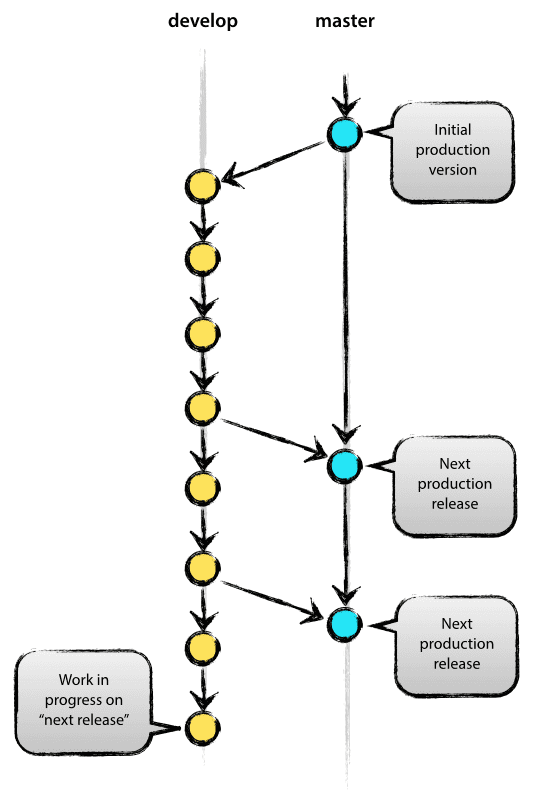
Git branching model

# Main branches

The central repo holds two main branches with an infinite lifetime:

* master
* develop



The master branch at origin. 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.

We consider origin/develop to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release.

When the source code in the develop branch reaches a stable point and is ready to be released, all of the changes should be merged back into master and then tagged with a release number.

Therefore, each time when changes are merged back into master, this is a new production release *by definition*. We could use a Git hook script to automatically build and roll-out our software to our production servers every time there was a commit on master.

# Supporting branches

The different types of branches we may use are:

* **Feature branches** (usually prefixed with “feature/”)

When you start work on anything non-trivial, you create a feature branch. When finished, you will merge this branch back into the development branch to queue it for the next release

* **Release branches** (usually prefixed with “release/”)

When you are about to package a new release, you create a release branch from the development branch. You can commit to it during your preparation for a release, and when it’s ready to be deployed, you merge it into both the development branch and the master branch (to indicate that the release has been deployed).

* **Hotfix branches** (usually prefixed with “hotfix/”)

If you need to patch the latest release without picking up new features from the development branch, you can create a hotfix from the latest deployed code in master. Once you have made your changes, the hotfix branch is then merged back into both the master branch (to update the released version) and the development branch (to make sure the fixes go into the next release too).

**SourceTree** helps you utilise these branches via **git-flow** actions.

**Git-flow**

There’s a handy new addition to the toolbar in SourceTree from version 1.5 onwards (keyboard shortcut Cmd-Alt-F):

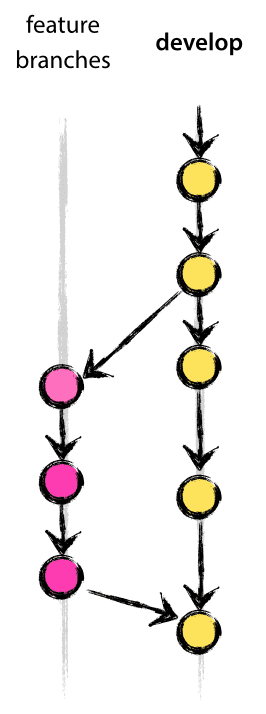


Based on the current state of the repository, the Git-flow button will bring up a dialog with the most likely action you’d want to perform next. So if you haven’t set up git-flow on this repo yet, it’ll help you do that by default. If you’re on the development branch, it will default to starting a new feature. If you’re already on a feature branch, it will offer to finish your current feature and merge it back into the development branch, and so on. You can always get to all the other git-flow actions via this button as well, but most of the time the default option will be the action you’ll want SourceTree to perform.

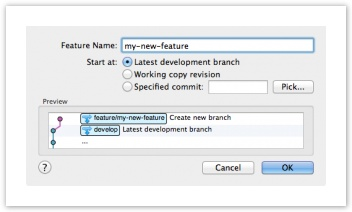
If you haven’t used git-flow already on this repository, the first thing SourceTree will do is initialise your repository to use it. You’ll probably just want to use the defaults in the initialisation window so that’s not covered here.

**Starting a new Feature**

You can commit trivial changes directly to the development branch (‘develop’) if you like, but any time you start on something non-trivial you should explicitly start a new feature. This ‘Start Feature’ action will be the default action when you click the Git-flow button if you are currently on the dev branch.



