Setup GitLab Access

I. Get into GitLab

1. Go to git.doit.wisc.edu.
2. At the login page, click the "UW Madison NetID" button.
3. You're probably already authenticated since you're reading these instructions, so wait for the redirect or authenticate if needed.

II. Create a Personal Access Token (PAT)

adapted from: GitLab docs

1. On the left sidebar, select your avatar (the circle that probably has a weird pattern in the top-right corner of the left sidebar) to open a dropdown.
2. Select Edit profile.
3. On the left sidebar, select Access Tokens.
4. Select Add new token.
5. Enter a name and expiry date for the token. Name it "pat-all" and set the expiry date to 2025-01-01. This means it will last you the whole semester without lingering too long beyond when you stop needing it. You can always create a new PAT.
6. Check the boxes to select all scopes.
7. Select Create personal access token.
8. A green box will appear titled "Your new personal access token" and containing a password-masked field. Do not close the page or navigate away before copying and saving the personal access token! We recommend saving it to a password manager. This isn't granting access to sensitive bank information or anything, so if necessary you can just store it in a file on your computer.

III. Confirm that your PAT is working

To check whether your PAT creation was successful, you can run a read-only command on an existing GitLab repository. It won't make any changes to the GitLab repository or to your computer, so it's safe to run as many times as you want. Run it now:

```
git ls-remote https://git.doit.wisc.edu/cdis/cs/courses/cs506/devtech0.git
```

When you hit enter, one of two things should happen:

1. It will "just work" and you'll get output (see below). This will only happen if you've set up a PAT with the DoIT GitLab instance before (for another class, for example).
2. It will interactively ask you for a username and password. Your username is your NetID - that is, your wisc email without the @wisc.edu. Your password is your PAT - copy it from where you have it saved, and paste it in.

Once you've entered your username and PAT, you should get some output. You may or may not get some kind of warning about OAuth that you can disregard. The output should be as follows:

```
c54d79c4284ee2a54c28fcf87394613aa84d5863 HEAD
c54d79c4284ee2a54c28fcf87394613aa84d5863 refs/heads/master
```

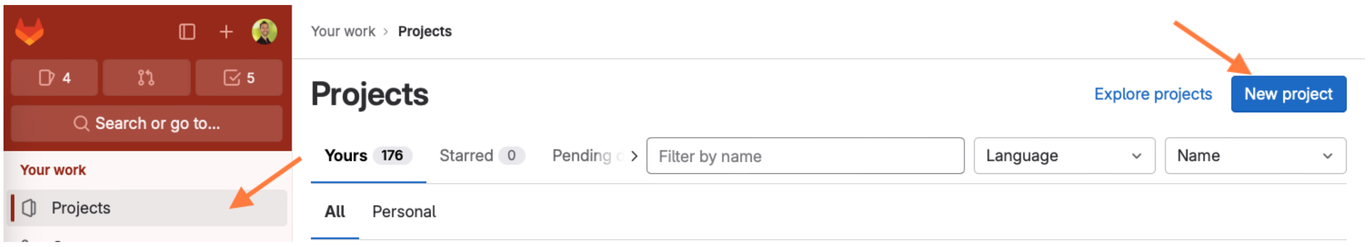
If you got the above output, congratulations, your PAT setup was successful. If you got some kind of permissions error, try again and be extra careful with your copying and pasting when it comes to entering the PAT for the password.

Using Gitlab

I. Create a remote repository

1. Go to git.doit.wisc.edu and select the "Projects" link in the left navigation bar.

2. There is a button in the top-right corner labeled "New project." Click it now.



A screenshot of the GitLab Projects page with arrows pointing to the left navigation "Projects" and the "New project" button

