COM1001 Introduction to Software Engineering

Version Control with Git

1 GitLab

For this lab session you will need to create your own git repository. To do this, open a web browser and navigate to the following URL:

`https://git.crossover-project.org.uk`

This is the GitLab installation provided for COM1001 (formerly the Crossover Project). GitLab is a web-based Git repository manager, similar to the well-known GitHub. You can log in to the website using your CICS username and password.

Once logged in, you can create a new repository by clicking on the "New project" button:

Choose a name, for example "COM1001". You can provide a description if you want to, but this is optional. The access level determines who gets to see and access the repository. Private repositories can only be accessed by you or any user you explicitly give access to. Internal repositories can be accessed by any authenticated user, i.e., any student taking COM1001. Public repositories will be accessible to anyone on the internet. Make "COM1001" a private repository.

GitLab will now show you a page with details on how to access the repository. You can choose either SSH or HTTPS; for SSH you need to upload your public key by following the link shown. The advantage of using SSH is that you can set up authentication without password, whereas for HTTPS you will always need to enter your password if you access the repository. Choose the method you prefer (if you haven't used SSH before you might want to use HTTPS to start with, although SSH has advantages) and note the URL shown.

2 Git on the Command Line

After creation on GitLab, the remote repository is empty (not initialised). This means that you can take any local Git repository and push it to the server, or you can pull from the remote repository and will get an empty local repository. The GitLab page shows you the commands to create a new repository locally and then push that to the remote repository just created on GitLab, and this is what we will do. To better understand how Git works, we will start by using it at the command line again, using the `git` tool. For this exercise, start by launching a command line prompt (On Windows, select 'Start Command Prompt with Ruby' from the 'Ruby 2.1.6' entry).

1. Create a new directory (`mkdir COM1001`)
2. Change into that directory (`cd COM1001`)
3. Create a new local Git repository (`git init`)
4. Query the status of your repository using `git status`. Are there any files?
5. Create a simple hello world Ruby script `lab.rb` in this directory, which prints out "Hello World".
6. Query the status of your repository again using `git status`. Are there any files now?
7. Add the `lab.rb` file to the repository using `git add lab.rb`
8. Query the status of your repository using `git status` (note the status of the added file.)
9. Commit this change to the repository using `git commit -m "<`*Some meaningful message*`">`. Try to form a habit from writing descriptive commit messages, it will be useful in the long run.
10. Query the status of your repository using `git status`. Are there any files with changes now?

11. Now let's push the changes to the remote repository on GitLab. As Git is intended to be used in a distributed way rather than with a centralised server, we first need to tell Git where the remote clone is located. You need to use the URL GitLab gave you; below is an example (but don't forget to change the URL):

    `git remote add origin https://git.genesys-solutions.org.uk/ac1gf/COM1001.git`

    After this, you can push the changes to your local repository to the remote clone using

    `git push -u origin master`

    Note that if you try to push again, Git will automatically use the same remote repository again. That is, if you push again, you will just need to write `git push`, and if you use `git pull` it will also default to the same remote repository.

12. Now change `lab.rb` such that it reads the name of the user, and then prints out a greeting to that name instead of "world".

13. Query the status of your repository using `git status`; note that the status of the file changed.

14. Commit your changes again, using an appropriate commit message: `git commit -m "`*<Some meaningful message>*`"`

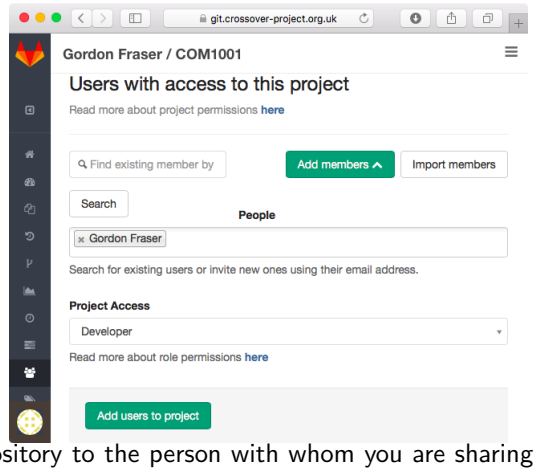
15. Query the history of changes of `lab.rb` using `git log lab.rb`. There should now be two entries for your last two commits.

