ELECTRICAL AND INFORMATION ENGINEERING
University of the Witwatersrand, Johannesburg
Software Development III

Laboratory 1 — Git Collaboration, Azure Deployment

In this lab we will go over the basics of Git usage as well as show how a static website can be deployed to Microsoft's Azure cloud service.

1 Install Git and Git Bash

Download the official Git command-line client and follow the instructions for your operating system. Visual Studio Code, which we will install next, relies on this for its Git integration.

2 Install Visual Studio Code

Visual Studio Code, or VS Code as it is commonly known, is a cross-platform, open-source, lightweight code editor. It is very popular and is used daily by millions of developers. We will primarily be using it for editing JavaScript but it also supports Markdown and multiple other languages through extensions, including LaTeX and SQL.

It is interesting to note that it is built using Electron which is a framework for deploying JavaScript applications to the desktop using Node.js as the runtime.

Download and install VS Code.

In order to familiarise yourself with the editor's functionality it is worth watching the following short videos:

1. Getting Started
2. Code Editing (go here for more detail on basic editing)
3. Productivity Tips

As you will have seen, keyboard shortcuts, are prevalent throughout VS Code and these can greatly increase productivity. Take the time to learn some of the ones related to the basic use of the editor.

Lastly, the Code 2020 YouTube channel has an extensive collection of tips on all aspects of VS Code. If you want to really maximise your use of the editor it is worth checking out.

2.1 Using Git Bash in VS Code's Integrated Terminal

The default shell, which runs within the integrated VS Code terminal, is Windows PowerShell. You may wish to change this to use Git Bash. To change the default shell, go to the command palette in VS Code by typing Ctrl-Shift-P, and then type `Select Default Shell` and choose Git Bash. If you do choose to use Git Bash, then you will need to use Unix commands for changing directories, etc. so it is worth reviewing the basic Bash commands.

VS Code: Using VS Code for version control

Many Git tasks can be accomplished directly in VS Code without having to use the command line. For some of the sections in this lab, the VS Code procedure is shown in a box like this one.

3 Setting up Git

Once you have installed Git, there are a few things to do before you can start working.

3.1 Setting up your global identity

Whenever you make a change using Git, your name and email address are recorded. These can be set up per-repository or globally (per-installation). We will be configuring a global ID. To do so, open the terminal/git bash and type the following in *mutatis mutandis* (i.e. use what is below as a template, changing that which must be changed):

```
git config --global user.name "Micky Duck"
git config --global user.email "micky.duck@students.wits.ac.za"
```

3.2 Setting up your editor

Git brings up an editor at several points for you to make comments. The default editor is `vi`, which, while it has its share of devotees, does not have obvious keystrokes. To use VS Code instead, type the following:

```
git config --global core.editor "code --wait"
```

Note, you are free to use any editor that you wish, including the command line editors, `emacs`, `nano` and `pico`, or the default text editor for your system: `notepad` for Windows, `textedit` for OS X or `gedit` for Ubuntu (and most other Linux distros). If you wish to use one of these, instead of VS Code, then Google the appropriate command for the Git config file.

4 Creating and using a local repository

Now we will create a local repository and put some files under version control.

4.1 Initialising the repository

Create a folder where you want your repository to be. Now open the terminal/git bash again and type the following in *mutatis mutandis*:

```
cd /path/to/my-git-repo # From now on I'm going to assume you are in the
                        # repository directory unless directed otherwise

git status              # This line should give you an error
git init
git status              # This line should not give you an error
```

Review the output of the `git status` commands. We're going to be using this command a lot.

