Development Infrastructructure

Development/Build/Collaboration Tools

Introduction

- Stepan Bolotnikov
- MSc Software Engineering from University of Tartu (2018)
- Software Engineer @ Proekspert

Motivation

- Development infrastructure & collaboration
- Projects are done in teams
- Need to have structures for communication
- Need to have processes automated
- Less error-prone
- More standardised, easier to introduce to people
- Easier to remember processes

Topics-tools-technologies

- Automation - gradle
- VCS - git
- Branching
- VCS hosting - Bitbucket
- Issue tracking - Bitbucket
- Pull requests - Bitbucket
- Wiki - Bitbucket

Automation - why

- Software projects include many repetitive processes
- Often need to pass arguments to commands
- Doing so by hand is error-prone
- Hard to remember
- Repetitive
- Boring

Automation - what

- Building your project
- Deploying to a server
- Generating documentation
- Running tests
- Managing dependencies
- etc

Automation - how

- Build scripts
  - Mini programs that describe the actions you wish to automate
  - Meant for a specific build script running software
- Different languages and formats for different development environments
- For Java projects, Gradle is often used

Gradle

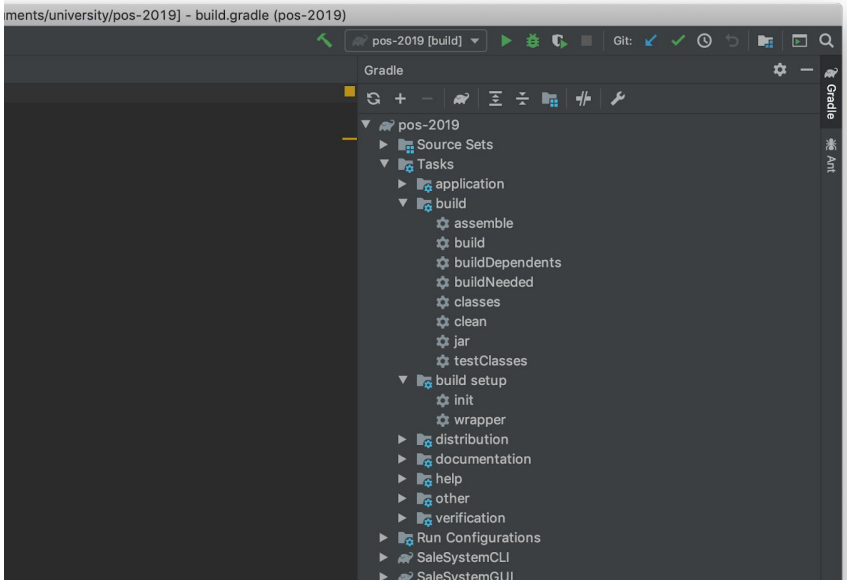
- Open source build automation system
- Builds on ideas of Ant and Maven
- Build scripts are programs written in Groovy
- Supported by popular IDEs
- Scripts can be extended with Gradle plugins
    - Many are available online

Gradle file example

```
apply plugin: ‘java’
```

- Defines tasks for a standard Java project
  - Tests
  - Building
  - Packaging
- Assumes a standard Maven-inspired directory structure
- https://docs.gradle.org/current/userguide/java_plugin.html

Running Gradle tasks



Conclusion

- Build automation is an essential part of modern software development
- Each language/environment has their own tools
- Get comfortable with yours to make your work more efficient
- Google for plugins/dependencies before inventing the wheel

Version Control System (VCS)

- As a software project progresses, the code changes.
- In a team, project changes in different ways, at different times, on different computers
- You need to know:
  - What is the latest (stable) version of the project?
  - What has changed since last stable state?
  - What changes a person has made?
  - When was a particular change made, why, and by whom?

What is a VCS

- A software tool that helps you keep track of changes in your data (code) over time.
- SVN, Mercurial, Git, ...

Git - the most popular VCS

- Released by Linus Torvalds in 2005
- Distributed version control system
- Free and open-source (written primarily in C)
- Cross-platform
- Can be used for huge complicated projects with many collaborators
  - Linux kernel
- Widely supported by popular tools
  - IDEs
  - Repository hosting providers (Bitbucket, Github, Gitlab, …)
  - GUI clients

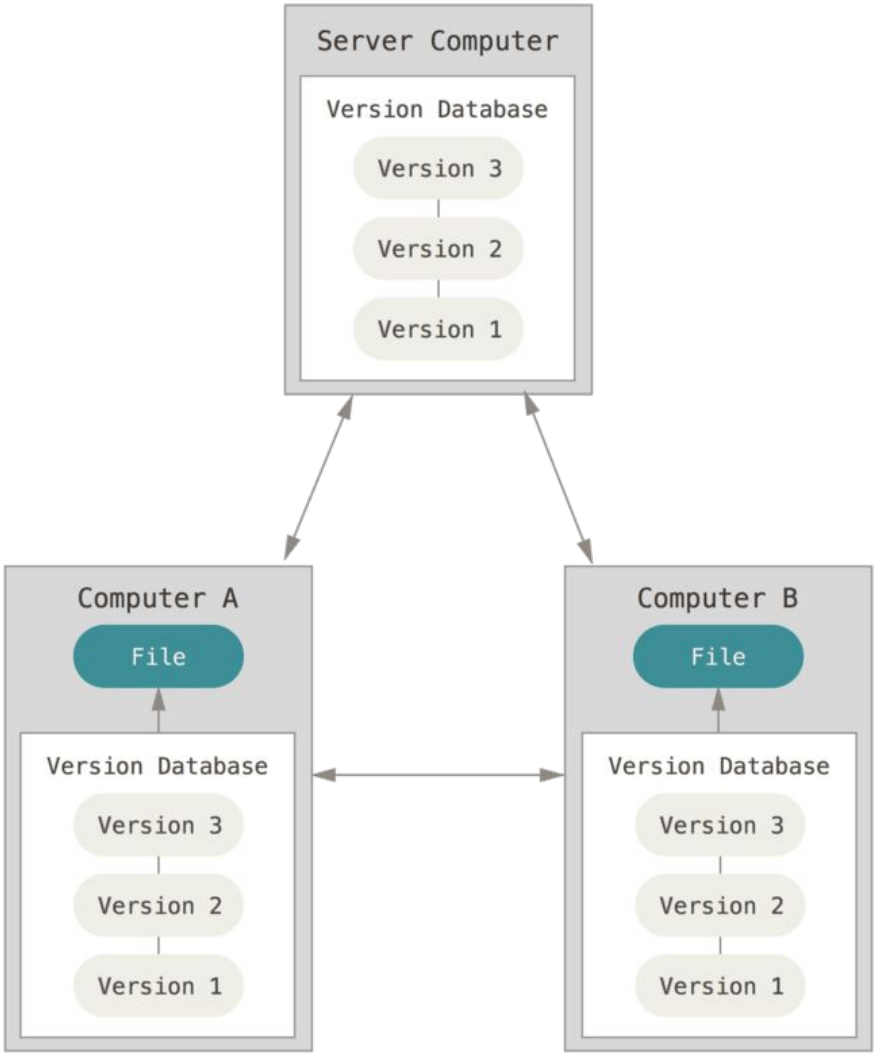
How to get started with Git command line

