Git/GitFlow/GitHub Workflows

|  |
| --- |
|  |
|  |
| Barbara Kocinski  4/9/2018 |

Contents

[Overview 2](#_Toc511312651)

[What is Git/GitFlow/GitHub? 2](#_Toc511312652)

[Configuration 3](#_Toc511312653)

[Rules of the GitFlow Process 3](#_Toc511312654)

[Development Workflow 4](#_Toc511312655)

[Release Workflow 5](#_Toc511312656)

[Production Defect Workflow 7](#_Toc511312657)

[Securing master and develop branches from “accidental” commits 9](#_Toc511312658)

[Pull Requests 10](#_Toc511312659)

# Overview

## What is Git/GitFlow/GitHub?

**Git** is a distributed open-source version control system which enables developers to track code changes and easily collaborate on a software project repository.

**GitFlow** is a branching model for Git which offers a clear model on how to structure branches and track releases, features, current development state, and hotfixes.

In this model, a repository has two main branches:

* **Master**: A highly stable branch that is always production-ready and contains the last release version of source code in production.
* **Develop**: A branch derived from the master branch and serves as a branch for integrating different features planned for an upcoming release. It is where developers collaborate and merge feature branches.

There are also many other types of branches that are created:

* **Feature**: These are branches that the developers create from the develop branch. A feature branch will be created for a specific feature (ie: Eligibility Profile).
* **Release**: These also derive from develop branch but are used during releases. Developers normally do not create these branches, but may submit pull requests to have bug fixes added to these types of branches.
* **Hotfix**: These derive from the ***master*** branch and are used to fix bugs in the production branch that was identified after a release.

**GitHub** is a web-based hosting service for version control using git. It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.

Organizations are at the highest level in the GitHub structure. They simplify management of group-owned repositories and enable you to control access permissions on repositories contained within them.

Organizations consist of teams and repositories.

* **Teams** enable you to control who has access to the repositories. You can create independent or nested teams to manage repository permissions. Organization owners and team maintainers can give teams admin, read, or write access to organization repositories. Permissions are maintained at the **repository level** though, not at the branch level. For instance if you allow a team to have read/write access to a repository, they will have this access to all branches in the repository, including the master branch.
* **Repositories** are where the code lives. There should be **one** repository for each code project. For example if you have an Angular application and a SpringBoot application, there should be one repository for the Angular application and another project for the SpringBoot application. Each repository can be accessed by its own unique URL.

Example Repositories: barb-angular, barb-services

Configuration

In order to protect our production code, we will need to have two separate organizations: RxClaimProd and RxClaim.

**RxClaimProd**: This organization will contain the production repositories. Only a few team members should have permissions to the repositories in this organization. The Git maintainer will need to create this organization and team, and assign specific members to this organization. Only the Git maintainers should be allowed to create and maintain repositories in this organization.

**RxClaim:** This organization will contain all the development repositories. Developers have read/write access to the repositories in this organization in order to collaborate in the development process.

Rules of the GitFlow Process

* No code should ever be directly committed to a remote repository. Remote repositories will be updated through pull requests.
* The master branch is used to store versions of the production code. Development should be done in the develop branch by creating feature branches derived from it and should not be done from master.
* Git hooks should be used to prevent accidental commits to the remote repositories’ master branch.
* The develop branch is used to store code for the upcoming release. If a feature is not identified as being part of the upcoming release, it must not be merged into develop until after a release branch is created.
* Once a release branch is created, no features should be added to this branch. Only hotfixes will be allowed to fix any defects found in testing.

# Development Workflow

|  |  |  |
| --- | --- | --- |
| Local RxClaim/barb-angular  Repository | Remote RxClaim/barb-angular  Repository | Remote RxClaimProd/barb-angular  Repository |
|  | master  develop | develop  master |
| Developer clones barb-angular repository and switches to develop branch  master  develop | develop  master | develop  master |
| Developer creates a feature branch off of develop and works on the feature  feature/screen1  develop  master | develop  master | develop  master |
| When developer is done with the feature, he pulls from the develop branch to get the latest changes to develop and merges this branch into his feature branch and tests again. Once done he pushes his branch to the Remote RxClaim barb-angular repository and issues a **pull request** and requests a code review  feature/screen1  develop  master | feature/screen1  develop  master | develop  master |
| feature/screen1  develop  master | Peers review code. Once all approve the code, the Git maintainer merges the Feature into the develop branch **ONLY** **IF** that feature was planned for the upcoming release, and then deletes the  feature/screen1  develop  master  feature branch. | develop  master |

