

TDDC88/TDDC93: Software Engineering

Lab 5

Software Configuration Management (SCM) and Continuous Integration (CI)

Objectives

- To give you a fundamental understanding and practical experience of how a version control system works in general, some of the things that are possible with Git
- To gain an understanding of git branches and manage branches whenever you want to add new work and push it to the original repository via merge request
- To gain a fundamental understanding and hands-on experience on continuous integration (CI)

This lab consists of two parts. The initial part gives you a fundamental understanding and practical experience of how a version control system works in general. The subsequent part provides you with fundamental insights and hands-on experience with continuous integration.

Part 1 - Software Configuration Management (SCM)

Introduction

“Version control is to programmers what the safety net is to a trapeze artist. Knowing the net is there to catch them if they fall, aerialists are free to fly. In the same way, version control enables you to take programming risks that you would never otherwise consider. If something goes wrong, you can always revert back to a known, good-working version of your code. You can experiment in a branch¹ off the main trunk without interfering with other team members. When bugs are discovered in an older version of a shipped product, you can easily check out that specific version to confirm, fix, and generate a patch for the bug. Without version control, you would have to be much more cautious, move more slowly, and generally be less productive”

¹ The normal convention is to use three root folders for a software project: *branches*, *tags*, and *trunk*. Branches are for experiments. Tags normally identify older, already released versions of the software. However, most of the time, you'll want to work on the main branch, which is called trunk.

- *Elliotte Harold, Adjunct Professor, Polytechnic University*

The main idea of a version control system is to contain the software project in a so-called *repository*; from which developers can *check out* copies of a project and in this way create local *working copies*. These working copies can then be edited and eventually uploaded, or *committed*, back to the repository, as a new version, or *revision*, of the file that has been modified. If several developers simultaneously are editing working copies of the same file, the version control system must be able to deal with *conflicts* that might occur when the developers later try to commit their different working copies to the repository. To be sure that a developer always has the latest revision of a file he or she should *update* his copy from the repository frequently, and at least before commit.

Git was developed by Linus Torvalds in 2005 for the development of the Linux kernel. Git is by default included in the standard Eclipse version. It is possible to work with Git from the terminal (by typing the commands) or from a graphical user interface (e.g., standalone clients or plugins to IDEs such as eclipse).

Very Important -

All instructions in this lab are inter-dependent. If you make a mistake in one instruction, the whole flow will fail, and you will be asked to do the lab again. To avoid the inconvenience, it is important to take a screen shot of each command, you ran in the terminal. These screen shots can be placed in Word along with the number or what you did. These screen shots will help your instructor to debug the error in case the workflow fails. This does not guarantee (i.e., particularly in cases, where you made many mistakes) that you will not be asked to do lab again.

These tasks are for demonstrating some features and problems that may occur using version control and the features in Git. To make life a bit easier we will in the later part of the exercise pretend that we are monitoring **two different developers**, named Peter and Sally, who are working for the same company. This company has recently bought a project called HelloWorld. However, the product manager of the company would like to make some changes to HelloWorld and has put both Peter and Sally to work on the project. Of course, since Peter and Sally do not exist, you will have to carry out the tasks for them. When you have finished the tasks, you should report to the lab assistant and give an oral explanation and a demonstration of what you have done and what you learned from it. Be prepared to answer questions about the details of your solution.

Note: Using several workspaces/terminals/folders can be a bit confusing. Therefore, it is very important that you read what you should do before you do it! In the worst case, you will have to re-do the lab from the start if you just miss a single instruction:

Task 1. Setting up a Git repository and importing a project

In this, task you will create a Git repository and import the *HelloWorld* project to the repository. However, since this is something that developers usually don't have to bother about, we will not pay it that much attention, but only go through the task.

Open a terminal and run the following commands:

```
git config --global user.name "<first name> <surname>"
```

Change <first name> to your name and <surname> to your surname

```
git config --global user.email <liu-id>@student.liu.se
```

Change <liu-id> to your liu-id

```
mkdir -p ~/TDDC88/git_lab
cd ~/TDDC88/git_lab
mkdir remote
cd remote
git init --bare HelloWorld.git
```

*This will create an empty git repository HelloWorld placed in the folder **git_lab/remote** placed in your **home** folder. The ~ is short for your home folder "/home/<liu-id>/".*

**This folder corresponds to a remote repository where everyone has access to. Usually reachable through a network (e.g., internet).*

```
mkdir ~/TDDC88/git_lab/tmp
cd ~/TDDC88/git_lab/tmp
```

This will create and navigate to a temporary folder.

```
git clone ~/TDDC88/git_lab/remote/HelloWorld.git
```

This clones the repository we just created into a folder HelloWorld in the current directory. You should now have a copy of the repository in the current folder (tmp). The project is still empty.

Run these commands:

```
wget https://www.ida.liu.se/~TDDC88/labs/HelloWorld.tar.gz
tar xzf HelloWorld.tar.gz
cd HelloWorld
git add -A
git commit -m "Added HelloWorld files to repository."
git push origin master
```

This will be explained later, but, in short, we copied new files into the project. Added them to version tracking with Git, committed the new changes, then pushed our changes to the remote origin (which is located at git_lab/remote) so that everyone else can see them.

Remove the temporary folder. We don't need it anymore.

```
cd ~/TDDC88/git_lab
rm -Rf tmp
```

Task 2. Checking out working copies for Peter and Sally

In this task, we will set up two new workspaces/folders/terminal sessions, one for Peter and one for Sally, and check out one working copy from the Git repository to each of the workspaces. You will from now on need **two terminal sessions** running at the same time, one for Sally and one for Peter. Employing separate terminal sessions might lead to confusion, so please ensure that each session corresponds to either Peter or Sally.

Note: You have the option to leverage Eclipse for code modification and execution, or you can use other text editors to edit the source code.

Now we are ready to start working as Peter.

Launch a terminal designated for Peter (referred to as Peter's terminal) and execute the subsequent commands to establish a Git folder specific to Peter (referred to as Peter's folder). *Create and navigate to Peter's folder:*

```
mkdir ~/TDDC88/git_lab/peter
cd ~/TDDC88/git_lab/peter
```

*Clone a working copy of the repository to Peter. You should now have a folder named **HelloWorld** in the current folder (which is ~/TDDC88/git_lab/peter/). This is your clone.*

```
git clone ~/TDDC88/git_lab/remote/HelloWorld.git
```

Enter the folder using *cd HelloWorld*, then:

```
git config user.name "Peter"
```

Change the name in this repo to Peter. (This is only for the lab and is normally not needed since you have entered a global configuration earlier.)

It is Sally's turn now.

Open a **new** terminal for Sally (referred to as Sally's terminal) and do the same for Sally (to create Sally's working folder):

```
mkdir ~/TDDC88/git_lab/sally
cd ~/TDDC88/git_lab/sally
git clone ~/TDDC88/git_lab/remote/HelloWorld.git
cd HelloWorld
git config user.name "Sally"
```

Tip: If you forget which terminal is Peter and which is Sally you can run "git config user.name" to get the name from the terminal.

Task 3. Modify-Commit-Conflict that can be automatically resolved

Both Peter and Sally will now begin their modifications of HelloWorld (You have the option to leverage Eclipse for code modification and execution, or you can use other text editors to edit the source code). Their first assignment is to add some comments in the file *HelloWorldFrame.java*. Although it is the only file in the project, the CEO still wants to clarify that it is the main file.

- 1) Sally starts by adding a comment at the top of the file *HelloWorldFrame.java* (located in ~/TDDC88/git_lab/sally/HelloWorld/src/helloworld/), clarifying that this is the main file:
//This is the main file.
Be sure that you are in Sally's workspace/folder, add this comment, and save the file!
- 2) Sally then checks if there has been a new revision to the repository and performs an update. To do this run *git pull* in **Sally's terminal**. Has anything happened since the latest revision?
- 3) Switch to Peter's workspace/folder.

