Week 8

GIT and Continuous Integration

Robert Merkel has written an outstanding article on Git which is accessible at following link (also available in week 8 as a mandatory readings).

https://www.alexandriarepository.org/module/introduction-to-version-control-with-git/

In addition, I have added few important sections here which explains extra concepts that are specific to GIT INFOTECH server on Monash and general practice in using GIT. READ CAREFULLY.

URL for GIT Infotech

Git infotech is accessible at https://git.infotech.monash.edu/. It is only accessible through https.

NO SSH FOR GIT INFOTECH

Git server on Git Infotech is not accessible through ssh so use https to clone your repository.

ACCESS TOKENS

You cannot access Monash GIT through your authcate password - weird? Well we have a solution. You need to create an ACCESS TOKEN before accessing it. Here I demonstrate how to create an access token. Press the image on the right top corner and press settings.

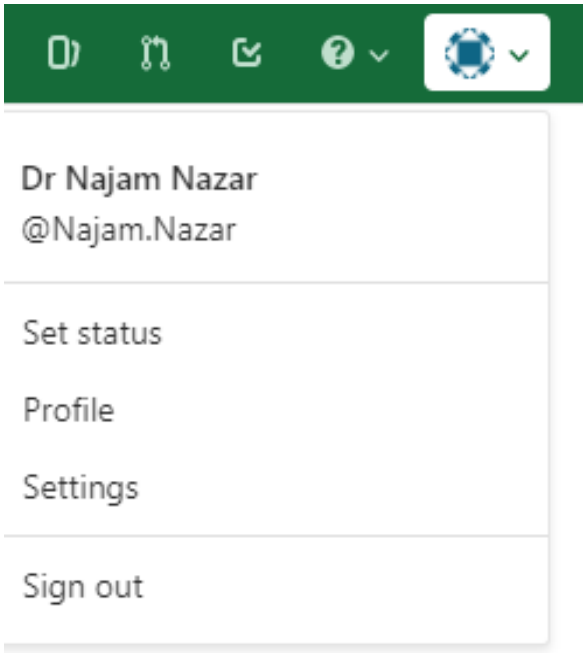

Figure 1: Accessing Settings.

On pressing settings, I will see an access token page as shown in Figure 2.

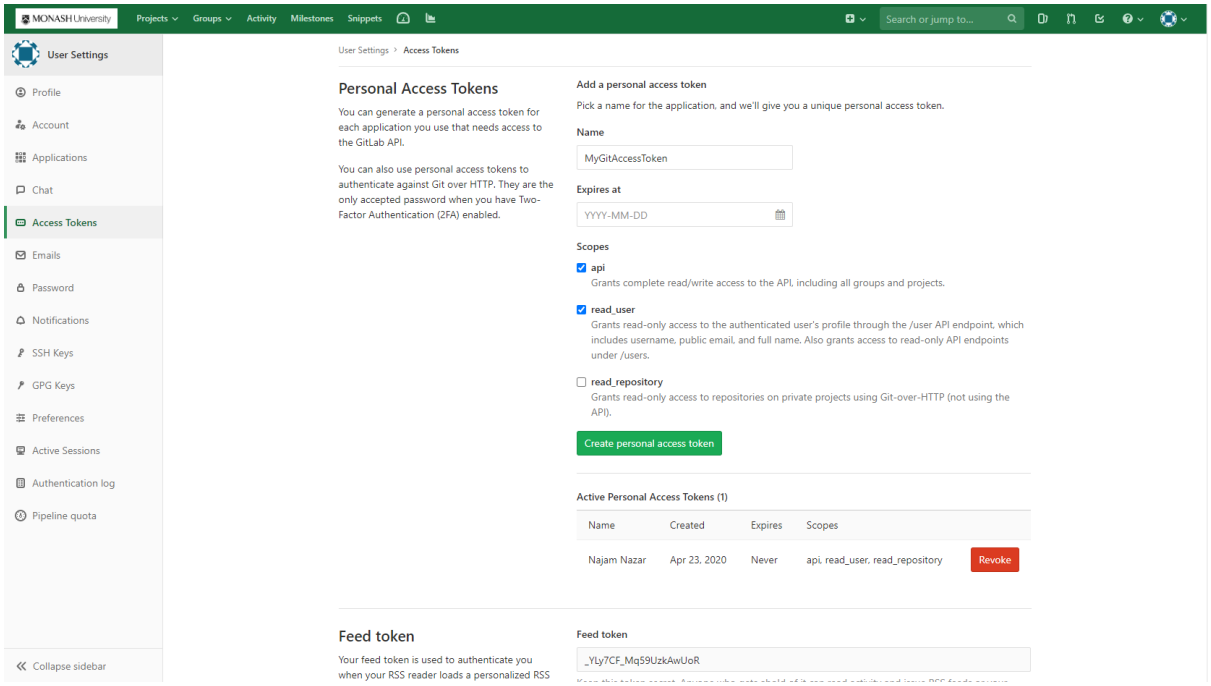

Figure 2: Access Token Page.