# Release Workflow

|  |  |  |
| --- | --- | --- |
| Local RxClaim/barb-angular  Repository | Remote RxClaim/barb-angular  Repository | Remote RxClaimProd/barb-angular  Repository |
|  | master  develop | develop  master |
|  | Git maintainer creates a Release branch off of develop. The release branch should then be deployed to the non-prod environments to be tested.  release/1.0.01.0.0  develop  master | develop  master |
| If there are defects found in the release, the developer pulls the latest code and switches to  the release branch  release/1.0.0  develop  master | release/1.0.01.0.0  develop  master | develop  master |
| The developer then creates a hotfix branch off of the release branch. When the fix is complete, the developer pushes the hotfix branch to the remote RxClaim barb-angular repository and issues a **pull request** and requests a code review  hotfix/fix-bug123  release/1.0.0  develop  master | hotfix/fix-bug123  release/1.0.01.0.0  develop  master | develop  master |
| master  hotfix/fix-bug123  release/1.0.0  develop  master | Peers review code. Once all approve the code, the Git maintainer merges the hotfix into the release branch, then deletes the hotfix branch and the release is deployed for further testing.  hotfix/fix-bug123  develop  release/1.0.0 | develop  master |
| master | Once the release has been tested and validated, the Git maintainer will be merge the release branch into both master and develop in both the RxClaim barb-angular repository **and** the RxClaimProd barb-angular repository.  The release branch can optionally be pushed to the RxClaimProd barb-angular repository to be saved.  release/1.0.0  develop | release/1.0.0  develop  master |

# Production Defect Workflow

|  |  |  |
| --- | --- | --- |
| Local RxClaim/barb-angular  Repository | Remote RxClaim/barb-angular  Repository | Remote RxClaimProd/barb-angular  Repository |
|  | master  develop  release/1.0.01.0.0 | release/1.0.01.0.0  develop  master |
|  | Git maintainer creates a HotFix branch off of master.  hotfix/fix-bug456  release/1.0.01.0.0  develop  master | release/1.0.01.0.0  develop  master |
| The developer pulls the latest code and switches to  The HotFix branch.  hotfix/fix-bug456  hotfix/fix-bug456  release/1.0.0  develop  master | develop  master  release/1.0.01.0.0 | release/1.0.01.0.0  develop  master |
| The developer makes the required changes to this branch, then pushes the hotfix branch to the remote RxClaim barb-angular repository and issues a **pull request** and requests a code review  hotfix/fix-456  master  release/1.0.0  hotfix/fix-bug456  develop | master  develop  release/1.0.01.0.0 | release/1.0.01.0.0  develop  master |
| release/1.0.0  hotfix/fix-bug456  develop  master | Peers review code. Once all approve the code, the Git maintainer merges the hotfix into the master branch and the master branch is deployed in a non-prod environment for further testing.  master  develop  release/1.0.0  hotfix/fix-bug456 | release/1.0.0  develop  master |
| master | Once the master branch has been tested and validated and the code has been deployed in production, the Git maintainer will merge the hotfix branch into develop in both the RxClaim barb-angular repository **and** the RxClaimProd barb-angular repository and will delete the hotfix branch.  hotfix/fix-bug456  develop | release/1.0.0  develop  master |

# Securing master and develop branches from “accidental” commits

We may want to control what goes into the develop branch and certainly what goes into the master branch from “accidental” commits. We can do this by either using Git Hooks.

<https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks>.

An example of a git hook to Example to prevent commits to master: <https://gist.github.com/aaronhoffman/ffbfd36928f9336be2436cffe39feaec>

pre-commit file:

#!/bin/sh

# prevent commit to local master branch

branch=`git symbolic-ref HEAD`

if [ "$branch" = "refs/heads/master" ]; then

echo "pre-commit hook: Can not commit to the local master branch."

exit 1

fi

exit 0

pre-push file:

#!/bin/sh

# Prevent push to remote master branch

while read local\_ref local\_sha remote\_ref remote\_sha

do

if [ "$remote\_ref" = "refs/heads/master" ]; then

echo "pre-push hook: Can not push to remote master branch."

exit 1

fi

done

exit 0

# Pull Requests

When developers are done with their features or hotfixes, they will issue pull requests instead of committing their code directly to remote repositories.

Pull requests let others know about changes you've pushed to a GitHub repository. Once a pull request is sent, interested parties can review the set of changes and discuss potential modifications. Once the pull request is approved, the Git maintainer can commit the code though the pull request.

Pull Request workflow:

1. Developer selects a story and creates a feature branch off of develop for the story.
2. Developer works on the story and tests it locally.
3. Developer is done with the work and ready to have it merged into develop.

* Developer issues a pull on the develop branch to get the latest changes and merges it into his local branch so his feature can be retested with the latest code changes and any conflicts resolved.
* Once the code is tested and any conflicts are resolved, the developer issues a pull request to have his branch merged into the develop branch. At this time the developer also can request his peers and an expert to review his code.
* Peers and expert review the code and either approve it or ask for modifications.
* Once the code is approved, the Git maintainer is notified to merge the pull request into the develop branch.
* Jenkins detects the change and deploys the code to the cloud or destination server.