CONTINUOUS DELIVERY KIT – TRAINING

Software configuration management

2017-2018

V. 4.0

Delivering Transformation. Together.

sopra steria

Software configuration management

*In software engineering, **software configuration management** (**SCM** or **S/W CM**)[1] is the task of tracking and controlling changes in the software, part of the larger cross-disciplinary field of configuration management.[2] SCM practices include revision control and the establishment of baselines. If something goes wrong, SCM can determine what was changed and who changed it. If a configuration is working well, SCM can determine how to replicate it across many hosts.*

*The acronym "SCM" is also expanded as **source configuration management process** and **software change and configuration management**.[3] However, "configuration" is generally understood to cover changes typically made by a system administrator.*

Continuous Delivery Kit

VERSION CONTROL SYSTEM (REVISION CONTROL)

A version control system (also known as a Revision Control System) is a repository of files, often the files for the source code of computer programs, with monitored access.

Every change made to the source is tracked, along with who made the change, why they made it, and references to problems fixed, or enhancements introduced, by the change.

Continuous Delivery Kit

Developed since 2005 from Linux development community (and in particular Linus Torvalds, the creator of Linux).
The goals of Git were:

- ✓ Speed
- ✓ Simple design
- ✓ Strong support for non-linear development (thousands of parallel branches)
- ✓ Fully distributed
- ✓ Able to handle large projects like the Linux kernel efficiently (speed and data size)
- ✓ Linux Kernel 4.9.2: 56,233 files, 22,345,566 lines of code.

CONTINUOUS DELIVERY KIT

WHY GIT?

AS A CONSEQUENCE OF ITS SIMPLICITY AND REPETITIVE NATURE, BRANCHING AND MERGING ARE NO LONGER SOMETHING TO BE AFRAID OF. VERSION CONTROL TOOLS ARE SUPPOSED TO ASSIST IN BRANCHING/MERGING MORE THAN ANYTHING ELSE.

Continuous Delivery Kit

GIT

- ✓ Install and config

CONTINUOUS DELIVERY KIT
GIT – INSTALL

- https://git-scm.com/;
- PenDrive/software/git;

- Open Git Bash and verify it:

```
$ git --version
git version 2.11.0.windows.1
```

CONTINUOUS DELIVERY KIT
GIT – CONFIG

- Configuration level:
  - **system**: for writing options to system-wide;
  - **global**: for writing options to o global level (user level);
  - **local**: for writing options to the repository level;

```
$ git config --global user.name "Mario Rossi"
```

```
$ git config --global user.email "mario.rossi@email.com"
```

```
$ git config --global --list
user.name=Mario Rossi
user.email=mario.rossi@email.com
```

Continuous Delivery Kit

GITFLOW WORKFLOW

- ✓ Repository
- ✓ Tree
- ✓ How it works
- ✓ Let’s see a simple example

CONTINUOUS DELIVERY KIT
REPOSITORY

- Local Repository;
- Remote Repository;

How create a new Local Repository:

- Create a new directory, open it and perform a:

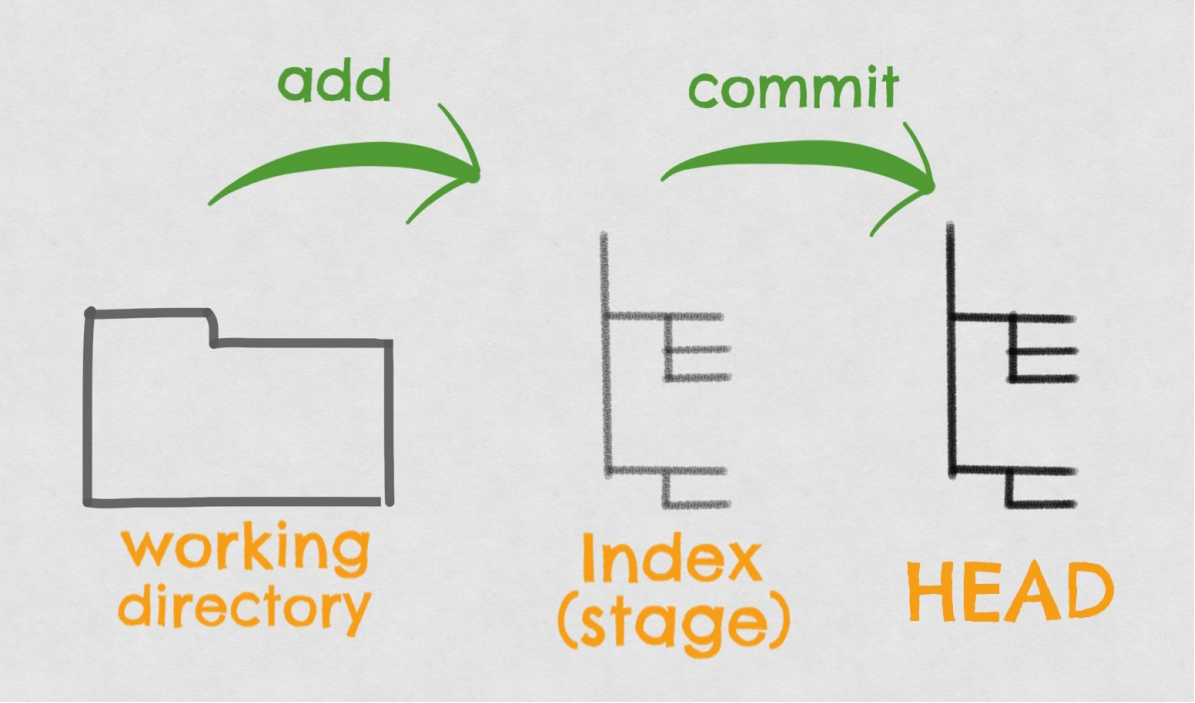
  mkdir dir_repo && cd dir_repo;
  git init