3. Choose "Create a blank project."

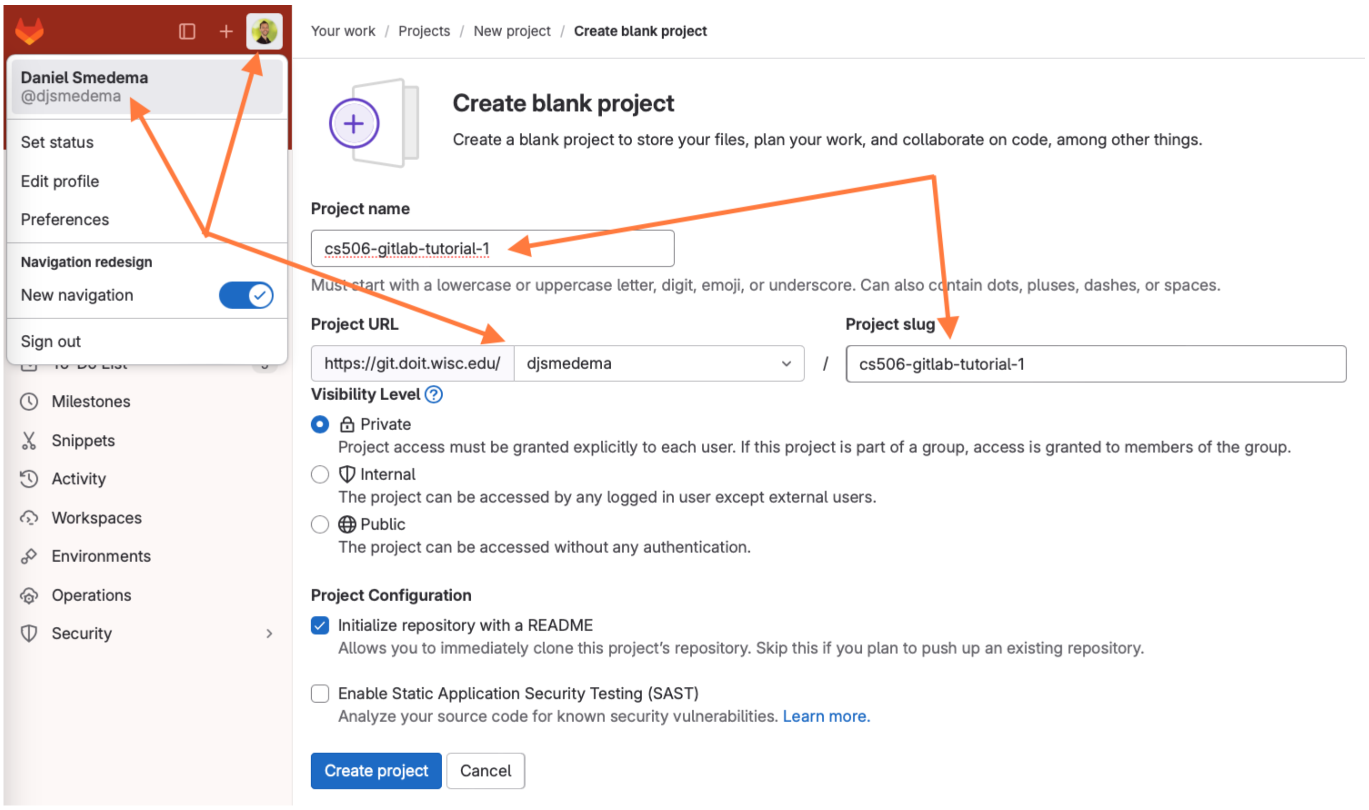
4. Give your project a name; for example, cs506-gitlab-tutorial-

5. Click the field "Pick a group or namespace" to open a dropdown with a search box. Enter your GitLab username (your Wisconsin NetID).

6. The default visibility level is "Private" -- this is fine for our purposes.

7. "Initialize repository with a README" should be checked by default; if not, make sure it is checked.

8. Click "Create project."



Create blank project

Create a blank project to store your files, plan your work, and collaborate on code, among other things.

Project name: cs506-gitlab-tutorial-1

Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.

Project URL: https://git.doit.wisc.edu/ djsmedema / Project slug: cs506-gitlab-tutorial-1

Visibility Level

[✓ Private] Project access must be granted explicitly to each user. If this project is part of a group, access is granted to members of the group.

Internal The project can be accessed by any logged in user except external users.

Public The project can be accessed without any authentication.

Project Configuration

[✓ Initialize repository with a README] Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

Enable Static Application Security Testing (SAST) Analyze your source code for known security vulnerabilities. Learn more.

