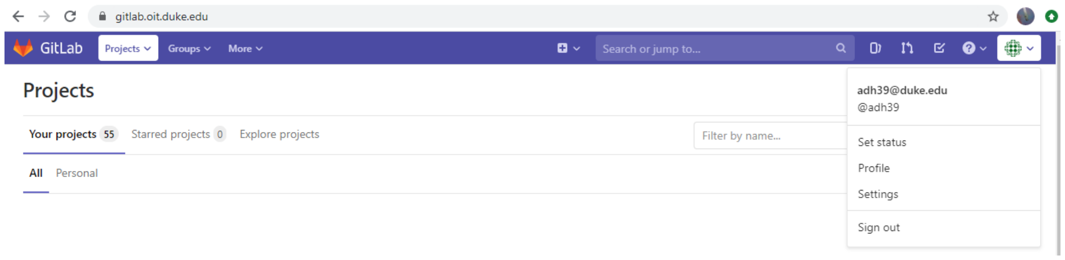
# ECE 651
# Forge: GitLab/Emacs Integration

This tutorial will walk you through using **forge** so that you work with GitLab's Issues + Merge Requests directly from the comfort of Emacs! If you prefer to just use GitLab's web interface instead, that is your choice.

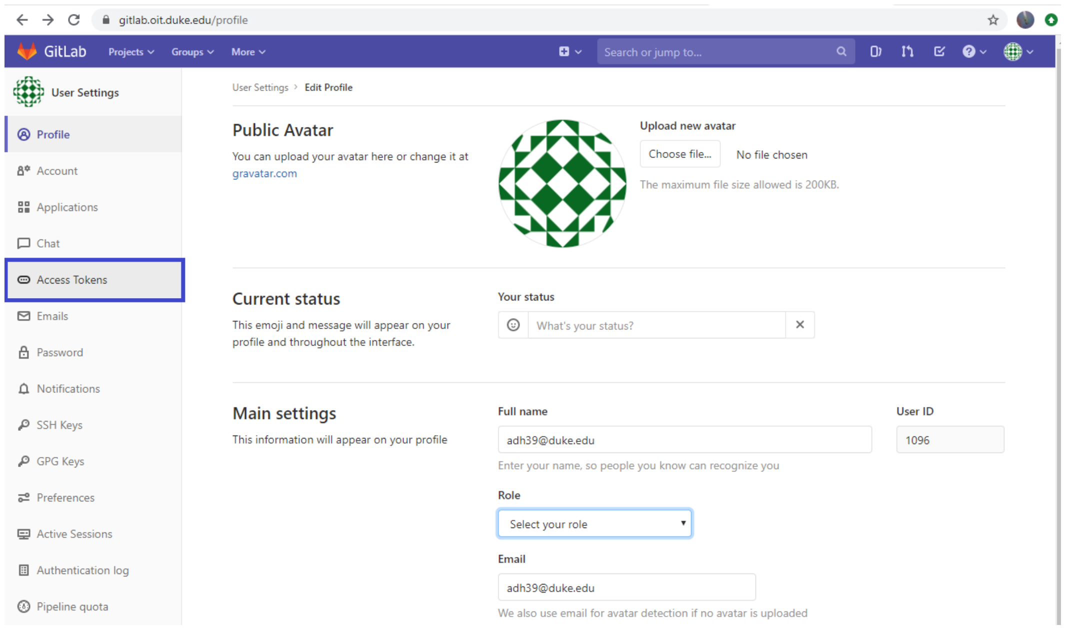
## 1 Setup: API Token Creation

Before we get started, you will need to create an API token on GitLab. This token will serve as the way for Emacs to authenticate you GitLab.

First, login to GitLab. Next, find the drop down menu in the upper right corner:



Select "Settings", and you should be presented with a screen that looks like this (except that it won't have the blue box around "Access Tokens" that we added to call your attention to it):



You can upload your avatar here or change it at gravatar.com

No file chosen

The maximum file size allowed is 200KB.

