example, if you were working on a large feature it might have many somewhat independent pieces and you might want to work on them in separate branches. Doing so would provide a safety net in case work on one subfeature didn't go well. In which case, you could just ditch that branch and do no harm to your main feature branch. In which case, it makes sense to branch the subfeature branch from the feature branch and name it accordingly.

This is where "/" delimiters come in useful in Git. In our case, we always use an id from our project management system in the branch name to make them easy to find.

So, if we had a feature branch named "ajax_12345" and we wanted to create a subfeature for widgets then we might create "ajax/widgets_12345". That would make the relationship clear to anyone looking.

It is important to note that all branches should end their lifecycle by merging back into their parent. That means that feature branches are deleted after merging back into master and subfeature branches are deleted after merging back into the feature branch from whence they came.

Conclusion

This branching strategy, which could maybe be called "Git Web Flow", has worked very well for us for a few years now. It allows us to work on several different features at any given time and have multiple features under review at any given time while still retaining the ability to take any given feature live independent of any other feature.