Lecture 2: Getting started with Git: configuration, repositories and staging
Last time

● Basics of VCS
● History of VCS
● Practice session: installing Git, first repository and first commit
Today

At the end of today’s lesson, you will be able to:

- Edit the Git configuration to set the default editor and add aliases
- Create a new repository
- Clone a repository
- Configure linked repositories
- Stage a change
- Reset a change
- Commit a change
- Pull changes
- Push changes
Git configuration

Git is usable out of the box with (almost) no configuration. Often it’s useful to customise Git to your needs. That’s where configuration comes in.
What can you configure

https://git-scm.com/docs/git-config#_variables
“...this list is non-comprehensive and not necessarily complete.”

External tools (Git GUI etc) can have their own configuration parameters.

Most of the variables you never ever need to touch.

“...git is not so much a version control system as it is a tool for building your own version-controlled workflow.”
(https://importantshock.wordpress.com/2008/08/07/git-vs-mercurial/)
Popular things to configure

- User credentials
- Editors
- Aliases
- Colors and output format
- HTTP and SSH settings
Three levels of configuration

1. Local, repository-specific
2. Global, user-specific
3. System-wide

Overrides
Two ways to change configuration

1. Manually change configuration file
2. The git-config tool
Configuration file syntax

#comment
[section] ; comment
  variable=1 ; integer
  key=true ; boolean
[section "subsection"]
  variable=/path/to/ ; pathname
  key ; true
  color=yellow ; color
[include]
  path=/path/to/config
Configuration file locations

- **Local**
  
  /path/to/project/.git/config

- **Global**
  
  /Users/username/.gitconfig

- **System**
  
  /path/to/git/install/gitconfig
Git-config example from last time

$ git config --global user.name "Your Name"
Git-config example from last time

$ git config --global user.name "Your Name"

Level

Key

Value
Git-config example from last time

$ git config --global user.name "Your Name"

“In all my repositories, use “Your name” as the user’s name”
The git-config tool

- **List**
  - $ git config -l

- **Edit**
  - $ git config -e

- **Set**
  - $ git config --system <key> <value>
  - $ git config --global <key> <value>
  - $ git config --local <key> <value> or git config <key> <value>

- **Get**
  - $ git config <key>

- **Unset**
  - $ git config --unset <key>

- **Replace all, remove and replace sections etc**
Aliases

- Allow you to make alternative names for Git commands
- For speed
  - `git co` vs `git checkout`
- To group common arguments
  - `git patch` vs `git add --patch`
- Personal preference
  - `git update` vs `git pull`
Creating aliases

$ git config alias.co checkout
$ git co ➞ git checkout

- Aliases pass arguments to the original commands

$ git co path/to/file
External aliases

- Can also be used for external shell commands

$ git config alias.all '!gitx --all'
$ git all ⇒ gitx --all

- External aliases are executed from the root directory of the repository
Common aliases

[alias]
co=checkout
ci=commit ; check-in
br=branch
st=status
up=push
dn=pull

https://github.com/GitAlias/gitalias
https://git.wiki.kernel.org/index.php/Aliases
Repository creation

- Last time: github.com → “New repository”
- Usually: Github/Gitlab/Bitbucket/Gogs/etc → ”New repository”
- What happens behind the curtains?
- How to do it without an external Git-based service?
- What if you have already created the project?
- What do command-line wizards do?
Cloning vs initializing

- Last time:

  $ git clone path/to/repository

- Doesn’t create a repository
- Copies an entire existing repository from a given location
- AND checks out the latest version
Git-init tool

$ git init

- “Initializes an empty Git repository”
- Creates a .git directory with all the necessary internals.
- Use in the project directory (working copy)
- Safe to run in an existing project directory
Initializing a bare repository

$ git init --bare

- Creates the **internals** of the `.git` directory
- No working copy associated with the new repository
- Mainly used on “servers” - where repositories are backed up to
Scenario 1

Creating a repository on remote server, getting in to local machine
Scenario 1: on the remote server

1. $ cd /path/to
2. $ mkdir project
3. $ cd project
4. $ git init --bare

    Initialized empty Git repository in /path/to/project
Scenario 1: on the local machine

1. $ cd /where/to/clone
2. $ git clone server.com:/path/to/project
   Cloning into ‘project’...
   Warning: You appear to have cloned an empty repository.
   done.