https://git-scm.com/

- Linux - mostly pre-installed
  - `apt-get install git-core`
- mac OS - not pre-installed
  - Run `git --version` in your Terminal application to launch wizard
- Windows - not pre-installed
  - Download from Git website

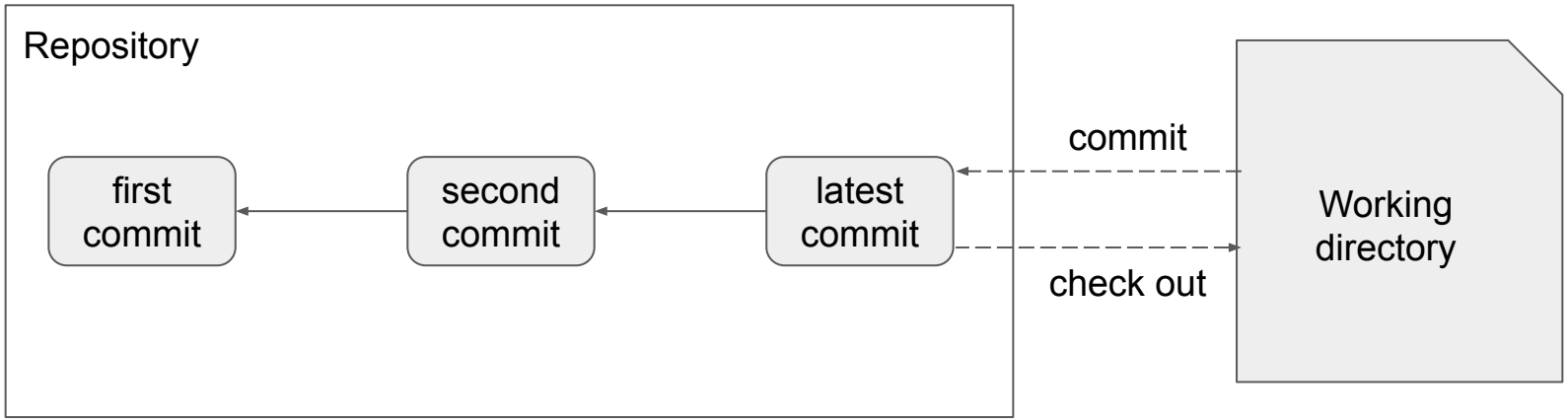
How information is stored

- The whole history of your project (a *repository*) is stored on your computer
- You can synchronize the history with other repositories (*remotes*) by sending your changes to them (*pushing*) or downloading changes (*pulling*)



How information is stored II

- A repository is a database of versions (*commits* or *revisions*).
- You can *check out* a commit into your *working directory* to work on it.
- You can *commit* your changes into the repository, creating a new commit.
- Each commit has a *parent* commit that came before it



Data in a commit

- Commit hash
  - Uniquely identifies a commit. Calculated from other contents of the commit and the hash of the parent
- Author
  - Who?
- Timestamp
  - When?
- Commit message
  - Why, what?
- Changes
  - What exactly?
- Parent commit(s)

Using Git - getting repositories

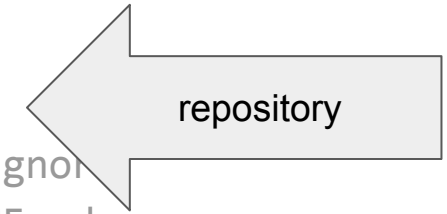
```
$ git clone git@bitbucket.org:stepan_ut/avalah.git
Cloning into 'avalah'...
remote: Counting objects: 676, done.
remote: Compressing objects: 100% (171/171), done.
remote: Total 676 (delta 91), reused 0 (delta 0)
Receiving objects: 100% (676/676), 3.58 MiB | 1.32 MiB/s, done.
Resolving deltas: 100% (301/301), done.
```