- Clone from remote repository by perform a:

  git clone username@host:/path/to/repository

CONTINUOUS DELIVERY KIT
TREE

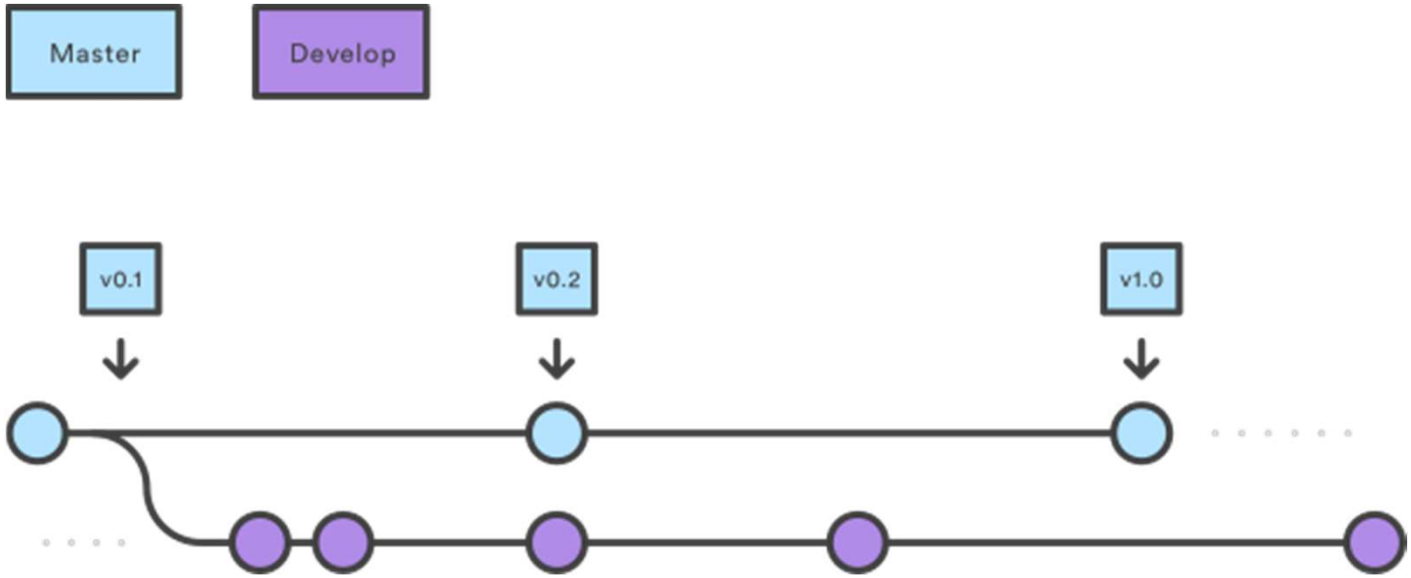
- Local repository consists of three "trees" maintained by GIT:
  - **Working Directory:** which holds the actual files;
  - **Index:** which acts as a staging area;
  - **HEAD:** which points to the last commit made.



CONTINUOUS DELIVERY KIT

HOW IT WORKS - HISTORICAL BRANCHES

This workflow uses two branches to record the history of the project. The master branch stores the official release history, and the develop branch serves as an integration branch for features. It's also convenient to tag all commits in the master branch with a version number.