You can write anything in the name section, I mostly use the name of the application. Tick api, read_user add read_repository options.

VERY IMPORTANT:

- Do not set the expiry date. Then it will never expire. If you put the expiry date, you can't access the repository after the expiry date.
- Expiry date is not date of birth so do not put your date of birth in the 'expires at' text box.
- Use access token as password when you clone or access your repository through GIT clients.
- Most importantly save the access token at a secure place. I mostly save it in a text document.

GIT CONFIGURATION FOR PROXY

You may need to configure a proxy server if you are having trouble cloning or fetching from a remote repository or getting an error like `unable to access '...' Couldn't resolve host '...'`. etc.

- Open a command prompt
- Run the following commands replacing USERNAME, PASSWORD, PROXY_ADDRESS, and PROXY_PORT with your network's information:

```
git config --global --add http.proxy http://USERNAME:PASSWORD@PROXY_ADDRESS:PROXY_PORT

git config --global --add https.proxy http://USERNAME:PASSWORD@PROXY_ADDRESS:PROXY_PORT
```

You may need to set in manually through credential manager on Windows OS.

GIT LIFE CYCLE

General workflow is as follows:

- You clone the Git repository as a working copy. This is done through a CLONE command.
- You modify the working copy by adding/editing files.
- If necessary, you also update the working copy by taking other developer's changes. This is done by using a PULL command.
- You review the changes before commit.
- You commit changes. This is done through the COMMIT command.
- If everything is fine, then you push the changes to the repository. This is done through PUSH command.
- After committing, if you realize something is wrong, then you correct the last commit and push the changes to the repository.

VERY IMPORTANT: Always pull the repository before committing and pushing the changes into the remote repository.

The GITIGNORE file

Always add `.gitignore` file as instructed in the document shared on Moodle.

GIT Clients

Almost every IDE provides a built-in functionality for versional control systems. If you know the basic concepts of clone, pull, push, fork, commit etc you can use it easily. I always prefer using GIT through command line or through and IDE such as VSCode. A git cheat sheet which contains a list and usage of basic commands is provided on Moodle readings.

I am not providing a detailed resource here, however; I am providing the list of some common third-party git clients. You can use any you like.

- Gitkraken accessible at https://www.gitkraken.com/
- Github Desktop accessible at https://desktop.github.com/
- Tortoise Git accessible at https://tortoisegit.org/

To use git with command line on Windows you can download it at https://git-scm.com/. To install it on Linux, follow the instructions at https://git-scm.com/download/linux.

Integration Testing

Software integration is the process by which the individual units which make up a software system are integrated together to make a complete system.

The only reason one has to integrate some of the software individual units of a complete system into something that isn't the complete system, is to test the integration of those units in one way or another. Therefore, the objective of any integration strategy should be to allow the most effective, efficient testing of the system possible.

In other words, integration testing and integration are peas in a pod. Without one, the other is useless.

We will start with a look, mainly for historical and box-ticking reasons, at some "traditional" approaches to integration, and then look at the modern context where continuous integration is practiced.

Top-down, bottom-up, sandwich integration

“Top-down, bottom-up, sandwich integration is gritty, they're just about as bad as the big bang city, this is old skool (integration)” -- with no credit to Beats International

If you read a software engineering textbook, or SWEBOK, on the topic of integration, they will spend quite a lot of time rabbiting on about lots of different approaches.

We will start with the straw man known as "big bang integration". In this approach, the developers write every single module in the system independently, unit test them, and only once they are all complete stick them all together and test the completed system.

Think about what would happen if you actually tried this for a minute. Potentially hundreds, or thousands, of faults, interacting with each other in unpredictable ways, spreadeagled across a huge system. About the only thing worse would be to do this without adequate system testing. And of course, nobody would try that, would they?