A freshly cloned repository

```
$ ls -lah
total 392
drwxr-xr-x  11 afterhours  staff   352B Oct  1 21:24 .
drwxr-xr-x   3 afterhours  staff 96B Oct  1 21:26 ..
drwxr-xr-x  12 afterhours  staff   384B Oct  1 21:26 .git
-rw-r--r--   1 afterhours  staff   110B Oct  1 21:24 .gitignore
-rw-r--r--   1 afterhours  staff   736B Oct  1 21:24 README.md
drwxr-xr-x  17 afterhours  staff   544B Oct  1 21:24 dist
-rw-r--r--   1 afterhours  staff   102K Oct  1 21:24 package-lock.json
-rw-r--r--   1 afterhours  staff   858B Oct  1 21:24 package.json
drwxr-xr-x   4 afterhours  staff   128B Oct  1 21:24 src
drwxr-xr-x   8 afterhours  staff   256B Oct  1 21:24 voog
-rw-r--r--   1 afterhours  staff 78K Oct  1 21:24 yarn.lock
```

A freshly cloned repository

```
$ ls -lah
total 392
drwxr-xr-x  11 afterhours  staff   352B Oct  1 21:24 .
drwxr-xr-x   3 afterhours  staff   96B Oct  1 21:26 ..
drwxr-xr-x  12 afterhours  staff   384B Oct  1 21:26 .git
-rw-r--r--   1 afterhours  staff   110B Oct  1 21:24 .gitignor
-rw-r--r--   1 afterhours  staff   736B Oct  1 21:24 README.md
drwxr-xr-x  17 afterhours  staff   544B Oct  1 21:24 dist
-rw-r--r--   1 afterhours  staff   102K Oct  1 21:24 package-lock.json
-rw-r--r--   1 afterhours  staff   858B Oct  1 21:24 package.json
drwxr-xr-x   4 afterhours  staff   128B Oct  1 21:24 src
drwxr-xr-x   8 afterhours  staff   256B Oct  1 21:24 voog
-rw-r--r--   1 afterhours  staff   78K Oct  1 21:24 yarn.lock
```



Using Git - inspecting history

$ git log

```
commit d9ee1e7b1b2df9cb0fc14cbc648451c1ba073044 (HEAD -> master, origin/master,
origin/HEAD)
Author: Stepan Bolotnikov <stoopa@gmail.com>
Date:   Sun Aug 11 20:30:48 2019 +0300

    Updated source from Voog Kit, updated README, added Gmaps api key

commit 3dbc00a38aa56eab6d6c4b528dc52cd7a22c0cd5
Author: Stepan Bolotnikov <stepan.bolotnikov@guardtime.com>
Date:   Mon Nov 5 01:25:05 2018 +0200

    Cleaned up CSS
```

Using Git - status

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

- Status is possibly the most important command in Git
- Crucial to knowing what is the state of your local repository and working copy
- Run it as much as possible to avoid errors and confusion

Status after creating a new file

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    test.txt

nothing added to commit but untracked files present (use "git add" to
track)
```

Using Git - creating commits

- With the new file added, let’s make a commit

```
$ git commit
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
    test.txt

nothing added to commit but untracked files present
```

But first - possible states of a file in Git

- Torvalds called Git “a stupid content tracker”
- Doesn’t do a lot of things for you
- Most actions need to be explicitly specified
- Including showing exactly what parts of what files Git should add to a new commit
- A change can exist in one of three states:

1. Up to date - just like it is in the repository
2. Changed - something about the file has been changed by you
3. Staged - something has been changed and you want to commit the changes

Using Git - adding a change

```
$ git add test.txt

$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   test.txt
```

Using Git - creating commits

```
$ git commit
[master 3ab10af] Added file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 test.txt
```

Commits are local

- By default, Git doesn’t send your commits to other repositories

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Using Git - synchronizing commits

```
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 277 bytes | 277.00 KiB/s, done.
Total 3 (delta 1), reused 1 (delta 0)
To bitbucket.org:stepan_ut/avalah.git
   d9ee1e7..3ab10af  master -> master
```

Using Git - updating your repository

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From bitbucket.org:stepan_ut/avalah
   3ab10af..f526ea5  master  -> origin/master
Updating 3ab10af..f526ea5
Fast-forward
 test2.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 test2.txt
```

Lifehack - add changes line by line

- When working with many changes, it’s easy to lose track
- Avoid unnecessary additions, debugging, typos, empty lines
- Get a reminder on what exactly you changed to write a better commit message

```
$ git add -p .
```

- Shows you each line that you’ve changed(/added/removed), lets you chose whether you want to stage it or not

Basic commands

- These commands you will use 80% of the time
- Enough for managing a 1-person project and reap the benefits of Git
- What changes when working in a team?

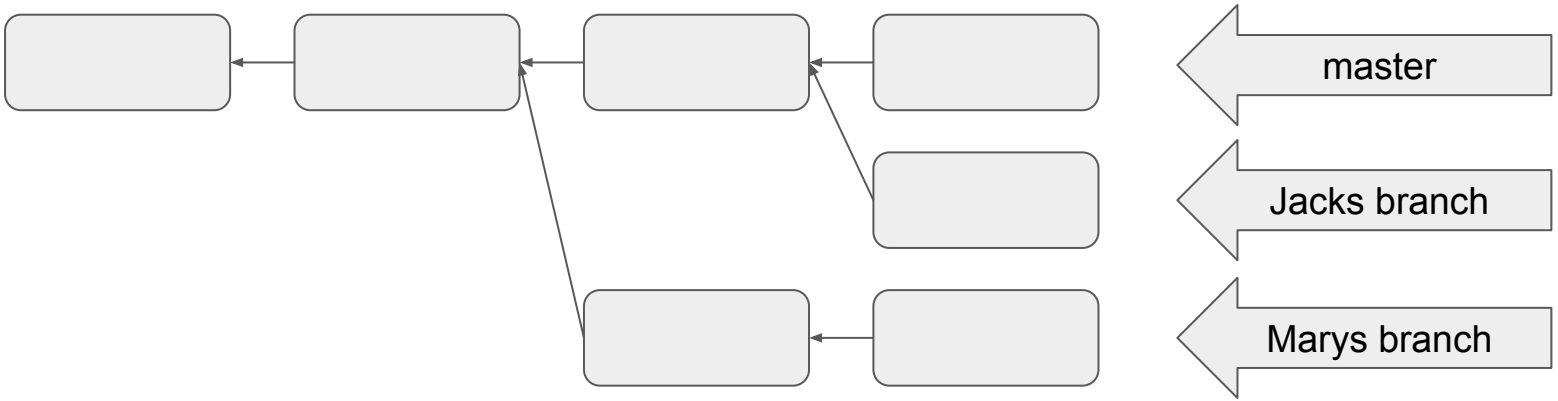
```
git clone
git pull
git add
git commit
git push
git log
```