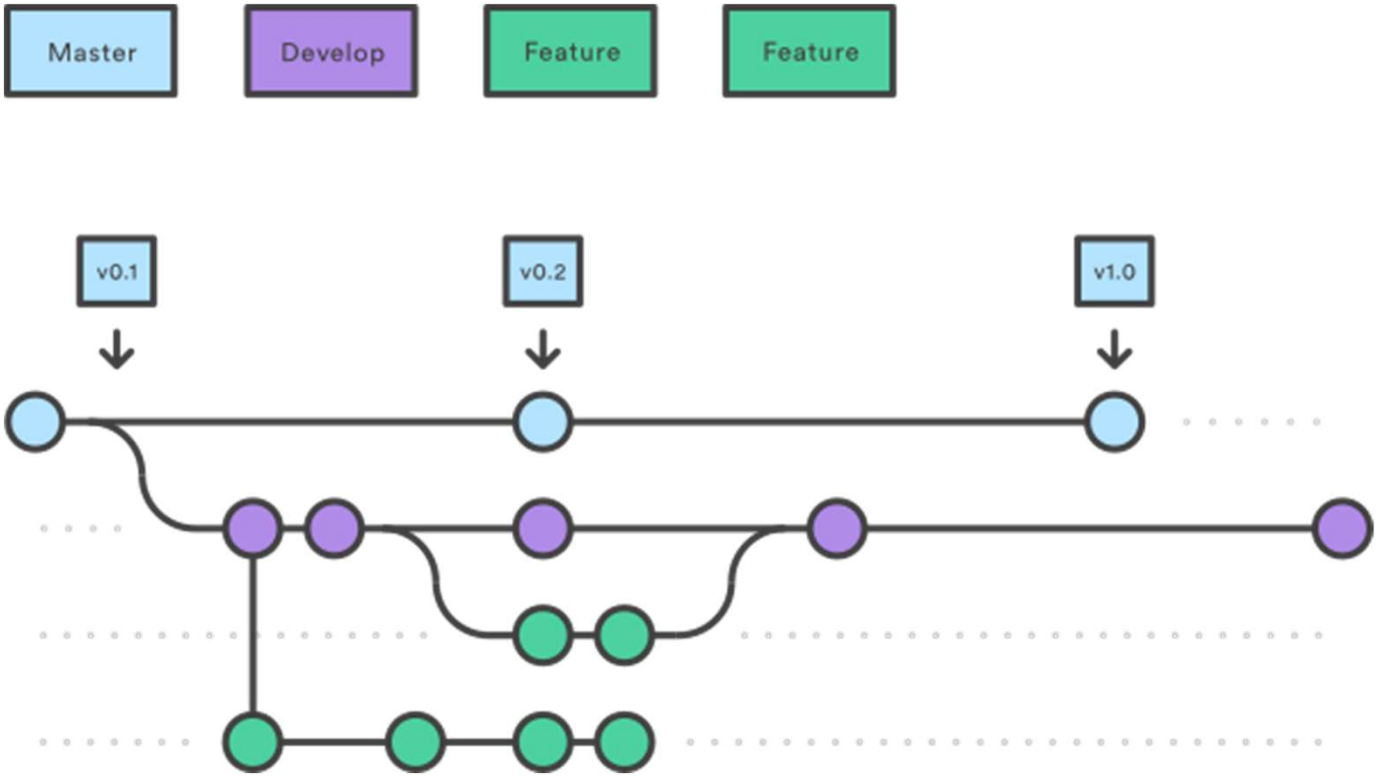


The rest of this workflow revolves around the distinction between these two branches.

CONTINUOUS DELIVERY KIT

HOW IT WORKS - FEATURE BRANCHES

Each new c should reside in its own branch, which can be pushed to the central repository for backup/collaboration. But, instead of branching off of master, feature branches use develop as their parent branch. When a feature is complete, it gets merged back into develop. Features should never interact directly with master.