Result: /where/to/clone/project contains the cloned repository that is linked to repository on server.com
Scenario 2

Creating a repository for an existing project on local machine, getting it to remote server
Scenario 2: on the remote server

1. $ cd /path/to
2. $ mkdir project
3. $ cd project
4. $ git init --bare
   Initialized empty Git repository in /path/to/project
Scenario 2: on the local machine

1. $ cd /path/to/project
2. $ git init
   Initialized empty Git repository in /path/to/project
3. $ git remote add origin server.com:/path/to/project
Scenario 2: on the local machine

4. $ git commit -a -m "Initial commit"
   [master (root-commit) 737513b] Initial commit
   1 file changed, 0 insertions(+), 0 deletions(-)
   create mode 100644 test.txt

5. $ git push origin master
   Counting objects: 3, done.
   Writing objects: 100% (3/3), 217 bytes | 0 bytes/s, done.
   Total 3 (delta 0), reused 0 (delta 0)
   To /path/to/project/
   * [new branch] master -> master
Git-remote tool

- Every repository can track one or more remote repositories that you can synchronise with
- Necessary for git-push; git-pull and others
- Remote has a name; addresses for fetching and pushing (usually the same)
List remotes

$ git remote -v

origin  server.com:/path/to/project  (fetch)
origin  server.com:/path/to/project  (push)
staging other.com:/different/path  (fetch)
staging other.com:/different/path  (push)
Add remote

$ git remote add <name> <url>

URL can be address on another machine or another local repository
Choosing what remote to use

- Normally, repositories default to remote “origin” and branch “master”
- Can be specified in arguments:

  $ git pull <remote> <branch>
  $ git push <remote> <branch>

- If new branch/empty repository, need to set default remote on first push:

  $ git push --set-upstream origin master
Intermission

Until now:

- Git configuration
- Creating repositories
- Cloning repositories
- Remotes

Next:

- What to do with changes
Lifecycle of a file in Git

- Untracked
- Unmodified
- Modified
- Staged
- Committed
- Synchronised

- git pull
- Add the file: `git add`
- Edit file
- Stage file: `git add`
- Reset the file
- git commit
- git commit
- git push
- Remove the file
Git-status tool

- Shows the differences between current working copy and latest state in the (LOCAL!) repository
- Often aliased to “st”
Sample git-status output

$ 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)
      modified:   voog/layouts/reference.tpl

Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)
      modified:   src/scss/_fonts.scss

Untracked files:
    (use "git add <file>..." to include in what will be committed)
      src/javascripts/a.txt
## Sample git-status output

```bash
$ git status
```

<table>
<thead>
<tr>
<th>On branch master</th>
<th>Branch information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your branch is up-to-date with 'origin/master'.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Changes to be committed:</th>
<th>Staged</th>
</tr>
</thead>
<tbody>
<tr>
<td>(use &quot;git reset HEAD &lt;file&gt;...&quot; to unstage)</td>
<td></td>
</tr>
<tr>
<td>modified: voog/layouts/reference.tpl</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Changes not staged for commit:</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>(use &quot;git add &lt;file&gt;...&quot; to update what will be committed)</td>
<td></td>
</tr>
<tr>
<td>(use &quot;git checkout -- &lt;file&gt;...&quot; to discard changes in working directory)</td>
<td></td>
</tr>
<tr>
<td>modified: src/scss/_fonts.scss</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Untracked files:</th>
<th>Untracked</th>
</tr>
</thead>
<tbody>
<tr>
<td>(use &quot;git add &lt;file&gt;...&quot; to include in what will be committed)</td>
<td></td>
</tr>
<tr>
<td>src/javascripts/a.txt</td>
<td></td>
</tr>
</tbody>
</table>
Sample git-status output

$ 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)
    modified:  voog/layouts/reference.tpl

Changes not staged for commit:

  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:  src/scss/_fonts.scss

Untracked files:

  (use "git add <file>..." to include in what will be committed)
    src/javascripts/a.txt
Lifecycle of a file in Git

1. **Untracked**
2. **Unmodified**
3. **Modified**
4. **Staged**
5. **Committed**
6. **Synchronised**

- Add the file: `git add`
- Edit file
- Stage file: `git add`
- Reset the file
- Remove the file
- **git pull**
- **git commit**
- **git push**
Staging changes: git-add tool

$ git add <path>

- Path can be a file, a directory or wildcard

$ git add .

