GIT Branching and Merging

> The GIT branching model we have chosen for FreePBX is modeled directly after Vincent Driessen's git "flow" branching model. The documentation provided here on out is a modified form of his blog at http://nvie.com/posts/a-successful-git-branching-model/.

Overview

FreePBX has a long standing history of using branches for releases. In recent years this has started to cause some growing pains which have started limiting features that we can introduce into the current stable branch, while we actively work on the next stable release. This also means that end users have to wait anywhere from 6 months to a year before they can play and test new features, since they aren't available on their existing release

The Plan

We plan to address this by moving FreePBX away from the traditional branch release system it has maintained for so long, to a new supported release system. This system will allow modules to specify what frameworks they support, and allow us from a project standpoint to pack modules more dynamically. This will also reduce the amount of version bumping we need to do to each module every time we release a new framework version, allowing us to focus more on the actual modules that require attention than those that don't.

What's a module's naming convention

Under the current project structure, all modules are grouped together in subversion under /modules/branches/[branch number]. However we don't see this as maintainable long term, as we want to take advantage of git and the features it has to offer. To do this each module has been broken up into it's own git repository. This allows for us to have history on a per module basis, making information quicker and easier to find. Each module repository, will bear the modules raw name as the repository name in git. Each module can have as many branches as needed (currently we have release branches based off the svn branch), however we are trying to maintain git-flow, which is a git branching model (more info at http://nvie.com/posts/a-successful-git-branching-model/) for maintainability.

Everything is now a module. There is no longer a special case for framework and fw_ari.

What is a branch

A branch is no longer necessarily a release, as each module should at bare minimum have the following branches:

- master
  - latest stable code, also includes the latest localization files from Weblate
- feature/*
  - used to develop new features for the upcoming or a distant future release.
- hotfix/*
  - Hotfix branches are very much like release branches in that they are also meant to prepare for a new production release, albeit unplanned. They arise from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.
- bugfix/*
  - Used to fix bugs against tickets
- release/*

- Release branches support preparation of a new production release.

How does a release happen?

Previously a release, used to package up a given svn commit and store the tarball in a release directory in subversion. With git, however, there is no reason to store tarballs. Instead the proposed new release cycle will be to do the following:

1. Checkout module (it's the developer's responsibility to be at the right release)
2. Write code
3. Decide if it's release ready……after a lot of testing, right?
4. Make changes to module.xml (can also be done with options in package.php)
5. run ./package.php -m [modulename] -r [releaseversion]
    1. TODO: get release from user and verify it matches the release they are on. We can handle failures either interactively or just fail.

When running package.php, we are:

1. checking if the module.xml is valid
2. that there are no PHP syntax errors
3. Check for rawname, supported tag, and version and fail if they aren't there
4. TODO (client side and server side): Check for legal tag format and repository names
5. TODO: Get file exclude list working in client side package.php, from separate file in module
6. creating a git tag with the version specified in the module.xml file

The tag in git will look like:

- release/2.11.0.2
- release/2.11.0.1beta2

What about repositories

To do this we will be doing away with the old repo concept, in which repos were hardcoded into FreePBX. This concept was flawed and doesn't offer expansion. Instead we will be using the repo tag to automatically display the repositories available to users based on what the module.xml has in it when fetched via module_admin.

