Git Development Model

We will be using a git development model to facilitate multiple developers working on the same project.

The estimated time to work through this tutorial is 2 days.

1. Read this article on a successful git branching model:

http://nvie.com/posts/a-successful-git-branching-model/

You need to understand the purpose of a master, develop, feature and release branch.

2. For NG-STAR we will have two main branches:

master – The master branch is the main branch that contains released versions of the project to the UAT (User Acceptance Testing) server. After each release to UAT, we will Tag the release with a version number. The UAT server is for internal testing purposes by Irene and Walter. It is located in ngstar-uat.corefacility.ca

development – Development branches off of the master branch and contains code that passes all tests but is currently not released. The development branch contains functionality that will be included in the next release to UAT (and eventually production). During a release, we merge the development branch to the master branch and tag the master branch HEAD with a version number. Code from feature branches will be merged into the development branch via merge requests.

feature branches – A feature branch is branched off of development each time a developer wants to implement a particular feature. A feature is encapsulated in a single feature branch which makes it easy for multiple developers to work on separate features without disturbing the main codebase (development). The functionality contained in a feature branch are meant to be included in a future release. After all tests have passed, a feature branch is merged back into the development branch by submitting a merge request to another developer on the team. During a merge request, the other developer reviews the code and accepts the merge request if everything is ok. When the merge request is accepted the feature branch is merged into the development branch as part of the next release.

Eventually, when the project goes into a public release you will need to create a **production** branch (which will be **master** in this case) and you will need to move **UAT** to its own separate branch. Production will be **master** (on its own production server), **UAT** (on its own UAT server) will branch off of **production** and merge back into **production**, **development** will branch off of **UAT** and merge back into **UAT**, and **feature branches** will branch off of **development** and merge back into **development**.

Ideally, you will want a production server (prod) that serves the public, a UAT server (uat) that serves internal users (Irene and Walter) for testing before changes are deployed to production,

and a development server (dev) which will be a mirror of the uat and prod server **environment** that you can use to test the latest version of the application from your development branch (to ensure that the application passes all tests on a production-like server environment).

When setting up the servers the git branching model will have a natural translation:

Git Branch	**Server**
master	prod
uat	uat
development	dev
feature branches	locally on your machine only

3. Read this tutorial on using branches in git:

https://www.atlassian.com/git/tutorials/using-branches/

For this tutorial, create a project on GitLab and follow along with the examples. You will need to understand git branching extremely well for our git development model so the best way to do this is to try some examples.

4. Understand how to create a local branch and push a local branch remotely:

http://stackoverflow.com/questions/1519006/how-do-you-create-a-remote-git-branch

5. Understand how to delete a local branch and push deletion of a branch remotely:

http://stackoverflow.com/questions/2003505/delete-a-git-branch-both-locally-and-remotely

6. Read this tutorial on Feature Branch Workflow:

https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow

Create a project on GitLab and follow the example.

7. Read this tutorial on Gitflow Workflow:

https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Create a project on GitLab and follow the example.

8. Read these articles on Continuous Integration:

http://en.wikipedia.org/wiki/Continuous_integration
http://www.thoughtworks.com/continuous-integration

Git Development Model Tutorial

1. Create a new GitLab project and clone the project using git clone.

2. Add a file to the GitLab project with the following contents:

```
#include <iostream>
using namespace std;

int main(){
    cout << "Hello World!" << endl;
    return 0;
}
```

Save this file as hello.cpp

3. Add the files to staging, save changes made to staging and push to master:

```
git add -A
git commit -m 'Initial commit'
git push origin master
```

4. View git branches by running the following command:

```
git branch
```

5. Create a development branch that branches off of master. The `git branch <branch_name>` command creates a new branch that branches off of the current branch, which is master:

```
git branch development
```

6. View your git branches again. You should see a development and a master branch with the current branch set to master.

7. Push the development branch remotely:

```
git push origin development
```

8. Switch your current branch to development:

```
git checkout development
```

9. When viewing your git branches, you should see that you've switched to the development branch:

```
git branch
```

10. Make a changes to hello.cpp as follows and save the changes:

```
#include <iostream>
using namespace std;

int main(){
    cout << "Hello World!" << endl;
    cout << "Goodbye World!" << endl;
    return 0;
}
```

11. Next, add and save to staging and push your changes to the development branch by running the following commands:

```
git add -A
git commit -m 'Added to development branch'
git push origin development
```

12. On GitLab, go to **Files**. By switching between master and development in the dropdown you should see the difference between the files of each branch.