- 4) Peter, who is lagging behind somewhat, adds a comment right above the main Method of the file HelloWorldFrame.java (located in `~/TDDC88/git_lab/peter/HelloWorld/src/helloworld/`):
//This is the main method.
Add this to his working copy of the file and save it. Peter also checks for new revisions. Do this! Has anything happened? Why/why not?
- 5) Switch to Sally's workspace.
- 6) Sally now feels pretty satisfied and decides to commit her working copy to the repository. Do this for her using the following commands (in Sally's terminal):

git status
git pull

To check for unstaged changes, HelloWorld/HelloWorld/src/helloworld/HelloWorldFrame.java Should be marked in red. If you have unstaged .class files don't add those.

git add <file path>

*This command puts the file in the staging area. Change <file path> to the unstaged file ex.
HelloWorld/HelloWorld/src/helloworld/HelloWorldFrame.java*

git commit -m "Enter commit message here"

This commits the changes.

git push

This updates the server with your changes.

git log -2

This shows the last 2 commits.

- 7) What is the git log output after the push?
- 8) What is the staging area for?
- 9) Why should you do a git pull before pushing?
- 10) Switch to Peter's workspace/folder.
- 11) Peter also feels it is time to make a commit. Therefore, commit the file to the repository using the same procedure as above (in Peter's terminal). Remember to enter a commit comment. What happened?
*Tip. Git may ask you to pull the changes first before pushing Peter's files. To pull the changes, simply run **git pull**. You may also find yourself trying to resolve a merge*

*conflict in VIM (MERGE_MSG). To do this, press **i** (to insert), enter a message on the next line, and exit using **ESC** + **:wq** (Once resolved, commit and push the files)*

- 12) Would it have been possible to solve the problem just as easily if Peter had added a comment at the top of the file as well?
- 13) Which are the three latest commit messages after Peter successfully has committed the file?
- 14) What are the key differences between git merge and git rebase? How do these commands affect the commits history?

Note: Once the conflict is resolved, make a commit from Peter's end. Ensure that both Peter and Sally are aligned with the same version upon concluding this section, which may necessitate a pull if required.

Task 4. Modify-Commit-Conflict that must be manually resolved

The product manager has also asked Peter to change the color of the “Hello World”-message from blue to green when pressing the button (in *HelloWorldFrame.java*). However, due to a misunderstanding Sally thought she was assigned to do this, she also thought the color should be red instead of blue.

- 1) Switch to Sally’s workspace/folder.
- 2) Make a pull for Sally (in Sally’s terminal) so that she has the latest revision. Locate the section in the file containing the information about the color and enter the code for red and save the file. Then make a commit with a suitable commit comment and push it to the server.
- 3) Switch to Peter’s workspace/folder.
- 4) **Without** first making an update with pull, help Peter change the color to green. Save the file! When you are done, try to commit and push the file. If it does not work, you must solve the conflict in it and commit the merge!

***Tip:** switch to Peters workspace in Eclipse and look for rows looking like these and replace it with what it should be:*

```
<<<<<<< HEAD
// This is your local version
=====
// This is the conflicting change from the server
>>>>>>> 9c421ebb4def402d2204d05301aec9b1b07e148a
```

Note: Do not push the commit to the remote after the merge has been resolved.

- 5) What happens? Can this problem be solved like before? Why/why not?
- 6) Why is the use of git rebase not preferred in a team project?

Task 5. Roll-back to an earlier revision

The CEO of the company, who has been under a lot of stress lately, suddenly realizes that it is probably best to go with the blue color after all. The product manager is notified about this and asks Sally to fix it.

- 1) Switch to Sally's workspace and make sure she has the latest version.
- 2) Sally, being a practical developer, thinks the best way to fix this is to make a so-called *roll-back* to an earlier revision instead of changing the code. Help her with this.

Terminal:

Run *git log* and find a suitable commit that you want to check. On the commit you want to check copy the string following "commit" looking something like this 775b4b53e2de4d85f794c5932af3e5f133ffde07.

Then run:

```
git diff HEAD 775b4b53e2de4d85f794c5932af3e5f133ffde07
```

This command will show what will be changed if we roll back or revert that commit. When you find the commit you want to revert run:

```
git revert 775b4b53e2de4d85f794c5932af3e5f133ffde07  
git push
```

- What does the log say now, why?
- What are the key differences between *git reset* and *git revert*? How do these commands affect the files in the staging area?

Examination

When you are done and understand all steps in part A contact your assistant during a lab occasion. Be ready to answer questions concerning what you did when demonstrating. You don't need to hand in anything.

References and resources:

Git
git-scm.com

Part 2 - Continuous Integration (CI)

Introduction

This lab will give you some hands-on experience in using continuous integration tools to automate the integration when members of a team push new/modified code into the remote repository. There are several CI tools to choose from, for example Jenkins, GitLab CI, TeamCity, and Travis CI etc...

In this Lab, you will be able to set up a CI system using GitLab, which is one of the more popular CI tools. In GitLab, you can create projects for hosting your codebase, collaborate on code, and automate all sorts of tasks related to building, testing, and delivering or deploying software continuously with built-in GitLab CI/CD. Builds can be triggered by time or event based.

You will be using the FreeCol codebase (FreeCol is a turn-based strategy game), which resides at <https://gitlab.liu.se/jesji387/group3>, along with GitLab from LiU (<https://gitlab.liu.se/>), and docker hub images available in <https://hub.docker.com/> to build FreeCol application. FreeCol application [<http://www.freecol.org/>] is a Java module, aims to create an open-source version of the game Colonization. FreeCol project's build script, using **Ant**, is configured to build, generate HTML documentation, code coverage reports, automated testing report etc. Apache Ant (<https://ant.apache.org/>) is a Java library and command-line tool with a number of built-in tasks allowing to compile, assemble, test, and run Java applications

Recommended reading before you start working on this lab:

Introduction to CI/CD with GitLab

<https://docs.gitlab.com/ee/ci/introduction/index.html#how-gitlab-cicd-works>

GitLab CI/CD Pipeline Configuration Reference

<https://docs.gitlab.com/ee/ci/yaml/README.html>

Creating and Tweaking GitLab CI/CD for GitLab Pages-

https://docs.gitlab.com/ee/user/project/pages/getting_started_part_four.html

JUnit test reports - https://docs.gitlab.com/ee/ci/junit_test_reports.html

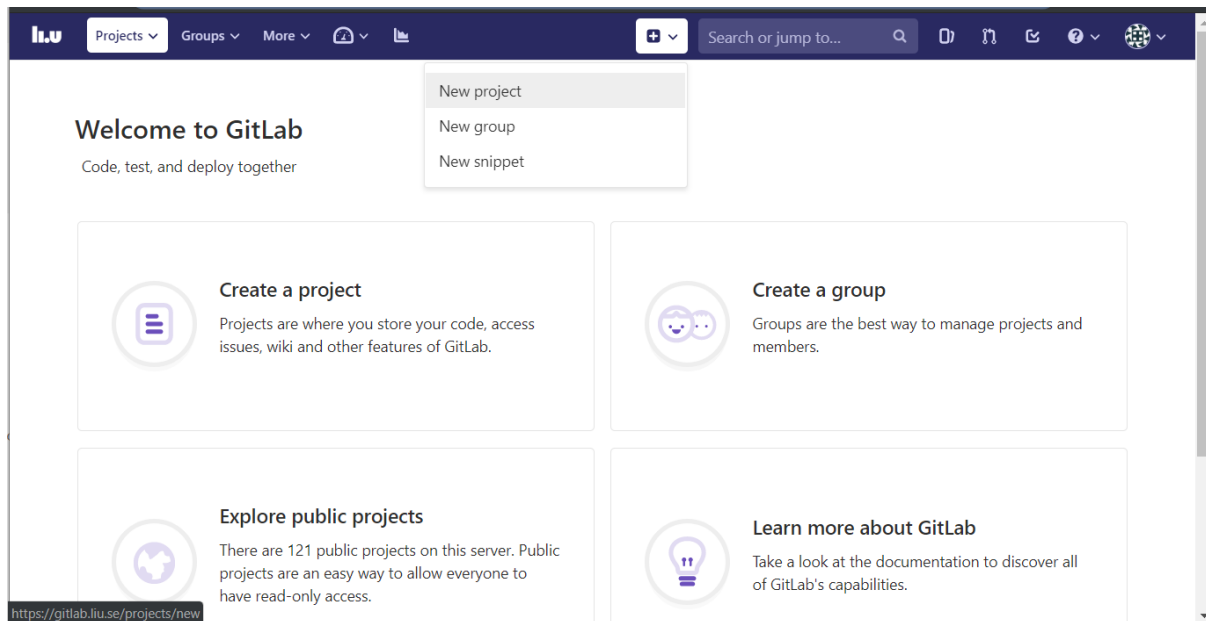
Part A – Tutorial on how to set up a CI using Gitlab.....	11
1. Create a “helloworld” project in GitLab.....	11
2. Clone your remote repository.....	12
3. Working on your local repository.....	13
4. Gitlab CI configuration for “helloworld” project.....	14
Question: Explain each line in the figure above.....	14
Part B – GitLab CI Configuration and Git Branches.....	16
1. Setting up a shared Git repository.....	16
2. Forking the FreeCol repository.....	18
3. Cloning the repository.....	18
4. Run build scripts in FreeCol in your local computer.....	19
5. Set up Continuous Integration in GitLab.....	20

<u>6. Step 4. Create a merge request in GitLab to the original repository.....</u>	<u>21</u>
<u>7. Merge the proposed changes.....</u>	<u>22</u>
<u>8. Fixing the test cases.....</u>	<u>22</u>
<u>9. Merge test fix into the master branch.....</u>	<u>23</u>
<u>Examination.....</u>	<u>24</u>

Part A – Tutorial on how to set up a CI using Gitlab

1. Create a “helloworld” project in GitLab

1. Sign in on <https://gitlab.liu.se/> using your LiU account.
2. In your dashboard, click the green **New project** button or use the plus icon in the navigation bar.

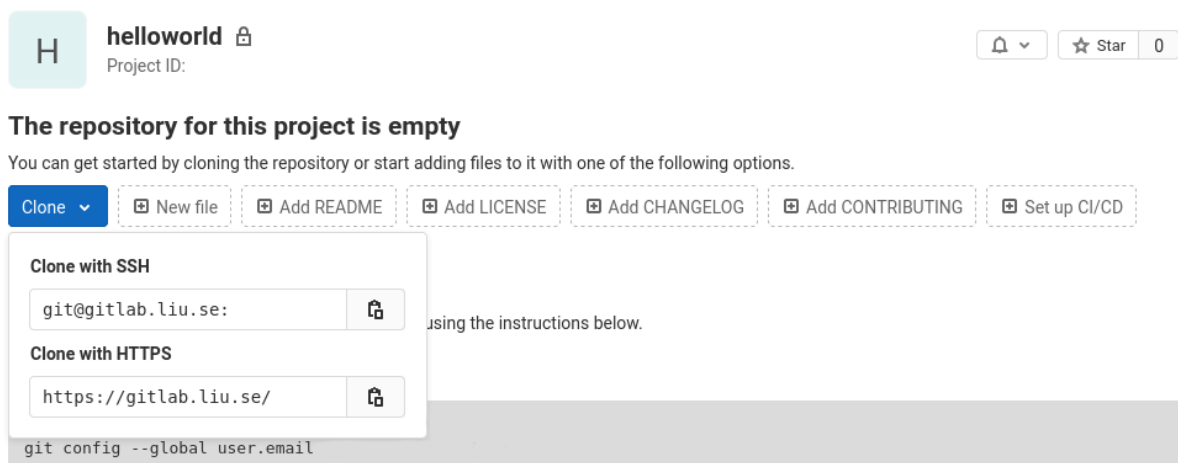


This opens the **New project** page.

Blank project	Create from template	Import project	CI/CD for external repo
<p>Project name</p> <input type="text" value="helloworld"/>			
<p>Project URL</p> <input type="text" value="https://gitlab.liu.se/alash325/"/>		<p>Project slug</p> <input type="text" value="helloworld"/>	
<p>Want to house several dependent projects under the same namespace? Create a group.</p>			
<p>Project description (optional)</p> <div>Description format</div>			
<p>Visibility Level ?</p> <p><input checked="" type="radio"/> Private Project access must be granted explicitly to each user.</p> <p><input type="radio"/> Internal The project can be accessed by any logged in user.</p> <p><input type="radio"/> Public The project can be accessed without any authentication.</p> <p><input type="checkbox"/> 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.</p>			
<input type="button" value="Create project"/>			<input type="button" value="Cancel"/>

3. On the **New project** page, choose the **blank project** tab.
4. On the **Blank project** tab, provide the following information:
 The name of your project (i.e., **helloworld**) in the **Project name** field.
 Select the Initialize repository with a README option to create a README file.
5. Click **Create project**. Now your Git repository is initialized with a default branch (**master**).

2.Clone your remote repository



To start working locally on your remote repository, you must clone (download) a copy of its files to your local computer. You can clone it via HTTPS:

- From your **helloworld** project click on the **Clone** button
- Copy the URL from “**Clone with HTTPS**” field
- Open a terminal in the directory you wish to clone the repository files into, and run the following command:
 - ***git clone [repository_URL]***
 where ***[repository_URL]*** is the URL (remote repository path) you copied.
- Enter your credentials if asked. Once completed, the following actions occur.
- A new folder called **helloworld** initialized as a Git repository is created in your local computer
- A remote named **origin** is created, pointing to the URL you cloned from. Go to **helloworld** folder and type ***git remote -v*** and press **Enter**. You will see the current configured remote repository:
 - **origin https://gitlab.liu.se/<USERNAME>/helloworld.git (fetch)**
 - **origin https://gitlab.liu.se/<USERNAME>/helloworld.git (push)**
- All of the repository's files and commits are downloaded there (for now only the README file)
- The default branch (usually called master) is checked out. Type ***git branch*** and press **Enter**. You will see your branches. A * will appear next to the currently active branch: * **master**.

Questions:

Write a few lines (i.e., what, and why mostly) about each of the following terms, with respect to GIT:

- 1) Origin
- 2) Remote
- 3) Branching (master or other) – Creating a new branch
- 4) Forking
- 5) Merge Request
- 6) Continuous Integration
- 7) Continuous Delivery
- 8) Making a pull request
- 9) Build Script
- 10) .yml file in CI workflow

3. Working on your local repository

1. Create a file called **HelloWorld.java** in you project root, which contains the simplest program of Java printing “Hello World” to the screen.

```
class HelloWorld
{
    public static void main(String args[])
    {
        System.out.println("Hello, World");
    }
}
```

2. Push all your changes to Gitlab (your remote repository):
 - Run the command “git status” to know how many files have been added/changed.
 - Add the file to your local repository and stage it for commit to your local repository:

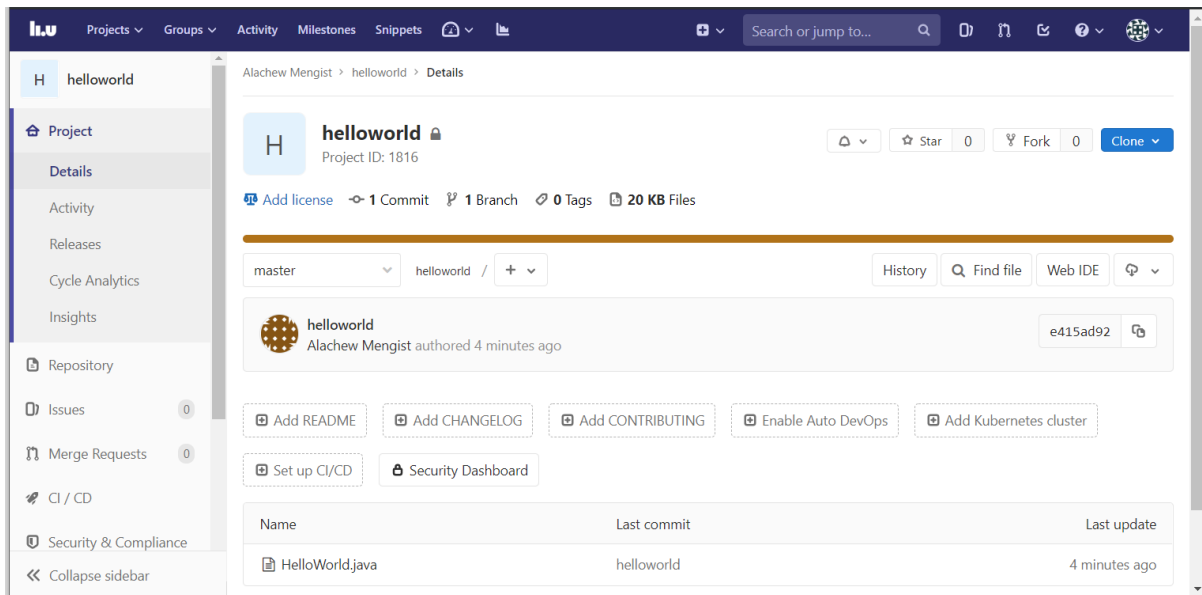
git add . or git add HelloWorld.java

Note: The . character means all file changes in the current directory and subdirectories.

- Commit the file that you've staged in your local repository:
git commit -m "helloworld", where -m stands for message, you want to associate with these changes so you remember, why you made these changes.
- Push all commits in your local repository to GitLab (remote repository)

git push [name_of_your_remote] [name_of_your_branch]

On successful completion, your new file “HelloWorld.java” will be added to your remote repository in GitLab webpage as well as on local computer.



4. Gitlab CI configuration for “helloworld” project

- Create `.gitlab-ci.yml` and save it in your root directory of “helloworld” project. The contents of `.yml` file is as follows:

```
image: openjdk:8
stages:
  - build
build:
  stage: build
  script:
    - javac HelloWorld.java
    - java HelloWorld
artifacts:
  paths:
    - ./HelloWorld.*
```

NOTE: YAML does **not** allow tab indentation, so use 2 spaces instead. If you are using vim to create the ci file, run `set ts=2 sw=2 et` and add a line containing `# vi: ts=2 sw=2 et` at the bottom of the file. Ensure that the file `~/.vimrc` contains the line `set modeline`.

Question: Explain each line in the figure above.

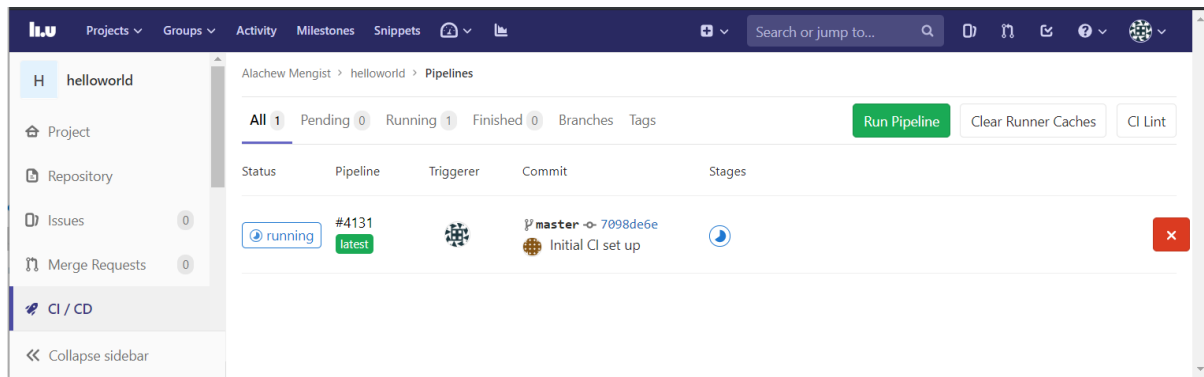
- Push all your changes to Gitlab (your remote repository)