CONTINUOUS DELIVERY KIT

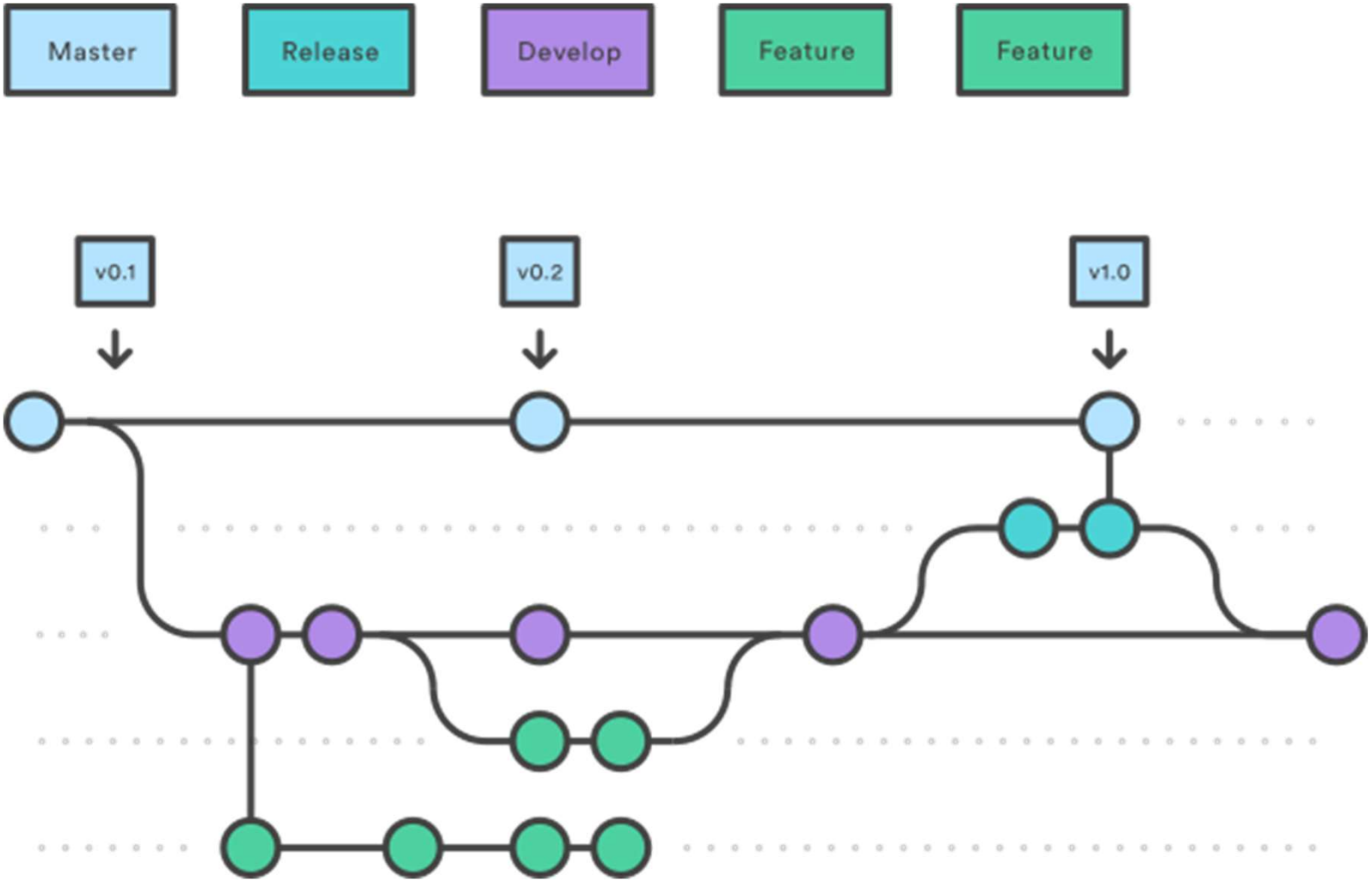
HOW IT WORKS - RELEASE BRANCHES

Once develop has acquired enough features for a release (or a predetermined release date is approaching), you fork a release branch off of develop. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it's ready to ship, the release gets merged into master and tagged with a version number. In addition, it should be merged back into develop, which may have progressed since the release was initiated.

Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. It also creates well-defined phases of development (e.g., it‘s easy to say, “this week we’re preparing for version 4.0” and to actually see it in the structure of the repository).

