Git Branching Strategy: Version Control & Release Management

The purpose of this strategy is to have unified development practice across different team members for maintaining **version control** and **release management** of their respective codebases.

Version Control

Any product or service has a version, representing the released state of that particular product/service. In git based version controlling, it's encouraged to use tags tied with semantic versioning to keep track of different versions for your product/service.

Defining Versions:

Naming: v<major>.<minor>.<patch>

Major: A breaking release, which may or may not be backward compatible, and require certain upgrade or changes to happen on consumer side of your service to continue the usage.

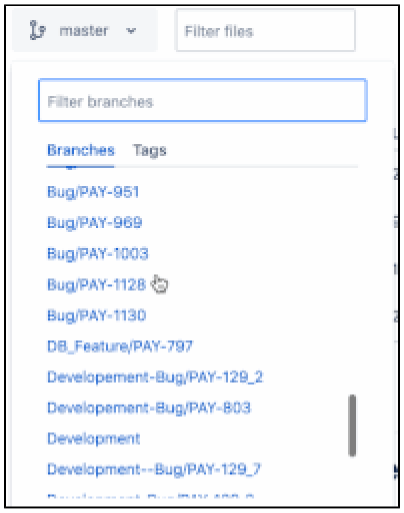
Minor: A non-breaking release, which has new features and improvements. This does not require consumers to do any upgrade on their side. If they consume new feature, that's an addition on their side, but older features should be working fine as well.

Patch: A bug fix, performance improvement or any subtle changes; which is not affecting any existing interface or interaction with consumers in any ways.

Example: v1.0.1, v2.0.0

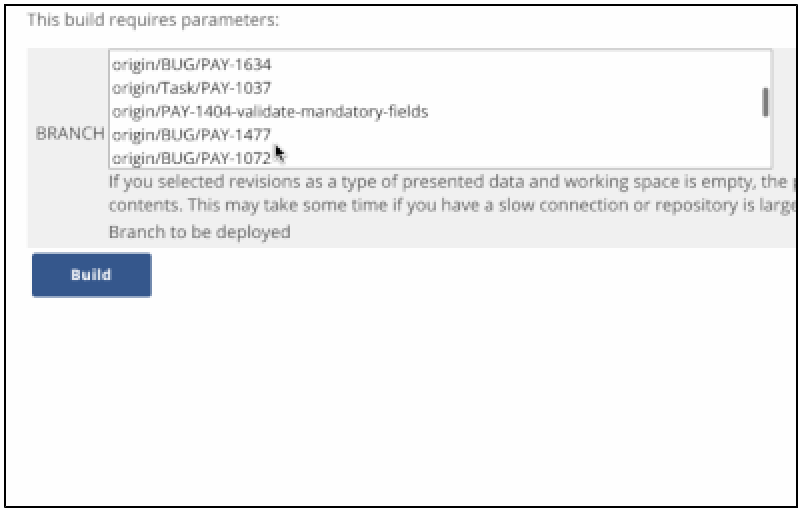
Forking

Before you start development on a certain codebase, you should create a fork of the main repository. The reason is with a big enough codebase where pull requests are created in an enormous volume everyday, the main codebase will get polluted with so many branches from different devs created over the time for raising PRs.



Bug/PAY-951
Bug/PAY-969
Bug/PAY-1003
Bug/PAY-1128
Bug/PAY-1130
DB_Feature/PAY-797
Developement-Bug/PAY-129_2
Developement-Bug/PAY-803
Development
Development--Bug/PAY-129_7

It's also becomes extremely difficult to give a build from specific branch, let's say in jenkins or any other param based build environment-


This build requires parameters:

BRANCH
origin/BUG/PAY-1634
origin/Task/PAY-1037
origin/PAY-1404-validate-mandatory-fields
origin/BUG/PAY-1477
origin/BUG/PAY-1072

If you selected revisions as a type of presented data and working space is empty, the
contents. This may take some time if you have a slow connection or repository is large
Branch to be deployed

So, the recommendation towards proper version controlling using Git is having your own forked repository. It also has other benefits as listed here .

Other Recommendations:

If you have created a fork, then cloned it into your local, it will have the name `origin` as remote, hence add the base repository remote url as well which should be named as `upstream`. And then rename your origin as your name, since that's your fork. So, your remote urls should be like below-

fork-origin-upstream

```
$ git remote -v
dibosh git@bitbucket.org:dibosh/apg-transaction-api.git (fetch) // my
forked repo
dibosh git@bitbucket.org:dibosh/apg-transaction-api.git (push)
upstream git@bitbucket.org:DigitalPlumbing/apg-transaction-api.git
(fetch) // base repo
upstream git@bitbucket.org:DigitalPlumbing/apg-transaction-api.git
(push)
```

This is mostly required when you are a **merger**, and hence you need to fetch other's branch in your local- test it, merge it and then push back to main repo.

Main Branches

There are 2 main branches with infinite lifetime:

- `master`
- `develop`

The `master` branch is stable branch, which is the perfect reflection of your production deployed code.

The `develop` branch is essentially deviated off from master, containing the latest development changes which got merged from all the team members.