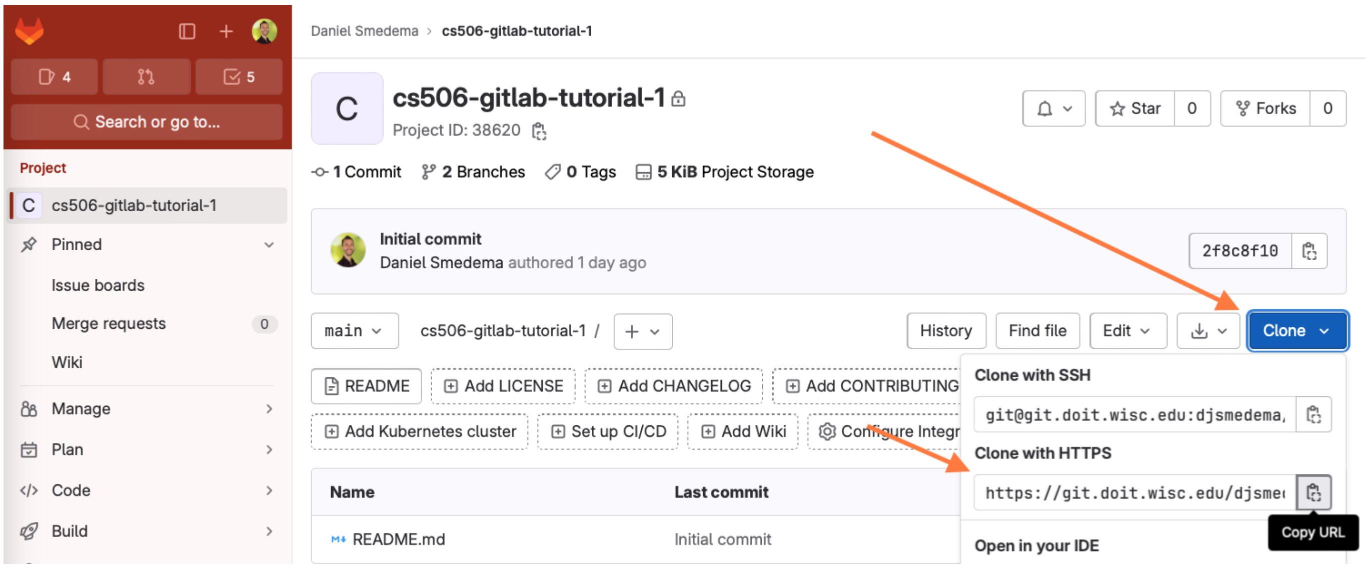
A screenshot of the "Create project" screen filled out according to these instructions, with arrows pointing to important points like where you can find your GitLab ID

Congratulations, you have created a GitLab project, which contains a Git repository! A repository hosted in some other place besides your computer is called a *remote repository* or often just a *remote*.

II. Clone the remote repository to your machine

Now we are almost ready to get back to Git on the local machine. Leave the web browser open; we will be going back there to observe how the actions we take in our local terminal affect the GitLab repository.

Find the "Clone" button in the top right of the GitLab page. Click it to open a dropdown. The dropdown will give you two options: "Clone with SSH" and "Clone with HTTPS." The options are very similar, but they are not the same. Use the copy button to copy the **HTTPS** option.



cs506-gitlab-tutorial-1

Project ID: 38620

1 Commit 2 Branches 0 Tags 5 KiB Project Storage

Initial commit
Daniel Smedema authored 1 day ago 2f8c8f10

Name	Last commit
README.md	Initial commit

Clone with SSH: git@git.doit.wisc.edu:djsmedema,

Clone with HTTPS: https://git.doit.wisc.edu/djsme

A screenshot showing the clone button and the HTTPS option within it

Now, open a shell on your computer (if on Windows, make sure this is a Git Bash shell). Navigate to the directory where you want to put files for this class. You are about to do what's called *cloning a remote repository*. This will automatically create a directory which will contain all of the repo's files, so you don't need to create a directory first (unlike git init). Run:

```
git clone <HTTPS_URL_FROM_GitLab>
```

Now you should be able to change your directory to the newly created one, and confirm that it is in fact a Git repository with git log:

```
cd cs506-gitlab-tutorial-1/ # tab-completion is your friend!
git log
```

Compare this log message to those you have seen before. It should look pretty similar. In particular, note the Author field: you created this commit within the GitLab web application, and because you were authenticated there with your NetID, you should see your name and wisc email in the log Author field.

One thing that is different: in the parentheses, there are several more entries than what you saw when you ran git init locally, specifically, a couple of new ones prefixed with origin/.

Origin is Git's default name for a remote repository. However, this is just a default; the name that your local repository gives to a remote repository does not matter.

You can check your repository's *remotes* by running the command:

```
git remote -v
```

Here you can see that origin is just a label given to the URL we copied in earlier.

