**7/1/2020, 8/8/2021, 3/15/22**

**CI/CD flowchart** (under constructions)

1. **Code done**
2. **unit test**
3. **Check-in feature branch**
4. **Trigger build and QA test (CI/CD Server)**

* **Jenkins CI server automate build.**
* **The build artifact will be hosted on a server.**
* **Jenkins automate pytest on feature-branch**
* **CI/CD server also hosts as local server as API server, right?**
* **CI build and deploy app on server(different PC from CI PC) with publish with SSH**
* **Pytest is running on CI server. Confirmed.**

1. **pass** 🡪 **merge to Master branch / trunk** (**Need to figure out how to merge, automatically or manually, by who**)
2. **Nightly Build test from Trunk running on staging Server**

**Q: run nightly regression from trunk or feature? If running in trunk, who will merge the feature, and needs code review?**

**The merge is performed by dedicated project maintainer, once QA test pass.**

Each merge will be preferably automatically deployed to a staging environment for internal QA & testing (nightly regression testing)

**Key**: testing and server is located at the same PC; so API server is localhost

* **There are several approaches I have used to solve similar issue**:

1. Check **execution time**, and find all most slowest tests and then **analyze why they take so much time to execute**.

pytest --durations=0 — Show all times for tests and setup and teardown

pytest --durations=1 — Just show me the slowest

pytest --durations=50 — Slowest 50, with times, … etc

1. You have 100 projects, may be you do not need to build and test them every time? Could you run all unittest only at a night builds? **Create several 'fast' build configurations for daily use**. CI server will **perform only limited set of unittests projects related to 'hot' parts of your current development process**.
2. **Mock and isolate everything you could**, avoid disk/network I/O whenever it's possible
3. When it isn't possible to isolate such operations, may be you have integration tests? May be you could **schedule integration tests to night builds only**?
4. Check all occasional singletons, that keep references to instances/resources and that consume memory, this could leads to performance degradation while running all tests.

* I usually set up my CI to run on every single commit. Branches don't get merged into master until the branch has been tested. If you're relying on running tests on master, then that opens a window for the build to be broken.
* **Running the tests on a CI machine is about reproducible results**. Because the CI server has a known clean environment pulled from your VCS, you know that the test results are correct. When running locally, you could forget to commit some code needed for them to pass, or have uncommitted code that makes them pass when they should be failing
* **Comments**

A developer creates the feature in a dedicated branch in their local repo.

The developer **pushes** the branch to a public Bitbucket repository.

The developer files a **pull** request via Bitbucket.

The rest of the team **reviews** the code, discusses it, and alters it.

**Code review before merging to trunk from feature branch?**

**The project maintainer merges the feature into the official repository and closes the pull request.**

* The CI server configuration doesn't usually include all the developer tools and configuration and thus is closer to the production system
* CI systems build the project from scratch every time, meaning builds are repeatable
* A library change could cause problems downstream - a CI server can be configured to build *all* dependent codebases, not just the library one
* running them on a CI environment usually means you set up your project *from* ***scratch***, ensuring your build is **repeatable**
* Also, two changes could be committed that tested okay separately, but break together (e.g. one removing an unused API, and the other starting to use it)

Unit tests should run in-mem and instantaneously. 1 second max for thousands of tests. Otherwise you are doing something wrong (io). They should be enabled to run as a **pre-commit hook**. **Integration tests (or other slow tests) should be enabled on your build server**. **Merges should not be allowed to happen unless the tests pass.**

You can also have your CI server run the unit tests on each **feature branch** as commits are made. This way the developer can make some changes, commit the code, and let the server run the tests in the background while they continue to work on additional changes or other projects.

The majority of Unit Tests should take under 10 milliseconds each or so. Having 'almost a thousand tests' is nothing and should take **maybe** a few seconds to run.

If they're not, then you should stop writing highly coupled integration tests (unless that's what the code needs) and start writing good unit tests (**starting with well decoupled code and proper usage of fakes/mocks/stubs/etc**). That coupling will impact test quality and the time it takes to write them too - so it's not just a matter of reducing test run time.

**1. Make your tests 100% reliable.**

If you ever have tests that fail with false negatives, deal with that right away. Fix them, change them, eliminate them, whatever it takes to guarantee 100% reliability. (It's OK to have a set of unreliable, but still useful tests that you can run separately, but the main body of tests must be reliable.)