How we do we beta/alpha

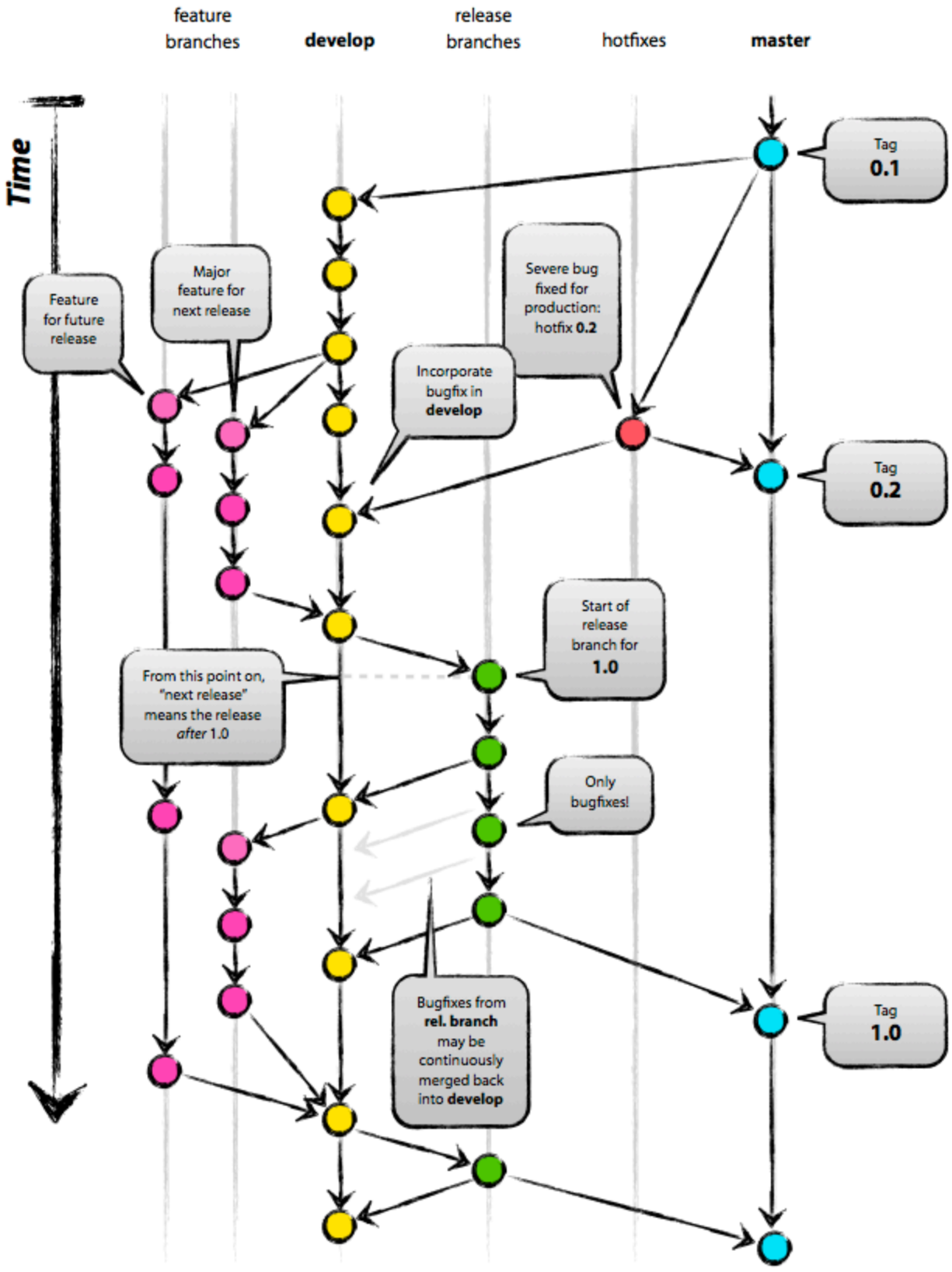
Beta is a new tag we are introducing through module.xml with two values, 0 or 1. If 1 or true, we will given users the option to install either the stable or beta release but default to stable.

TODO: Client side needs to display this. If customer is already on beta, default to beta not stable unless only stable is presented.