$ git add images/*.png

- Currently staged snapshot is formally known as the “index”
Adding by hunks

$ git add --patch <path>

- Interactively goes through modified files
- Allows you to decide for each hunk if it should be staged or not
- Hunk is a line difference in a file
- Lets you only commit only what needs to be committed
- Reminder of what you’ve changed
Adding by hunks: sample output

$ git add -p

diff --git a/src/scss/_fonts.scss b/src/scss/_fonts.scss
index 6351aef..8e116dd 100644
--- a/src/scss/_fonts.scss
+++ b/src/scss/_fonts.scss
@@ -1,2 +1,5 @@
+@import url('https://fonts.googleapis.com/css?family=Open+Sans:300');
  @import url('https://fonts.googleapis.com/css?family=Roboto+Condensed:700');
+body {
+  background: white;
+}

Stage this hunk [y,n,q,a,d,/,s,e,?]?
Adding by hunks: sample output

```bash
$ git add -p
```

<table>
<thead>
<tr>
<th>File</th>
<th>Hunk</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff --git a/src/scss/_fonts.scss b/src/scss/_fonts.scss</td>
<td>@import url('<a href="https://fonts.googleapis.com/css?family=Open+Sans:300,400">https://fonts.googleapis.com/css?family=Open+Sans:300,400</a>');</td>
<td>Stage this hunk [y,n,q,a,d,/,s,e,?]</td>
</tr>
<tr>
<td>index 6351aef..8e116dd 100644</td>
<td>@import url('<a href="https://fonts.googleapis.com/css?family=Open+Sans:300">https://fonts.googleapis.com/css?family=Open+Sans:300</a>');</td>
<td></td>
</tr>
<tr>
<td>--- a/src/scss/_fonts.scss</td>
<td>@import url('<a href="https://fonts.googleapis.com/css?family=Roboto+Condensed:700">https://fonts.googleapis.com/css?family=Roboto+Condensed:700</a>');</td>
<td></td>
</tr>
<tr>
<td>+++ b/src/scss/_fonts.scss</td>
<td>body {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>background: white;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
```
Checking in: git-commit tool

$ git commit -m “Commit message”

- Commits the currently stashed changes (the “index”) to the repository with a given message
- -m can be omitted - then Git opens the default editor (git config core.editor) where you can write the message
- Often aliased to “ci” (check-in)
Splitting hunks

$ git add -p

diff --git a/src/scss/_fonts.scss b/src/scss/_fonts.scss
index 6351aef..8e116dd 100644
--- a/src/scss/_fonts.scss
+++ b/src/scss/_fonts.scss
@@ -1,2 +1,5 @@
+@import url('https://fonts.googleapis.com/css?family=Open+Sans:300');
 @import url('https://fonts.googleapis.com/css?family=Roboto+Condensed:700');
+body {
+  background: white;
+}

Stage this hunk [y,n,q,a,d,/,s,e,?]?
Splitting hunks

$ git add -p

diff --git a/src/scss/_fonts.scss b/src/scss/_fonts.scss
index 6351ae0..8e116dd 100644
--- a/src/scss/_fonts.scss
+++ b/src/scss/_fonts.scss
@@ -1,2 +1,5 @@
+@import url('https://fonts.googleapis.com/css?family=Open+Sans:300');
+@import url('https://fonts.googleapis.com/css?family=Roboto+Condensed:700');
+body {
+  background: white;
+
+
}

Stage this hunk [y,n,q,a,d,/,s,e,?]?
Splitting hunks

$ git add -p

diff --git a/src/scss/_fonts.scss b/src/scss/_fonts.scss
index 6351aef..8e116dd 100644
--- a/src/scss/_fonts.scss
+++ b/src/scss/_fonts.scss
@@ -1,2 +1,5 @@
+@import url('https://fonts.googleapis.com/css?family=Open+Sans:300');
 +@import url('https://fonts.googleapis.com/css?family=Roboto+Condensed:700');
+body {
+  background: white;
+}

Stage this hunk [y,n,q,a,d,/,s,e,?]? s
Splitting hunks

Stage this hunk [y,n,q,a,d,/,s,e,?]? s
Split into 2 hunks.
@@ -1,2 +1,2 @@
+@import url('https://fonts.googleapis.com/css?family=Open+Sans:300');
   @import  
Stage this hunk [y,n,q,a,d,/,s,e,?]?
Editing hunks

- Sometimes split can’t split where you want it to (consecutive lines)
- Sometimes you discover that you need to edit one word/character on one line
# Manual hunk edit mode -- see bottom for a quick guide.
@@ -1,2 +1,5 @@
+@import url('https://fonts.googleapis.com/css?family=Open+Sans:300');
 @import
+body {
+  background: white;
+}
# ---
# To remove '-' lines, make them ' ' lines (context).
# To remove '+' lines, delete them.
# Lines starting with # will be removed.
#
# If the patch applies cleanly, the edited hunk will immediately be
# marked for staging.
# If it does not apply cleanly, you will be given an opportunity to
# edit again. If all lines of the hunk are removed, then the edit is
# aborted and the hunk is left unchanged.
# Manual hunk edit mode -- see bottom for a quick guide.
@@ -1,2 +1,5 @@