4.2 Committing Files

Now that we have a repository, let's add some files. Since we're going to be doing web development, let's make a simple website. In your repository folder, create a file called `index.html` and type the following in *exactly as below*:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Hello World</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>Hello World</h1>
    <p>
    &#127760 - Hello (your name here)!
    </p>
</body>
</html>
```

Also create a file called `dummy.html` without any text in it.

Open `index.html` in your web browser. Not pretty, but it works!

Commits: Every time you save changes with Git, you need to make a commit. A commit is like a snapshot of the state of your repository when you made the commit. This allows you to see all the previous versions of your files by moving between commits.

Commit this file with Git:

```
git commit -m "My first commit"
```

The output should be ... `nothing to commit`?

Check the status of the repository:

```
git status
```

File tracking: Git doesn't automatically track files because there are often files that we don't want to keep under revision control such as binaries, compilation artefacts, debug logs, and any other content that would bloat the repository unnecessarily. To keep a project small and efficient, you should only track *source* files and omit anything that can be generated from those files.

So, our files need to be tracked. This is done by staging them.

Staging: Continuing on the photography metaphor, in order to include something in a snapshot (commit) it needs to be staged. This is done using the `git add <files>` command which tells Git to include the listed files in the next commit. Staging has to be done before every commit, even if the file is tracked.

Stage `index.html` and `dummy.html`:

```
git add index.html dummy.html
git status
```

Try committing the files again:

```
git commit
```

Your chosen text editor should come up. Type the message ”My first commit” and save the file. Now close the editor. This signals to Git that you are done entering your message and that it should continue with the commit.

Make sure the commit worked:

```
git status
```

What does the `-m` flag do?

Working Directory: Note that you could have many Git repos. Git, by default, assumes you are referring to the repo in the current directory. The contents of the current directory is the working directory. Usually there will be some minor to moderate changes of the contents of the directory compared to the most recent commit. You can restore files from previous commits into your working directory, if you need to.

Staging more efficiently: The `git add` command supports flags to make staging files faster. By giving the relative folder path to the repository folder (`git add .` ← the dot is the file path) Git will stage all files that are new or modified as well as files that have been deleted. `git add -u .` will only stage tracked files that have been modified or removed and `git add --ignore-removal .` will ignore deletions, but stage both tracked and untracked files.

Staging and committing as a single command: the `-a` flag can be used when committing to automatically stage changes to and deletions of tracked files. Thus, in most cases where new files have not been created, a commit can be reduced to:

```
git commit -am "<commit message here>"
```

VS Code: Repo initialisation staging and committing

Most Git tasks can be achieved by using the Source Control sidebar (Ctrl-Shift-G). Watch this video on using Git within VS Code for a good introduction. For now, you only need to watch up to 5:21. We will return in a later section to the rest of this video.

Git tasks can also be done via the Command Palette (Ctrl-Shift-P). Call up the Command Palette and type `initialize`, `stage`, or `commit` to see the different options pop up.

5 Moving between commits

Edit line 10 of `index.html` to reflect your name rather than the placeholder. Delete `dummy.html`. Stage and commit `index.html` with an appropriate commit message.

Now, edit your name in line 10 of `index.html` replacing the first vowel with a ”q”. Stage and commit this change with the message ”A Mistake”.

Look at the commit history of the repository using:

```
git log
```

Commit checksums: Note that the log gives a history of all commits. Next to each commit is a checksum that acts as an ID. However, as we will see later, you don't need to use the entire checksum to refer to a commit, just the first part.

5.1 Reverting a commit

That last commit was clearly a mistake, let's undo it.

To do this we are going to call `git revert <commit ID>` with the ID of the last commit. Remember that you don't need to type the whole checksum, just the first few digits. So run the following (*mutatis mutandis*):

```
git log --oneline  # To find out the checksum of the previous commit
                   # Notice that --oneline only shows the first few digits
                   # of the commit checksums
git revert 3eb20e6 # Substitute the first few digits of your commit checksum
```

Notice that instead of deleting the “A Mistake” commit, Git figures out how to undo the changes it contains, and then tacks on another commit with the resulting content. So, our fourth commit and our second commit represent the exact same snapshot. Git is designed to never lose history. The third snapshot is still accessible, just in case we want to continue developing it.

Let's add an author page to our website. Create a file called `author.html` with the following content:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Authors</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>This page was made by:</h1>
    <p>
    (your name here)!
    </p>
</body>
</html>
```

Let's make some more mistakes:

Create two new files: `tracked.html` and `untracked.html` with anything in them.