How do we differentiate between php version and components in tags

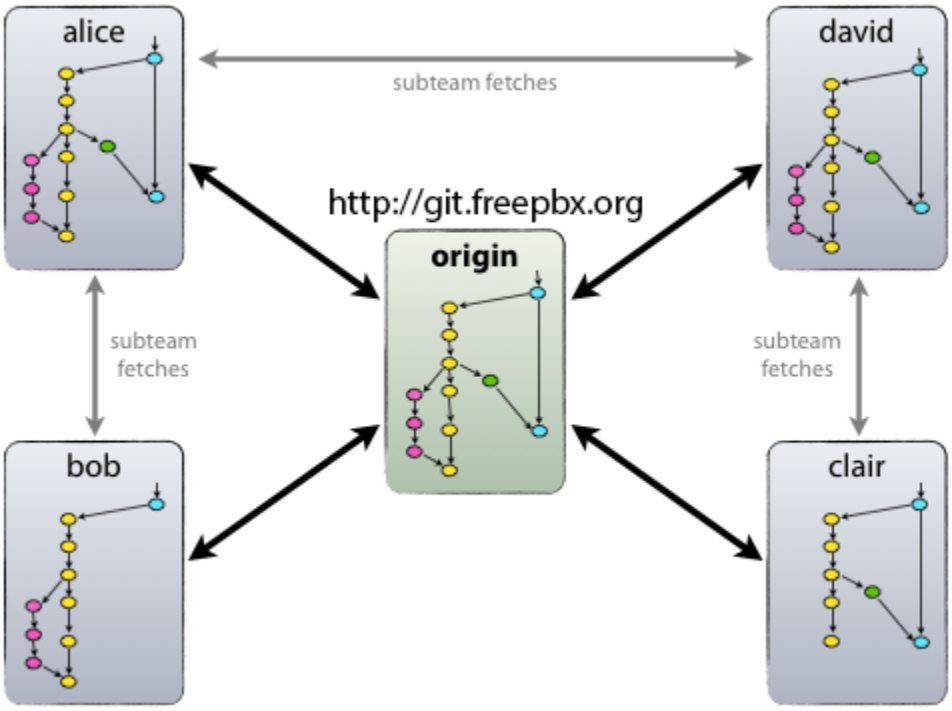
We aren't going to do anything component wise on the server side at this time. We currently get the PHP version from the client side, allowing us to lookup modules based on the PHP version they were published for (at this time this is mostly for Zended modules).

The Why



The repository setup that we use and that works well with this branching model, is that with a central “truth” repo. The central "truth" repo for FreePBX.org is http://git.freepbx.org.

The repo at http://git.freepbx.org is considered to be the central one (since Git is aDVCS, there is no such thing as a central repo at a technical level). We will refer to this repo as origin, since this name is familiar to all Git users.



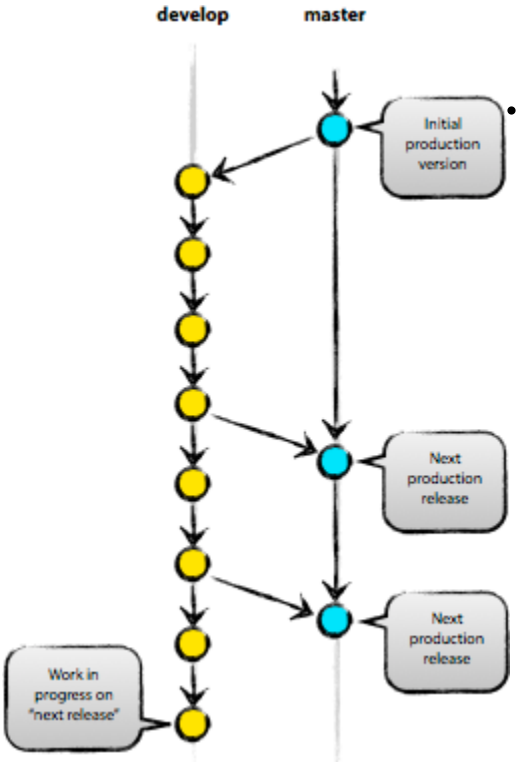
Each developer pulls and pushes to origin. But besides the centralized push-pull relationships, each developer may also pull changes from other peers to form sub teams. For example, this might be useful to work together with two or more developers on a big new feature, before pushing the work in progress to origin prematurely. In the figure above, there are subteams of Alice and Bob, Alice and David, and Clair and David.

Technically, this means nothing more than that Alice has defined a Git remote, named bob, pointing to Bob's repository, and vice versa.

Such terminology can also be used when dealing with github, we normally refer to github as one of our remotes. Everything from git.freepbx.org mirrors directly to github but pull requests are managed through git.freepbx.org by assigning a remote to the github remote that wants to do said pull request and applying it on git.freepbx.org first, which then replicates back to github. This way we are applying patches and fixes directly to our "truth" repo.

The How

The main branches



At the core, the development model is greatly inspired by existing models out there. The central repo holds one main branch with an infinite lifetime:

- master

The master branch at origin should be familiar to every Git user. Parallel to the master branch, another branch exists called develop.

We consider origin/master to be the main branch where the source code of HEAD always reflects a production-ready state. In our case it is usually the STABLE branch code and thus is reflected back into weblate.