III. Remote branches and fetch

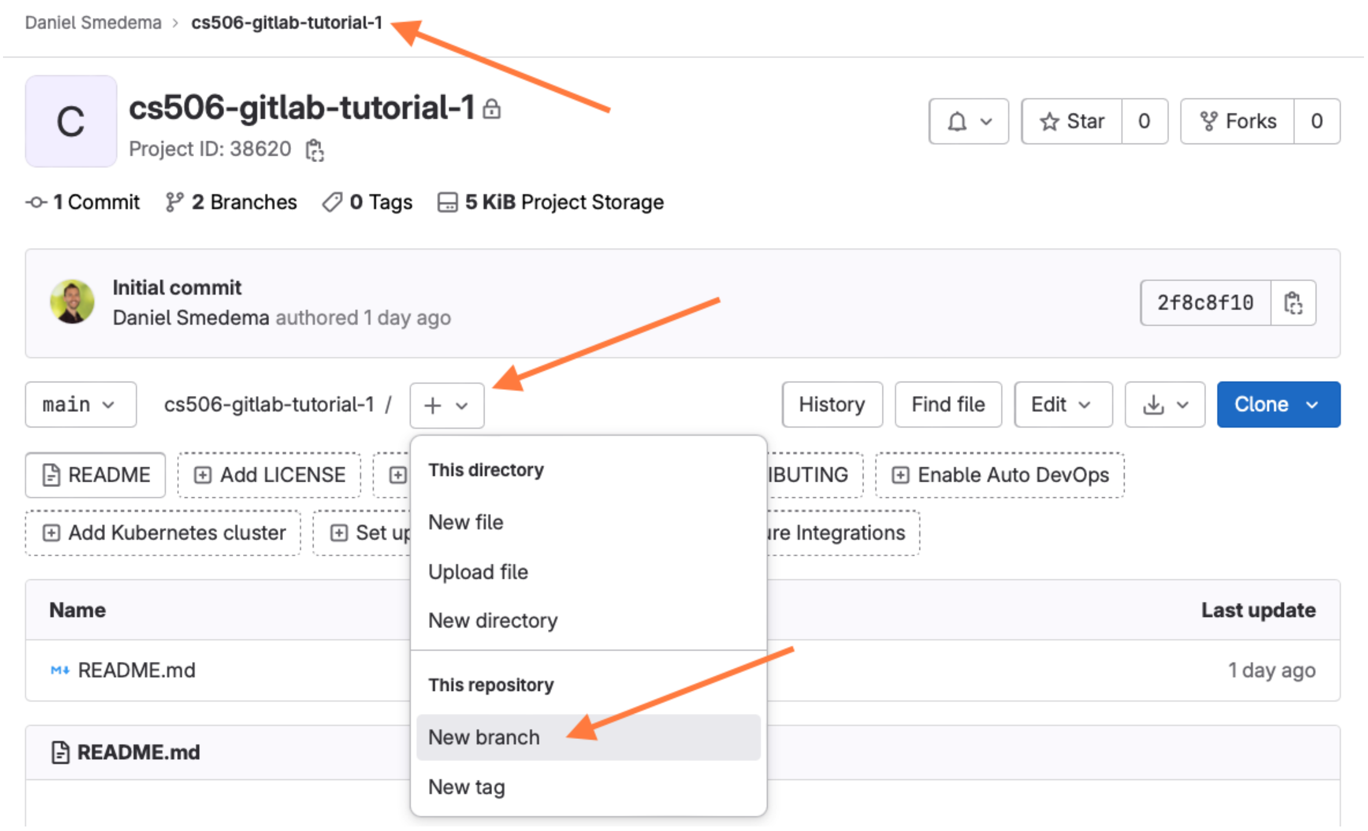
Now recall the git branch command from the DevTech 0 assignment. It shows you your branches. If you pass the -a option, it will include branches that it is tracking from your remote repositories:

```
git branch -a
```

These remote branches are local copies of the branches in your remote repository.

Go back to the GitLab webpage. You should still have your project's main page open; if not, go there now (using the navigation breadcrumbs at the top of the page is an easy way to do this). Near the top of the

page, you should see a dropdown labeled "main", then the repo name, cs506-gitlab-tutorial-1 /, then another dropdown button labeled with a + symbol.



A screenshot of the GitLab project home page with the navigation breadcrumbs, + button, and "New branch" button indicated by arrows

Click the dropdown, and in the section "This repository," click "New branch." This will take you to another screen where you will provide a name for the branch; use new-in-gl. The "Create from" dropdown should say "main." Create the branch.

Now return to your terminal. Run git branch -a again:

```
git branch -a
```

Nothing has changed. Although you created a new branch in GitLab, your local machine is not aware of it; it only knows about the branches that were present when you first cloned it.

In order to update your local repository's remote references, we use the git fetch command. This command does *not* make changes to your local copy of the repository or your local branches. Run it now:

```
git fetch
```

Now your *local* repository has updated its references to the *remote* branches. Run:

```
git branch -a
```

You can see that your local repo now has a reference to the remote branch new-in-gl. However, you still don't *quite* have that branch on your local machine. Run:

```
git branch
```

Your local machine is in a bit of an odd limbo state with respect to the new-in-gl branch. It knows the branch exists on the remote repository, but it doesn't show up among your branches without the -a option to git branch.