Feature branches typically exist in developer repos only, not in origin.



The first thing to note is the ‘Preview’ window, which is present on all of the action dialogs and shows you what will actually happen when you confirm the dialog. In this case, a new feature branch called ‘feature/my-new-feature’ will be created (default prefix when initialised in git-flow is initialised in SourceTree). You commit your feature implementations to this branch.

If at any time you want to switch branches, either to another feature branch or to somewhere else, just use the normal mechanisms in SourceTree to do that, such as double-clicking a log entry or a branch in the sidebar. Git-flow determines your context simply from the branch you currently have checked out, so it’s fine to jump around if you like.

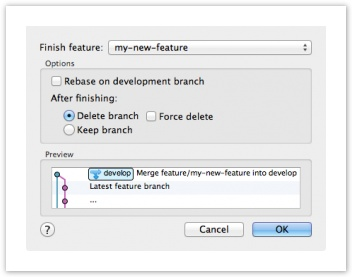
Equivalent git command:

$ git checkout -b my-new-feature develop

Switched to a new branch "my-new-feature"

**Finishing a feature**

Eventually when you’re done implementing a feature, use the ‘Finish Feature’ action (again, this will be the default action from the toolbar button if you’re on a feature branch):



Again, the Preview shows you what will happen — the feature branch will merge into the main development branch, essentially queueing it for inclusion in the next release. Feature branches are deleted by default but you can opt to retain them if you like.

Equivalent git commands:

$ git checkout develop

Switched to branch 'develop'

$ git merge --no-ff my-new-feature

Updating ea1b82a..05e9557

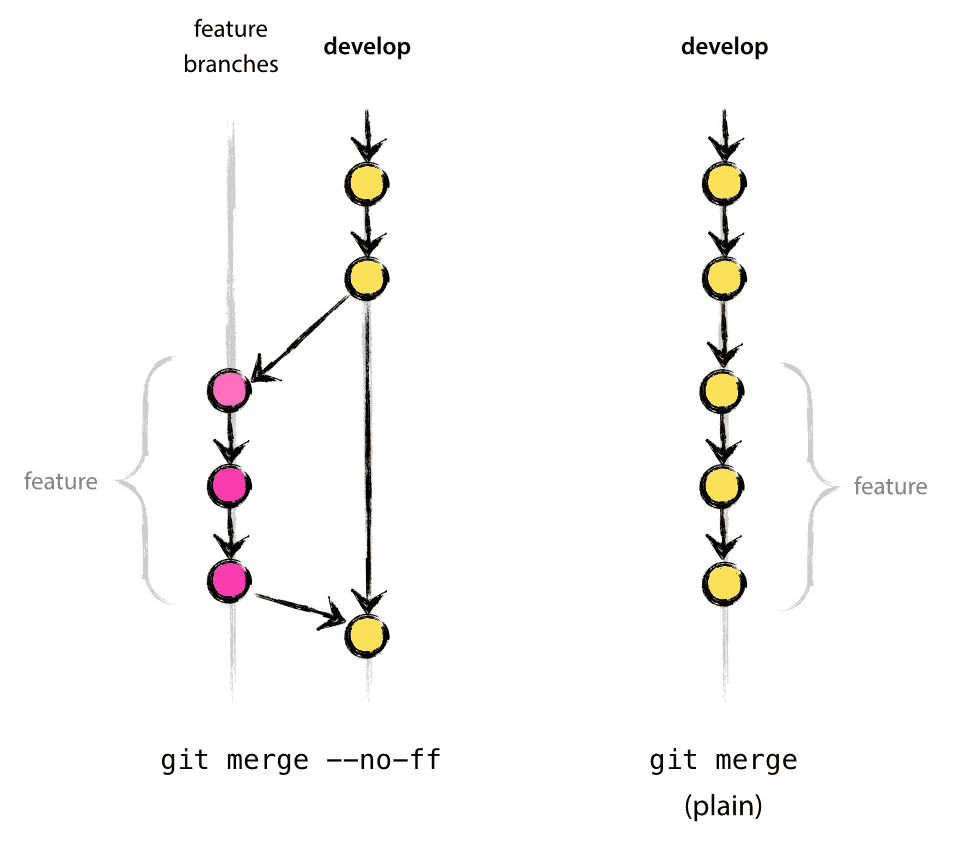
(Summary of changes)

$ git branch -d my-new-feature

Deleted branch my-new-feature (was 05e9557).

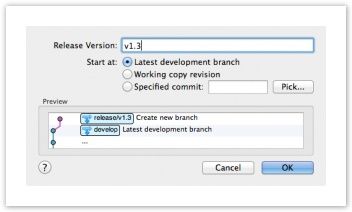
$ git push origin develop

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.