**2. Change your systems to guarantee that all tests pass all the time.**

Use continuous integration systems to ensure that only **passing commits get merged in to the main/official/release/whatever branch.**

**3. Make the tests fast and less dependency**

I have worked on projects where tests take a second, and on projects where they take all day. **There is a strong correlation between the time it takes to run tests and my productivity**.

The longer tests take to run, the less often you'll run them. That means you'll go longer without getting feedback on the changes you're making. It also means you'll go longer between commits. Committing more often means smaller steps that are easier to merge; commit history is easier to follow; finding a bug in the history is easier; rolling back is easier, too.

**4. Make tests useful.**

If the tests don't fail when you screw up, then what's the point? Teach yourselves to write tests that will catch the bugs you're likely to create. This is a skill unto itself, and will take lots of attention.

* **You test the code from the repository**. It may work on your machine with your files... that you forgot to commit. It may depend on a new table that does not have the creation script (In liquibase for example), some configuration data or properties files.
* **You avoid code integration problems.** One developer downloads the last version, creates unit and integration test, adds code, pass all test in his machine, commits and push. Another developer has just done the same. Both changes are right on their own but when merged causes a bug. This could be the repository merging or just that it is not detected as a conflict. E.g. Dev 1 deletes file that was not used at all. Dev 2 codes against this file and tests without Dev 1 changes.
* **You develop a script to deploy automatically from the repository.** Having an universal building and deploying script solves a lot of issues. Some developer may have added a lib or compiling option that is not shared by everybody. Not only does this save you time, but more importantly, it makes the deployment safe and predictable. Furthermore, you can go back in your repository to version 2.3.1 and deploy this version with a script that works with this version. It includes database objects like views, stored procedures, views, and triggers that should be versioned. (Or you won't be able to go back to a workable version).
* **Other tests**: Like integration, performance and end to end tests. This can be slow and might include testing tools like Selenium. **You may need a full set of data with a real database** instead of mock objects or HSQL.
* the result is not that important for some special settings - for example, running unit tests to check the code coverage requires special compiling flags. For normal developers, code coverage is not that important - it is more for people taking care that code maintains certain quality, like team leads

It is possible to imagine cases when the change A does not break the test, and change B does not break the test, but A and B *together* do. If A and B are made by different developers, only CI server will detect the new bug. A and B may even be two parts of the same longer sentence

The are several reasons for doing CI, but the main point of CI is to get an idea what the state of the code is over time. The main benefit (out of several) this provides, is that we can find out when the build breaks, figure out what broke it, and then fix it.

If the code is never broken, why do we even use CI? To deliver builds for testing, nightly builds would be good enough.

A build can be useful even if some tests fail. **Not every build is intended to be delivered to the customer.** And a failing test does not always mean that the tested code is broken. I have seen a number of cases where the tests were simpliy too strict and sometimes simply didn't make sense at all

**The *individual developer* verifies by running the UnitTests that the code he wrote really implements the desired behavior and does not break already existing desired behavior.**

**The *integration process* verifies by running the UnitTests that merging different developement paths did not break any desired behavior**.

Seems to me a : "oh no, If I merge into master and someone's else code make my tests to fail, I will have to waste time reviewing why. No, I'm better working solo in my branch, where everything is wonderful and works". Then, the lead dev has to merge branches with tons if changes each. A drama. I missing something here

Firstly, **separate projects for unit tests** are **good**. Keeping the unit test code away from the main code is separation of concerns and all that kind of cool stuff.

Secondly, there is no getting away from the fact that **unit tests, if run as part of the build will slow things down and queuing for builds can happen**. You don't mention what you're using to build (TFS, TeamCity, Jenkins etc) but there are things you can do to mitigate this like multiple queues and priorities. Speak to your build manager.

If there is an urgent build that has to jump the queue, the build manager again should be your first port of call. They may be able to can lower priority builds and push the urgent build higher up the queue.

Finally, you mention CI but don't mention whether it is **gated**. That is: new code is kicked out in its entirety if the build or unit tests fail. This can undoubtedly be contentious as the code can in theory work, but some conditions may cause the tests to fail.

There is alas, no silver bullet here. When I was a build manager, I often encountered huge resistance when code got kicked out - especially if a deadline was looming. How often I heard the refrain: "TFS keeps crashing. Just turn the setting off and let me check the code in!". Without exception the developer was at fault. They'd either forgotten to include a file in the check in or simply failed to update the test when the code changed as it would have taken too long.