This emoji and message will appear on your profile and throughout the interface.

This information will appear on your profile

Full name: adh39@duke.edu

User ID: 1096

Enter your name, so people you know can recognize you

Role: Select your role

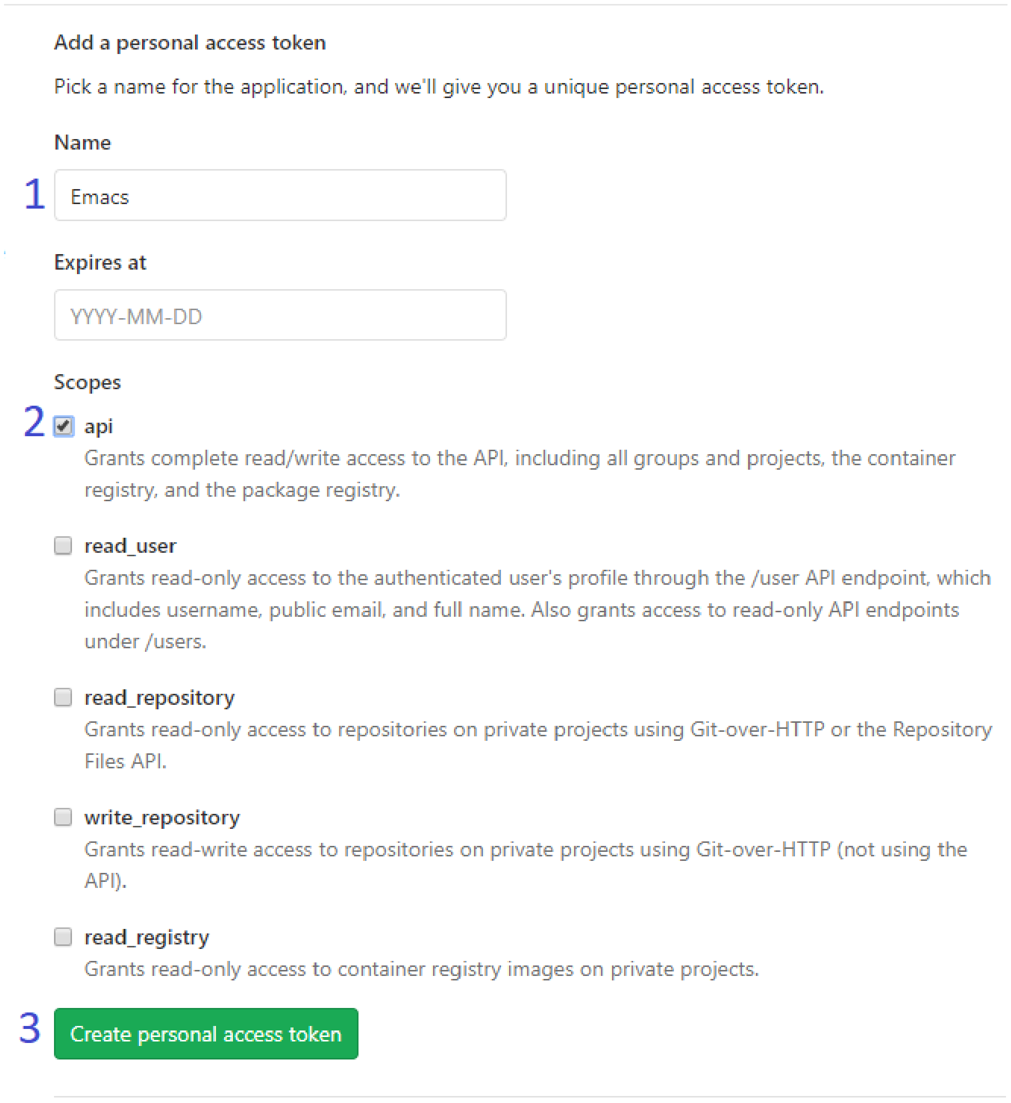
Email: adh39@duke.edu

We also use email for avatar detection if no avatar is uploaded

Now, select "Access Tokens" on the menu at the left. You need to do three things on this page:

1. Type "Emacs" in the "Name" box.
2. Check the "api" box under "Scopes."
3. Click the "Create personal access token" button (which should turn green after doing steps 1 + 2).