Now run the following:

```
rm index.html
git add tracked.html
```

5.2 Undoing changes without committing

Now, we don't want to commit those mistakes. Let's revert them. However, we don't want to use `git revert` this time. We don't want to just commit and revert because we'll lose all our useful work in creating `author.txt`. So we're going to have to remove the incorrect files manually.

Any newly created file that has not been Git added can just be deleted. But a file that we have asked Git to add, must be unstaged using reset. Finally we can undo a change to a file by restoring it from the repo. Here HEAD is a Git key word, which refers to the commit we are about to operate on. Usually this refers to the most recent commit (as it does in the example below), but we can change the HEAD.

```
rm untracked.html           # untracked files can just be deleted
git status

git reset HEAD tracked.html # tracked files must be unstaged using reset
git status

rm tracked.html
git status

git restore index.html
git status
```

The `restore` command we used got the file from the most recent commit. But you can also get a file (or files) from a previous commit. As an example, suppose we realise that it was a mistake to delete the `dummy.html` file. We do a `git log` to remind us of when we had it and find the commit ID. Then we change our status to checkout all the files from that commit into the working directory. Do the following (*mutatis mutandis*) using your commit ID, not mine:

```
git checkout d82778f7b9
```

Now you can do an `ls` and check that this was the commit where you had the `dummy.html` file. If not, you checkout another commit until you find it.

Now you can copy the file into a temporary place:

```
mkdir ../tmp
cp dummy.html ../tmp
```

And we go back to where we came from and restore the file (`master` points to the most recent commit).

```
git switch master
mv ../tmp/dummy.html .
rm -rf ../tmp
git add dummy.html
git commit -a -m "Restoring dummy.html"
```

As an aside, if we knew for sure which commit the dummy.html was in, we could have just done

```
git restore --source d82778f7b9 dummy.html
```

Note that in this case we don't have to do a `git add` because, although we deleted the file from the directory, it knows this is the same file we used to track.

6 Branching and merging

Suppose you wanted to try out a new idea without using Git, you might copy all of your project files into another directory and start making changes. If you liked the results, you would copy the affected files back into the original project. Otherwise, you would simply delete the entire folder and forget about it.

This is the functionality offered by Git branches... with some key improvements. First, branches present an error-proof method for incorporating changes back into the main project. Second, they let you store all of your experiments in the same directory, with the same version control as your main project.

Let's see our existing branches:

```
git branch
```

The master branch: The master branch is Git's default branch and is usually used to denote the main branch of a project, with experiments being branched from it.

Notice the * next to master? That means that it is the active branch (i.e. the branch currently reflected in the working directory).

Let's create a branch where we can develop a new feature:

6.1 Creating a branch

To create a branch from the active commit, type the following:

```
git branch <branch name>
```

Create a branch called "newFeature".

```
git branch newFeature
```

To switch to the branch, call `git switch` with the branch name:

```
git switch newFeature
```

6.2 Committing to a branch

Lets add the links between our index and author pages in this branch. In `index.html`, add a line above the body closing tag (`</body>`) with the following:

```
<p><a href="author.html">Authors</a></p>
```

Similarly, in `author.html`, add:

```
<p><a href="index.html">Return to home page</a></p>
```

Open `index.html` in your web browser and confirm that your new links work. Then stage and commit your changes.