CONTINUOUS DELIVERY KIT

HOW IT WORKS - RELEASE BRANCHES



CONTINUOUS DELIVERY KIT

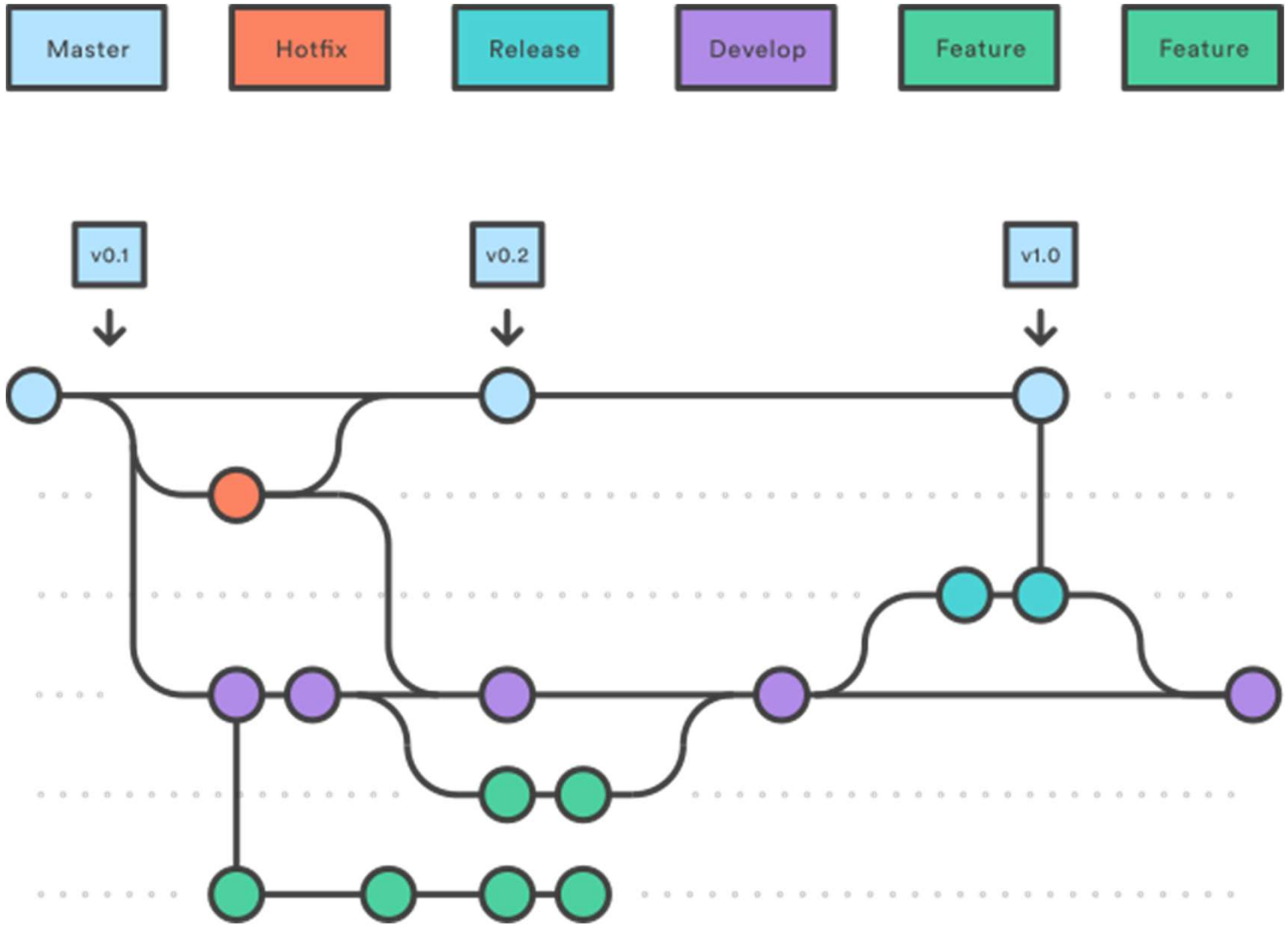
HOW IT WORKS - MAINTENANCE BRANCHES

Maintenance or “hotfix” branches are used to quickly patch production releases. This is the only branch that should fork directly off of master. As soon as the fix is complete, it should be merged into both master and develop (or the current release branch), and master should be tagged with an updated version number.

Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle. You can think of maintenance branches as ad hoc release branches that work directly with master.