How to effectively manage a team project

- Ideally, different people work on different features
- Need to separate your work from that of others
- Need to be able to only add your work when it’s done

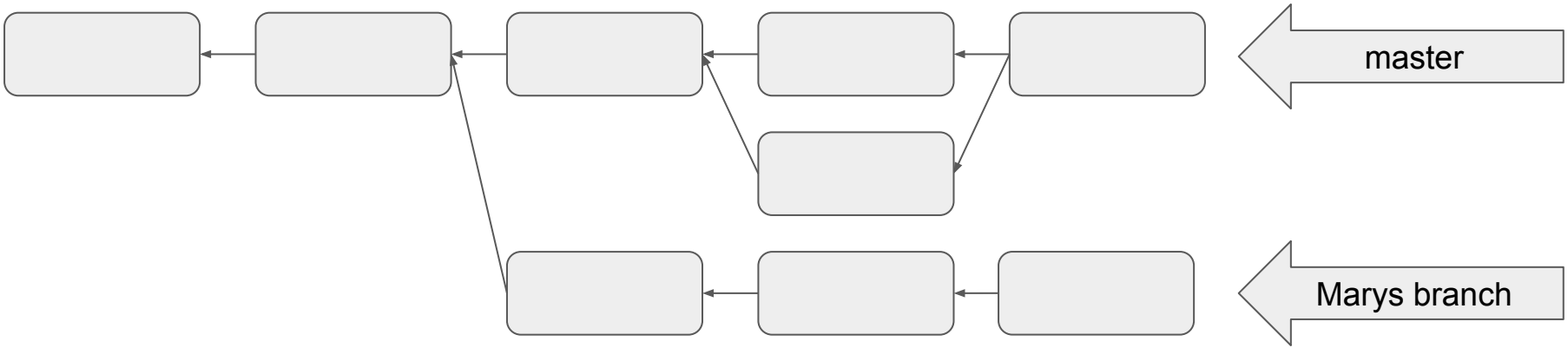
The other 15%: branches

- A *branch* is where the history of your project diverges in two or more directions.
- In Git, branches have names for ease of use
- The main branch, created by default, is usually named *master*



Merging branches

- When a feature on a branch is ready, it can be *merged* into another branch, uniting the histories and code.
- After this, the temporary branch can be deleted



Using Git - branches

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Listing branches

```
$ git branch
* master
```

Adding branches

```
$ git branch new-branch
```

Populated branch list

```
$ git branch
* master
  new-branch
```

Changing branches

```
$ git checkout new-branch
Switched to branch 'new-branch'
$ git status
On branch new-branch
nothing to commit, working tree clean
$ git branch
  master
* new-branch
```

Remember - “stupid content tracker”

- Git will not do many branch-related things for you:
  - Commits on a branch will not be available on other branches until you merge
  - Branches other than the active one will not be pushed
- Be mindful of where you are and what you’re doing

