Git Flow

Isaac Griffith

Fall 2021

Idaho State University | Software Engineering

SE 5520 - Software Construction and Configuration Management

Outcomes

At the end of Today's Lecture you will be able to:

- Understand the concept of Semantic Versioning
- Understand the need for and how to adopt a Changelog
- Understand git repo management using the git flow method
- Apply git flow to your own repos
- Use the git flow tool

Inspiration

Semantic Versioning

SE 5520

Semantic Versioning

Version numbers for releases should follow the Semantic Versioning 2.0.0 approach:

- Each version number is specified as: MAJOR.MINOR.PATCH
- We increment:
  1. MAJOR version when you make incompatible API changes
  2. MINOR version when you add functionality in a backwards compatible manner
  3. PATCH version when you make backwards compatible bug fixes
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format

Project Docs

SE 5520

Project Documentation

- Typically a GitHub project is documented in a few ways
  - Changelogs
  - Readme
  - Project Wiki
  - GitHub Pages
- We will discuss the first two, and I will leave the latter for your own discovery

Keeping a Changelog

- Normally keps as the file `CHANGELOG.md` in the project root folder
- A **changelog** is simply a file containing a curated ordered list of notable changes for each version of a project
- Provides documentation so that other contributors know what happened in the project
- All projects need a changelog

Changelog Guiding Principles

- Changelogs are for humans, not machines.
- There should be an entry for every single version.
- The same types of changes should be grouped.
- Versions and sections should be linkable.
- The latest version comes first.
- The release date of each version is displayed.
- Mention whether you follow Semantic Versioning.

Types of Changes

- `Added` for new features
- `Changed` for changes in existing functionality
- `Deprecated` for soon-to-be removed features
- `Removed` for now removed features
- `Fixed` for any bug fixes
- `Security` in case of vulnerabilities

Reducing Effort

- You should keep a section titled `Unreleased` which tracks upcoming changes
- Serves two purposes:
  - Allows people to see changes that are expected in upcoming releases
  - Allows developers to simply move the `Unreleased` section to the next released version

Project README.md

- Project readme's follow a specific format in order to immediately orient developers to the most important aspects of a project.
  - Normally kept as `README.md` in the project's root folder
- This format is as follows:
  - **Project Name** - the project name, and the first thing they will see
  - **Description** - A clear and concise description of the importance of your project and what it does
  - **Table of Contents** - Optional, but allows for quicker navigation
  - **Installation** - Informs users how to locally install your project (use pictures or an animated gif to improve)
  - **Usage** - Describes how to use the project once it has been installed (screenshots help)
  - **Contributing** - Describes how others may contribute to the project
  - **Credits** - Highlights and links to authors of the project
  - **License** - License of the project (may be a link to another file)