Let's also make a second page to greet the universe in `dummy.html`. Change the contents of `dummy.html` to:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Hello Universe</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>Hello Universe</h1>
    <p>
    Hello (your name here)!
    </p>
  <p><a href="index.html">Return to home page</a></p>
</body>
</html>
```

Add a link to your second page into `index.html` and test that all your new pages and links work. Then, stage and commit your changes.

Now that we're happy that our feature is finished, let's switch back to `master`:

```
git switch master
```

Do you notice that all your changes have been undone? Let's look at the log:

```
git log --oneline
```

Your new commits are missing, this is because the commits were made on the `newFeature` branch.

Since we're done with that branch let's merge it back into master. This is like copying your changes back into your main folder:

```
git merge newFeature
git log --oneline
```

Feature Branches: By creating branches for features in development, we can ensure that no incomplete code is on `master`. This allows us to have a master branch that is always ready to be deployed, since it only contains stable, tested code. What are the drawbacks of this approach?

7 Git Collaboration

For the next section pair up with another member of your group. Where instructed work with your partner.

In this course we will be using GitHub to host our repositories. Create an account and a new repository for you and your partner. You only need to create *one repo* for the both of you. Add your partner as a collaborator in the repo settings window so that you both have permission to work on the code.

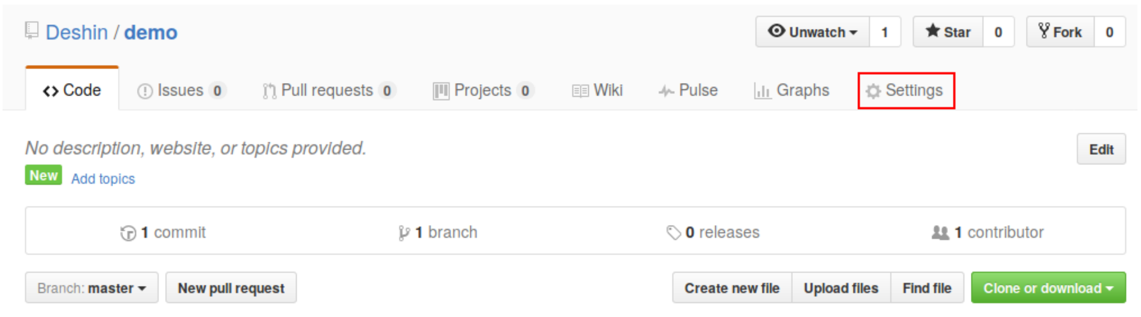


Figure 1: The settings window on GitHub.

7.1 Local and Remote

In order to work on the code in your new repository, you need to create a local (on your machine) copy of it.

This is achieved by `cloneing` it from your remote repository (GitHub). Get your repository URL from GitHub using the ”Clone or Download” button on you repository page. Copy the SSH URL.

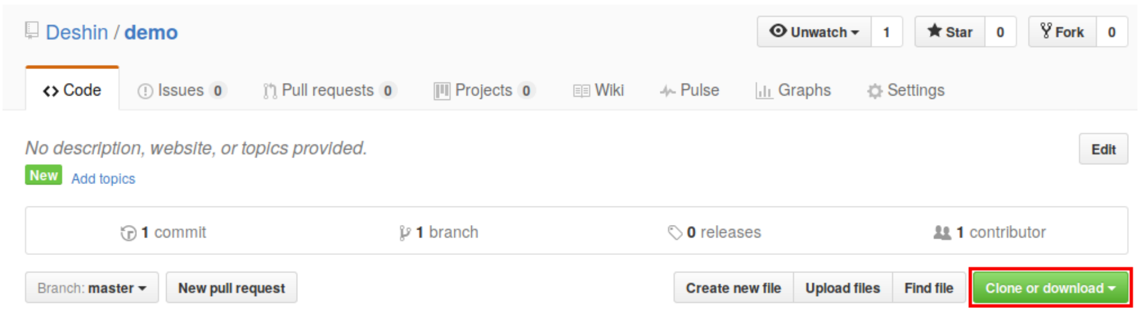


Figure 2: The clone/download button on GitHub.

Now, in the folder where you want to clone your repository (it will create a new subfolder) type the following command:

```
git clone <Repository URL> # e.g. git clone https://github.com/Student/Lab1.git
```

Git should create a local copy of your repository.

VS Code: Collaboration using a remote repo on GitHub

Return to the video on using Git within VS Code and watch from 5:21.

7.2 Pushing and Pulling

When working with this repository, most interactions are the same as previously with the local-only repository. However, in order to synchronize the remote repository with your local copy, two new concepts are defined, `pushing` and `pulling`.

Let’s make some changes and see how to sync them to the remote.

Open `index.html` and edit it so that the placeholders in brackets are replaced with their correct values. Confirm that your changes are correct using a browser, stage and commit your changes.

Confirm that your commit has worked.