16. Push the changes to the remote clone again. This time you do not need to specify the remote clone, simply type `git push`

17. If you look at the GitLab page for this repository in your browser, you should see all your activities on this repository documented there.

3 Sharing Repositories in GitLab

We will now work in teams of two on the same software project. Find a partner (e.g. just ask your neighbour) and decide on one of your two GitLab repositories to work on. On the chosen repository, the owner needs to grant write access to the other team member. To give access to other users, find the "Members" menu entry on the GitLab page of your repository. There, click on "Add members", and enter the username of your colleague into the "People" text field. Choose "Developer" project access, such that your colleague gets write access to the repository.



Give the URL of your repository to the person with whom you are sharing the repository. (If you are not sitting next to each other, how about using Email, Google Hangouts, or similar to collaborate remotely?) Each of you should now have the repository URL of a repository created by someone else.

4 Git in RubyMine

In order to do some more serious programming, let's use RubyMine with Git this time. Each of the team members should now launch RubyMine, and create a new project by selecting "Check out from Version Control" in the RubyMine welcome dialog. Select "Git", enter the URL of your repository (see GitLab), and then click on "Clone". You should now see the file `lab.rb` in the project.

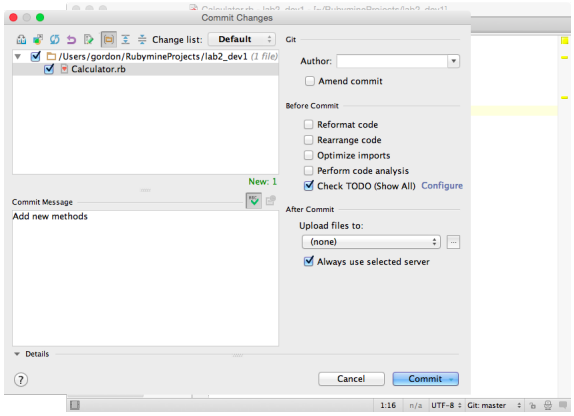
Note: If you have already opened a project in RubyMine, you need to close the project (File → Close Project) to get to the dialog to create/checkout new projects. Make sure you use the correct URL provided by GitLab. If you have not set up SSH, then you will need to use the HTTPS-based URL!

Your task is now to implement a Ruby program that reads a decimal value and two currency codes (e.g. "EUR" or "GBP"), and then converts the number from the first currency to the second currency. The program consists of six methods, which should all be implemented in the file `lab.rb`. Split the implementation task of these methods between the two of you. In the first step, each partner implements one of the following two methods *on their own computer*. Obviously, you will need to synchronise to make sure you are not both implementing the same method:

Developer 1: Write a method `print_error` with one parameter that is an error message, writes this error message to the output, and quits the program execution using `abort`.

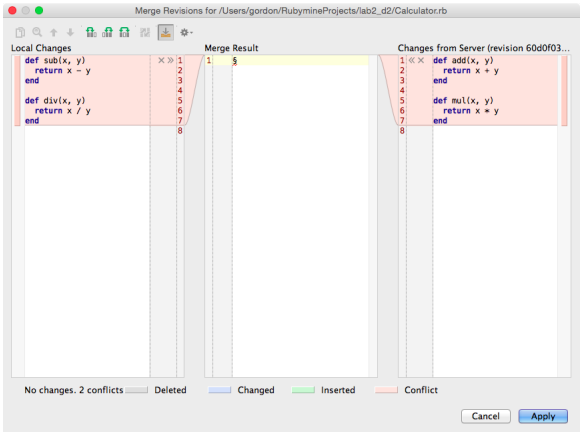
Developer 2: Write a method `print_greeting` that simply prints a welcome message (e.g., "This program converts values from one currency to another.").

Now both commit your changes. In RubyMine, this is done by selecting the menu item VCS → Commit Changes. This will show you the "Commit Changes" dialog where you can select which changed files to commit, and enter a commit message.