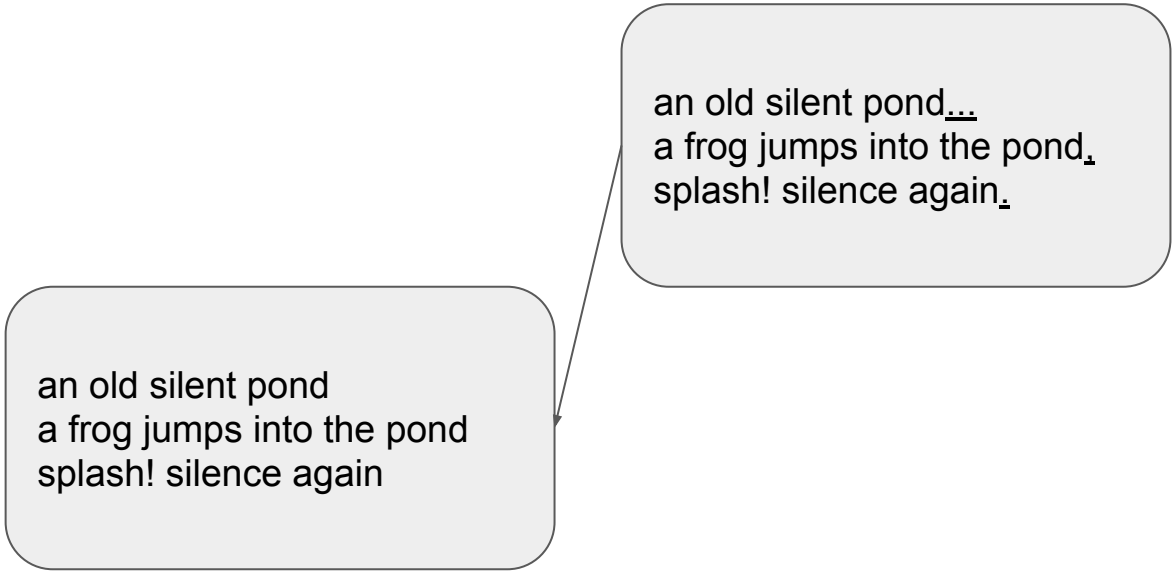
Merging branches

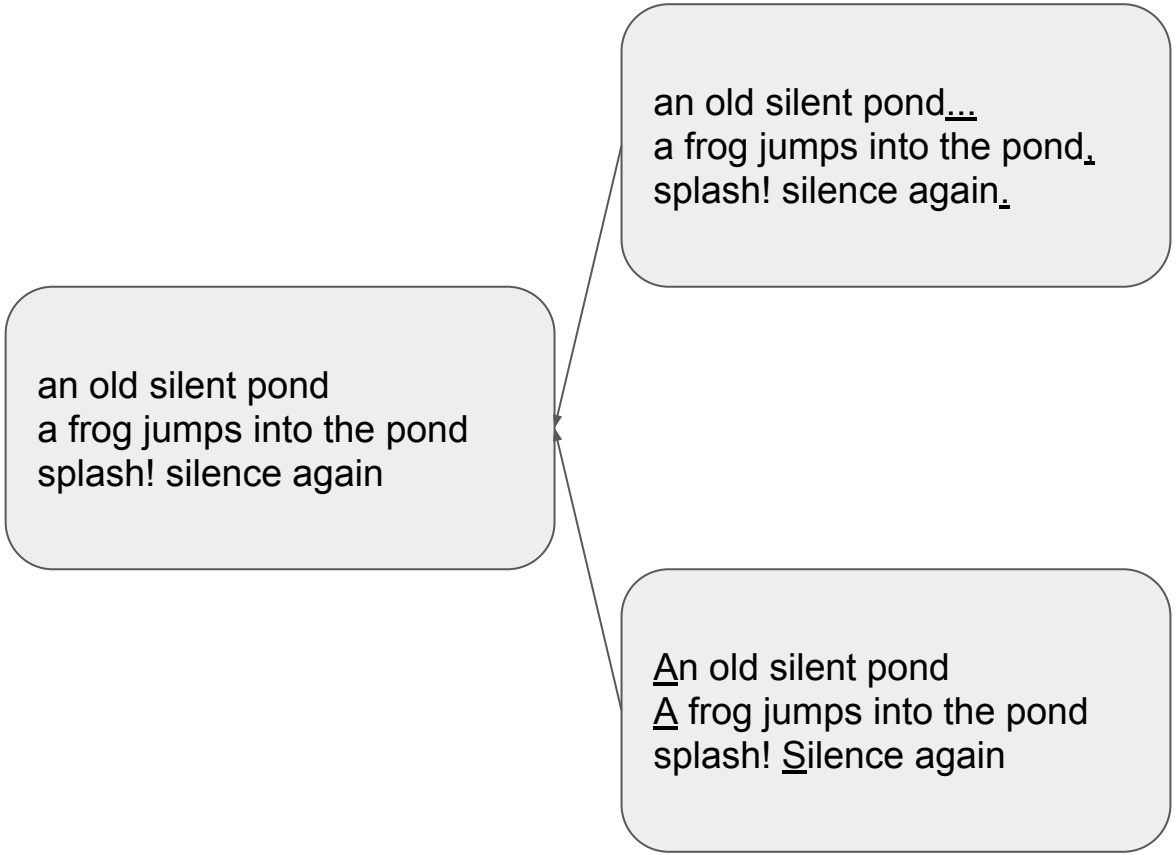
```
$ git status
On branch master
nothing to commit, working tree clean
$ git merge new-branch
Updating f526ea5..5d46150
Fast-forward
 a.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 a.txt
```