These are labeled with the blue numbers matching the above steps in this picture:



Add a personal access token

Pick a name for the application, and we'll give you a unique personal access token.

Name

Emacs

Expires at

YYYY-MM-DD

Scopes

[✓] api
Grants complete read/write access to the API, including all groups and projects, the container registry, and the package registry.

[ ] read_user
Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.

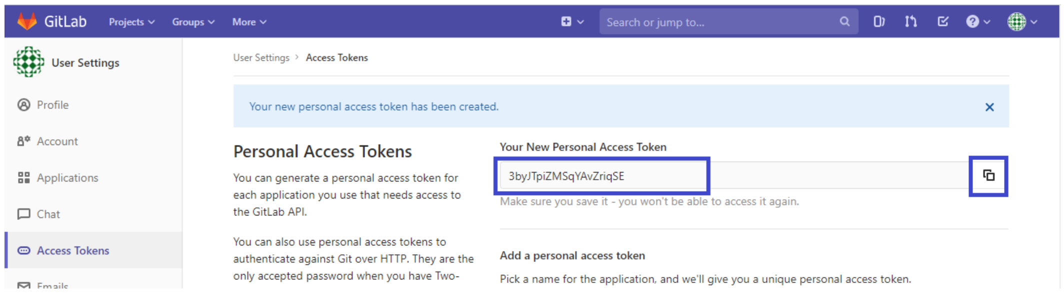
[ ] read_repository
Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.

[ ] write_repository
Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).

[ ] read_registry
Grants read-only access to container registry images on private projects.

Create personal access token

You will be shown the token at the top of the page, along with a "copy" button to the right of the text field. You should NOT share this token with anyone (the one displayed here is one I made just for this and then immediately deleted!)—it will grant full access to your gitlab repositories! Also note that once you leave this page, you will not be able to find this token again (so if you lose it, you will have to make a new one):



Your new personal access token has been created.

Personal Access Tokens

You can generate a personal access token for each application you use that needs access to the GitLab API.

You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-

Your New Personal Access Token: 3byJTpiZMSqYAvZriqSE

Make sure you save it - you won't be able to access it again.

Add a personal access token

Pick a name for the application, and we'll give you a unique personal access token.

Now, in your virtual machine, go to your home directory (`cd ~`), and create a file called `.authinfo`. Note that the name of this file must be exactly as above, and that the name starts with a dot. In this file, you need to write one single line as follows:

```
machine gitlab.oit.duke.edu/api/v4 login NETID^forge password TOKEN
```

where you replace NETID with your NETID and TOKEN with the API Token you just created.

For example, my netid is adh39, and the token I just created is 3byJTpiZMSqYAvZriqSE, so I would write:

```
machine gitlab.oit.duke.edu/api/v4 login adh39^forge password 3byJTpiZMSqYAvZriqSE
```

## 2 Pulling Topics

To do the rest of this tutorial, you are going to need to work in one of your repositories that is on GitLab. If you forked the development walkthrough tutorial's repository at the start of the semester, that is a great one to work with. If not, we recommend you fork that now and work in your fork of it. Whatever repository you choose to use, if you do not already have it cloned into your VM, clone it now. Then cd into the directory for it.