13. On GitLab, take a look at **Network** to see the branching structure. You should see an image that looks like this:

14. Next, create a feature branch off of development called my-cool-feature. First, make sure that you are on the development branch. If you are not on the development branch, then switch to the development branch:

```
git checkout development
```

15. Create a feature branch called my-cool-feature:

```
git branch my-cool-feature
```

16. Push the my-cool-feature branch to the remote repository:

```
git push origin my-cool-feature
```

17. View your git branches. You will see that you are currently on the development branch. Switch to the my-cool-feature branch:

```
git branch
git checkout my-cool-feature
```

```
git branch
```

18. Next, make the following changes to hello.cpp in the my-cool-feature branch and save the changes:

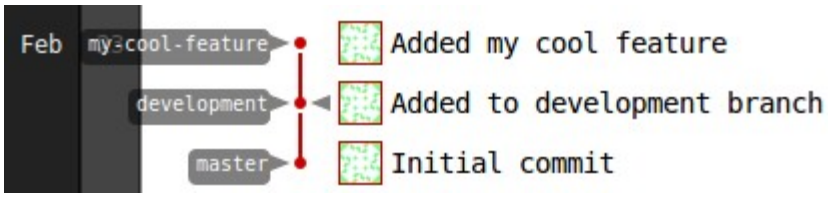
```
#include <iostream>
using namespace std;

int main(){
    cout << "Hello World!" << endl;
    cout << "Goodbye World!" << endl;
    cout << "This is my cool feature!" << endl;
    return 0;
}
```

19. Next, add and save to staging and push the changes to the remote repository:

```
git add -A
git commit -m 'Added my cool feature'
git push origin my-cool-feature
```

20. On GitLab, go to **Files** and toggle between master, development and my-cool-feature in the dropdown to see the difference in the hello.cpp file.

21. On GitLab, take a look at **Network** to see the branching structure. You should see an image that looks like this:



22. Next, submit a merge request to a team member. In this case, you will act as the team member to review the code and accept the merge request. What this merge request will do is merge the functionality contained in the my-cool-feature branch to the development branch. To submit a merge request on GitLab follow these instructions:

a) Click on **Merge Requests**
b) Click on **New Merge Request**
c) Set source branch to the branch you want to merge into development, which is your feature branch called my-cool-feature.
d) Set target branch to the branch that source branch will merge into, which is development.
e) Click on **Compare branches**
f) Include an appropriate title
g) Include a good description that will be useful to the person reviewing and accepting the merge request.

h) In the **Assign to** field, this will normally be set to a team member. You will assign the team member this merge request. The team member will receive the merge request and review the code that was written as part of your feature branch. If everything looks ok, the team member will then accept the merge request and your feature branch will be merged into the development branch. But for now (and only in this tutorial), you will act as the team member. Click on **Assign to me**.
i) Take a look at the **Inline Diff** and the **Side-by-side Diff** between the files in my-cool-branch and development branch.
j) Once you're done taking a look at the diff, click on **Submit merge request**.

In the actual project, you need to make sure that all tests pass before submitting a merge request (as well as before committing code to your branch).

23. On GitLab, click on **Project**. You will see that your merge request has been posted as an activity for other team members to see.

24. You will now be acting as the other team member who has been assigned a merge request. In the actual project, you will be assigned a merge request when the team member has completed the functionality in their feature branch which gives you the chance to review their code. To accept a merge request on GitLab (by assuming that you are the other team member), follow these instructions:

a) Click on **Merge Requests**
b) To view merge requests that have been assigned to you to accept, click on **Assigned to me**.
c) Click on the merge request that has been assigned to you.
d) In this page it shows the branches we are merging (From my-cool-feature into development), the title of the merge request, the description, the commits and most importantly the diff under the **Changes** tab. During a merge request you will be reviewing the code written by the team member before their feature branch is merged into development. To do this, click on the **Changes** tab.
e) Review the code by looking at the **Inline Diff** or the **Side-by-side Diff**. The diff compares the code in the feature branch (my-cool-feature) and the current code in the development branch.
f) Let's assume that we spotted a problem with the team member's code. We can let them know there is an issue by making a comment under the **Discussion** tab. Click on the **Discussion** tab.
g) Submit a comment for the team member explaining the problem. For now, just submit a comment such as "You can make some improvements here ..." and click on **Add Comment**. We will not be accepting this merge request until the other team member has fixed their code.
h) Next, on GitLab click on **Project**. You will see the comment made by your team member saying "You can make some improvements here ...". If your team member makes comments make sure to review the comments and make the appropriate changes.
i) Let's assume we are fixing our code based on the comments our team member made. Ensure that you are currently on the my-cool-feature branch. Change hello.cpp to have the following contents and save it:

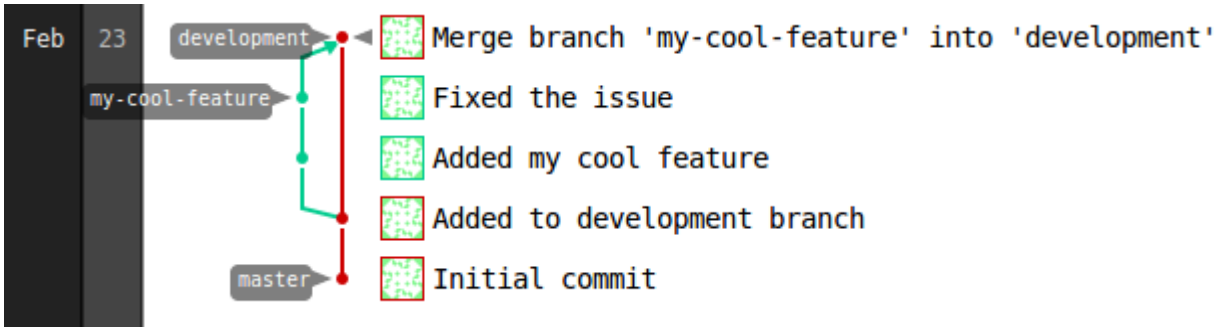
```
#include <iostream>
using namespace std;
```

```
int main(){
    cout << "Hello World!" << endl;
    cout << "Goodbye World!" << endl;
    cout << "This is my cool feature!" << endl;
    cout << "I am fixing my cool feature!" << endl;
    return 0;
}
```

j) Add and save the changes to staging and push the changes to the remote repository:

```
git add -A
git commit -m 'Fixed the issue'
git push origin my-cool-feature
```

k) On GitLab, go to **Files**. You can toggle between the development branch and the my-cool-feature branch and observe that we will eventually merge two lines of code from the my-cool-feature branch to the development branch.
l) Once you have fixed all the issues by possibly making multiple separate commits to the my-cool-feature branch, you can inform your team member that you have completed all fixes by clicking on **Merge Requests**.
m) Click on **Created by me**.
n) Click on the merge request that you submitted before.
o) Inform your team member that you have completed all fixes by adding a comment in the Discussion by saying "I have fixed all issues, can you review my code again?". Click on **Add Comment**.
p) Next, we will be assuming the role as the other team member again (the one that you submitted your merge request to and is reviewing your code for the merge). On GitLab, go to **Project**.
q) The team member has posted a message saying that they are ready to have their code reviewed again. To review their code again, go to **Merge Requests**, click on **Assigned to me** and click on the merge request that you have been assigned to.
r) You can review the team member's code again by clicking on the **Changes** tab and looking at the diff to review the code and compare the changes between branches. In this case, we will assume that the code looks ok and that we have not spotted any issues. Since this is the case, we can go ahead and accept our team member's merge request.
s) To accept the team member's merge request, click on **Accept Merge Request**.
t) Next, on GitLab, go to **Project**. You will see that there is an activity message saying "Merge branch 'my-cool-feature' into 'development'". This means we have successfully merged our team member's my-cool-feature branch into the development branch.
u) On GitLab, go to **Files**. You should see that the development branch will contain all the code added from the my-cool-feature branch.
v) On **Network**, you should see a branching diagram such as this one:



You can see that we have a branch called my-cool-feature (the green branch) that initially branches off of the development branch (the red branch). After accepting the merge request, we can see that the my-cool-feature branch has merged into the development branch. The timeline goes from the down and up.

25. Next, switch out to the development branch:

```
git checkout development
```

26. You can see that we are now on the development branch:

```
git branch
```

27. In your local files, take a look at hello.cpp in the development branch. You can see that the changes from the merge have not been reflected to hello.cpp yet.

28. In the local version of the project, make sure you pull all changes that resulted from the merge (your local files will not reflect the merge unless you pull) by running the following command:

```
git pull origin development
```

29. Take a look at hello.cpp in the development branch again. We are now updated with what is in the remote repository.

30. Since we have successfully merged the my-cool-feature branch into the development branch, we no longer need the my-cool-feature branch locally or remotely.

31. To delete the my-cool-feature branch locally, run the following command (if you are ever unsure, create a backup of your project by creating a tarball of your project):

```
git branch -d my-cool-feature
```

If you run into a message that says "error: The branch 'my-cool-feature' is not fully merged" then you may have forgotten to pull all the changes after the merge (see step 28).

32. We can also delete the my-cool-feature branch remotely by running the following command:

```
git push origin --delete my-cool-feature
```

In general, for the actual project, each team member will create a feature branch off of development for each new feature or bug fix that they have chosen to work on (for simplicity, we will treat bug fixes and requests from clients are features to implement, meaning we will create a separate feature branch for each of these). Each feature branch should implement only a single feature that is well defined (if possible). Once the team member has completed implementing their feature and all tests have passed, they can submit a merge request that they will assign to the other team member. The other team member will be responsible for evaluating this merge request by reviewing the code in the diff. If the code looks ok then the team member will accept the merge request and the feature branch will be merged to the development branch. If there are issues or suggestions relating to the code in the merge request then the team member that has been assigned the merge request will make the appropriate comments and post the comments. The other team member will then fix the code, commit their changes (making sure all tests pass before committing) and inform the other team member that they have fixed all issues. The other team member will review the code again the the process will repeat itself (reviewing, commenting, fixing issues) until there are no longer anymore issues in the merge request in which the reviewer will finally accept the merge request. After the merge request has been accepted, the team member who submitted the merge request can then locally and remotely delete the feature branch that was just merged (they should make sure they have a backup beforehand).

You have successfully completed the tutorial!