**Tests should go in a separate project. This is standard practice**

**mySolution**

**myProject**

**myProject.Tests**

Our developers are not allowed to commit to master. **They push to a feature branch, the CI server then merges with master and runs tests.** **If they succeed, *then* changes are merged to master.** So code with broken tests **cannot** be on master.

Very good workflow indeed. master should always be sane, and **preferably automatically deployed on each merge to a staging environment for internal QA & testing**

1. Attempt to determine exactly what the failing tests are trying to validate.
2. Triage - if some tests are trying to test unimportant things like (an old) state of the world, delete them. If you realize that some of them are trying to validate something important, try to determine if those tests are doing so correctly. If they're testing incorrectly, make them test correctly.
3. Fix whatever is wrong with your production code now that you have good tests.

**workflow**:

* code something
* run local tests
* **check in to working branch**
* (optional) build server builds ant tests
* **QA/Business testing**
* bugfixes (loop back to top)
* **merge to deploy branch**
* deploy

We have automated acceptance tests on the same feature branch. When you make a **release candidate, it includes the automated tests** you ran to see if the feature passes. You also test the release candidate. When everything passes, **you promote it then by merging to maste**r.

However, when the point arrives where everything *appears* to be done and tested - we call it a release and *then* it is merged with trunk. Essentially, QA can certify the RC by taking a fresh check out on the HEAD of the branch and if he/she Okey's it, the same is merge back with trunk.

So essentially we use the concept of task-branches or private-branches so people are free to make check-ins as much as they need. At the same time, trunk is relatively *free* from any *broken* check-ins.

# Clone the git repo:

**git clone** https://remote-git-repo-url/demoproject

# **Create new dev branch**, do your work, and commit changes locally

git **checkout -b** dev

vi index.html

git commit -m "Made the change.." index.html

# **Push your changes to remote dev branch**

git **push** --set-upstream **origin dev**

# Merge dev branch to master

**git checkout** master

git merge dev

# Finally, **delete dev branch** both locally and remote

git branch -d dev

git branch -d -r origin/dev

For explanation and example output of the above steps, read the rest of the article.

1. Create New Dev Branch

In the following example, I’ve cloned demoproject from remote git repository to **work on it locally**.

# git clone https://remote-git-repo-url/demoproject

By default, **the current working branch is master branch**.

# **git status**

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean

The following commands displays all available branches for this repository (both local and remote).

# **git branch -a**

\* master

remotes/origin/HEAD -> origin/master

remotes/origin/master

As you see from the above output, there is no additional **local or remote** branch except the master branch.

For more details on git branch command, refer to this: [15 Git Branch Command Examples to Create and Manage Branches](https://www.thegeekstuff.com/2017/06/git-branch/)

So, to do our development work, let us **create a new local dev branch** as shown below.

# **git checkout -b dev**

Create a new branch 'dev' and then Switched to it

In the above:

* git checkout command will try to checkout the given branch.
* In our case, we don’t have a branch called “dev”. So, the above command will create a new “dev” branch.
* Once the **empty dev branch** is created, it will also switch to the dev branch and make that as our working branch.

Verify that the new branch got created as shown below.

# git branch -a

**\*** dev

master

remotes/origin/HEAD -> origin/master

remotes/origin/master

As you see from the above output, the **\* i**s now in front of dev, which indicates the current working branch is dev.

The following git status command indicates that we are currently on the new “dev” branch.

# git status

On branch dev

nothing to commit, working tree clean

2. Work on your local Dev Branch

Now that we have a new “dev” branch, start making your changes here. Any change that you do from now on will be only on the “dev” branch.

In this example, let us make a change to index.html file.

# vim index.html

**Commit the change to the dev branch**. Since we are already inside the dev branch, any commit that we do will happen only on the dev branch.

# git commit -a -m "Fixed email address"

[dev b0147e6] Fixed email address

1 file changed, 1 insertion(+), 1 deletion(-)

It’s important to understand that when we initially created our branch, **it exists only locally on our local laptop**. It’s better to push our local “dev” branch to the **remote git repository**.

**This way, you can push all your changes to the remote dev branch, and someone else who is working on the “dev” branch can then checkout the changes**. Even if you are the only person who is working on the “dev” branch, it is still a good idea to push your local changes to remote git repo to keep a remote backup of your changes.

When you do a git push at this stage, it will give the following error message, as we don’t have the “dev” branch in the remote git repository.

# git push

fatal: The current branch dev has no upstream branch.