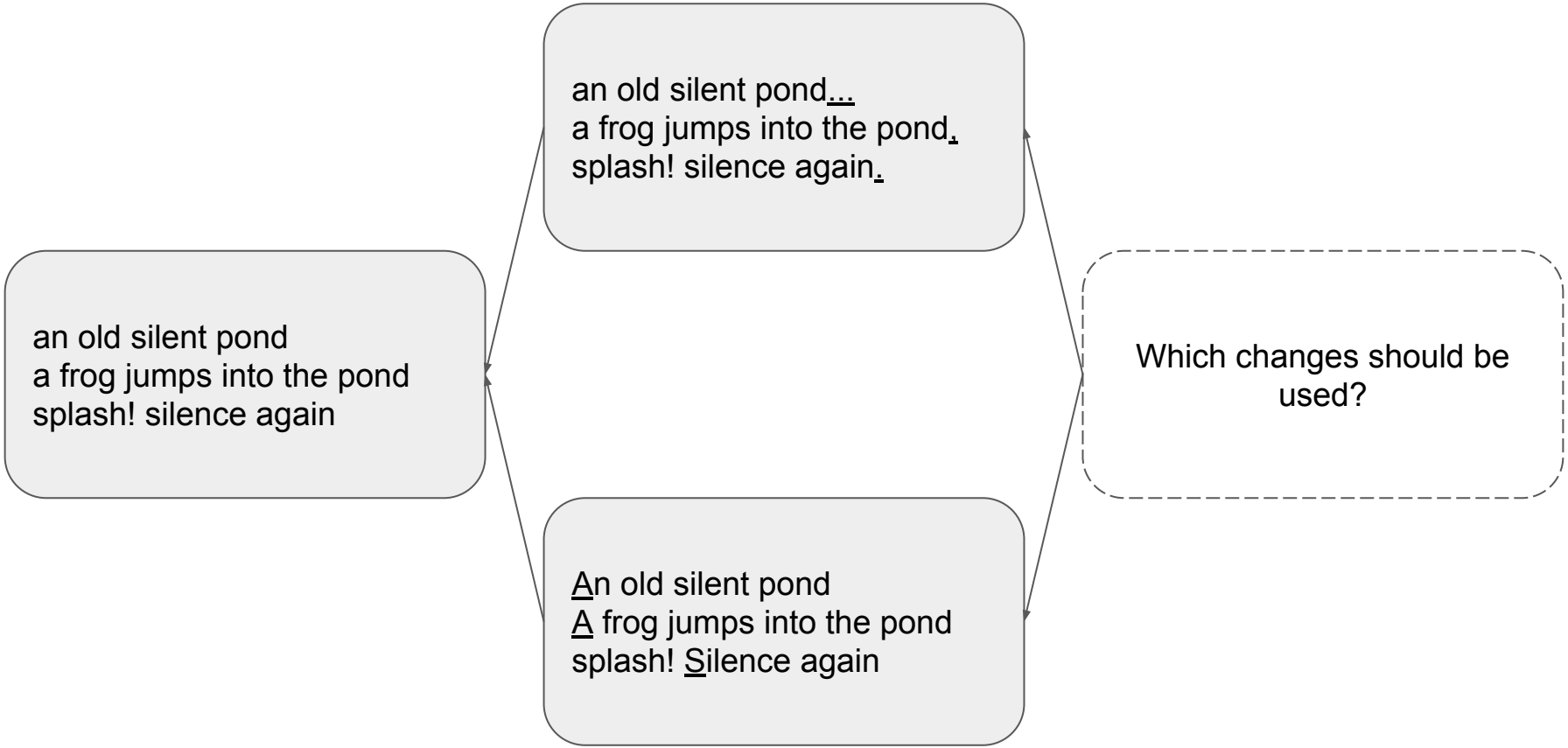
Merge conflicts

- In a perfect world, that would be it
- No real project is perfect
- Git gets confused when the same parts of the files are changed in different ways on different branches

an old silent pond
a frog jumps into the pond
splash! silence again


an old silent pond...
a frog jumps into the pond,
splash! silence again.
an old silent pond
a frog jumps into the pond
splash! silence again


an old silent pond
a frog jumps into the pond
splash! silence again
an old silent pond...
a frog jumps into the pond,
splash! silence again.
An old silent pond
A frog jumps into the pond
splash! Silence again


an old silent pond...
a frog jumps into the pond,
splash! silence again.
an old silent pond
a frog jumps into the pond
splash! silence again
Which changes should be used?
An old silent pond
A frog jumps into the pond
splash! Silence again

Merge conflict

```
$ git merge new-branch
Auto-merging haiku.txt
CONFLICT (content): Merge conflict in haiku.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Status in conflicted state

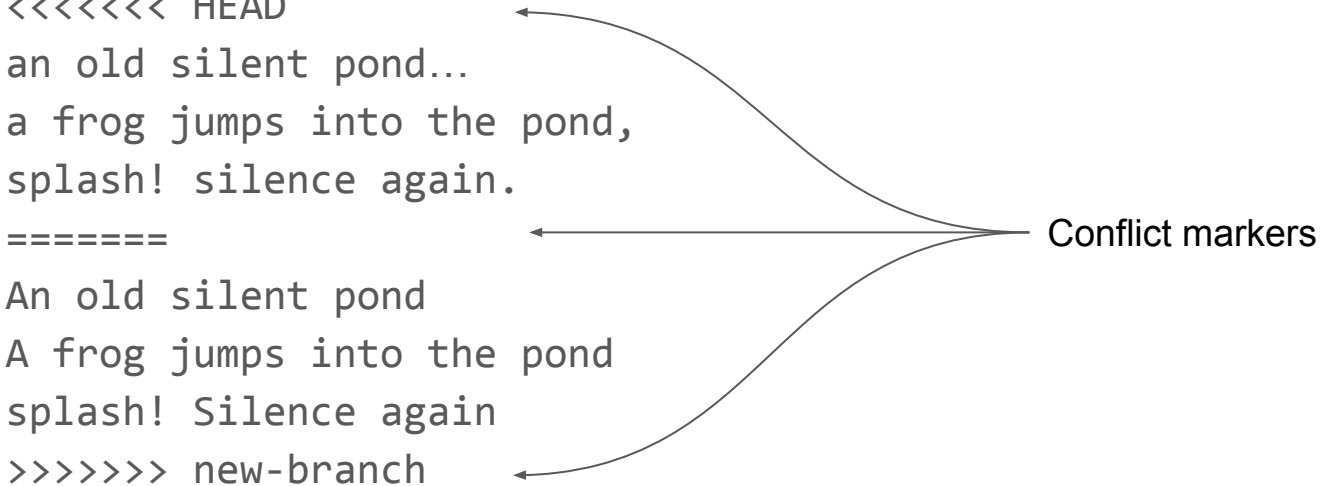
```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:   haiku.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Haiku.txt in conflicted state

```
<<<<<<< HEAD
an old silent pond…
a frog jumps into the pond,
splash! silence again.
=======
An old silent pond
A frog jumps into the pond
splash! Silence again
>>>>>>> new-branch
```

Resolving a merge conflict

- Find conflicts in files
- Decide what is the desired final state
- Delete the conflict markers, leaving only the desired state
- Add the resolved files and commit


<<<<<<< HEAD
an old silent pond…
a frog jumps into the pond,
splash! silence again.
=======
An old silent pond
A frog jumps into the pond
splash! Silence again
>>>>>>> new-branch
Conflict markers

```
<<<<<<< HEAD
an old silent pond…
a frog jumps into the pond,
splash! silence again.
=======
An old silent pond
A frog jumps into the pond
splash! Silence again
>>>>>>> new-branch
```

```
<<<<<<< HEAD
an old silent pond…
a frog jumps into the pond,
splash! silence again.
=======
An old silent pond
A frog jumps into the pond
splash! Silence again
>>>>>>> new-branch
```

Changes on branch that we merge

Haiku.txt with resolved conflict

```
An old silent pond…
A frog jumps into the pond,
splash! Silence again.
```

```
$ git add haiku.txt
$ git commit
[master 4b1bee1] Merge branch 'new-branch'
```