This is actually a good thing. This way, your collaborators can create as many branches as they want, and you won't have to look at a whole long list of them whenever you run git branch.

But let's say you do want to do some work with the new-in-gl branch. Now that your local repo knows it exists on the remote, you can run checkout *as if* you already had a local copy of the branch. Do so now:

```
git checkout new-in-gl
```

```
git branch
```

Git has now set up a local copy of new-in-gl for you that knows it is connected to the branch from the remote, origin/new-in-gl.

IV. push your code up to the remote

We are now going to hit on one of the most commonly used commands, git push. This is used to send our local changes to the remote repository.

Simulate doing some dev work, then add and commit it, then run git log:

```
touch urls.py
git add.
git status # should show only urls.py as a new file
```

```
git commit -m "adding urls file"
```

```
git log --oneline
```

Look at the log output: our new commit is where our local branch new-in-gl is; but the remote-tracking branch origin/new-in-gl is still on the previous commit.

Now run git push:

```
git push
```

Check GitLab. Now this change in our branch should be there.

Creating a branch locally and pushing to remote

Most of the time, you will actually create the branches on your local machine and then push them up. Let's demo this - create a branch, then push it:

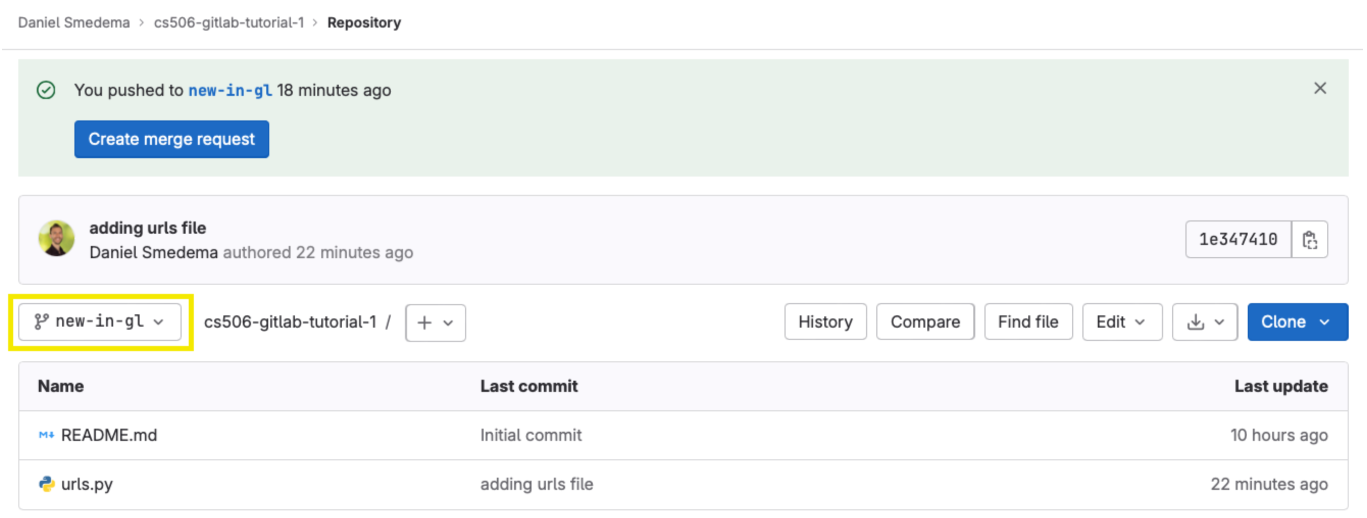
```
git checkout -b local-branch
```

Now to push our branch to our remote repository, we need to use a special command to tell GitLab which branch to match with remotely:

```
git push -u origin local-branch # Short for git push --set-upstream origin
local-branch
```

V. Exploring GitLab and making changes to the remote branch

Let's check back in on our friend GitLab. Use the branch dropdown to switch from main to new-in-gl if you're not already there. The branch dropdown is on the left side of the main content, near the top, just above the list of files.