Before we proceed, we are going to take a moment to set up default labels for issues. Go to this project in the GitLab web interface. Go to the menu on the right, and expand the "Issues" menu. Find "Labels" and click on it. Unless you have already setup labels for your project, you should see a screen that tells you that "Labels can be applied to issues and merge requests to categorize them." There should be two buttons below that. Click on "Generate a default set of labels." This is generally a good starting point for your labels. However, as you work on larger projects, you might want to add other labels (*e.g.*, "kernel module," "user interface," "database," etc—these are up to you and your team). Take a moment to note that the labels come in various colors to help you mentally categorize them. For example, "enhancement" is green, and "suggestion" is blue, while "bug" and "critical" are red.

Open up any file in Emacs, and hit "C-x g" (not "C-x C-g", just "C-x g") to open the Magit status buffer[1] Press "f y" (which is for "forge pull"). You will get prompted

[1]We covered Magit at the end of the development environment setup walkthrough. If you didn't work through that, it is ok, but you might want to cover it too.

```
Git variable 'gitlab.gitlab.oit.duke.edu/api/v4.user is unset. Set to:
```

Type your netid and hit enter[2].
You will get asked

```
Set gitlab.gitlab.oit.duke.edu/api/v4.user=adh39 [g]lobally (recommended) or [l]ocally?
```

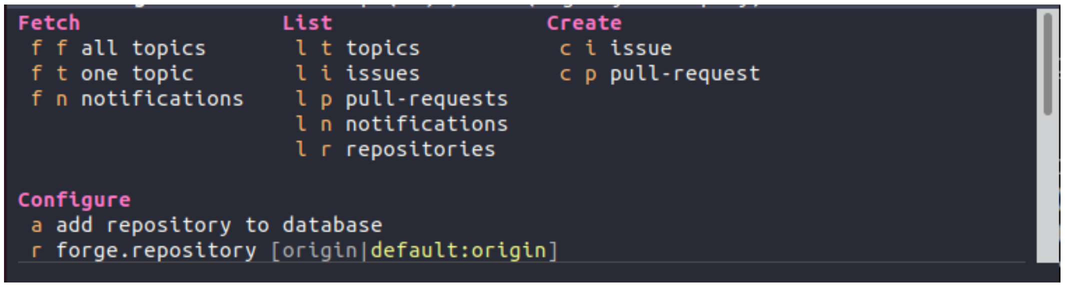
Hit "g" for "globally". You should then be told that it is pulling the repository (username/projectname), and a few seconds later it will tell you "Git finished." If you get an error message like "glab-repository-id: wrong type..." then it most likely means that you did not put your toke into `.authinfo` correctly.

At this point, you probably will not see much else change. Unless the repository you pulled from already has issues or pull requests, nothing new will show up.

Let us fix that by creating an issue, making a merge request for it, merging, and closing the request.

## 3 Creating An Issue

In your magit status buffer, press the tick (or single quote) key ('). This is the key you would use for character literals in Java or C/C++, do not confuse it with backtick (‘). You should see the following at the bottom of your window:



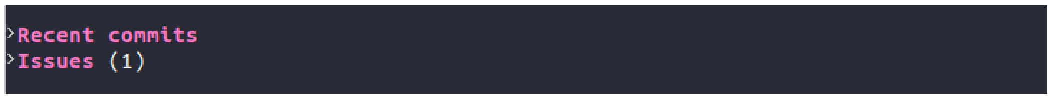
```
Fetch                 List                  Create
 f f all topics        l t topics            c i issue
 f t one topic         l i issues            c p pull-request
 f n notifications     l p pull-requests
                       l n notifications
                       l r repositories

Configure
 a add repository to database
 r forge.repository [origin|default:origin]
```

This interface (which you can always access with tick (') from the magit status buffer is how you do most things with forge.

As you can see on the right, "c i" will create an issue. Go ahead and hit "c i" and you should get a buffer named "new-issue". At the top, it will say "Create new issue on netid/project" and then have the point right after a pound sign. Type the title of the issue on the line with # (it will turn green). On the remaining lines, write the text to describe the issue. When you are done, hit C-c C-c (this is generally how you finish things in magit).