`git add .`

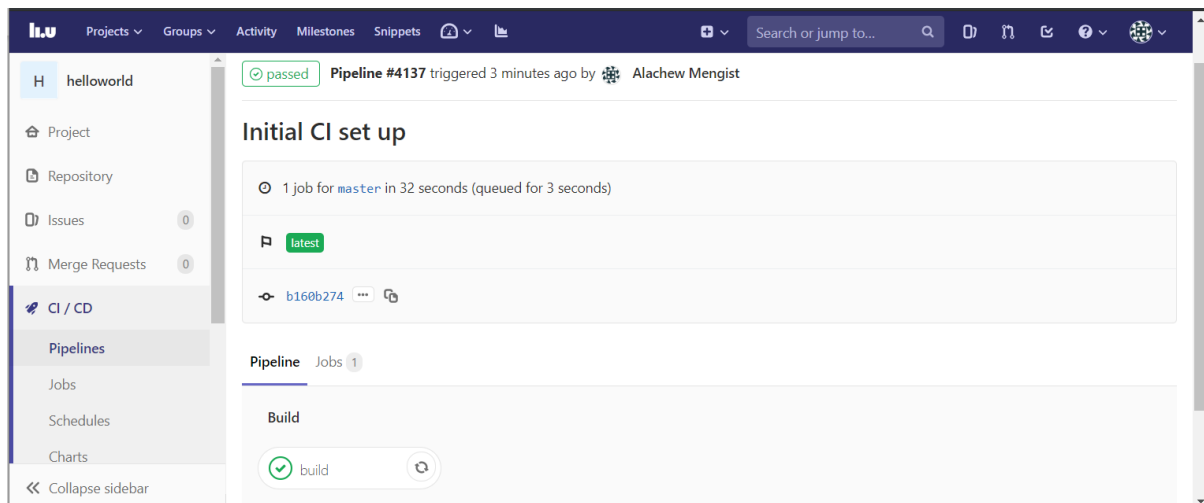
`git commit -m "initial CI"`

`git push [name_of_your_remote] [name_of_your_branch]`

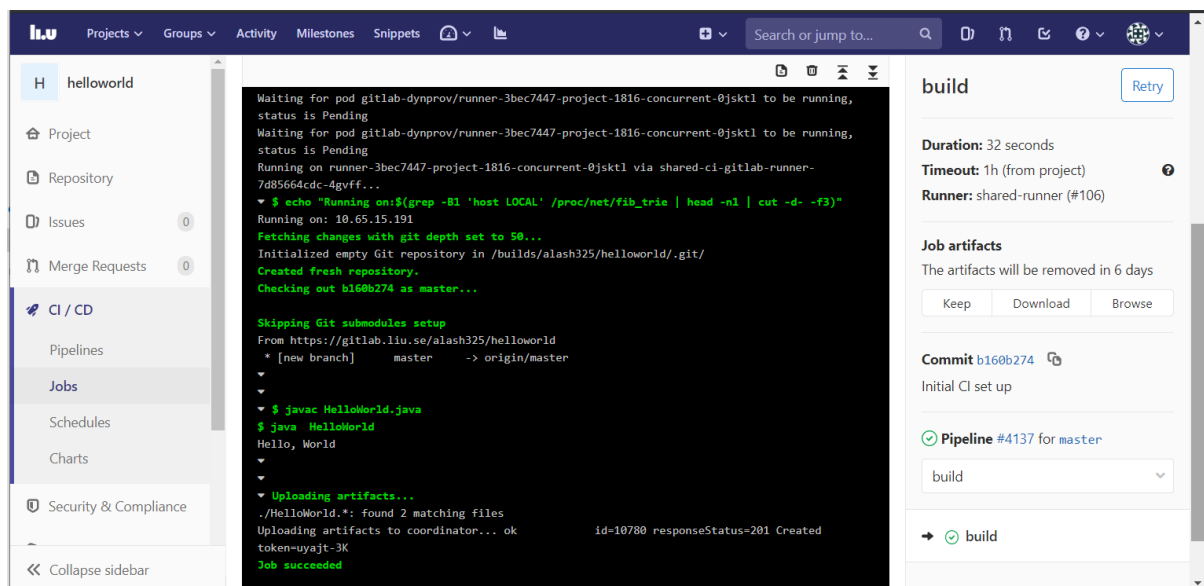
On successful completion, GitLab CI automatically starts to execute the jobs you’ve set. Go to option on left side on gitlab.liu.se “Build” and click on “pipelines” (Build → Pipelines). You should see the status of you last commit changes from **pending** to either **running**, **passed** or **failed** as shown below (see your Gitlab repository).



You can view all pipelines by going to the **Pipelines** page in your project.



In GitLab CI, Runners run the code defined in **.gitlab-ci.yml**. GitLab CI not only executes the jobs you've set, but also shows you what's happening during execution, as you would see in your terminal. You can also view if artifacts were stored correctly using the build artifacts browser that is available from the build sidebar.



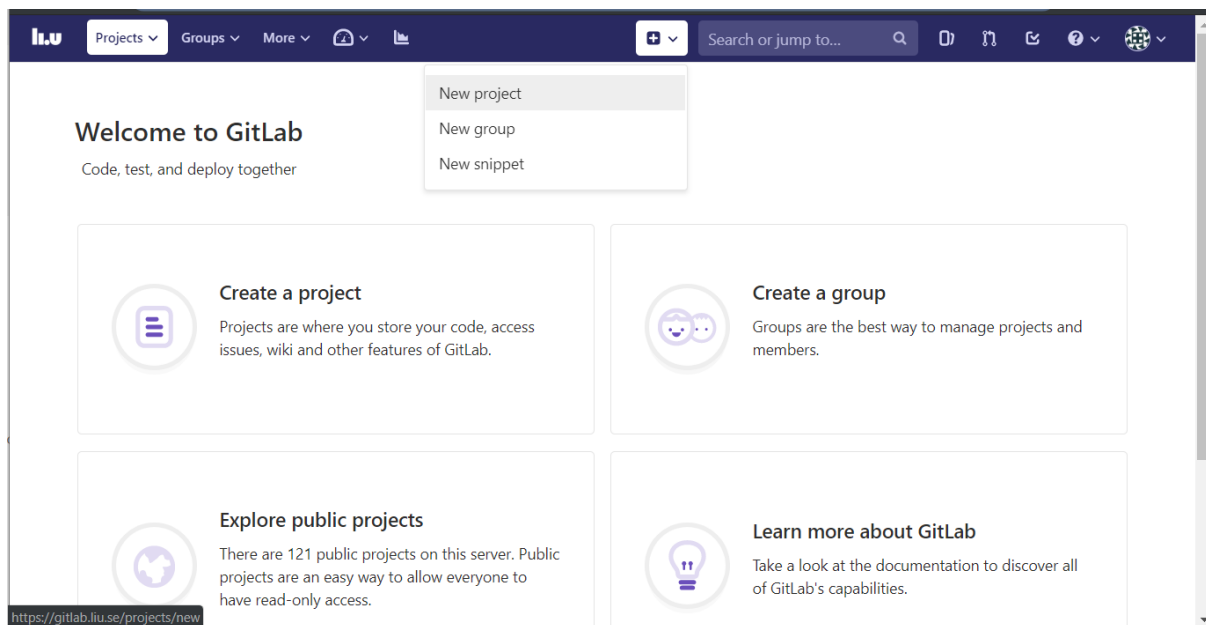
Part B – GitLab CI Configuration and Git Branches

In Part B, you have to work in pair, and the project owner (In this case student-A) want the GitLab CI tool to continuously integrate, build and test his FreeCol project automatically triggered by push event, and publish the test results.

1.Setting up a shared Git repository

In this task, you need to import an existing FreeCol repository via HTTP by providing the Git URL from the New Project page:

- Sign in on <https://gitlab.liu.se/> using “**Student_A**” LiU account



- From your GitLab dashboard click **New project**
- Switch to the **Import project** tab
- Click on the **Repo by URL** button
- Fill <https://gitlab.liu.se/jesji387/group3.git> in the “Git repository URL”. Use Public Checkbox. By default, it is private.
- Click **Create project** to begin the import process

liu

Projects Groups More

Search or jump to...

New project

A project is where you house your files (repository), plan your work (issues), and publish your documentation (wiki), [among other things](#).

All features are enabled for blank projects, from templates, or when importing, but you can disable them afterward in the project settings.

To only use CI/CD features for an external repository, choose **CI/CD for external repo**.

Information about additional Pages templates and how to install them can be found in our [Pages getting started guide](#).

Tip: You can also create a project from the command line. [Show command](#)

Blank projectCreate from templateImport projectCI/CD for external repo

Import project from

GitLab export

GitHub

Google Code

Fogbugz

Gitea

git Repo by URL

Manifest file

Git repository URL

https://gitlab.liu.se/tddb84/freecol.git

Username (optional)

Password (optional)

- The repository must be accessible over `http://`, `https://` or `git://`.
- If your HTTP repository is not publicly accessible, add your credentials.
- The import will time out after 180 minutes. For repositories that take longer, use a clone/push combination.
- To import an SVN repository, check out [this document](#).
- Once imported, repositories can be mirrored over SSH. Read more [here](#).

☐ Mirror repository

Automatically update this project's branches and tags from the upstream repository every hour.

Project name

My awesome project

Project URL

https://gitlab.liu.se/alash325/

Project slug

freecol

Want to house several dependent projects under the same namespace? [Create a group](#).

Project description (optional)

Description format

Visibility Level

☐ Private

Project access must be granted explicitly to each user.

☐ Internal

The project can be accessed by any logged in user.

☒ Public

The project can be accessed without any authentication.

Create project

Cancel

Once completed, you will be redirected to your newly created FreeCol project accessible at `https://gitlab.liu.se/<student_A_LiU_ID>/freecol`, which will be your original remote repository.

Your task is now to configure CI for FreeCol project using built-in GitLab CI/CD and propose changes to the original repository (i.e., to the **GitLab repository reside in student A**). You are expected to set up the following stages: build, test, and publish HTML test report page in GitLab. The second requirement for this lab is to create a new branch with git whenever you want to add a new work and push it to the remote/original repository via create merge request.