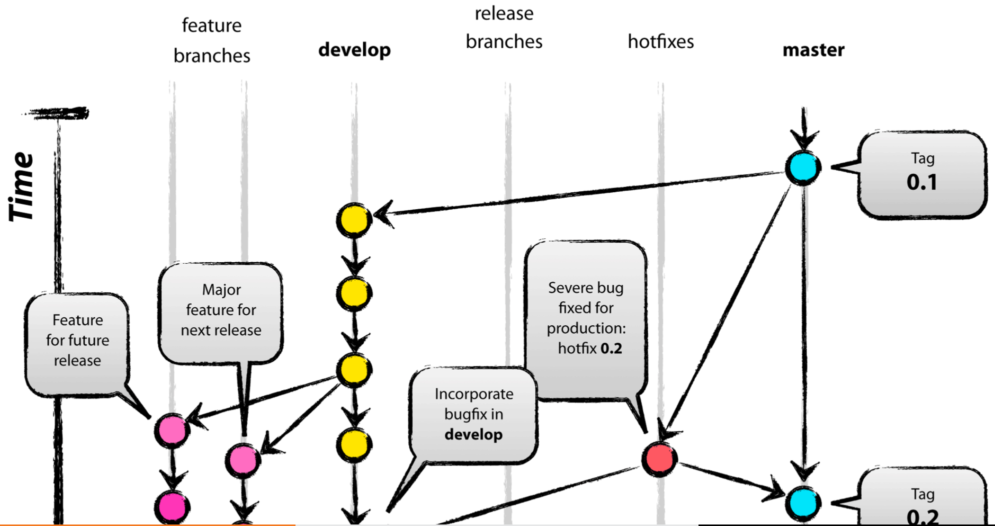
GitFlow

SE 5520

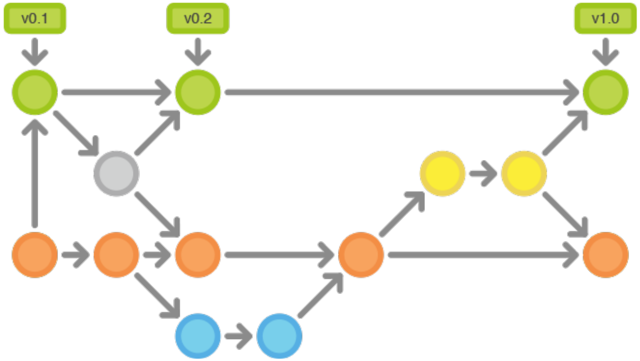
Git Flow?

- Git Flow is a method and tool for managing the workflow of git.
- Is it better than other approaches
  - Yes and No, but it does simplify the majority of git operations within a project
- Just like all techniques and approaches there are champions and detractors
  - But, if you follow the approach it works quite well

Git Flow Workflow

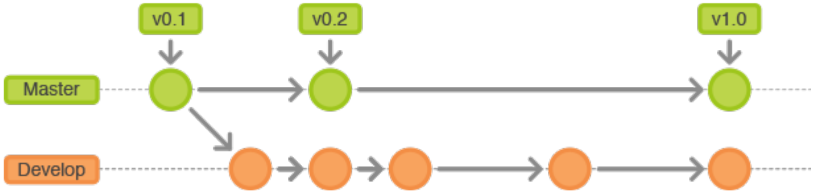

feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Major feature for next release
Feature for future release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2

How Git Flow Works



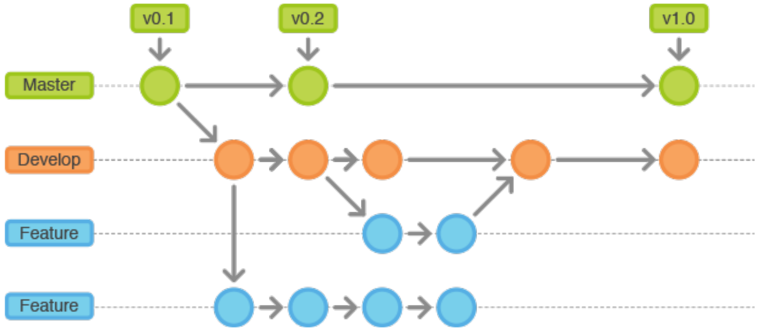
- The Git Flow workflow uses a central repository as the communication hub for all developers.
- Developers work locally and push branches to the central repo.

Historical Branches



- Instead of a single `main` branch, this workflow uses two branches to record the history of the project.
  - The `main` branch stores the official release history
  - The `develop` branch serves as an integration branch for Features
  - You should also tag all commits in the `main` branch with a version number

Feature Branches

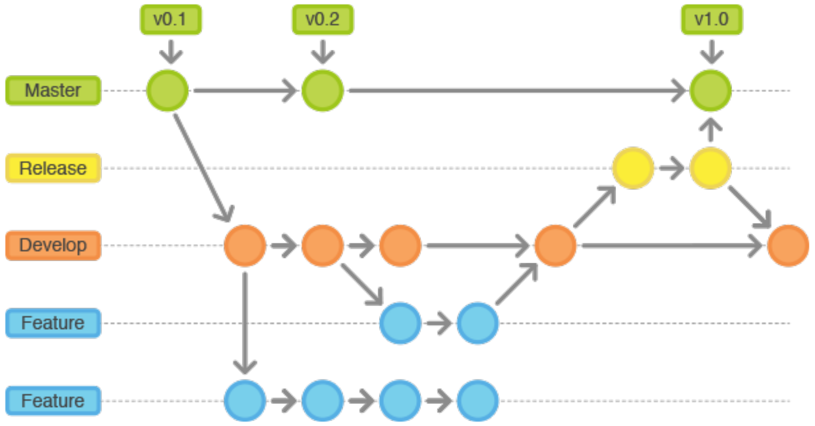


- Each new feature should reside in its own branch
  - Which is pushed to the central repo for backup/collaboration
  - `develop` is the parent branch for feature branches
  - Upon completion a feature branch is merged into `develop`
  - Features should never interact directly with `main`