+@import url('https://fonts.googleapis.com/css?family=Open+Sans:300');
  @import url('https://fonts.googleapis.com/css?family=Roboto+Condensed:700');
+body {
+  background: white;
+}
#
# ---
# To remove '-' lines, make them ' ' lines (context).
# To remove '+' lines, delete them.
# Lines starting with # will be removed.
#
# If the patch applies cleanly, the edited hunk will immediately be
# marked for staging.
# If it does not apply cleanly, you will be given an opportunity to
# edit again. If all lines of the hunk are removed, then the edit is
# aborted and the hunk is left unchanged.
--Patch elsewhere

- `-p/--patch` can be used in reverse
- By `git-checkout` (in working copy) and `git-reset` (in index)
- Allows selectively discarding changes in current working tree
Lifecycle of a file in Git

1. Untracked
2. Unmodified
3. Modified
4. Staged
5. Committed
6. Synchronised

- git pull
- Add the file: git add
- Edit file
- Stage file: git add
- Reset the file
- git commit
- git commit
- Remove the file
- git push
Committing best practices

- Divide work into smaller commits
- Bigger chunks of work should be grouped in a separate branch
- Commits should be **atomic**
  - Commit each fix or task as a separate change
  - Each commit should have only one fix or task
  - Only commit when a block of work is complete
  - Do not commit broken and incomplete code
- Leave an informative commit message
- Never commit commented out code
What files to never commit

- Packages (NPM, Bower)
- Configuration files (database.yml for Ruby on Rails)
- Log files
- Files built & compiled from other source files in the project
- Operating system files
  - .DS_Store on Mac OS X
  - Thumbs.db on Windows
Lifecycle of a file in Git

- Untracked
- Unmodified
- Modified
- Staged
- Committed
- Synchronised

Steps:
1. Add the file: `git add`
2. Edit file
3. Stage file: `git add`
4. Reset the file
5. git commit
6. git push
Resetting files: git-reset tool

$ git reset <source> <path>

- Reset index (staging area) to specified state
- If source is not specified, defaults to latest committed snapshot (HEAD)
Lifecycle of a file in Git

- Untracked
- Unmodified
- Modified
- Staged
- Committed
- Synchronised

Add the file: `git add`

Edit file

Stage file: `git add`

Reset the file

Remove the file

`git pull`

`git commit`

`git commit`

`git push`
Removing files: git-rm tool

$ git rm <file>

- Removes specified file from working directory and index
- Stages deletion of file
- If file was already deleted from working copy (e.g. with rm) and you want to record that change in the index, use add:

$ git add path/to/deleted/file
What files to never commit

- Packages (NPM, Bower)
- Configuration files (database.yml for Ruby on Rails)
- Log files
- Files built & compiled from other source files in the project
- Operating system files
  - .DS_Store on Mac OS X
  - Thumbs.db on Windows

- Git-add lets us choose what files we want to commit
- Sometimes even finer control is needed
What files to never commit

- Packages (NPM, Bower)
- Configuration files (database.yml for Ruby on Rails)
- Log files
- Files built & compiled from other source files in the project
- Operating system files
  - .DS_Store on Mac OS X
  - Thumbs.db on Windows
- Git-add lets us choose what files we want to commit
- Sometimes even finer control is needed

Surely there’s a better way than just manually avoiding them
Ignoring files: .gitignore

- Gitignore file tells Git what files are intentionally unstaged
- File must be named .gitignore
- There can be several .gitignore files (Linux kernel has 206)
  - Applies to the directory it is in
- List of paths that are to be ignored
- Can use globbing - *, ?, [0-9], etc
- Comments start with #
- GitHub has an extensive collection of useful .gitignore files: [https://github.com/github/gitignore](https://github.com/github/gitignore)
Ignoring files: .gitignore

- NOTE: .gitignore doesn’t work on files that are already staged/committed
- They need to be manually removed:

  $ git rm --cached path/to/file

  $ git commit

- “--cached” removes file only from index, not working directory
Moving files

- Unlike most VCS, Git doesn’t explicitly track file movement