```
git log --oneline
```

Browse to your repository in GitHub and look at the commits to the repository. Do you notice that your new commit isn't there? It's because your changes to your local repository haven't been synced with the remote yet.

Before we sync the repository, ask your partner to clone the repo on their machine.

Now we can sync your repository. Before we can do that, lets find out what our remote is called:

```
git remote -v
```

Now we can see that our remote is called `origin`

So let's tell Git to push the changes in our local repository up to the remote:

```
git push -u origin
```

The `-u` argument tells Git to set the remote as the upstream (default remote) repository so that we can just call `git push` next time.

Now switch to your partner's repository. Let's get the changes that have been made:

```
git pull -u origin
```

Open `index.html` in your browser to confirm that the pull worked.

In order to ensure that you are always committing changes to the newest version of the repository, you should always `pull` before you `push`. This ensures you receive any new commits that have happened to the remote repository before you push your own commits.

Now that we have seen that we can synchronise our repositories, let's look at what happens when two people edit the same file.

7.3 Merge Conflicts

Switch back to your repository and edit your `index.html`, add a group bio to your home page describing the group. To do this, add another paragraph (`<p>`) below the existing one and type your bio in there. Open `index.html` in your web browser to confirm your changes, stage and commit them.

Do the same for the `index.html` in your partner's repository. In your partner's repository, however, `pull` and then `push` the changes up to GitHub.

Now switch to your repository and `pull` their changes from GitHub. Your output should tell you that a merge conflict occurred and it failed to automatically merge the files, instructing you to fix conflicts.

When two people attempt to commit changes to the same section of a file, Git is unable to decide which change to use (or whether to keep both and in what order). This is called a *merge conflict*. To fix the conflict, it delegates the decision on what to keep to the developer who attempts to commit their changes second. This is why you must pull before you push.

Okay, let's resolve this merge conflict. Open `index.html` and look at the section where you added your code. Notice that Git has added both versions of the line(s) with big delimiters ("`<<<<<<< HEAD`", etc...) between them? To fix the conflict, make the file valid again by removing the delimiters and deciding what to keep. Now call:

```
git commit
```

Notice that Git has already filled your message in for you?

Finally, `pull` and then `push` your changes up to GitHub.

Doing a merge using an ordinary text editor can become tedious and difficult in situations where conflicts happen in many parts of many files. VS Code, however, has excellent, built-in, support for handling diffs and merges.

VS Code: Merge conflicts

Clicking on the Source Control icon on the left-hand side of the screen, and then clicking on a particular file within your repo allows you to easily see the changes between different versions of the file, as well as deal with merge conflicts. For merge conflicts, differences are highlighted and there are inline actions to accept either one or both changes. Once the conflicts are resolved, you can stage the conflicting file so that you can commit those changes. This is illustrated in this video on resolving merge conflicts.

Induce a merge conflict with your partner by adding another paragraph to both your and their `index.html`. The second partner to `push` will be notified of a conflict. Use VS Code to resolve the conflict. When you're done, save the file and commit the fix as usual, then `pull` and `push`. Make sure that both you and your partner have an opportunity to resolve a merge conflict by changing the order in which you push your changes.

VS Code: Viewing the repo's commit history

VS Code has a useful extension which allows you to view the commit history of a repo. The commit history displays all of the commits and branches of the repo and helps you to keep track of the work that has been done. Install Git Graph and use it to review the work that has been done in this lab. Git Graph can also be used to create branches, tag commits, and so on.

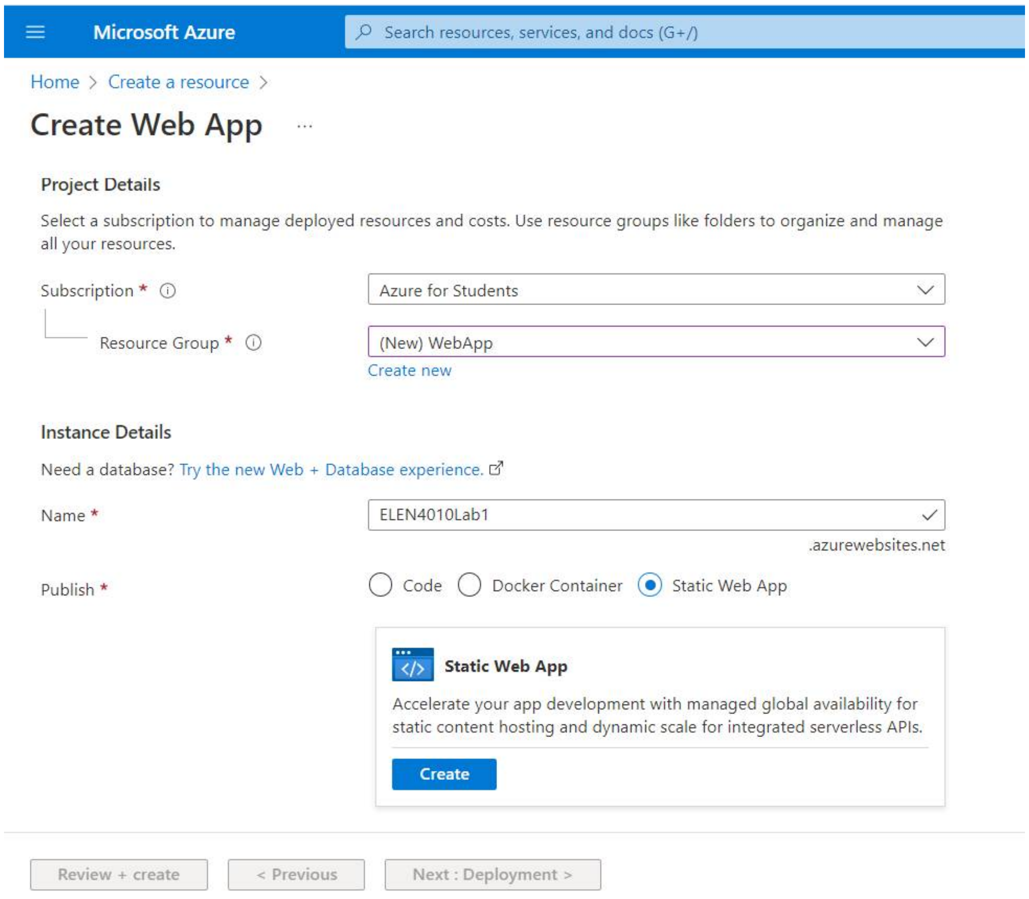
8 Hosting Static Web Pages in Azure