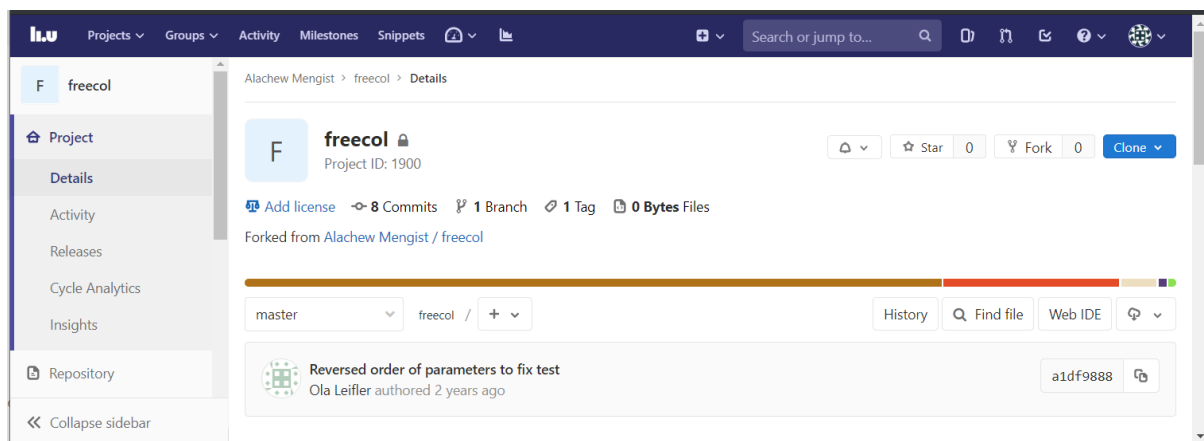
2. Forking the FreeCol repository

A fork is a copy of original repository. Forking a repository allows you to freely experiment with changes without affecting the original project. In order to push to the original repository, you will push to your own forked repository and create a merge request from your forked repository in GitLab.

- Sign in on <https://gitlab.liu.se/> using “**Student_B**” LiU account
- Navigate to the FreeCol repository of **student_A** in your browser (https://gitlab.liu.se/<student_A_LiU_ID>/freecol)
- In the top-right corner of the navigated page, click **Fork**
- Select your namespace to fork the project
- Now if you look at the URL of your forked repository it is changed into **<student_B_LiU_ID>**:

https://gitlab.liu.se/<student_B_LiU_ID>/freecol

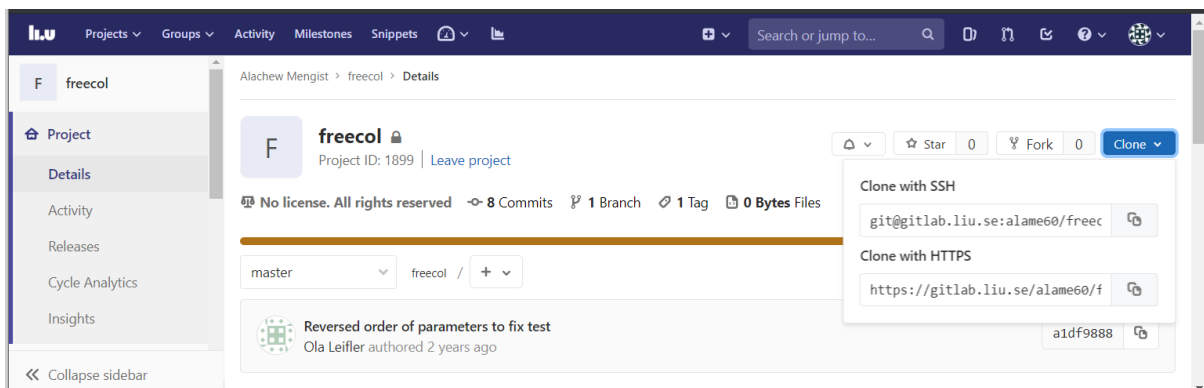
It also shows you where your repository is forked from (see Figure below).



3. Cloning the repository

Log in on the lab terminal as **Student_B**.

To start working locally, you must clone (download) a copy of the original repository files (available in **Student_A** GitLab repository) to **<Student_B>** local computer. You can clone it via HTTPS:

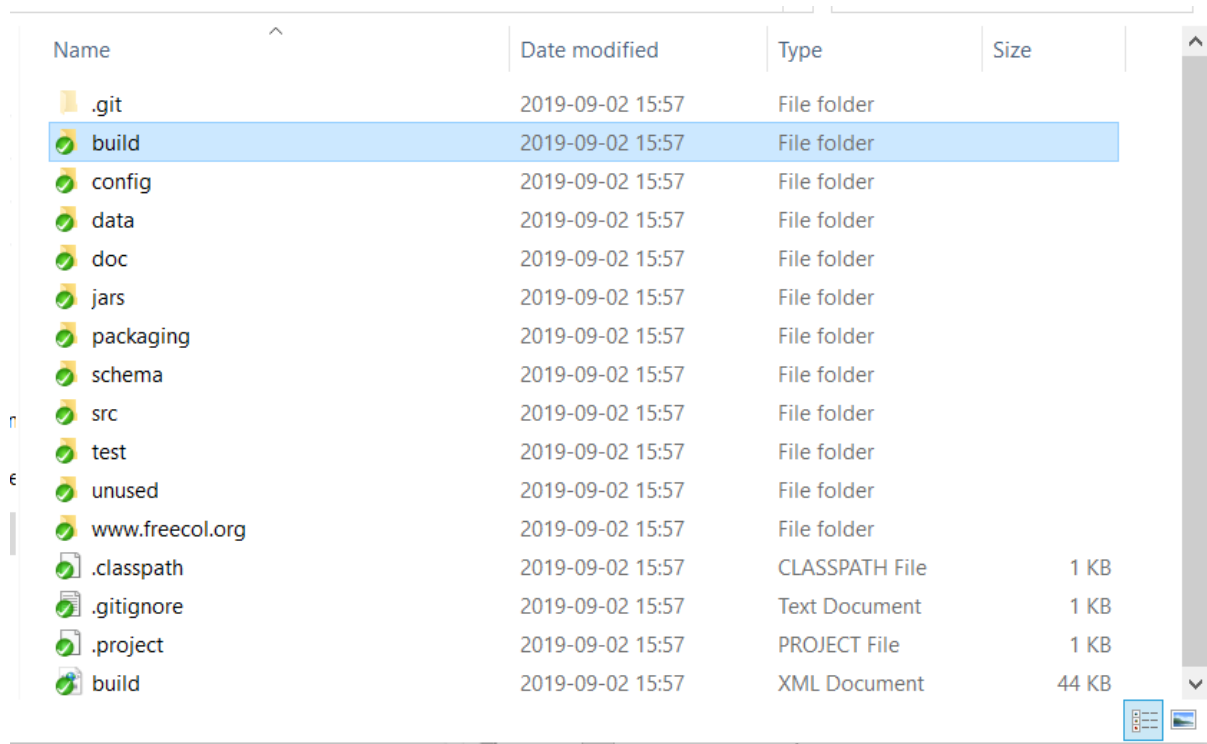


- Go to https://gitlab.liu.se/<student_A_LiU_ID>/freecol
- Click on the **Clone** button

- Copy the URL from “Clone with HTTPS” field
- Open a terminal in the directory you wish to clone the repository files into and run the following command.

git clone [repository_URL] , where ***[repository_URL]*** is the URL (original remote repository path) you copied.

After cloning the repository successfully, you should see the copy of files in your local git repository as shown below (see the directory where you cloned the repository).