Daniel Smedema › cs506-gitlab-tutorial-1 › Repository

You pushed to new-in-gl 18 minutes ago

adding urls file
Daniel Smedema authored 22 minutes ago

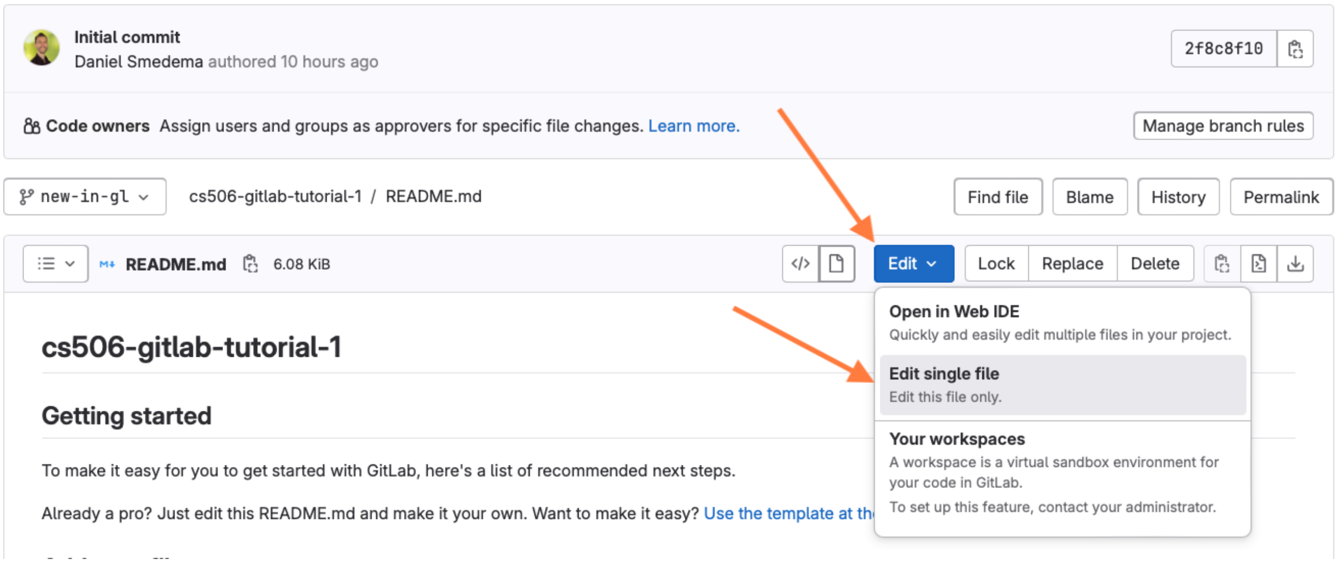
1e347410

new-in-gl cs506-gitlab-tutorial-1 /

Name	Last commit	Last update
README.md	Initial commit	10 hours ago
urls.py	adding urls file	22 minutes ago

A screenshot of the GitLab project home page on the new-in-gl branch, with the branch dropdown box highlighted with a yellow rectangle

Click the README.md file. Then click the "Edit" dropdown and select "Edit single file."


Initial commit
Daniel Smedema authored 10 hours ago
2f8c8f10
Code owners Assign users and groups as approvers for specific file changes. Learn more.
Manage branch rules
new-in-gl
cs506-gitlab-tutorial-1 / README.md
Find file
Blame
History
Permalink
README.md
6.08 KiB
Edit
Lock
Replace
Delete
Open in Web IDE
Quickly and easily edit multiple files in your project.
Edit single file
Edit this file only.
Your workspaces
A workspace is a virtual sandbox environment for your code in GitLab.
To set up this feature, contact your administrator.

cs506-gitlab-tutorial-1

Getting started

To make it easy for you to get started with GitLab, here's a list of recommended next steps.

Already a pro? Just edit this README.md and make it your own. Want to make it easy? Use the template at th

The "Edit single file" option highlighted in the "Edit" dropdown

This will open the editor. Now delete all the text and replace it with something like:

```
# GitLab Tutorial 1

This repository helps teach `git` commands `clone`, `fetch`, `push`, and
soon, `pull`.
```

Make sure the **Target Branch** field says new-in-gl! Leave the default commit and click "Commit changes."

Now, go back to the History for the new-in-gl branch: use the breadcrumb navigation links at the top to go back to your project home page, use the branch dropdown to switch to new-in-gl, then click the History button.

You should see three commits, with the most recent one being the "Update README.md" commit you just made.

Interlude: Git does not magically update in the background

Go back to your terminal and run git log:

```
git log --oneline
```

origin/new-in-gl is still pointing to the "adding urls file" commit; in fact, there are only two commits. Your local repository will not know about changes made to the remote unless you run a command that tells it to check!