In case RubyMine asks you whether to add the ".idea" directory as well, select "no". This is the place where RubyMine stores some project metadata that is irrelevant for this exercise.

- Click "Commit and push" (which appears if you hover your mouse over the "Commit" button), such that the changes are in the local clone as well as the remote repository. (You could also select only "Commit", in which case only the local clone is updated. You can then push your changes to the remote clone at a later point with the menu "VCS" → "Git" → "Push"). In the push dialog, just accept the suggested master clone.
- The second person to push their changes will receive information that the remote repository has changes that need merging; you will have to update your own clone first. This is done by selecting "Merge" in the shown dialog. In the following dialog, RubyMine will offer you a choice of how to deal with the merge conflict. The choice is to "Accept Yours", in which case the remote changes are overridden, "Accept Theirs", in which case your own changes are lost, or "Merge", in which case you will get a chance to merge your changes with the remote changes. Click on "Merge", and then you will get to see a three-way merge dialog:



Here you can manually select which changes to add. Make sure you find a way to integrate both your changes!

- Once the merge is complete, RubyMine will save the merge in your local repository, but inform you that the push was rejected. To push your merge, select "VCS" → "Git" → "Push".
- After that, the other developer will have to pull as well to ensure you are both working with the same version. To do so, select "VCS" → "Update Project". RubyMine will ask you about the "Update Type" (just use the default of "Branch Default"), and how to clean the working tree before update (again, the default of "Using Stash" is fine.)

Now let's implement two more methods, each of which performs some simple input validation checks. Again, each of the partners in a pair selects and implements one of the methods:

Developer 1: Write a method `check_number` that takes a string parameter as input and verifies that it represents a valid floating point number. A simple way to do this is to convert the string to a number (`to_f`), and the number back to a string (`to_s`), and then check whether the result was equal to the original string. If the input is not a floating point number, then call the `print_error` method to quit with an error message such as "`Input has to be a valid number`".

Developer 2: Write a method `check_code` that takes a parameter as input and verifies that the parameter is a valid code (i.e., it is either "GBP", "EUR", or "USD"). If it is not, then quit with an error message such as "`Input has to be a valid currency code`" by calling the `print_error` method.

Commit and push your changes again. Resolve any merge conflicts that may arise. Once you both have the same version of the program, implement the last two remaining bits. Again, each of you should implement only one of the two tasks:

Developer 1: Write a method `convert` that takes three parameters: A floating point number, and two currency codes. The method should return the converted numerical value. Here are some recent conversion factors:

GBP	EUR	1.27
EUR	GBP	0.78
USD	EUR	0.80
EUR	USD	1.24
GBP	USD	1.59
USD	GBP	0.63

To check which currency codes were passed in, you need to use conditional statements such as `if`. Note that in this example the result is dependent on two conditions (the value of the first and of the second currency code). To achieve this, you can either nest `if` conditions:

```
if condition1 then
  if condition2 then
    ...
  end
end
```

Or, you can connect several conditionals using the `&&` (and) and `||` (or) operators you should already know from Java.[1]

```
if condition1 && condition2 then
 ...
end
```

If a combination of currency codes is not known then use the `print_error` method to output an error message and quit the execution.

Developer 2: Write a method `main` where you read the value as well as the two currency codes from the user and store them in variables. For each of these values, check that they are valid using `check_number` and `check_code`. If they are valid, pass the parameters to `convert` (don't forget to convert the value to a floating point number), and assign the return value of `convert` to a variable. Finally, print out the converted value. Also add a call to `main` to the end of the script, outside any method definitions, such that the script can be run with RubyMine or by invoking it on the command line.

Once again, commit and push your changes to synchronise your work spaces.

Now the program is complete, and should work for both developers in a team. For example, an interaction with the program could look like this:

```
Please enter a value: 112.34
From currency: USD
To currency: EUR
112.34 USD = 89.87 EUR
```

Here is an example of how an error case could look like:

```
Please enter a value: 112.34
From currency: XYZ
Error: XYZ is not a valid currency code.
```

[1] In Ruby, there are also alternative versions of these operators denoted as `and` and `or`, but these have different precedence than the operators you know from Java, so if you want to use them make sure you read the docs.