Example. If there are currently three active releases of FreePBX: 13,14,15, at any time any one of those branches might be mirrored into Master.

This means the code base in master will directly match the code base of one of the release branches (talked about later in this article)

Therefore, each time when changes are merged back into master, this is a new production release by definition.

Supporting branches

Next to the main branches master and develop, our development model uses a variety of supporting branches to aid parallel development between team members, ease tracking of features, prepare for production releases and to assist in quickly fixing live production problems. Unlike the main branches, these branches always have a limited life time, since they will be removed eventually.

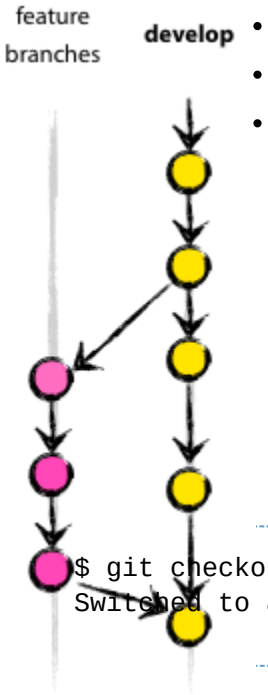
The different types of branches we may use are:

- Feature branches
- Release branches
- Hotfix branches

Each of these branches have a specific purpose and are bound to strict rules as to which branches may be their originating branch and which branches must be their merge targets. We will walk through them in a minute.

By no means are these branches “special” from a technical perspective. The branch types are categorized by how we use them. They are of course plain old Git branches.

Feature branches



- May branch off from: *release/**
- Must merge back into: *release/**
- Branch naming convention: *feature/**

Feature branches (or sometimes called topic branches) are used to develop new features for the upcoming or a distant future release. When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at that point. The essence of a feature branch is that it exists as long as the feature is in development, but will eventually be merged back into develop (to definitely add the new feature to the upcoming release) or discarded (in case of a disappointing experiment).

Feature branches typically exist in developer repos only, not in origin. Feature branches are also never merged back into master or a release/* branches as features are not usually back ported to older versions of FreePBX unless explicitly noted

Creating a feature branch

When starting work on a new feature, branch off from the release branch you want to start from

```
$ git checkout -b myfeature release/13.0
Switched to a new branch "myfeature"
```

Release branches

May branch off from: *release/**
Must merge back into: *master*
Branch naming convention: *release/**

Release branches support preparation of a new production release. They allow for last-minute dotting of i’s and crossing t’s. Furthermore, they allow for minor bug fixes and preparing meta-data for a release (version number, build dates, etc.). By doing all of this work on a release branch, the develop branch is cleared to receive features for the next big release.

The key moment to branch off a new release branch from develop is when develop (almost) reflects the desired state of the new release. At least all features that are targeted for the release-to-be-built must be merged in to develop at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

It is exactly at the start of a release branch that the upcoming release gets assigned a version number—not any earlier. Up until that moment, the develop branch reflected changes for the “next release”, but it is unclear whether that “next release” will eventually become 0.3 or 1.0, until the release branch is started. That decision is made on the start of the release branch and is carried out by the project’s rules on version number bumping.

Creating a release branch

Release branches are created from the develop branch. For example, say version 1.1.5 is the current production release and we have a big release coming up. The state of develop is ready for the “next release” and we have decided that this will become version 1.2 (rather than 1.1.6 or 2.0). So we branch off and give the release branch a name reflecting the new version number:

```
$ git checkout -b release/14.0 release/13.0
Switched to a new branch "release/13.0"
```

This new branch may exist there for a while, until the release may be rolled out definitely. During that time, bug fixes may be applied in this branch (rather than on the develop branch). Adding large new features here is strictly prohibited. They must be merged into develop, and therefore, wait for the next big release.

Finishing a release branch

When the state of the release branch is ready to become a real release, some actions need to be carried out. First, the release branch is merged into master (since every commit on master is a new release by definition, remember). Next, that commit on master must be tagged for easy future reference to this historical version. Finally, the changes made on the release branch need to be merged back into develop, so that future releases also contain these bug fixes.

The first two steps in Git:

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff release/1.2
Merge made by recursive. (Summary of changes)
$ git tag -a release/1.2 -m 'release/1.2'
```

The release is now done, and tagged for future reference.

To keep the changes made in the release branch, we need to merge those back into develop, though. In Git:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff release/1.2
Merge made by recursive. (Summary of changes)
```

This step may well lead to a merge conflict (probably even, since we have changed the version number). If so, fix it and commit.