Branching commands

- This covers probably 95% of typical Git usage
- There is much more
  - Aliases
  - Rebasing
  - Cherry-picking
  - Bisecting
  - Git hooks
  - Traversing history and undoing changes

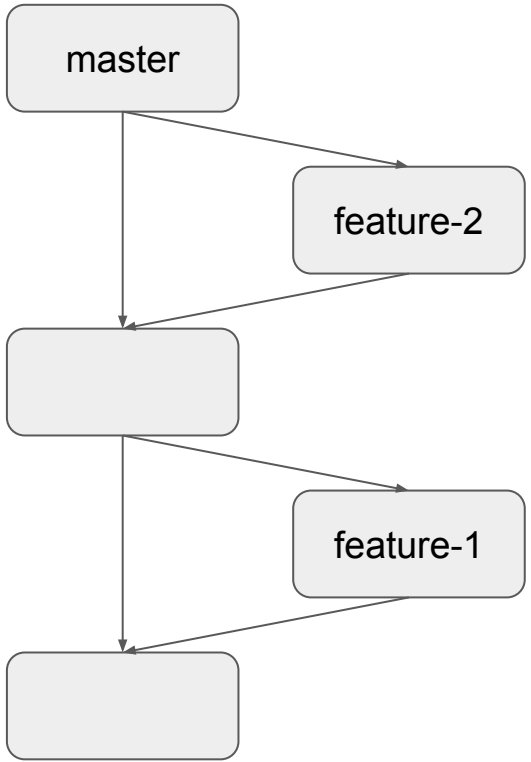
```
git branch
git branch branch-name
git checkout
git merge
```

Branching models

- To facilitate collaboration and ensure consistent practice, companies often enforce a branching model
- Describes:
  - How branches are named
  - When and how they’re created
  - When and how they’re merged
- Depends on project/team

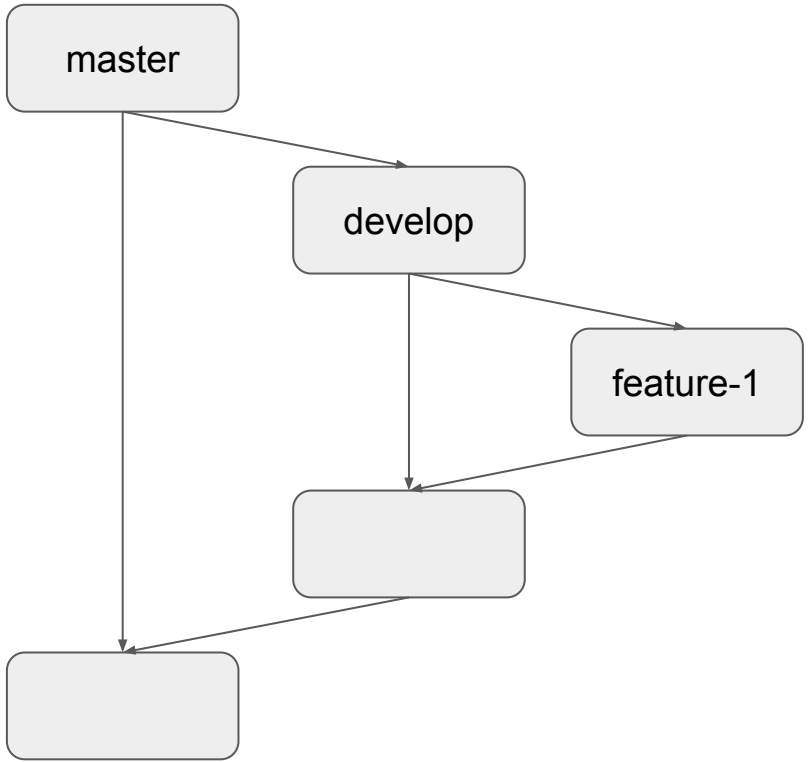
A simple branching model

- `master` always contains stable state
- Branches for new features are created from `master`
- Branches are merged into master when feature has been tested



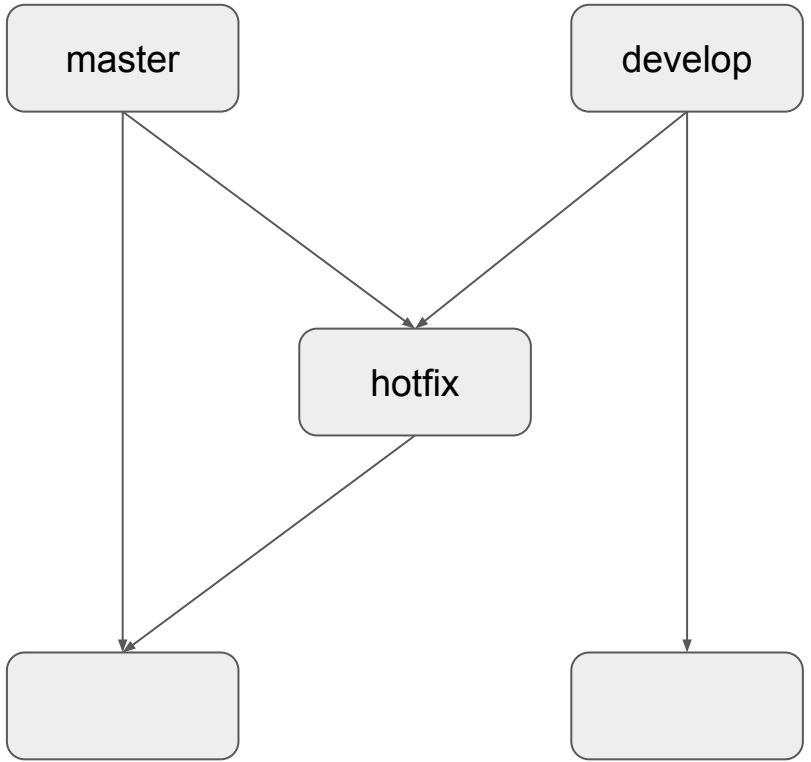
A simple branching model

- `master` always contains released state
- `develop` always contains stable state
- Feature branches are created from `develop`
- Feature branches are merged into `develop` when feature is tested
- `develop` is merged into `master` when it's time for a new release