This section of the lab will show you how to simply host the static web pages, that you created earlier, in the cloud. In subsequent labs, we will introduce Node as well as *con-*

tinuous integration and how to run tests etc. before deploying your code. Additionally, the group laboratory web application will need to be developed using Node.js. For the purposes of this exercise, everyone will need a remote repo that is managed by their account on GitHub. This will not be necessary for the group project as only one deployment will be needed. However, for this lab, everyone whose accounts weren't used in the previous exercise, must also create remotes and push their local repos up to them. Every person should have a repo they can access by going to `github.com/UserName/RepoName`.

Firstly, sign up for a Microsoft student account here: `https://azure.microsoft.com/en-us/free/students/`. You will need a Microsoft account to complete the Azure Student Account registration.

Go to the GUI interface for Azure at `https://portal.azure.com/` and select *Create a Resource* and then select *WebApp* as the resource that you would like to create. Choose to *Publish* a *Static Web App* as shown in Figure 3.


Microsoft Azure
Search resources, services, and docs (G+/)
Home > Create a resource >
Create Web App ...

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * : Azure for Students

Resource Group * : (New) WebApp

Create new

Instance Details

Need a database? Try the new Web + Database experience.

Name * : ELEN4010Lab1 .azurewebsites.net

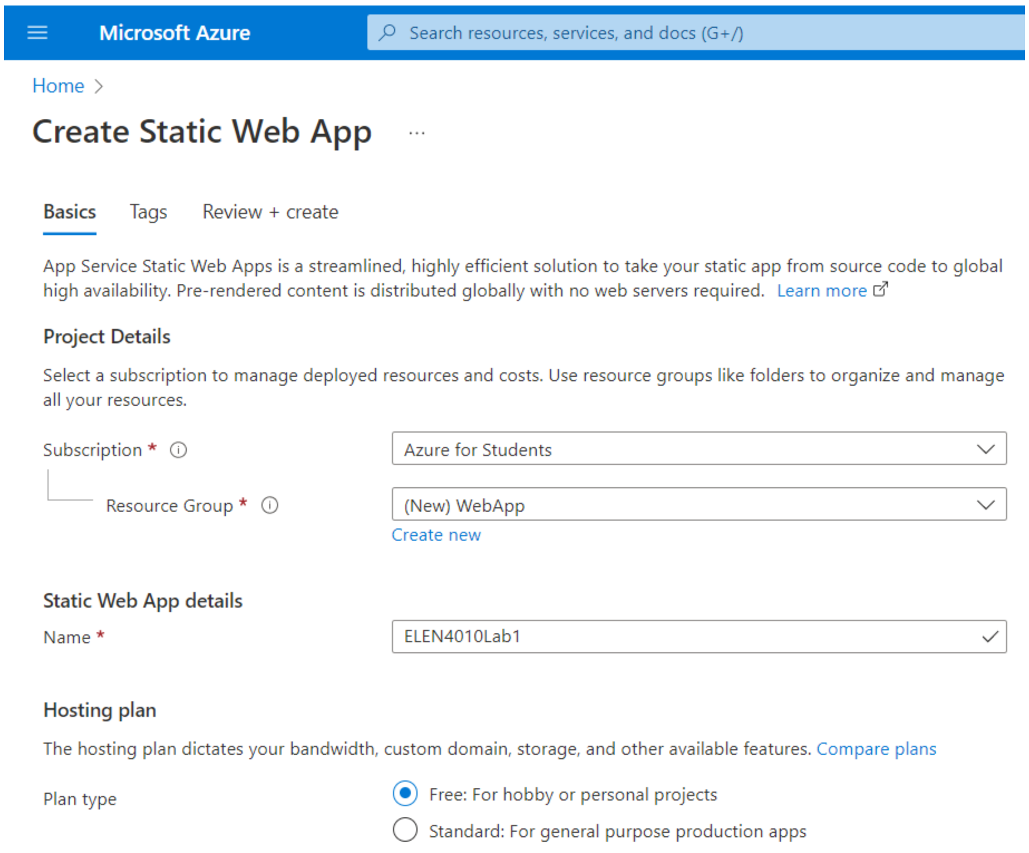
Publish * : Code Docker Container [✓ Static Web App]

Static Web App

Accelerate your app development with managed global availability for static content hosting and dynamic scale for integrated serverless APIs.

Figure 3: Create a Static Web App