Once you have done that, it will take emacs a few seconds to communicate with git lab, but then your magit status buffer should update to have "Issues (1)" as in this picture:



```
>Recent commits
>Issues (1)
```

[2]Note that this information can be changed later or overridden for other projects with git config.

If you put the point on the >Issues (1) line, and hit TAB, it will expand it to show the issues (TAB is generally expand/contract in magit). If you put your point on the issue you just created, and hit ENTER you will get a buffer with the details of that issue. You will see that the state is open, the labels, marks, and assignees are "none" and the description you just wrote is at the bottom.

Let us first assign this issue to ourself. Put the point on Assignees and hit "C-c C-e" (note that C-c C-e is how you generally edit fields inside of a topic). You will get prompted for the assignees. Type your own netid and hit enter. You will see forge communicate briefly with gitlab, then you will see that assignees now lists you.

You would do this in much the same way: put the point on Labels and hit "C-c C-e". You will get prompted for the labels to set. Here you enter a comma separated list of labels (and can tab complete: forge knows what labels you set up in your project). For now, let us put in

```
enhancement,suggestion
```

Note that you can type enh(TAB),sug(TAB) to enter this quickly. Once you hit enter, you will (as usual) see forge briefly interact with gitlab. You will then see the labels displayed in green and blue. Why these colors? Those are the colors for those issues that we saw when we set them up[3].

Note that these labels will show up in the magit status buffer where the issues are listed.

Next, suppose you wanted to add another comment to this issue. If you hit "C-c C-n" anywhere in the buffer for the issue, you will get another buffer to write a comment (similar to creating the initial description, but you can't add the title). As with creating the initial description, you hit "C-c C-c" when you are done.

For more detailed documentation on editing issues, see `https://magit.vc/manual/forge/Editing-Topics-and-Posts.html#Editing-Topics-and-Posts`.

Note that if you want to close this (or any buffer), you can just kill it with "C-x k".

## 4 Create a Pull Request

Before we proceed, note that GitLab calls these "Merge Requests" while forge calls them "Pull Requests" (because other platforms call them that). The terms are identical. Note that you will often hear pull request abbreviated "PR".

To create a merge request, let us first make something to merge. Go back to the magit status buffer with "C-c g". Now hit "b" (for "branch"). We want to create a new branch and check it out at the same time, so choose "c". You will be prompted for the branch to start from. The default of "master" is fine, so just hit enter. Then you will be prompted for the name of the branch to create. Let us call it "feature1"[4].

[3] I am not aware of a way to add labels or configure their colors through forge—this is not a big deal, since it is done quite rarely, so just do it in the web interface... or submit a feature to forge!

[4] You can put pretty much anything you want here, but as always good names are helpful. Your development teams may require certain naming conventions.

Now you are on the “feature1” branch. You can see this at the top of the magit status buffer (“Head: feature1”). Let us now do some “development”. In a real project, you would implement a feature, fix a bug, or otherwise do real development work here. However, we are just working on using forge, so there is nothing real to do. Instead, let us just create a new file, write something in it, and save it. Now hit “C-c g” to refresh the magit status buffer. You will see “Untracked files” with the file you just created listed under it. Put the point on that file and hit “s” to stage it. Now it should say “Staged changes (1)” and under it list “new file yourfilename”. In the magit status buffer, hit “c c” to **c**reate a **c**ommit. Emacs will display the staged changes and the place to edit the commit message. Go ahead and write a commit message and finish with “C-c C-c”. Now we want to push so from the magit status buffer, press “P u” (“P” for push, “u” for upstream). Since we have not set the upstream, we will get prompted for it. Type “origin/feature1” and hit enter.

Now let us do the actual creation of the merge request. From the magit status buffer, press tick (’) and then hit “c p” (which should be listed on the right under “Create” as “c p pull-request”). You will be prompted for the source branch (what you are merging from). If you hit TAB, it will almost certainly complete as “origin/” and then stop (since we have two choices). You want “origin/feature1” so type “f” and hit TAB. Then you will be prompted for the target branch (what you are merging into). You want “origin/master” so hit TAB to complete “origin/” then type “m” and hit TAB again.