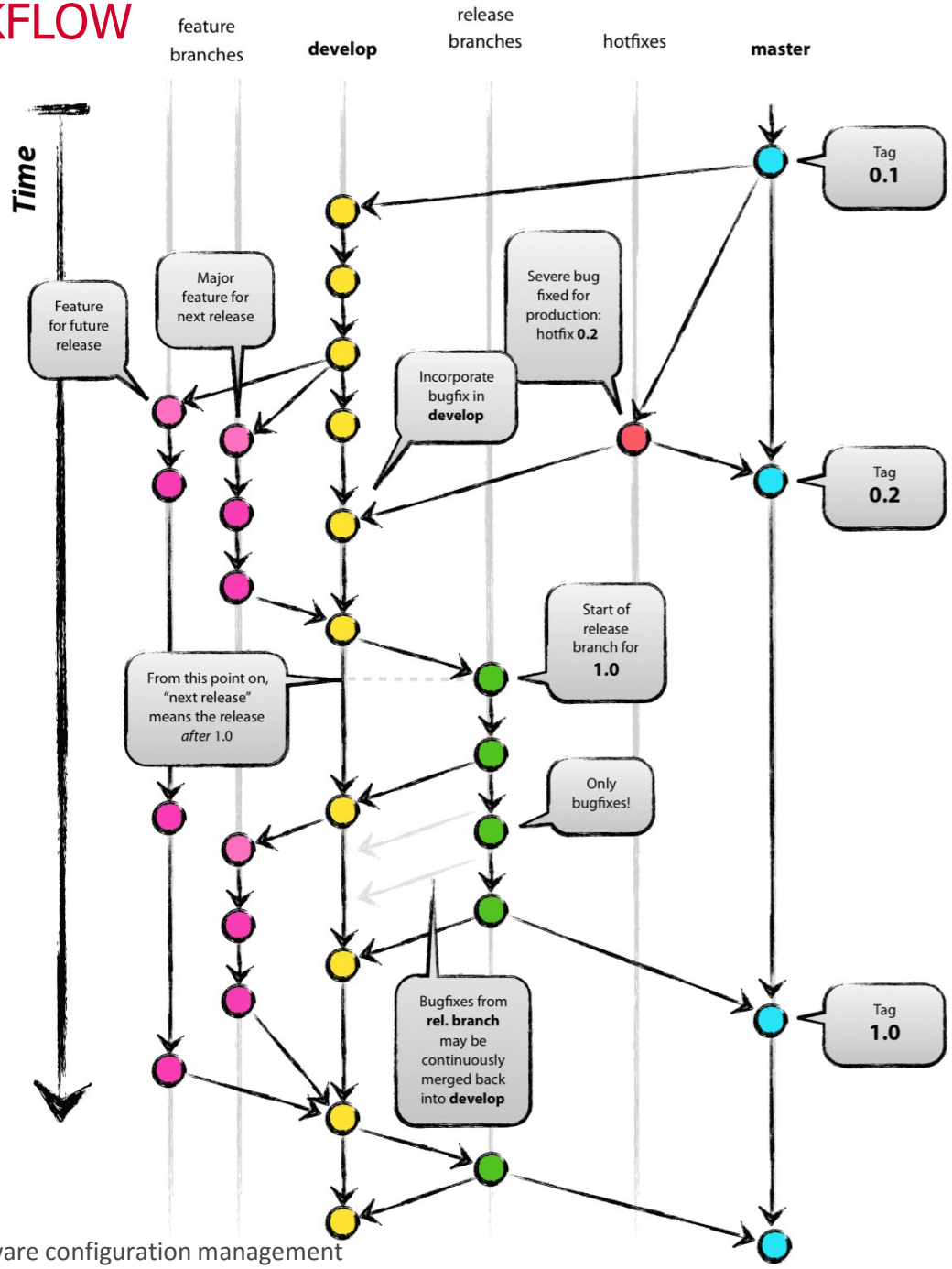
CONTINUOUS DELIVERY KIT

HOW IT WORKS - MAINTENANCE BRANCHES



CONTINUOUS DELIVERY KIT

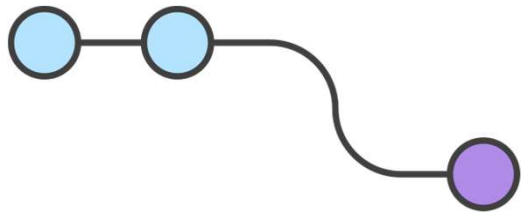
GITFLOW WORKFLOW



CONTINUOUS DELIVERY KIT

EXAMPLE - CREATE A DEVELOP BRANCH PT.1

The example below demonstrates how this workflow can be used to manage a single release cycle. We'll assume you have already created a central repository.



The first step is to complement the default master with a develop branch. A simple way to do this is for one developer to create an empty develop branch locally and push it to the server:

- *git branch develop*
- *git push -u origin develop*

CONTINUOUS DELIVERY KIT

EXAMPLE - CREATE A DEVELOP BRANCH PT.2

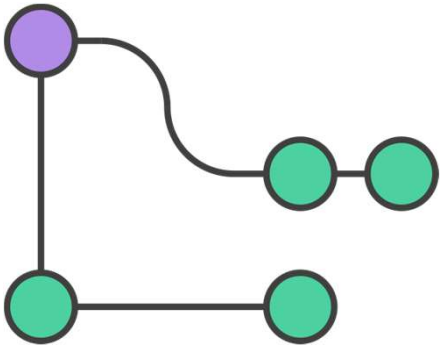
This branch will contain the complete history of the project, whereas master will contain an abridged version. Other developers should now clone the central repository and create a tracking branch for develop:

- *git clone ssh://user@host/path/to/repo.git*
- *git checkout -b develop origin/develop*

Everybody now has a local copy of the historical branches set up.