**Starting a Release**

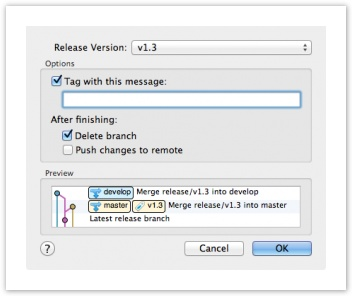
You start a release branch when you want to start preparing a new release, which probably coincides with a feature freeze. Accessing the Start Release option –either from the menu or from the toolbar action selection window — brings up the following dialog:



The preview shows that a new release branch will be created. Most of the time, you want to start the release from the latest commit in the development branch, but you can choose to base it on another commit if you wish. This essentially freezes the release, so it’s not affected by subsequent development. You can also perform preparatory tasks for the release process on this branch, such as updating the version number in source files, updating changelogs, or committing other tweaks. Once these tweaks are done, you can finish the release as described below.

**Finishing a Release**

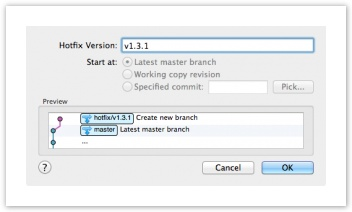
Once all the adjustments required for the release are done and committed, you can conclude the release process. If you’re on the release branch already, the git-flow button will show you the following dialog:

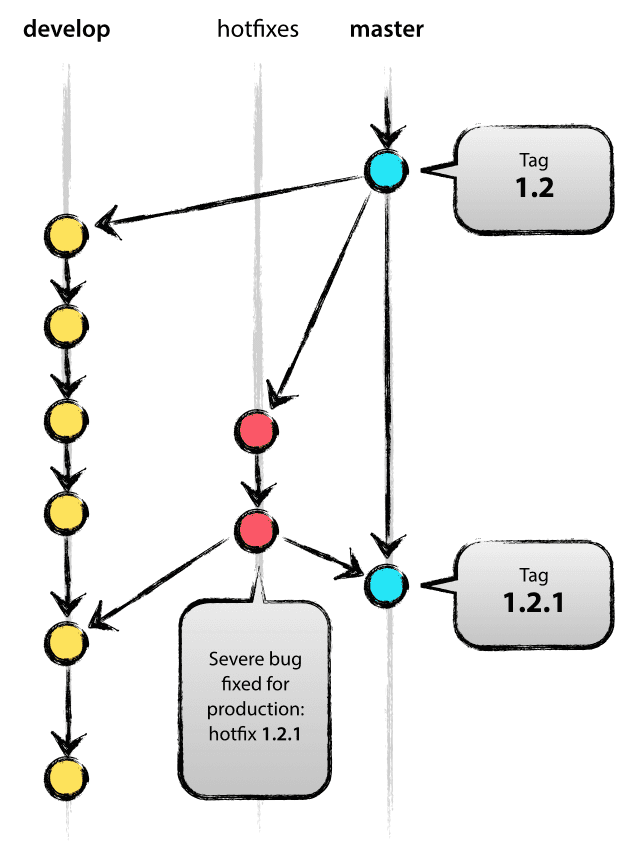


The preview illustrates that the release branch will be merged into the production branch (‘master’ or ‘default’ normally) to indicate an update to the deployed version. SourceTree will also create a tag here for the release. The release branch changes will also merge back into the development branch to make sure the develop branch is kept up to date.

**Starting a Hotfix**

What if you need to just fix a bug on the latest release? You don’t want to create a new release, because that will pick up the recent changes from the development branch. So instead, you can start a hot fix:





Hot fixes always start from the latest production code from the ‘master’ or ‘default’ branch. Other than that, they’re basically the same as release branches. Moreover, when you’re finished with the hot-fix, they behave the same way as finishing a release branch. The changes are merged back into both the production branch and the development branch, and a tag is created on the production branch for the hot fix release.

**Branching strategy**

While working on a story, create a branch from the current repository. The branch name should follow the convention *<<project name>>-<<story number>>-<<short description>>.* If the project name is ASUIB and story number is 319 then the branch name should be like ASUIB-319-ScanBoardingPass. All your commits related to your story should go into that branch with your id on every commit. If SSH keys are not enabled then mark your identity with below commands:

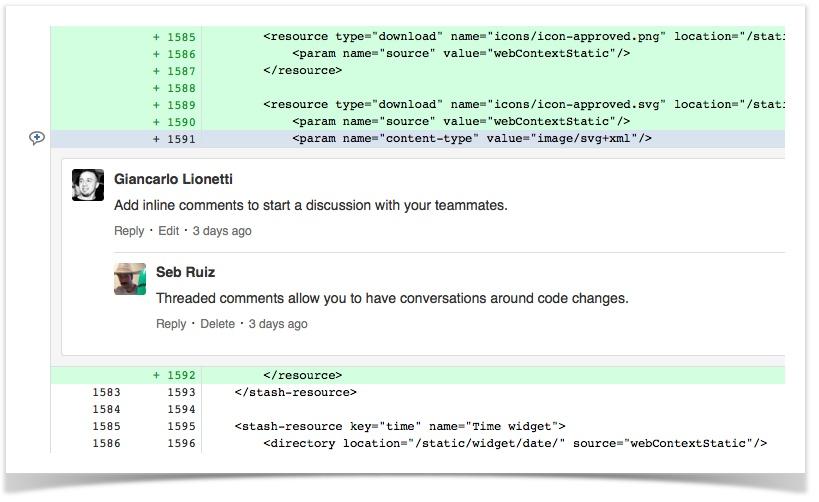
*git config - -config user.name “your name”*

*git config - -config user.email “your email”*

After your feature is tested on local, create a pull request that will be used in review process.

**Pull request review strategy**

After a feature is done and tested on local machine generate a pull request. The reviewer should put appropriate comments regarding code quality, test cases, design decision or refactoring options. Refer screenshot below:



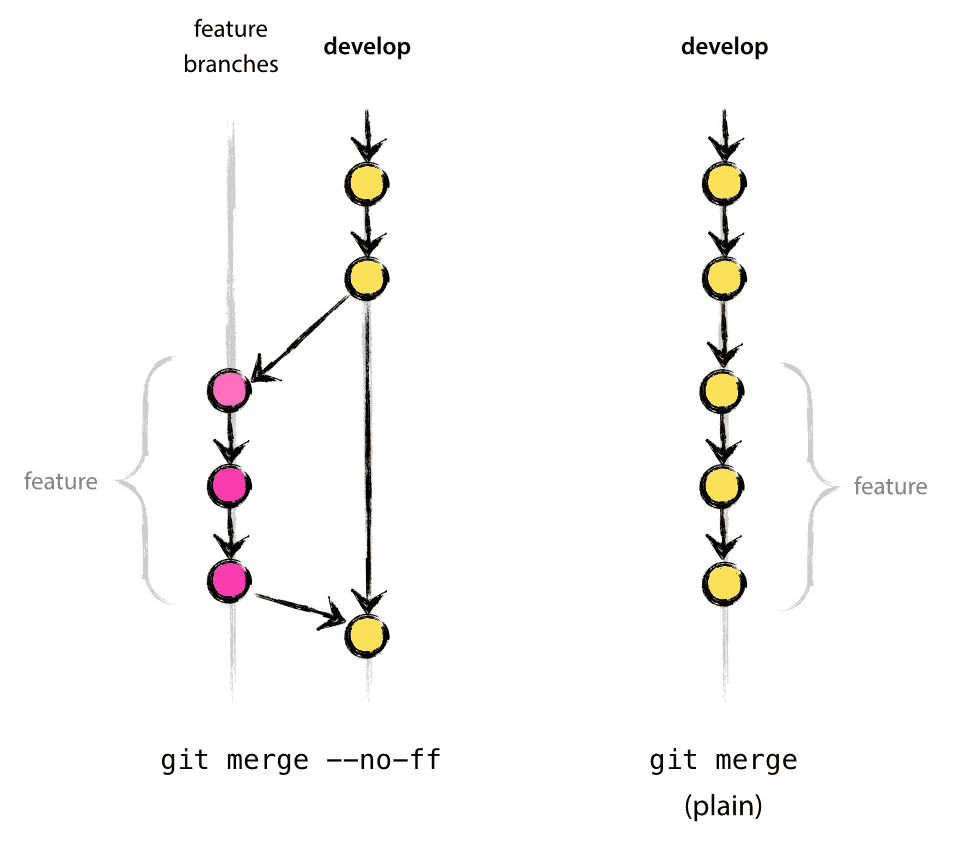
After the feedback has been worked upon by the team member, the pull request should be approved by the reviewer before it can be merged.

**Merge Strategy**

Git provides a plethora of merging choices.

* Explicit, non fast-forward merge
* Implicit via rebase or fast-forward merge
* Squash on merge

We strongly recommend using explicit merges. The reason is very simple: explicit merges provide great traceability and context on the features being merged.



The non fast-forward merge 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.

**References:**

<https://blog.sourcetreeapp.com/2012/08/01/smart-branching-with-sourcetree-and-git-flow/>

<http://nvie.com/posts/a-successful-git-branching-model/>

<https://github.com/GSoft-SharePoint/Dynamite/wiki/Getting-started-with-SourceTree,-Git-and-git-flow>