When the codes in the `develop` reaches a stable point and ready to be released, a new kind of temporary branches are created called `release` branches.

Release branches are merged to master and development

Supporting branches

The supporting branches has a limited life time, since they will be removed once they have got merged into their destination.

There supporting branch types can be:

- Feature branches
- Hot fix branches
- Bug fix branches
- Release branches

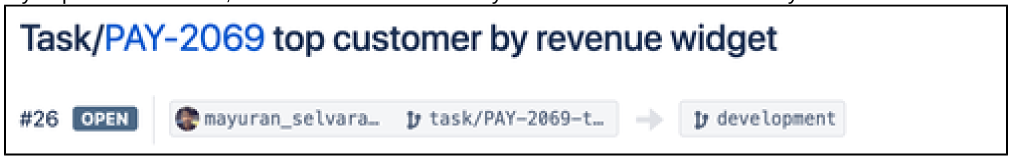
Each of these branches have a specific purpose and are bound to strict rules based on their source and destination.

Branch Naming Convention:

The branch name should have format like- **<type-prefix>/<Jira ticket number>-optional-work-slug** Example- **feature/PAY-1891**, **bugfix/PAY-1001-fix-requery-missed, task/PAY-1986**

Rationale

- The type prefix helps you to group your branches by same type in Bitbucket branch view or even in any git gui tool and obviously in your terminal with a simple grep
- If you put ticket number, then Bitbucket automatically attaches the Jira ticket link to your PR name as shown below



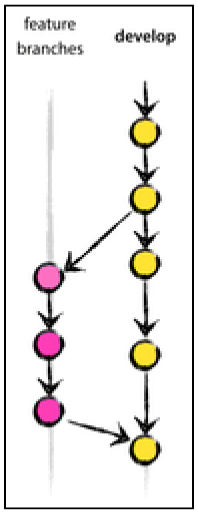
Task/PAY-2069 top customer by revenue widget

#26 OPEN mayuran_selvara… task/PAY-2069-t… development

- The optional work slug helps you to get a quick overview of what the work is about; like a glance- also helps you to remember which branch was for what when you are working in your local with so many branches at a time

Feature branches

A feature branch is essentially containing a piece of work which can be improvement on top of existing features, changes due to alter in feature requirements or even a new feature itself.



Naming Prefix:

feature/

Should branch off from:

`develop`

Must merge back into:

`develop`

Creating a feature branch

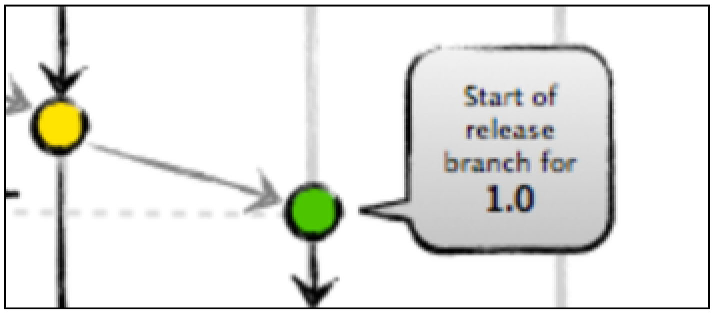
```
$ git checkout -b feature/JR-159 develop
Switched to new branch 'feature/JR-159'
```

Merging feature branch into develop

```
$ git checkout develop
$ git pull // to make sure you have the latest changes
$ git merge --no-ff feature/JR-159 // no-ff flag is important, we want a
merge commit
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d feature/JR-159 // it's a good practice to delete the
feature branch now
Deleted branch feature/JR-159 (was 05e9557).
$ git push upstream develop
```

Release Branches

After certain time, teams plan to make a release based on business requirement or may be they have a certain release cycle. A temporary release branch should be created off of from `develop`. This release branch then is thoroughly tested in the staging environment.



Naming Prefix:

release/

Should branch off from:

`develop`

Must merge back into:

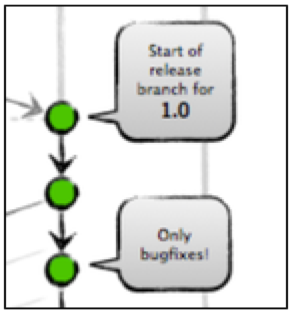
`develop`

Creating a Release Branch

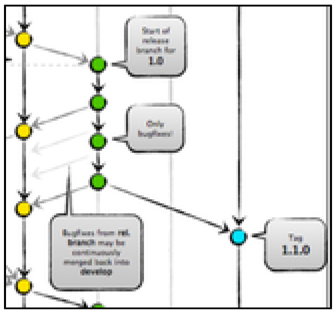
```
$ git checkout -b release/v1.0.1 developer
Switched to new branch 'release/v1.0.1'
```

Finishing release branch

While testing if any bugs comes out, new bugfix branches are created off of the release branch itself and then merged back to it upon resolution.



Once finalised, a release branch gets merged to master and then also to development(to keep it in sync with the changes).