Name	Date modified	Type	Size
.git	2019-09-02 15:57	File folder	
build	2019-09-02 15:57	File folder	
config	2019-09-02 15:57	File folder	
data	2019-09-02 15:57	File folder	
doc	2019-09-02 15:57	File folder	
jars	2019-09-02 15:57	File folder	
packaging	2019-09-02 15:57	File folder	
schema	2019-09-02 15:57	File folder	
src	2019-09-02 15:57	File folder	
test	2019-09-02 15:57	File folder	
unused	2019-09-02 15:57	File folder	
www.freecol.org	2019-09-02 15:57	File folder	
.classpath	2019-09-02 15:57	CLASSPATH File	1 KB
.gitignore	2019-09-02 15:57	Text Document	1 KB
.project	2019-09-02 15:57	PROJECT File	1 KB
build	2019-09-02 15:57	XML Document	44 KB

Now add the forked repository as well:

- Run **git remote add <Student_B> <URL to Student_B's fork>**

git remote -v should now output the following lines:

origin https://gitlab.liu.se/<Student_A>/freecol.git (fetch)

origin https://gitlab.liu.se/<Student_A>/freecol.git (push)

<Student_B> https://gitlab.liu.se/<Student_B>/freecol.git (fetch)

<Student_B> https://gitlab.liu.se/<Student_B>/freecol.git (push)

4.Run build scripts in FreeCol in your local computer

As mentioned in the introduction, FreeCol application uses the **Apache Ant** build script. By default, **Ant** uses **build.xml** as the name for a buildfile. You can find the FreeCol buildfile in your local repository (see **./build.xml**). It includes targets for building FreeCol, distribution packages, running tests, creating documentation etc.

Now try to build, run tests, and create documentation for FreeCol java application using the **Ant** build script:

- Open a terminal in the directory where your FreeCol project is copied and run the following command:
- **ant build** to build the FreeCol app

- **ant -lib test/lib/junit.jar -Dtest=AllTests testall** to run the JUnit tests and create a browsable HTML report (see the result **index.html** file under **./build/report/**).

NOTE: if the “ant” command does not work. Skip to next stage. We have provided the docker image including “ant” below.

5.Set up Continuous Integration in GitLab

The workflows required to have working CI and pass this lab is summarized as follows:

1. Create a new branch
2. Add your code for CI configuration. Your CI configuration should include the following:
 - Build the FeeCol application.
 - Run the FreeCol test suite.
 - Store the result as a build artifact (for the test suite).
 - Publish the test results with GitLab pages.
3. Push your changes to your forked repository.
4. **Create *merge request*** in GitLab to the original repository (i.e., to student A).
5. Merge the proposed changes in GitLab (from student_A Gitlab repository).

You are free to work with your own starting point but If you have trouble finding a starting point, follow the following steps:

1. Create a new branch called “**CI_<student_B_id>**” on your local machine and switch in this branch:
git checkout -b <your_new_branch_name> where **<your_new_branch_name>** is “**CI_<student_B_id>**”
 This will switch your local git repository to a new branch called “**CI_<student_B_id>**”. You can verify this by running **git branch** in your terminal. The output shall be:

```
* CI_<student_id>
  Master
```
2. Create a new file called **.gitlab-ci.yml**. The first line of yaml file i.e a Docker image to run your script written in .gitlab-ci.yml file is given below.
image: alash325/javaant:latest
3. Complete other parts using tutorials given in Part A and reference materials or your own material.
4. Once you are satisfied with your CI configuration, push your changes to your forked repository with **git push <Student_B> CI_<student_b_id>**.

On successful completion, GitLab CI automatically starts to execute the jobs you've set as shown below (Go to https://gitlab.liu.se/<student_B_LiU_ID>/freecol and see your Gitlab repository).

The screenshot shows the GitLab CI/CD interface for the 'freecol' project. The 'Pipelines' section shows a single pipeline (#4138) in a 'running' state. The 'Jobs' section shows three jobs: 'pages' (skipped), 'test' (failed), and 'build' (passed). The 'Jobs' section is highlighted with a red box, and the 'CI/CD' menu item is also highlighted with a red box.

Status	Job	Pipeline	Stage	Name	Timing	Coverage
skipped	#10859 master bc0c4445	#4184 by	deploy	pages		
failed	#10858 master bc0c4445	#4184 by	test	test	00:00:24 17 hours ago	
passed	#10857 master bc0c4445	#4184 by	build	build	00:02:37 17 hours ago	

Wait until all the jobs (stages) are executed. Once completed, go to **CI/CD->Jobs** from your project page and see the result for all the jobs you were expected to complete (i.e., build, test, publish html pages). If the results are as shown below in the graph which can be interpreted as: The CI for build job is **passed**, test job is **failed**, and publish html page **skipped**, then go to **Step 4** otherwise try to fix your CI configuration until you reached the expected results. The reason for the test job failure is due to the sound mixer is not available in our computer system. However, for the GitLab runner to execute and publish html page for the test, you are expected to fix those tests otherwise GitLab runner skips to execute the deploy (publish html page) stage. You are going to do this in **step 8**.

6.Step 4. Create a merge request in GitLab to the original repository

Go to https://gitlab.liu.se/<student_B_LiU_ID>/freecol

The screenshot shows the GitLab repository page for the 'freecol' project. The 'Create merge request' button is highlighted with a red box. The 'Details' section shows the project ID, commit count, branch count, tag count, and file count.

- Click on **Create merge request** button

From `alash325/freecol:CI_alash325` into `alame60/freecol:master` [Change branches](#)

Title

Start the title with **WIP:** to prevent a **Work In Progress** merge request from being merged before it's ready.

Description

Write **Preview**

Write a comment or drag your files here...

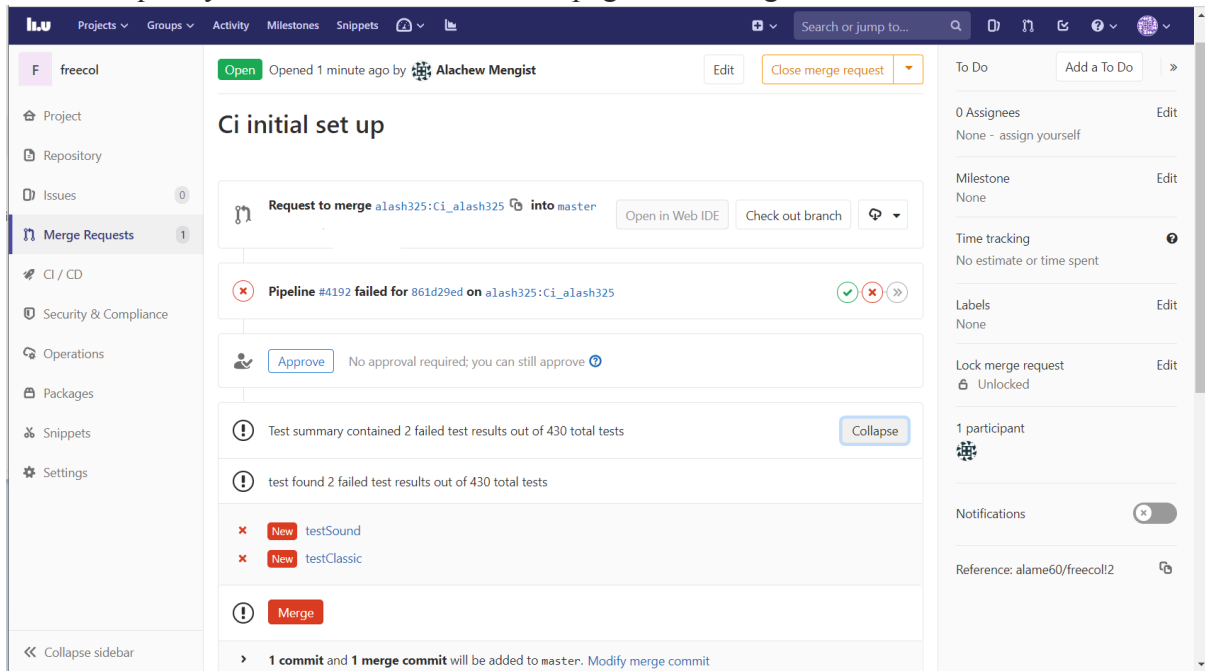
- Click on **Submit merge request** button

7. Merge the proposed changes

- Go to your original repository

https://gitlab.liu.se/<student_A_LiU_ID>/freecol

- Go to the **Merge Requests** tab and open the merge request. Once you open the merge request you will be redirected to the page like the figure below.