Unlike the normal GIT flow methods we keep release branches around indefinitely so that we can apply bug and security fixes if need be.

Hotfix branches

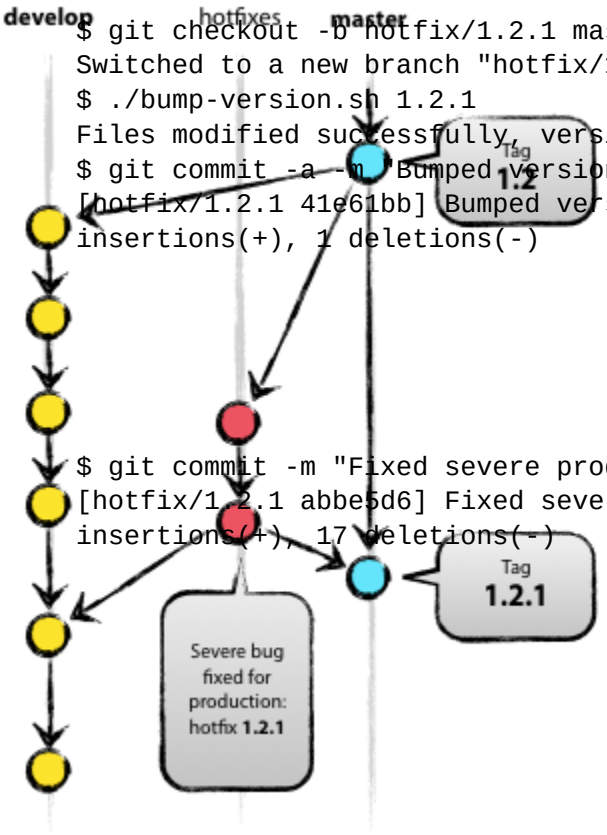
May branch off from: *master, release/**
Must merge back into: *master* and *release/**
Branch naming convention: *hotfix/**

Hotfix branches are very much like release branches in that they are also meant to prepare for a new production release, albeit unplanned. They arise from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

The essence is that work of team members (on the develop branch) can continue, while another person is preparing a quick production fix.

Creating the hotfix branch

Hotfix branches are created from the master branch. For example, say version 1.2 is the current production release running live and causing troubles due to a severe bug. But changes on develop are yet unstable. We may then branch off a hotfix branch and start fixing the problem:




```
$ git checkout -b hotfix/1.2.1 master
Switched to a new branch "hotfix/1.2.1"
$ ./bump-version.sh 1.2.1
Files modified successfully, version bumped to 1.2.1.
$ git commit -a -m "Bumped version number to 1.2.1"
[hotfix/1.2.1 41e61bb] Bumped version number to 1.2.1 1 files changed, 1
insertions(+), 1 deletions(-)
```

Don't forget to bump the version number after branching off!

Then, fix the bug and commit the fix in one or more separate commits.

```
$ git commit -m "Fixed severe production problem"
[hotfix/1.2.1 abbe5d6] Fixed severe production problem 5 files changed, 32
insertions(+), 17 deletions(-)
```

Finishing a hotfix branch

When finished, the bugfix needs to be merged back into master, but also needs to be merged back into develop, in order to safeguard that the bugfix is included in the next release as well. This is completely similar to how release branches are finished.

First, update master and tag the release.

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff hotfix/1.2.1
Merge made by recursive. (Summary of changes)
$ git tag -a release/1.2.1 -m 'release/1.2.1'
```

Next, include the bugfix in develop, too:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff hotfix/1.2.1
Merge made by recursive. (Summary of changes)
```

The one exception to the rule here is that, **when a release branch currently exists, the hotfix changes need to be merged into that release branch, instead of** develop. Back-merging the bugfix into the release branch will eventually result in the bugfix being merged into develop too, when the release branch is finished. (If work in develop immediately requires this bugfix and cannot wait for the release branch to be finished, you may safely merge the bugfix into develop now already as well.)

Finally, remove the temporary branch:

```
$ git branch -d hotfix/1.2.1
Deleted branch hotfix/1.2.1 (was abbe5d6).
```