Start of release branch for 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0

```
$ git checkout master
$ git merge --no-ff release/v1.0.1
$ git push upstream master

$ git push dibosh master // this optional step is to keep your forked
repository master updated as well

$ git checkout development
$ git merge --no-ff release/v1.0.1
$ git push upstream development


$ git push dibosh development // this optional step is to keep your
forked repository development updated as well
```

Tagging the Release Version

Once merged, a tag needs to be created for this release. A tag lets you to roll back to previous version very easily, it's also essentially the release history in your git when you do- `git tag -l`

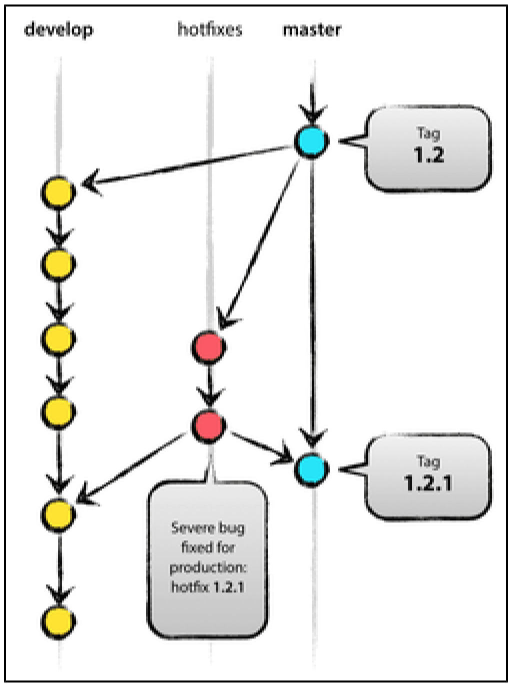
```
$ git checkout master
$ git tag -a v1.0.1 -m 'some message or simply v1.0.1'
$ git push upstream v1.0.1 // push the tag into base repo
```

We can always see the release branch merge commit in a particular tag/version

```
$ git show v1.0.1

Tagger: dibosh <md-abdul_munim@astro.com.my>
Date:   Thu May 9 00:21:00 2019 +0800

v1.0.1

commit 7a33f9c83dca05cc5210fa0a5d0d56b6a1df043e (tag: v4.1.1,
upstream/master, release/v1.0.1, development)
Merge: f966fdd9f 1046a7d22
Author: Rahul Kadam <rahul-dnyaneshwar_kadam@astro.com.my>
Date:   Wed May 8 10:01:51 2019 +0000

    Merged in fix/PAY-2034-adding-logging-paymentRefId-in-postTopartner
(pull request #349)

    PAY-2034 updating log for payment ref id

    Approved-by: Munim Dibosh <md-abdul_munim@astro.com.my>
    Approved-by: Saurabh Das <saurabh.das@blazeclan.com>
```

Hotfix branches

A hotfix branch is always unplanned and obviously created due to some production bug, which is time critical and has greater impacts on end users.



Naming Prefix:

hotfix/

Should branch off from:

`master` (`example scenario: in case of a release which introduced some unexpected side effect`**)**

Must merge back into:

`master` (`as a patch version`**)**

Creating a Hotfix Branch

```
$ git checkout -b hotfix/PAY-1234 master
Switched to a new branch "hotfix/PAY-1234" // like any other normal
work, a hotfix branch must have a related ticket against which it will
be tracked in future
```

Merging a Hotfix Branch

```
$ git checkout master
$ git merge --no-ff hotfix/PAY-1234
```

Releasing a Hotfix Branch

It will always be a patch version increment, hence if the previous version was v1.0.1, it now will be v1.0.2

```
$ git tag -a v1.0.2 -m 'PAY-1234 Fix partner payment not reflected in
topup' // referring the ticket and brief message helps in later
tracking; optional
$ git push upstream v1.0.2
```

CI/CD branching strategy

In continuous integration and deployment we will need at least 3 environments: develop, staging, production

Develop Environment

This environment is solely used by the developers only, and this environment contains nightly builds or incremental updates.

Branch used for this environment is: **`develop`**

Trigger action: **`push`**

Meaning, every time a developer merge something to the develop branch, it will automatically trigger the CI/CD for **develop** environment.

Staging Environment

This environment is shared environment to be used by developers, QA testers, and product owners and even partners who also does their UAT. This environment can be considered as identical copy of production- meaning whatever is tested and confirmed here is supposed to work in production as well.

Branch used for this environment: temporary **release** branches and **master**

- Why?
  A **release** branch is normally deployed into this env to do all the testing to confirm if it's actually releasable or not. In this phase, immediate bugfix and testing iteration keeps going on until the release is confirmed to be stable. Once the release branch gets merged into master, then again to have a sanity check, the master branch gets deployed- and a final round of testing happens just to confirm that release branch has been integrated with master without introducing any regression.

Trigger action: when a branch with **release/** prefix is pushed and also when **master** is tagged

Production Environment

This environment is the final environment used by the end-user, and must be the most stable of them all. As soon as **master** is tagged, a build cycle has already run by staging env and we have that artefact. The deployment to production happens manually- only after getting proper approvals through CM process. Once triggered the deployment job essentially uses the already built artefact in staging and deploys it into server machines.