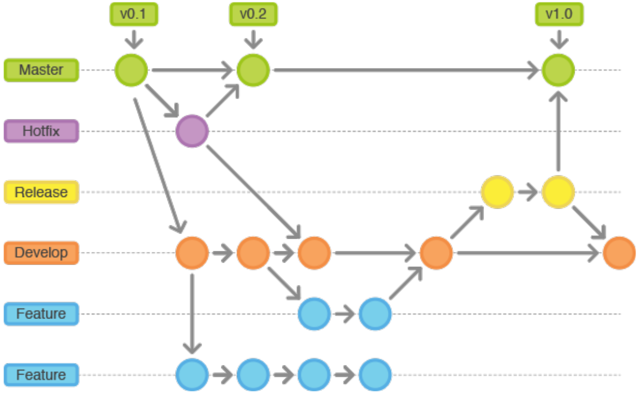
Feature Branches - Best Practices

- May branch off: `develop`
- Must merge back into: `develop`
- Branch naming convention: anything except:
  - `main`
  - `develop`
  - `release-*`
  - `hotfix-*`

Release Branches



Maintenance Branches



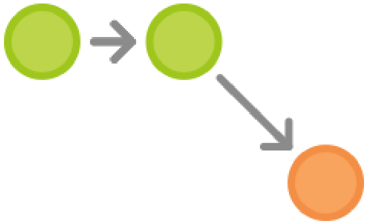
- Used to quickly patch production releases
- Upon complete it is to be merged both into `main` and `develop`

Maintenance Branches – Bests Practices

- May branch off: `main`
- Must merge back into: `main` and `develop`
- Tag: increment `patch` number
- Branch naming convention: `hotfix-*` or `hotfix/*`

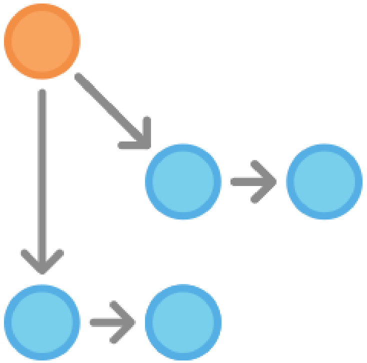
Git Flow Example

Create A Develop Branch



- Complement `main` with a `develop` branch locally and push it to the server.
- `develop` contains the project history, `main` contains an abridged version
- New developers should clone `develop` rather than `main`

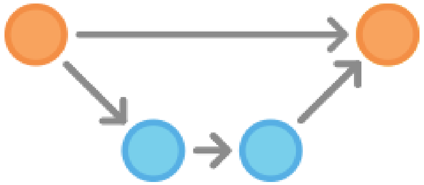
Beginning New Features



- Each developer should create a `feature` branch off of `develop`

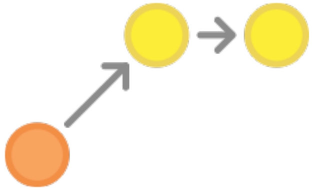
Git Flow Example

Finishing a Feature



- Once a feature is complete, the branch owner should either
  - make a pull request to have the branch merged with `develop`
  - or, merge it with their local copy of `develop` and push to the central repository

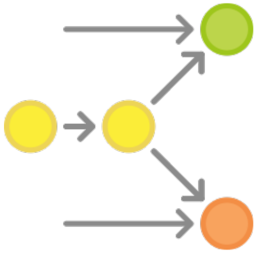
Preparing a Release



- Once ready to create a release, a new `release` branch off of `develop` should be created and named using Semantic Versioning
- The allows for cleanup of the release
- When ready it needs to be pushed to the central repository, where it becomes **feature-frozen**

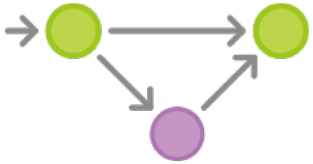
Git Flow Example

Finishing a Release



- Once ready to ship the `release` branch should be merged with both `main` and `develop`, and then it should be deleted.
- This is a great point at which to conduct a code review.
- At this point `main` should be tagged with the release version number

