U3a Siteworks GitHub

The repositories are all in an organization called u3a-siteworks-development, whose address is https://github.com/u3a-siteworks-development.

All repositories are public, which means anyone can read them. Everyone in the team has at least write access and some have administration access. If you feel you need more rights than you have then any administrator can change them.

Checking out

To clone a repository, open the main page of the repository, and click the drop-down "code" button.

Make sure the HTTPS tag is selected in the dropdown and click the 'copy' icon next to the https address of the repository.

Open a command prompt - (windows or linux or mac) and create a directory then cd into it. Alternatively -go to your plugins directory if you want to work directly in it (for example c:\users\ed\Local Sites\rc-98\app\public\wp-content\plugins).

git clone paste the repo name and hit enter.

(For example - git clone https://github.com/u3a-siteworks-development/u3a-siteworks-theme.git)

cd into the new repository and check the contents are as you expect.

git version - tells you which version of git you installed.

git branch - tells you which branch of the repo you cloned (main by default).

git status - tells you whether you have any local changes - you should not have any.

git help – gives you a command list.

git help <command> gives more details.

Next, you make your code changes in whatever way you like, test and then come back to the command prompt to deliver. I want to create a new branch called es-demo-change. It is useful if all branches begin with your initials, or some recognisable string of your choice.

Checking in

git branch - should still say main.

git branch es-demo-change - creates a new local branch called es-demo-change.

git checkout es-demo-change - moves to that local branch.

git branch – should now say es-demo-change.

git status - will show the files you changed - these should be in red, as they are locally changed only. Are these the files you expected to be changed?

git diff - will show all the changes you made - are these the changes you expected?

git diff <file> will show just the changes in one file - if there are many and you want to look at just one file. You should see only the expected changes or additional files.

Git add <file> For each changed or added file.

OR

git add \* - if you are sure that you have got no additional changes.

git status - should now show all the changed files in green - they are on a list for checking in.

git commit -m "my check in message" - this commits the change, but it is still local.

git push origin es-demo-change - pushes to the repository and creates the branch in the repo.

Go to the GitHub page for this repository and check the branch is now there.

Click on ‘create pull request’. This will open a pane in which you can see all the changes highlighted and fill in some text details if you want to. You can also choose a reviewer from the team.

If you know who you want to review it, select them as reviewer - you can add more than one. If you don't know (or care), leave reviewer unselected - they can self-select.

If you want to deliver without getting a reviewer, override the rule and click merge. You can use this if the change is minor and you are very confident, or if the team are taking too long a time to respond to a review request, or if there is an emergency fix and nobody is available to help.

The reviewer will look at the changes and if they are happy, merge the changes into main.

If they are not happy, they will chat with the author. If still not happy then they can decline the review with comments. The author can then respond to the comments and ask for another review.

When merging does not work

It may be that the merge does not work, because the main branch is ahead of the branch you are delivering. In almost all cases GitHub will offer and auto-merge of the changes and come out fine.

In some cases where it cannot decide, it will offer each change for you to fix up, and you can edit them directly in GitHub.

In a very few cases, GitHub mucks this up, usually when new code has been added, and old code deleted - it can be led to believe that both bits of code are needed, and you get duplicated lines.

So, if you have merged a change and GitHub has made changes - you have to check out main - and re-test your changes.

As an alternative to letting GitHub fix bad merges, you can do it yourself locally.

If you have your changes locally, you can git stash which will hide them in a safe place.

Next use git checkout main to switch to the main branch and git pull origin main.

Now you have main up to date - use git checkout es-mybranch to switch back to your branch and git stash pop. This will report errors and give you the code for both branches merged. Edit the code and choose what you want to do with each conflict (look for ==== in the code). Retest, and re-deliver using, as before, git add, git commit, git push.

Checking out someone else's branch.

You want to add to a branch that already exists, and you do not have it locally. (Example: The branch is called nt-changes and you can see it in the repository)

git clone <repo> as before.

git branch nt-changes - creates a new branch locally to mirror the one on the server.

git checkout nt-changes

git pull origin nt-changes - pulls from the branch on the server.

Other git features (probably never use them)

Revert a branch back to a previous level - you cannot rewrite history, but you can check out that version, then check it back in again over the top of the new version.

Squash changes. If you have been working for a while on a branch and have checked in many changesets, you can pull from that branch and use 'git squash' to reduce all the changes down to one single change. Some people like to have small histories, but it is harder to debug squashed changes as there are many changed files and a single commit. I do not recommend it.

Many other features are available and discoverable with git help – any which you find useful could be added here as a guide for new team members.

Github version control and update servers.

WordPress plugin projects normally have only one update server, which holds the current release of the plugins. This is fed from the main branch of the repositories, so there is never a need to have multiple different bug-fix versions available.

Contrast with MQ or similar, where you would have version 1,2 and 3 in the field, and customers would have multiple different bugfixes applied to their versions – all would need to be reproducible in the build systems.

Our requirements are much simpler. We need a single main branch into which features and bug fixes are fed - we really have no need to distinguish between a feature and a bugfix – except to say that in general bug-fixes are shorter development time, so features may require merging back after many bug-fixes have been merged.

This brings us to how to manage merges. There are two ways of merging, which are merge and rebase. They serve different purposes. Merge is generally ‘upwards’ – I.E you merge from other branches into main, and if you have a main and a preproduction and a production branch, then you merge from fixes to main to preprod to prod – never backwards.

Rebasing is the reverse, but you do it locally as its almost certain to require user interaction. In this case you have a branch with a long-running feature in it and want to have all the bug-fixes which have passed you by since you cloned main. You gather a new version of main onto your local machine, and use ‘git rebase’ on your local branch to update it before delivery, so it looks like it came from the current main. This avoids complex merges – which are at best slightly risky.

I recommend we have a single main into which nothing is ever pushed – and all features and bug-fixes are created as a branch. There is no need to distinguish between the two types, they are both just development branches.

As nobody outside development is ever likely to use code from a branch, most of the time there would be no need to rebase and gather bug fixes. Its more likely that a feature developer would need just a single bug-fix, and that would be much easier to just re-produce by looking at the differences in the changeset and applying them to the feature branch.

In summary:

Main is the branch from which the update server is populated.

A release is created every time the update server is modified to correspond to the version which is shipped. This release takes the form of a zip, but it’s a zip of the entire main branch, not just the shipped files – the purpose being that we can pull that release and get a working project which corresponds to the released code.

Other branches are created for each bug or feature – named with initials plus some reference, e.g es-bug777 or es-new-page-layout etc.

When a feature or a longer running bug-fix needs to be merged, the developer can take the decision to either merge and let github try to fix the conflicts or pull the head of the main branch and rebase on it. Rebase is quite hard to get right but complex merges where main is ahead by many fixes are liable to fail because the conflict is not resolvable without a human decision.

Hopefully we will have mostly short running features and small fixes – so the merges will be easy. Also, we have quite a few plugins, so the chances of needing a merge are smaller, as the bugs may not be in the same plugin as the feature.