Dynamics GitHub Branching Strategies

Number of repositories

Currently we have 5 repositories in Github for dynamics

1. **USFCECore-Dynamics**
2. **USFCEProcesses-Dynamics**
3. **USFCECodeComponent-Dynamics**
4. **USFCESecurityRoles**
5. **Dynamics-for-USF**

Git Hub Work flow:

Clone from development branch ---> work in local machine--->deploy to dynamics dev as plugin---> push to github developer ---> peer review-->Push to dynamics Test at plugin-->Push to git hub test---> QA review ---> Po review ---> push to master branch

No.Of Environments :- Dynamics Environments

devsandbox :- Environment where developers play-around experimenting new features. This is not integrated with Mulesoft. Hence integration related features cannot be developed or tested in this environment.

dev :- Environment which developers use for development. This is completely integrated with all the third party systems.

test :- Environment used for testing. This is where all the features completely developed in dev environment gets deployed. Users use this environment for User Acceptance Testing.

prod :- This is production environment.

Currently there are 4 releases annually. Which is a release per quarter. **(Need Mengdi to update this statement)**

No.Of Branches in each repository:-

1. 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-version*.
2. Dev : We consider `origin/dev` to be the main branch where the source code of `HEAD` always reflects a state with the work in progress for the next release.
3. Test : When the source code in the `dev` branch reaches a stable point and is ready to be released, all of the changes should be merged into *test* branch from where the code will be deployed to usftest environment.
4. Release : The source code that is tested in test environment and ready for release to production environment is merged to release branch.
5. Feature : Developers develop a new feature in feature branch and merge it to dev branch for peer review
6. Hotfix : To resolve a bug in production we use Hotfix branch.

Work Flow :-

- Master branch is the branch where we maintain released version or the version which is currently in production.
- From here developers create a dev branch for the first time when they start working with the repository.
- Developers start working in dev branch.

  Feature Branch :- 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`.

- When there is a new feature to be developed, developers branch off a feature branch from dev branch and start developing their feature.
- When the feature is completely ready, developers merge back the feature branch to dev branch.
- Each developer creates their own feature branch branch with branch name as the story name in Dynamics Jira Board.
- Once the feature is developed and merged to dev branch, this needs to be informed to their peer developers, so that they can do the peer review.

Creating a feature Branch :-

- Branch naming convention: anything except master, develop, release-*, or hotfix-*

```
git checkout -b myfeature develop
```

```
Switched to a new branch "myfeature"
```

- Incorporating a finished feature on develop

```
$ git checkout develop
```

```
Switched to branch 'develop'
```

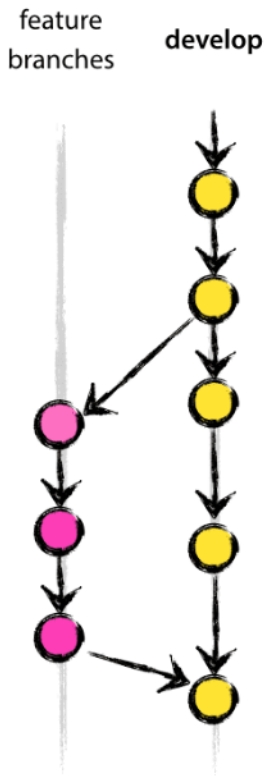
```
$ git merge --no-ff myfeature
```

```
Updating ea1b82a..05e9557 (Summary of changes)
```