CONTINUOUS DELIVERY KIT

EXAMPLE - MARY AND JOHN BEGIN NEW FEATURES



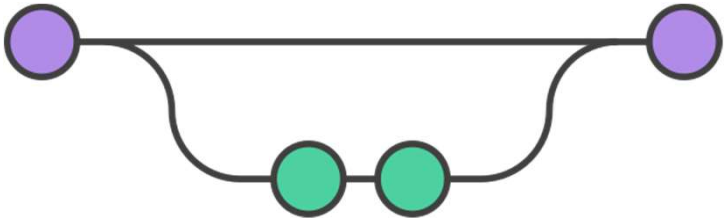
Our example starts with John and Mary working on separate features. They both need to create separate branches for their respective features. Instead of basing it on master, they should both base their feature branches on develop:

git checkout -b some-feature develop
Both of them add commits to the feature branch in the usual fashion: edit, stage, commit:

- *git status*
- *git add <some-file>*
- *git commit*

CONTINUOUS DELIVERY KIT

EXAMPLE - MARY FINISHES HER FEATURE



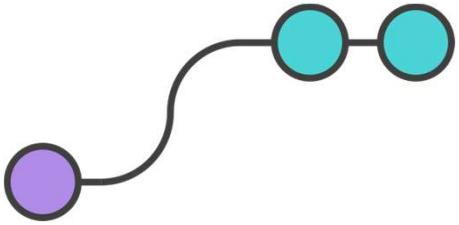
After adding a few commits, Mary decides her feature is ready. She can merge it into her local develop and push it to the central repository, like so:

- *git pull origin develop*
- *git checkout develop*
- *git merge some-feature*
- *git push*
- *git branch -d some-feature*

The first command makes sure the develop branch is up to date before trying to merge in the feature. Note that features should never be merged directly into master. Conflicts can be resolved in the same way as in the Centralized Workflow.

CONTINUOUS DELIVERY KIT

EXAMPLE - MARY BEGINS TO PREPARE A RELEASE



While John is still working on his feature, Mary starts to prepare the first official release of the project. Like feature development, she uses a new branch to encapsulate the release preparations. This step is also where the release's version number is established:

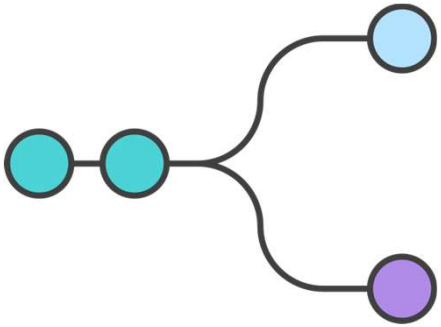
- *git checkout -b release-0.1 develop*

This branch is a place to clean up the release, test everything, update the documentation, and do any other kind of preparation for the upcoming release. It's like a feature branch dedicated to polishing the release.

As soon as Mary creates this branch and pushes it to the central repository, the release is feature-frozen. Any functionality that isn't already in develop is postponed until the next release cycle.

CONTINUOUS DELIVERY KIT

EXAMPLE - MARY FINISHES THE RELEASE PT.1



Once the release is ready to ship, Mary merges it into master and develop, then deletes the release branch. It's important to merge back into develop because critical updates may have been added to the release branch and they need to be accessible to new features. Again, if Mary's organization stresses code review, this would be an ideal place for a pull request.

- *git checkout master*
- *git merge release-0.1*
- *git push*
- *git checkout develop*
- *git merge release-0.1*
- *git push*
- *git branch -d release-0.1*

CONTINUOUS DELIVERY KIT

EXAMPLE - MARY FINISHES THE RELEASE PT.2

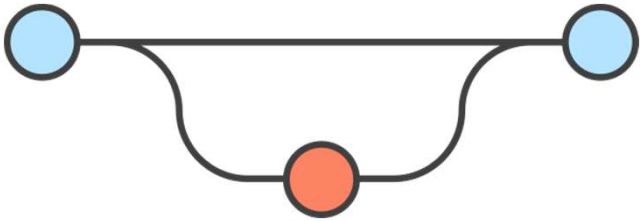
Release branches act as a buffer between feature development (develop) and public releases (master). Whenever you merge something into master, you should tag the commit for easy reference:

- *git tag -a 0.1 -m "Initial public release" master*
- *git push –tags*

Git comes with several hooks, which are scripts that execute whenever a particular event occurs within a repository. It’s possible to configure a hook to automatically build a public release whenever you push the master branch to the central repository or push a tag.