3. Push the dev Branch to Remote Git Repository Upstream

The following git push command will create the remote “dev” branch if it doesn’t exists, and push all your local “dev” branch changes to the remote “dev” branch.

# git push **--set-upstream** origin dev

Counting objects: 3, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 307 bytes | 307.00 KiB/s, done.

Total 3 (delta 2), reused 0 (delta 0)

To https://remote-git-repo-url/demoproject

\* [new branch] dev -> dev

Branch dev set up to track remote branch dev from origin.

Now if you do a git branch -a, you'll see both local and remote "dev" branch.

# git branch -a

\* dev

master

remotes/origin/HEAD -> origin/master

remotes/origin/dev

remotes/origin/master

4. **Merge Dev Branch to Master Branch**

Once you are done with your development work on the “dev” branch, and validated your changes, you may want to merge the changes to the master branch.

For this, first **switch your working branch from dev to master** as shown below.

# **git checkout master**

Switched to branch 'master'

Your branch is up-to-date with 'origin/master'.

Once you are in the master branch, execute git merge dev as shown below to merge the “dev” branch with master.

# **git merge dev**

Updating 03c769c..b0147e6

**Fast-forward**

index.txt | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

You’ll see the word “Fast-forward” in the above output. All git does in the master branch is to **move the HEAD pointer in the master branch to the latest commit from the “dev” branch**.

Once the merge is done, make sure to do a git push, to push your changes to the remote repository.

# **git push**

Total 0 (delta 0), reused 0 (delta 0)

To https://remote-git-repo-url/demoproject

03c769c..b0147e6 master -> master

5. Cleanup After Merging to Master

Now that you’e completed your development work, and merged the changes to master branch, you may want to delete your “dev” branch.

To delete the dev branch, execute the following command.

# git branch -d dev

Deleted branch dev (was b0147e6).

The above command deletes the branch only on your local git repository in your laptop.

As you see from the following output, the “dev” branch is still in the remote git repository.

# git branch -a

\* master

remotes/origin/HEAD -> origin/master

remotes/origin/dev

remotes/origin/master

The following command will delete the remote “dev” branch.

# git branch -d -r origin/dev

Deleted remote-tracking branch origin/dev (was b0147e6).

After the above clean-up, now we just have only the master branch which includes all the development work that we recently did.

# git branch -a

\* master

remotes/origin/HEAD -> origin/master

remotes/origin/master

**Creating local branches based on remote-tracking branches**

If you want to create a local branch based on a remote-tracking branch (i.e. in order to actually work on it) you can do that with *git branch –track* or *git checkout –track -b*, which is similar but it also switches your working tree to the newly created local branch. For example, if you see in *git branch -r* that there’s a remote-tracking branch called origin/refactored that you want, you would use the command:

git checkout --track -b refactored origin/refactored

In this example “refactored” is the name of the new branch and “origin/refactored” is the name of existing remote-tracking branch to base it on. (In recent versions of git the “–track” option is actually unnecessary since it’s implied when the final parameter is a remote-tracking branch, as in this example.)

The “–track” option sets up some configuration variables that associate the local branch with the remote-tracking branch. These are useful chiefly for two things:

* They allow *git pull* to know what to merge after fetching new remote-tracking branches.
* If you do *git checkout* to a local branch which has been set up in this way, it will give you a helpful message such as:

Your branch and the tracked remote branch 'origin/master'

have diverged, and respectively have 3 and 384 different

commit(s) each.

… or:

Your branch is behind the tracked remote branch

'origin/master' by 3 commits, and can be fast-forwarded.

The configuration variables that allow this are called “branch.<local-branch-name>.merge” and “branch.<local-branch-name>.remote”, but you probably don’t need to worry about them.

You have probably noticed that after cloning from an established remote repository *git branch -r* lists many remote-tracking branches, but you only have one local branch. In that case, a variation of the command above is what you need to set up local branches that track those remote-tracking branches.

*You might care to note some*[*confusing terminology*](http://longair.net/blog/2012/05/07/the-most-confusing-git-terminology/)*here: the word “track” in “–track” means tracking of a remote-tracking branch by a local branch, whereas in “remote-tracking branch” it means the tracking of a branch in a remote repository by the remote-tracking branch. Somewhat confusing…*

Now, let’s look at an example of how to update from a remote repository, and then how to push changes to a new repository.