Fill in the details as shown in Figure 4 and Figure 5, signing into GitHub and using your own app details, as required.



Create Static Web App ...

Basics Tags Review + create

App Service Static Web Apps is a streamlined, highly efficient solution to take your static app from source code to global high availability. Pre-rendered content is distributed globally with no web servers required. Learn more

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * : Azure for Students

Resource Group * : (New) WebApp

Create new

Static Web App details

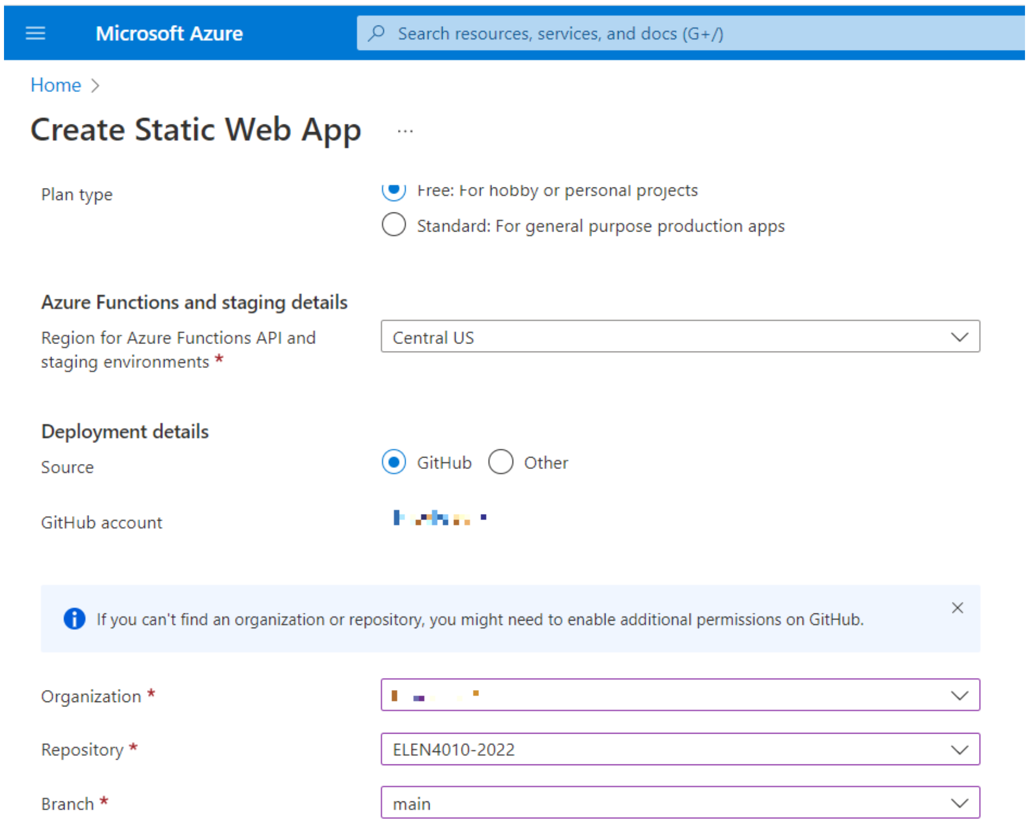
Name * : ELEN4010Lab1

Hosting plan

The hosting plan dictates your bandwidth, custom domain, storage, and other available features. Compare plans

Plan type: [✓ Free: For hobby or personal projects] Standard: For general purpose production apps

Figure 4: Choose a name for your Web App



Create Static Web App

Plan type	[✓ Free: For hobby or personal projects] Standard: For general purpose production apps

Azure Functions and staging details

Region for Azure Functions API and staging environments *	Central US

Deployment details

Source	[✓ GitHub] Other
GitHub account	[redacted]

If you can't find an organization or repository, you might need to enable additional permissions on GitHub.

Organization *	[redacted]
Repository *	ELEN4010-2022
Branch *	main

Figure 5: Link your app to your repo hosting the code and deploy the `main` branch

Your are now in a position to *Review + create* your application. Then choose *Create* and wait while your application is initialised and deployed. When the deployment is complete you can visit your (auto-generated) website URL to view your web pages. Do this by clicking *Go to resource* and then clicking the URL that is given. Whenever you push changes to your GitHub repo, your website will be updated automatically after a couple of minutes.