- Click on **Merge** to merge the proposed changes to your repository

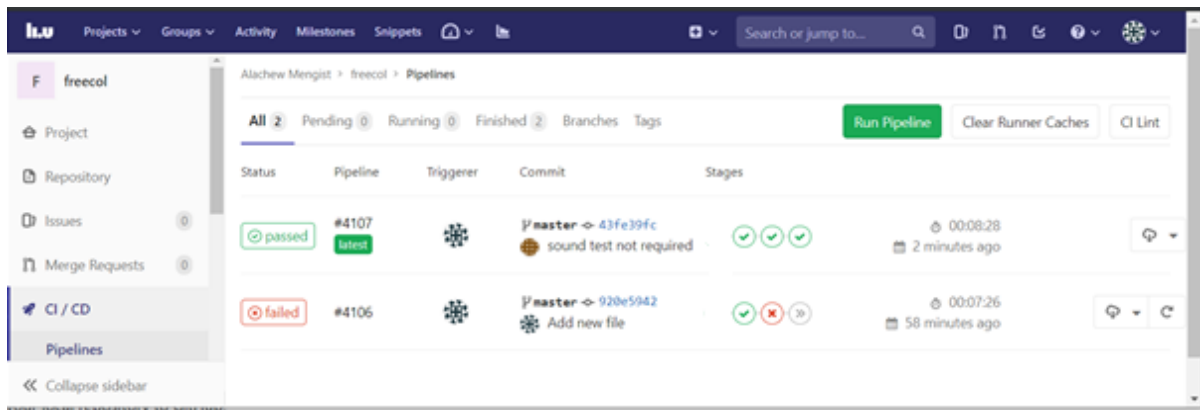
8. Fixing the test cases

As mentioned, the reason for the test job failure is due to the sound mixer is not available in our computer system. However, for the GitLab runner to execute and publish html page for the test, you are expected to fix those tests. In this lab, we will fix just by removing those tests. Instructions are given below.

1. Check out to the original repository i.e., git checkout master
2. Pulling changes from a remote repository to retrieve new work done by other people and combines your local changes with changes made by others
3. Create a new branch called “**Fix_tests**” and checkout your new branch
4. Go to **AllTests.java** in `./test/src/net/sf/freecol/common/AllTests.java` and comment the following line of code:

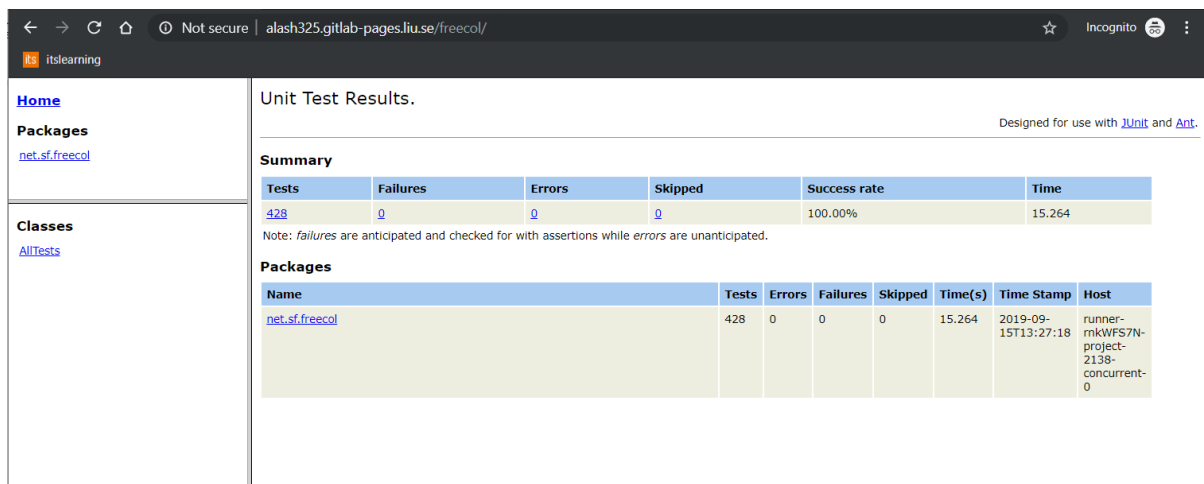
```
// suite.addTest(net.sf.freecol.common.sound.AllTests.suite());
```
5. Push your changes (See **Step 3 in Task 4**)

After pushing your changes successfully, GitLab CI automatically starts to execute the jobs you’ve set and you can view the status of the Pipelines.



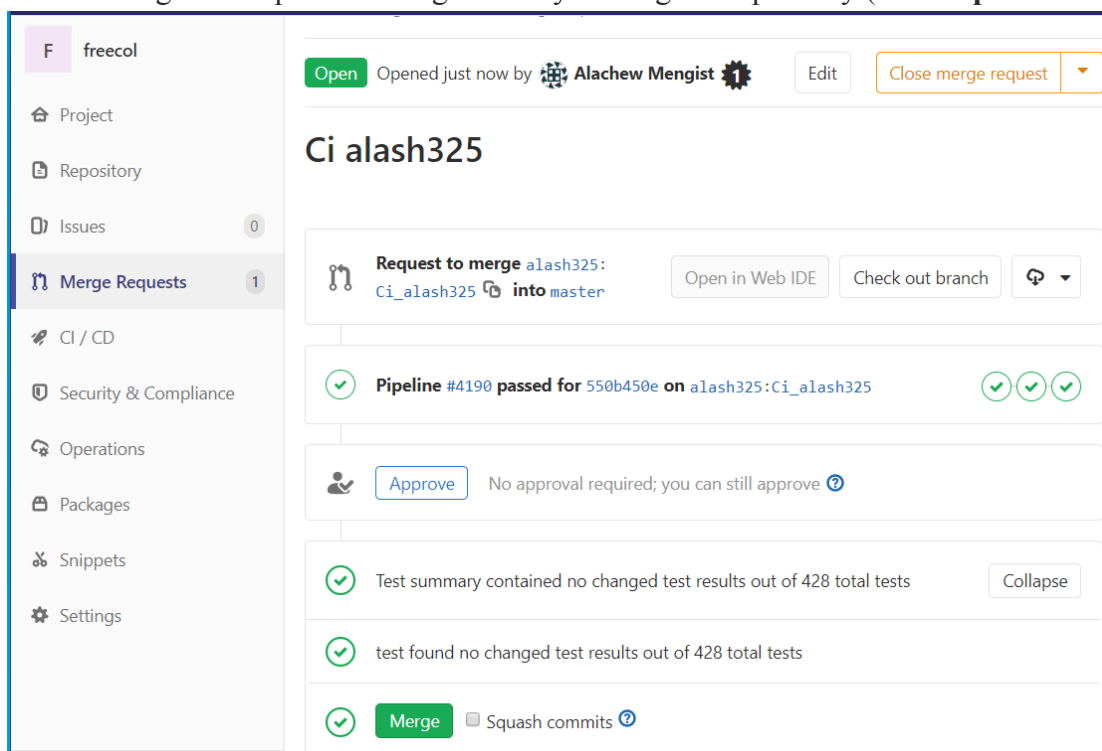
After successful Pipelines you can view your clickable HTML pages by going to your projects **Deploy->Pages**.

You can view your HTML report by clicking the pages your found.



9. Merge test fix into the master branch

1. Create a merge request (See Step 5 in Task 4)
2. Merge the requested changes from your original repository (See Step 5 Task 4)



Examination

When you are done with all tasks and ready to demonstrate your solutions, contact your assistant during a lab occasion. Be prepared to show the workflows described in Task 4 and answers to questions. The lab assistants may ask you to send in these answers for later evaluations.