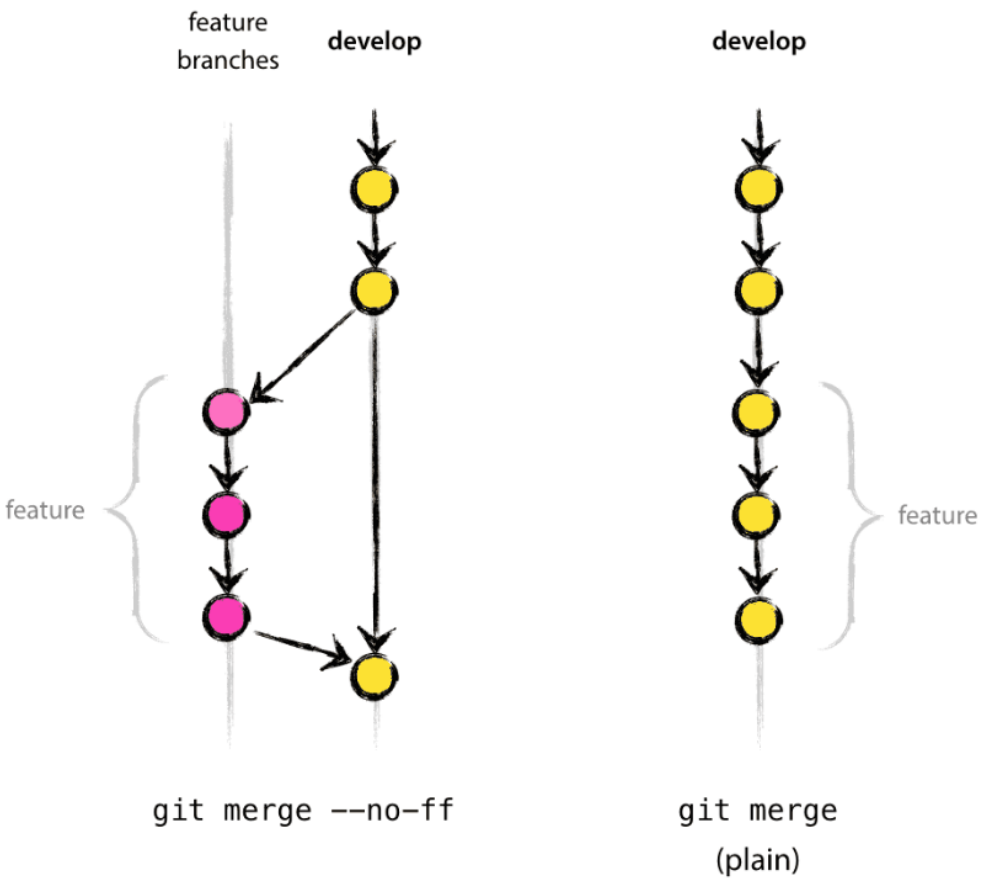
```
$ git branch -d myfeature
```

```
Deleted branch myfeature (was 05e9557).
```

```
$ git push origin develop
```



The `--no-ff` flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature. Compare:



In the latter case, it is impossible to see from the Git history which of the commit objects together have implemented a feature—you would have to manually read all the log messages. Reverting a whole feature (i.e. a group of commits), is a true headache in the latter situation, whereas it is easily done if the `--no-ff` flag was used. Yes, it will create a few more (empty) commit objects, but the gain is much bigger than the cost.

Test Branch :- This branch contains the code that is ready to be deployed to test environment. Developers need to create test branch for the first time branching off from Dev branch. Once all the features are developed, merged to dev branch and peer reviewed in dev branch, test branch needs to be branched off from dev branch for the first time. Then for every development of a feature, once it is peer reviewed, the dev branch needs to be merged to test branch.

- Once the code is merged to test branch from dev branch, Operations team needs to be informed that code that is ready to be deployed to test is merged to test from dev branch.
- Once the operations team is informed, code from test branch is taken as a zip file and deployed to Dynamics Test environment.
- Once the deployment is successful, P.O should be informed that deployment is successful and test environment is ready for P.O review.
- Once the P.O review is successful, the Test branch needs to be merged to release branch by operations person

Release Branch :- 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.

- Operations team, need to create a release branch for the first time by branching off from Test branch.
- Then on every time P.O confirms that code in test environment is tested and working fine, test branch should be merged to release branch.
- Once the release branch is updated with latest code from test branch, this branch should be given a release tag.
- The key moment to branch off a new release branch from `test` is when test(almost) reflects the desired state of the new release.

Creating a Release Branch :-

- Branch naming convention: release-*

git checkout -b release test

```
Switched to a new branch "release"
```

- Finishing a release branch

  $ git checkout test

  Switched to branch 'test'