```bash
$ mv package.json asd.json
$ git st
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

  deleted: package.json

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

  asd.json
```
Moving files

- Git mostly figures out renaming/moving by itself

$ git add package.json asd.json
$ git st
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  renamed:  package.json -> asd.json
Moving files: git-mv tool

- Git-mv tool moves file in the filesystem and records this change

```sh
$ git mv package.json asd.json
$ git st
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:  package.json -> asd.json
```
Synchronising with remotes

$ git st
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
Lifecycle of a file in Git

Untracked -> Unmodified -> Modified -> Staged -> Committed -> Synchronised

- Add the file: `git add`
- Edit file
- Stage file: `git add`
- Reset the file
- Remove the file
- `git commit` → `git commit` → `git push`
Publishing: git-push tool

$ git push <remote> <branch>

- Takes information from your repository and attempts to apply it to the given remote

$ git push origin master
To server.com:/path/to/repository
  ! [rejected] master -> master (fetch first)
error: failed to push some refs to 'server.com:/path/to/repository'
hint: Updates were rejected because the remote contains work that you do not have locally. This is usually caused by another repository pushing to the same ref. You may want to first integrate the remote changes (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
Updating local repository: git-pull tool

$ git pull <remote> <branch>

- Gets information from the remote or local repository or another branch
- Merges the changes into your local repository
- Opens core.editor to let you add an optional message to the merging commit
- Checks out the new latest state
Pulling while having uncommitted changes

$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From server.com/path/to/repository
  * branch master -> FETCH_HEAD
    a1570d5..d169cc0 master -> origin/master
error: Your local changes to the following files would be overwritten by merge:
    file.txt
Please, commit your changes or stash them before you can merge.
Aborting
Git-stash tool

$ git stash
Saved working directory and index state WIP on master: 6dd221b Merge branch 'master' of path/to/repository
HEAD is now at 6dd221b Merge branch 'master' of path/to/repository

- Saves a snapshot of the working copy for later use
Applying stash

$ git pull github master
From server.com/path/to/repository

  * branch           master  -> FETCH_HEAD

[main 2:23:32 AM] update#setState idle
Merge made by the 'recursive' strategy.
  File.txt | 2 +-  
  1 file changed, 1 insertion(+), 1 deletion(-)

$ git stash pop
Auto-merging file.txt
In conclusion

Now you should have information about

- Git configuration
- Repository creation
- Checking data in and out of repository, synchronising remotes
Practical part

1. Reminder on the unix command line
2. Exercise with all the tools introduced in the lecture
References

- https://git-scm.com/docs/
- https://www.freshconsulting.com/atomic-commits/