As it turns out, people did. Aside from the Queensland Government payroll system mentioned in the first chapter, there was an even higher-profile example in the United States. In 2013, as part of health care reforms that became known as "Obamacare", the US government created healthcare.gov, a complex website allowing people across the United States to compare and purchase health insurance plans. The site went live on October 1, 2013, after "a couple of days" of end-to-end system testing. It remained largely unusable for weeks afterwards and proved a major embarrassment to the Obama administration.

Anyway, big-bang integration, even with thorough system testing, is widely recognized as a bad approach, so a number of alternatives, more incremental approaches were devised. Schach provides a moderately readable description of these, which we will summarize. Consider a simple report

generation system that collects information from the file system and the network and produces a report by converting XML into a PDF. The system's class diagram is below:

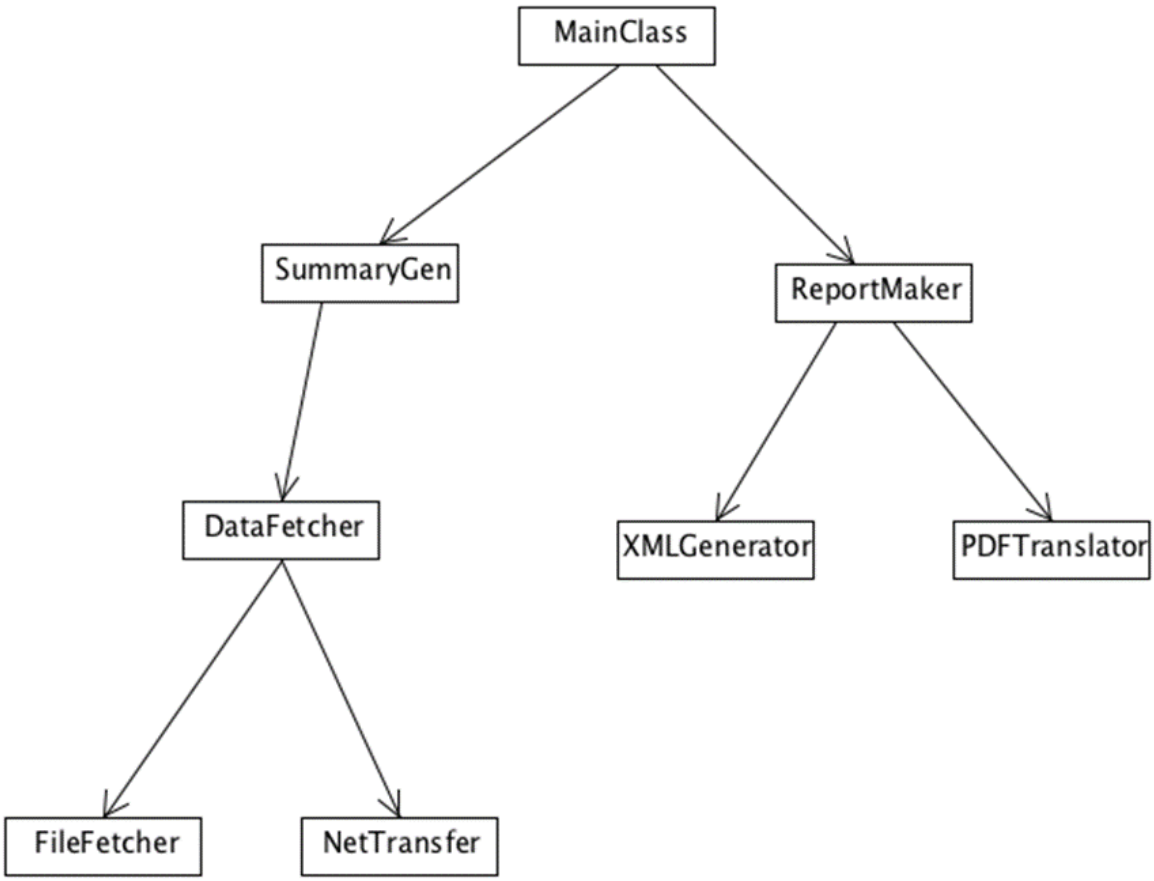

Figure 3: Class dependency diagram for a hypothetical reporting application.

To perform top-down testing, we would start with the module has no in-arrows (that is, it initiates interaction with other classes, but no classes initiate interaction with it), MainClass. We would write stubs for SummaryGen and ReportMaker to test it. Once it is tested, you would replace the stubs for SummaryGen and ReportMaker with their real implementations, write stubs for DataFetcher, XMLGenerator and PDFTranslator, and test that. Next, you would replace the stubs, add stubs for FileFetcher and NetTransfer, and test that. Finally, you would test the entire system as a whole.