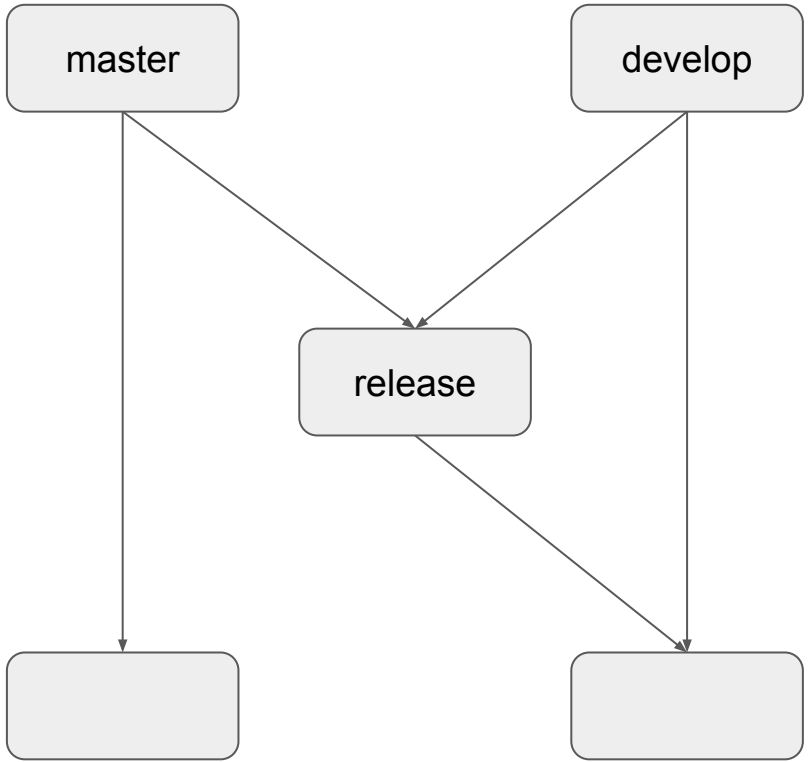
Hotfix branches

- Created from **`master`** for critical, small fixes
- Merged into **`master`** and **`develop`** when done
- = changes are both in the released state and in current development



Release branches

- Created from **`develop`** when it's time to prepare for release
- Used for final tests of feature-set
- Can commit bug-fixes to them
- When ready for release, merge into **`master`** and **`develop`**
- = isolate release from ongoing work



Complexity depends on project

- A branching model can have other types of branches
- Depends on the complexity of project/team/company
- Can enforce naming conventions
  - I.e. feature/<feature_number>_<short-description>
- For a small project, small team, enough:
  - `master` branch with stable, shippable state
  - Separate feature branches

Recap

- A VCS like Git helps you track changes in your project and collaborate
- Git is the most popular modern VCS enabling powerful collaboration in big projects
- Git is “stupid” and doesn’t do many things for you
- Use **`git status`** a lot, be aware of where you are, and what you’re doing

Repository hosting

- Typically, a repository is backed up to a server
- Considered the source of truth
- All changes are sent there and all team members get updates from there
- Often have additional features
  - More comfortable history viewing
  - Automation, CI/CD (next lecture!)
  - Documentation
  - Issue tracking

Bitbucket issue tracker

- Needs to be enabled in repository settings
- A relatively simple issue tracking solution
- Tightly integrated with your Git repository
- Can be made public or private
- In a public tracker, anyone can report issues
    - For open source projects

Anatomy of an issue

- Title - for quick overview
- Description - detailed description of work to be done
- Assignee - who is expected to work on the issue
- Kind - separate bugs from feature requests
- Priority - help developers pick more crucial tasks
- Component (optional) - separate issues by areas of the project
- Milestone (optional) - during what time should the issue be completed
- Version (optional) - in what version of the project is the issue completed

Life-cycle of an issue

- Issue tracking software typically defines several possible states of an issue
- Bitbucket has some default states

- Open → In progress → Resolved/Cancelled

Integrating issues with commits

- Often, issue tracking software lets you integrate issues with your Git repository
- This way, you can easily see what commits relate to what issues
- For this, include a specific syntax in your message:
  - https://confluence.atlassian.com/bitbucket/resolve-issues-automatically-when-users-push-code-221451126.html

Author	Commit	Message
 Stepan Bolotnikov	5df2036	see #1

Stepan Bolotnikov REPORTER

see ~~#1~~

→ <<cset 5df20367d5a3>>

Edit • Pin to top • Mark as spam • Delete • 3 minutes ago

Pull Requests (PR)

- Signal to maintainer of the repository
- “Please pull my changes (branch) into the repository”
- Sometimes called a merge request
- “Please merge my changes into the given branch”
- Gives a maintainer overview of what is being done
- One PR per one branch
- Crucial part of collaboration in modern software projects

Collaborating on projects - full life-cycle of a feature

1. Reporting an issue
2. Assigning an issue
3. Creating a new feature branch for the issue
4. Working on the issue
5. Creating a Pull Request for the issue
6. Waiting for approvals
7. Merging the Pull Request

Rinse and repeat

Bitbucket wiki

- A place to document your project
- Can create many pages, link them together
- Uses Markdown for rich text styling
- Is contained in a VCS repository, so you can see history and roll back changes

Topics covered today

- Automation, Gradle
- VCS, Git, branching
- Branching models
- Bitbucket issues, pull requests
- Collaboration flows

Questions?