VI. pull

So, how do we tell Git to check the remote? Previously, we used fetch for this, telling Git to update its remote-tracking branches without making any changes to our local branches. Then we used checkout to

start working on the new branch we fetched from the remote repo. Do this now:

```
git fetch

git log --oneline --all

git checkout new-in-gl
```

If you already have a local version of a branch, fetch will not mess with it. That makes fetch very safe - you can always run fetch without worry. However, it won't actually bring in changes from the remote!

You can see that trying to checkout the branch gave us a little informational message: our branch is behind origin/new-in-gl, and we can use git pull to update the local branch. Go ahead and do so, then run git status and git log:

```
git pull
git status
git log --oneline
```

Fantastic: we've retrieved changes from the remote.

Note that git pull automatically runs a git fetch behind-the-scenes. You don't need to run git fetch and then git pull unless you're trying to demonstrate the difference between them; most of the time, you can just run git pull.

VII. When local and remote branches diverge

Recall what it means for branches to diverge.

They have some shared history, but at some point, both branches were active at the same time, and different commits were made on each of them. This is a totally normal thing to happen when we're talking about two different branches, like master and testing in the image. That's kind of the whole point of branches.

However, when your local branch and the remote's copy of a branch diverge, things are a little different and this is called divergence. Here's how you deal with it:

Set up the divergence

Add a line to the bottom of README.md. Commit it, but do NOT push.

```
# note the two angle brackets >> for "append"
echo "This is the local branch" >> README.md
git add.
git status
git commit -m "local branch commit"
```

Now edit the file in GitLab.

- Project home using breadcrumbs
- Branch dropdown to new-in-gl
- Click README.md
- Click "Edit" to open the dropdown
- Click "Edit single file"
- Add the line "This is the remote branch" to the bottom of the file.
- Set the commit message to "remote branch commit"
- Make sure the Target branch is new-in-gl
- Click "Commit changes

Now back in your terminal, fetch:

```
git fetch
git status
```

Examine the parts of the git status output:

- Your branch and 'origin/new-in-gl' have diverged
  - This is exactly what we were trying to set up for demonstration
- and have 1 and 1 different commits each, respectively.
  - This should make sense, because we made 1 commit on our local machine and 1 commit directly in the remote repo.
- use "git pull" to merge the remote branch into yours
  - merge : because we have divergent branches, we have to merge them back together. This is also telling you that git pull will automatically initiate the merge for you.

Reconcile the divergence

As mentioned earlier, normally you won't run git fetch first and check whether you've diverged. You will just run git pull - and you will immediately find yourself in the following situation. Run git pull now to see it.

```
git pull
```

This is results in a failed merge conflict which requires manual resolution, and should look familiar from DevTech 0.

Open README.md; in this case, we will keep only the line that says "This is the local branch". So remove the conflict markers and the lower section that is coming in from the remote. Your file should look like this:

```
# GitLab Tutorial 1

This repository helps teach `git` commands `clone`, `fetch`, `push`, and
soon, `pull`.
This is the local branch
```

Save the file, then run:

```
git add README.md
git commit
```

An editor will open so you could edit the commit message. We don't need to make any edits, so close the editor without saving.

Now check the log, and in this case, let's add the --graph option:

```
git log --oneline --graph
```

Now push your local changes so your remote repo is synced:

```
git push
```

VIII. Best practices using branches with remote repos

Every branch should encompass one task that a single person works on. That means that Divergence between your local branch and your remote branch should not happen because you are the only one pushing to your branch.