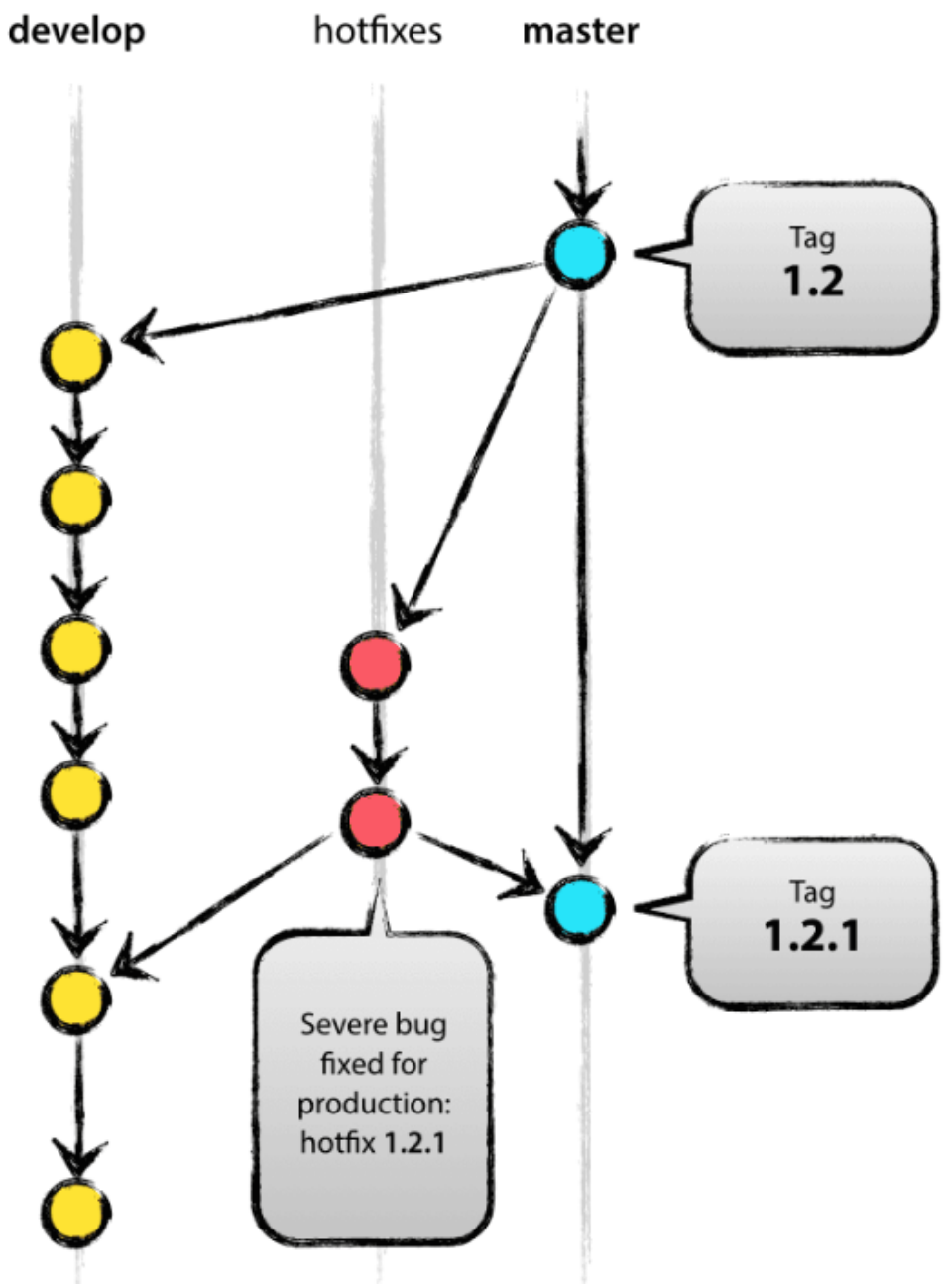
  $ git merge --no-ff release

  Merge Made by recursive (Summary of changes)

  $ git tag -a 1.2

HotFix Branch :- 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.
- Once the bug in production is fixed, the hotfix branch is merged back to master branch that has current production version.
- Once the things work as per expected in production, the hotfix branch is dissolved.



Naming the Tags :- (Discuss with mengdi)

Master Branch:-

Naming tag for release branch format looks lilke Rxxxx.y.z

xxxx : Year of release.

For example the release is happening in 2020 then it is going to be R2020

y : Number that represents the quarter release happened.

For example, the release is happened in Q3 then it is going to be 3

z : Number that increases every time the test branch is merged to release branch. Increases with the merge.

The number starts with 1 for every quarter and increases by 1 every time a new merge happens.

Eg :- R2020.3.1 : Release in 2020, 3rd quarter and 1st Merge from test environment.

R2020.4.5 : Release in 2020, 4th quarter and 5th Merge from test environment.

R2021.1.10 : Release in 2021, 1st quarter and 10th Merge from test environment.

Release Branch:-

Naming tag for release branch format looks like Rxxxx.y.sz.a

xxxx : Year of release.

For example the release is happening in 2020 then it is going to be R2020

y : Number that represents the quarter release happened.

For example, the release is happened in Q3 then it is going to be 3

z : Sprint of the quarter **(Need to confirm with mengdi if every quarter starts with Sprint 1)**

For example, the release is in sprint 4 of the quarter, then it is going to be s4

a : Number that increases every time the test branch is merged to release branch. Increases with the merge.

The number starts with 1 for every quarter and increases by 1 every time a new merge happens.

Eg :- R2020.3.s3.1 : Release in 2020, 3rd quarter, sprint 3 and 1st Merge from test environment.

R2020.4.s1.5 : Release in 2020, 4th quarter, sprint 1 and 5th Merge from test environment.

R2021.1.s6.10 : Release in 2021, 1st quarter, sprint 6 and 10th Merge from test environment.