End-User Discovers a Bug



- End-user opens a ticket about a bug in the current release.
- To address this a new maintenance branch, aka `hotfix`, off of `main` is created
- Fixes are added and committed to the new branch and when fixed the branch is merged back into `main`
- `main` is tagged at this point with a version number updated by incrementing the `patch` number
  - v0.1.0 -> v0.1.1

Using Git Flow

- To setup a repo to be a git flow repository simply execute the `init` command:

  ```
  git flow init
  ```

  - This setups both the main and develop branches, and what naming convention will be used for the feature and hotfix branches
- Working with a feature
  - To start a new feature (e.g., "initial-implementation") execute the `feature` command:

    ```
    git flow feature start initial-implementation
    ```
  - Once you are ready to finish the feature

    ```
    git add .
    git commit -m "some commit message"
    git flow feature finish initial-implementation
    git push origin --all
    ```
  - If you are working with others and want to share your progress

    ```
    git add .
    git commit -m "some commit message"
    git flow feature publish initial-implementation
    ```
  - You can pull a feature

    ```
    git flow feature pull initial-implementation
    ```

Using Git Flow

- Time to release
  - To start a new release (ensure that the current branch is clean) for version `v0.1.0`:

    ```
    git flow release start v0.1.0
    ```
  - Once you are ready to finish the release and merge with both `main` and `develop`

    ```
    git add .
    git commit -m "some commit message"
    git flow release finish v0.1.0
    git push origin --all
    git push origin --tags
    ```
  - Again, to publish your progress:

    ```
    git add .
    git commit -m "some commit message"
    git flow release publish v0.1.0
    ```
  - You can track a release

    ```
    git flow release track v0.1.0
    ```

Using Git Flow

- Users found an issue, time for a hotfix
- To start a hotfix:

```
git flow hotfix start v0.1.1
```

- To finish a hotfix:

```
git flow hotfix finish v0.1.1
git push origin --all
git push origin --tags
```

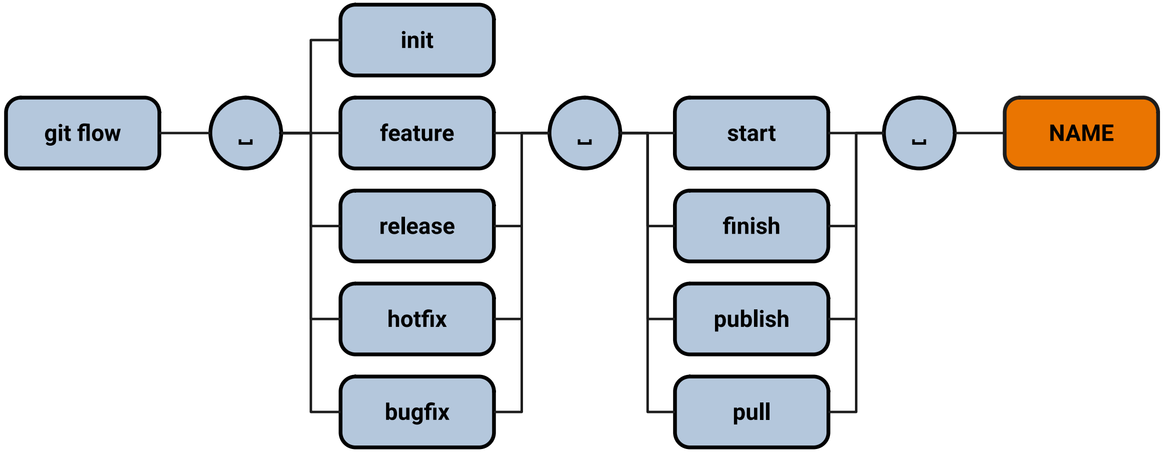
- To start a bugfix

```
git flow bugfix start v0.1.1
```

- To finish a bugfix

```
git flow bugfix finish v0.1.1
git push origin --all
git push origin --tags
```

Git Flow Reference



Resources

- Semantic Versioning
- keep a changelog
- Documenting your projects on GitHub
- A Successful Branching Model
- Atlassian's Tutorial on GitFlow
- GitFlow Cheatsheet

Summary

For Next Time

Are there any questions?