CONTINUOUS DELIVERY KIT

EXAMPLE - END-USER DISCOVERS A BUG PT.1



After shipping the release, Mary goes back to developing features for the next release with John. That is, until an end-user opens a ticket complaining about a bug in the current release. To address the bug, Mary (or John) creates a maintenance branch off of master, fixes the issue with as many commits as necessary, then merges it directly back into master.

- *git checkout -b issue-#001 master*
- *# Fix the bug*
- *git checkout master*
- *git merge issue-#001*
- *git push*

CONTINUOUS DELIVERY KIT

EXAMPLE - END-USER DISCOVERS A BUG PT.2

Like release branches, maintenance branches contain important updates that need to be included in develop, so Mary needs to perform that merge as well. Then, she's free to delete the branch:

- *git checkout develop*
- *git merge issue-#001*
- *git push*
- *git branch -d issue-#001*

CONTINUOUS DELIVERY KIT
GIT COMMAND: CONFIGURE TOOLING

Configure user information for all local repositories

$ git config --global user.name "[name]"

- Sets the name you want attached to your commit transactions

$ git config --global user.email "[email address]"

- Sets the email you want attached to your commit transactions

$ git config –list

- If you want to check your settings.

CONTINUOUS DELIVERY KIT
GIT COMMAND: CREATE REPOSITORIES

1. **$ git init [project-name]**

Creates a new local repository with the specified name in an Existing Directory (**$ mkdir dir_repo && cd dir_repo**)

Once you have a remote repo setup, you will need to add a remote repo url to your local git config, and set an upstream branch for your local branches

$ git remote add origin <remote_repo_url>

$ git push -u origin master

2. **$ git clone [url]**

Downloads a project and its entire version history

CONTINUOUS DELIVERY KIT

GIT COMMAND: CREATE BRANCH

1. **$ git branch [branch-name]**

   $ git push –u origin [branch-name]

   $ git checkout [branch-name]

   Creates a new branch in local repository, upload it to origin (remote repository), switch to new branch

2. **$ git checkout –b [branch-name]**

   Creates a new branch and switch to it using

$ git branch -d [branch-name] Deletes the specified branch

$ git branch Lists all local branches in the current repository

$ git branch -a Lists all local and remote (red color) branches in the current repository

$ git push origin :developer delete the specified branch in the remote repository

CONTINUOUS DELIVERY KIT
GIT COMMAND: WORKFLOW GIT

$ git checkout develop

$ git pull origin develop

Modify my file

$ git add . Move file in stage area

$ git commit –m 'my commit' Records file snapshots permanently in version history

$ git push origin develop to publish your local commits

CONTINUOUS DELIVERY KIT

GIT COMMAND: REPLACE LOCAL CHANGES

$ git checkout -- <file>

to discard changes in working directory

$ git revert

Create new commit that undoes all of the changes made in , then apply it to the current branch.

$ git reset HEAD <file>...

To remove in stage area

$ git fetch origin

$ git reset --hard origin/master

If you instead want to drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it like this

CONTINUOUS DELIVERY KIT
GIT COMMAND: MERGE BRANCH

$ git merge --no-ff [branch name]

- Merge into the current branch.

$ git merge origin/develop

- Merge develop into the current branch

$ git rebase <base>

- Rebase the current branch onto <base>. <base> can be a commit ID,a branch name, a tag, or a relative reference to HEAD.

CONTINUOUS DELIVERY KIT
GIT COMMAND: REVIEW HISTORY

$ git status

- Lists all new or modified files to be committed

$ git log

- Lists version history for the current branch

$ git log --author=bob

- To see only the commits of a certain author:

$ git diff <source_branch> <target_branch>

- Shows content differences between two branches

$ git diff --NomeFile

CONTINUOUS DELIVERY KIT
GIT COMMAND: SAVE FRAGMENTS

$ git stash

$ git stash save “message"

- Temporarily stores all modified tracked files

$ git stash list

- Lists all stashed changesets

$ git stash pop

- Restores the most recently stashed files

$ git stash drop stash@{1}

- Discards the most recently stashed changeset

Delivering Transformation. Together.