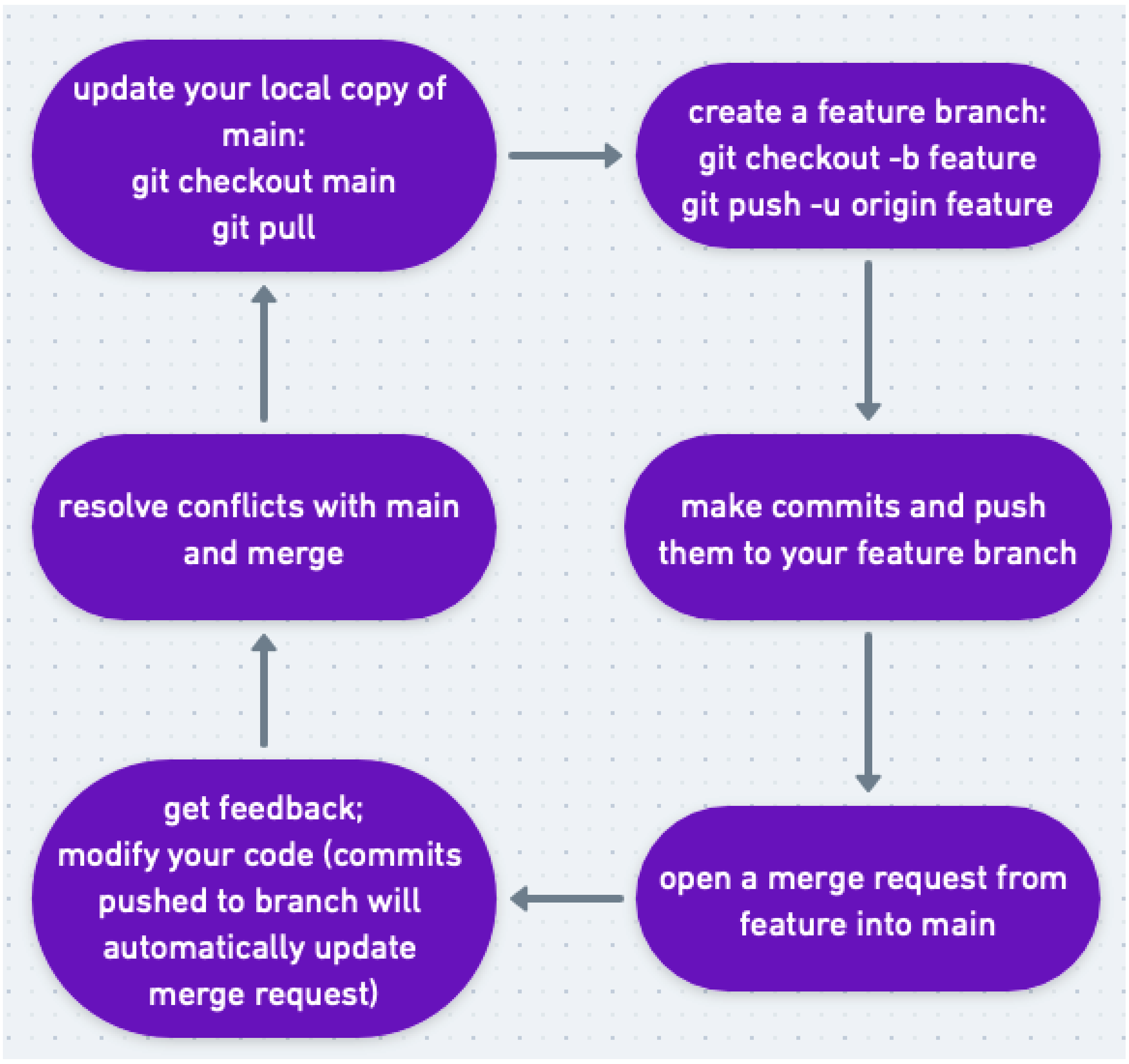
So how do you ever merge your code back together again? Eventually someone is going to have to merge into someone else's branch, right? Sort of - but you don't do this by directly interacting with their branch using Git. This is where *merge requests* come in.

A *merge request* is a workflow provided by GitLab where you *request* that some branch *merges* your branch into it.

The vast majority of merge requests are made into a single shared branch, like main (though we will get into more nuanced schemas with multiple shared branches, like GitFlow).

No one ever directly pushes to shared branches like main -- in fact, you usually set up the repo permissions in GitLab so that these branches can't be pushed to, so the *only* way to get code into them is via merge requests. Merge requests happen in the remote repository, so when main changes, it is immediately available to all collaborators via git fetch or git pull.

So a full development loop looks like this:


update your local copy of main:
git checkout main
git pull
create a feature branch:
git checkout -b feature
git push -u origin feature
make commits and push them to your feature branch
open a merge request from feature into main
get feedback; modify your code (commits pushed to branch will automatically update merge request)
resolve conflicts with main and merge

A simplified look at a development loop using the Git best practices of feature branches and merge requests

A note on paired programming

Paired programming means two programmers are working together on one computer, so you will only need one branch, not one each. If you switch computers when you switch off who is "driving" and who is the "copilot," then you should make sure that all code is pushed up before the handoff, and pulled down immediately after.

If you finish a paired programming session with a branch still open and you decide you will each continue working independently, at least one of you should create a branch off of your feature branch. When you begin your next paired programming session, merge your branches (you can use a merge request or do a local merge).

TL;DR

This is the most common workflow using the commands we discussed:

```
1. git pull
2. git checkout -b <NEW_BRANCH_NAME> OR git checkout EXISTING_BRANCH NAME>
3. Make changes to files
```

```
  4. git add.
  5. git status
  6. git commit -m "<YOUR_MESSAGE>"
  7. git push OR git push -u origin <NEW_BRANCH_NAME>
```