Now you will have a buffer for the pull request. This is where you enter the title and description of the pull request. Forge will pull the information from your last commit by default. That is for the most part reasonable, except we would also like to have this pull request automatically close our issue too. So we should write “Closes #1” (because we want to close issue number 1. We would put another number to close some other issue). Once we finish editing that, we should hit “C-c C-c” to complete it. As usual, you will see emacs/forge work with gitlab for a moment.

Now you should see “Pull requests (1)” in your magit status buffer, which is likely collapsed. If you put the point on “Pull requests(1)” and hit TAB it will expand. Now you can put the point on the pull request you just made (note that it starts with “!1” to indicate that it closes issue 1), and hit enter. Note that if you were working in a real team, you would go to “Assignees” and assign this to a team member for review. Note that we could also add labels (*e.g.*, “critical” if we want the reviewer to know this is urgent).

## 5 Reviewing a Pull Request

In a team setting, someone else would review our pull requests, and we would be reviewing the pull requests made by other team members. However, for the sake of this example, we are going to review our own pull request (and pretend we didn’t just create it). Before we review anyone else’s pull request, we would want to make sure our own changes were committed to our branch (though there isn’t anything to do with that here).

As usual, we will start from the magit status buffer ("C-c g")[5]. Now we want to checkout the branch for a pull request, so from the magit status buffer, we will hit "b y". We will then be prompted to enter the pull request we want to checkout. We can just hit "1" and TAB to tab complete the pull request's name. Now our source code is on the branch for this pull request.

We would then take the time to review the various changes—we might go read the source files, examine test cases (and see how well the tests cover the code), etc. We can narrow down what code we need to look at by going to the merge request (find it in the magit status buffer and hit enter), and then finding the "Commits" section. This will list the commits that were made on the branch that is being merged in. If you hit enter on one of these commits (in our example, there is only one), you will see a buffer appear that has the commit message and the changes to the files. In our case, we just create a file, so it will say there is a new file, and the contents will be listed with a green background (since it was added). If you put the point on the contents of the file and hit enter, it will take you to that file.

If you return to the pull request's buffer, you can also add additional comments (as before, "C-c C-n" will let you add new comments).

At this point, we might have a few different outcomes:

1. Close the pull request (*i.e.*, discard the changes). We do this by changing the State field ("C-c C-e") in the pull request.

2. Discuss with the other person. We might write comments back and forth and discuss what should be done.

3. Accept the pull request.

Let us assume we want to accept the PR. We should first merge these changes into master locally. So we go back to the magit status buffer ("C-c g") and hit "m" for "merge". We want to merge our current branch into something else (namely master), so we hit "i". We will then be prompted for what branch to merge into, and we type "master". This will merge all the changes from the current branch (feature1) back into master, and make our current branch be master.

At this point, in a real development cycle, we might want to do a bit more checking: build things, run tests, etc, in case there were any other changes that happened. When we are satisfied, we should return to the magit status buffer, and hit "P u" to push to upstream (should be "origin/master").

After we do that, hit "f y" again to update the issues from gitlab, and you will see that the merge request and the issue we were working on are now grayed out because we are done with them.

[5]If we were actually working in a team, we might need to do "f y" to pull in the latest pull requests and issues).

## 6 Beyond This Tutorial

We hopefully have given you enough information to get started in forge if you want to use it.

As with many Emacs things, we have barely scratched the surface. For more information, please consult the forge manual: `https://magit.vc/manual/forge.html`

Perhaps the most useful of the features we won't cover is that you can add to magit-status-sections-hook to display a section of pull requests (forge-insert-assigned-pullreqs) and issues (forge-insert-assigned-issues) that are assigned to you. This feature can be useful on much larger projects where there are many open issues and many pull requests.