**Updating from a Remote Repository**

So, if I want get changes from the remote repository called “origin” into my local repository I’ll type *git fetch origin* and you might see some output like this:

remote: Counting objects: 382, done.

remote: Compressing objects: 100% (203/203), done.

remote: Total 278 (delta 177), reused 103 (delta 59)

Receiving objects: 100% (278/278), 4.89 MiB | 539 KiB/s, done.

Resolving deltas: 100% (177/177), completed with 40 local objects.

From ssh://longair@pacific.mpi-cbg.de/srv/git/fiji

3036acc..9eb5e40 debian-release-20081030 -> origin/debian-release-20081030

\* [new branch] debian-release-20081112 -> origin/debian-release-20081112

\* [new branch] debian-release-20081112.1 -> origin/debian-release-20081112.1

3d619e7..6260626 master -> origin/master

The most important bits here are the lines like these:

3036acc..9eb5e40 debian-release-20081030 -> origin/debian-release-20081030

\* [new branch] debian-release-20081112 -> origin/debian-release-20081112

The first line of these two shows that your remote-tracking branch origin/debian-release-20081030 has been advanced from the commit 3036acc to 9eb5e40. The bit before the arrow is the name of the branch in the remote repository. The second line similarly show that since we last did this, a new remote tracking branch has been created. (*git fetch* may also fetch new tags if they have appeared in the remote repository.)

The lines before those are *git fetch* working out exactly which objects it will need to download to our local repository’s pool of objects, so that they will be available locally for anything we want to do with these updated branches and tags.

*git fetch* doesn’t touch your working tree at all, so gives you a little breathing space to decide what you want to do next. To actually bring the changes from the remote branch into your working tree, you have to do a *git merge*. So, for instance, if I’m working on “master” (after a *git checkout master*) then I can merge in the changes that we’ve just got from origin with:

git merge origin/master

(This might be a fast-forward, if you haven’t created any new commits that aren’t on master in the remote repository, or it might be a more complicated merge.)

If instead you just wanted to see what the differences are between your branch and the remote one, you could do that with:

git diff master origin/master

This is the nice point about fetching and merging separately: it gives you the chance to examine what you’ve fetched before deciding what to do next. Also, by doing this separately the distinction between when you should use a local branch name and a remote-tracking branch name becomes clear very quickly.

**Pushing your changes to a remote repository**

How about the other way round? Suppose you’ve made some changes to the branch “experimental” and want to push that to a remote repository called “origin”. This should be as simple as:

git push origin experimental

You might get an error saying that the remote repository can’t fast-forward the branch, which probably means that someone else has pushed different changes to that branch. So, that case you’ll need to fetch and merge their changes before trying the push again.

Aside

*If the branch has a different name in the remote repository (“experiment-by-bob”, say) you’d do this with:*

*git push origin experimental:experiment-by-bob*

*On older versions of git, if “experiment-by-bob” doesn’t already exist, the syntax needs to be:*

*git push origin experimental:refs/heads/experiment-by-bob*

*… to create the remote branch.  However that seems to be no longer the case, at least in git version 1.6.1.2 – see Sitaram’s comment below.*

*If the branch name is the same locally and remotely then it will be created  
automatically without you having to use any special syntax, i.e. you can just do*git push origin experimental*as normal.*

*In practice, however, it’s less confusing if you keep the branch names the same. (The <source-name>:<destination-name> syntax there is known as a “refspec”, about which we’ll say no more here.)*

The problem with rebase is that it corrupts the true history of the commits. This doesn't show up on a smaller repo, but if you have a busy repo, with lots of contributers, untangling a mess becomes much harder if you no longer have the true parentage of a given commit.

Git rebase destroys the context of the commit, leaving basically a diff apply instead of the much more contextually rich merge commit. Yes, your repo looks messier, but it more accurately reflects the lifecycle of the code, and what the developer intended at each commit. If you really want a straight-line for your repos, why aren't you using something like SVN? Is the distributed nature of git really the big selling factor?

I've run teams that have used git how you are describing, always making people use rebase (in this case it was because we were syncing to SVN). Reconstructing what a developer did 4 months ago is much simpler with a merge vs a rebase. You can go to the merge just before the developer's first commit and see what the repo actually looked like when they started work. You can never do that with a rebase. The context in which the developer was working has been lost. Also, with multi-commit branches, you can see what the repo looked like after each commit. With a rebase, those intermittent commits are almost useless, as the repo doesn't look like it did when they made that commit.

To sum up, don't subvert git just because you want to see a straight line in your commit graph. It's not worth it, and it is ultimately a lie.