By contrast, bottom-up testing would see you write drivers to test the four "leaf" units, FileFetcher, NetTransfer, XMLGenerator and PDFTranslator first. You would then integrate them with DataFetcher and ReportMaker, write drivers for those, and test the two assemblages. Finally, you would test the whole thing.

Schach then proposes a compromise method, sandwich testing, in which certain units near the bottom are designated "operational" units, which are tested bottom-up, and units near the top are designated logic units, and are tested top down, until you meet in the middle.

You may have observed that I am not going through these in a whole lot of detail. That's because they were never a particularly good idea and make far less sense now than when textbooks like Schach's were originally written in the 1990s.

Just in case we are not clear here, this is what I think of these integration approaches in the modern software development context:

Figure 4: Conventional approaches to integration.

What is wrong with these integration strategies? Lots:

- In any of the conventional integration approaches discussed, you do not end up with a working (if partially functional) system until right at the end of the process. This is a non-starter for any Agile methodology worth the name.
- They require masses of testing glue code.
- They require masses of very repetitive testing.
- It takes a long time for code to be exposed to the rest of the system.
- The nature of associations between modules in OO languages is not anywhere near as simple as the one-directional associations depicted in figure 1. Dependency inversion is now a thing, for instance.

Furthermore, the technological context in which integration is performed has completely changed. It used to be the case that just compiling a large piece of software could take many hours. These days, even behemoths like Firefox and the Linux kernel can be built from scratch in a few minutes on a high-end machine. Many applications are implemented in interpreted languages like Python which have no separate compilation step at all; adding functionality and running it is potentially near instantaneous.

Additionally, centralized source control repositories are now standard practice, making it trivial to make new code available to other developers as it is developed.

So virtually all projects using Agile methods, and an increasing number run under alternative process models, manage integration using Continuous Integration (CI).

Continuous Integration

Martin Fowler has written an outstanding description of continuous integration, which you should read in its entirety; but the very short summary is:

- Keep your code in a repository (duh).
- Everybody commits their work to the trunk (master/head/whatever you want to call it) daily, or if possible, more frequently.
- The trunk is always kept in a runnable state.
- Every commit triggers a build.
- All builds trigger the tests to be run.
- Fix anything that breaks the build.
- Test on something that matches the production environment as closely as possible.
- Make the resulting executable available and easy to install.

Again, YOU SHOULD READ MARTIN FOWLER'S ARTICLE! Accessible on Moodle.

For this to work, the more comprehensive your test suite, the better. At a minimum, you should ensure that as well as "pure" unit tests, or heavily mocked unit tests, you have tests that get large parts of this system to work together. Best practice, however, is that you have automated system tests as well. For systems with graphical user interfaces, that can mean using systems like Selenium to drive the user interface to perform system testing.

While it is not essential, many teams doing CI will use a continuous integration server to automate the builds and tests. A CI server will:

- Trigger an automated build of the system when a new revision is checked in.
- Run tests (and possibly analyse test coverage) on the new build.
- Make the results of running those tests conveniently available.

There are several continuous integration servers available, both commercial and open source. In the past, we have used Jenkins for CI in our testing units at Monash. It is a powerful and versatile CI server and can be configured to do all manner of things. Unfortunately, it is a bit of a pain in the backside to configure and run, and (at least when we tried to keep it running) had a nasty habit of crashing periodically. In recent years there are many companies offering CI servers as a service, sometimes free but normally for a monthly charge. For example, Travis-CI, Drone.io, CircleCI , to name a few.

In this unit, we are going to use a shiny new CI system, Gitlab. Gitlab has the advantage that your version control system, and your CI system, are fully integrated. This makes setting up Gitlab very, very easy, particularly if you are using Gitlab's CI servers rather than your own (though that is easy too). As mentioned, we use Gitlab installed on Monash server which we call Git Infotech.

Robert Merkel has written an excellent article on continuous integration and it is accessible at following link: (A MANDTAORY READING.)

https://www.alexandriarepository.org/module/introduction-to-continuous-integration-servers-with-gitlab/

As you see, I have created a repository for you all on git so with every commit you make the CI works automatically.

For more details on CI/CD and other concepts used in Git read the resources provided in the week 8 readings on Moodle.