Exercise: Project Management & Safe Git Collaboration

Overview

For this exercise, you and your team members will work collaboratively on a mini project. This exercise walks you through some safe collaboration methods for a team of 4 students collaborating on an RStudio project connected to a remote GitHub repo. Each student will contribute to a project by creating a unique element of the project by creating unique files, sharing files with other team members, and building a mini report or page for a website using the files from the collective.

When working on collaborative projects, team members do not work on the main branch of the repo but instead work on **feature branches**, which are later merged to the main branch. Unless you are a skilled Git user, I do not recommend diverging from the guidance approach or you may find yourself troubleshooting a Git conflicts or errors.

Things you will need:

- Git installed on your computer
- A team project GitHub remote repository
- An RStudio Git Version Control project connected to that remote repo

Member Tasks and Responsibilities:

- **Student 1**: Create/modify an `.R` script to clean the data (`src/data/clean_mpg_data.R`)
- **Student 2**: Create/modify an `.R` script to create and save a plot of the data (`src/figs/make_plot.R`) using the cleaned data file (`data/processed/cleaned_mpg.Rds`).
- **Student 3**: Create/modify a Quarto webpage file (`pages/my_third_page.qmd`) to source `src/data/clean_mpg_data.R` and `plot_cty_hwy_mpg_point.R` to clean and plot the data; and call the `.png` file for the plot referencing a saved png figure (`figs/cty_hwy_mpg_point.png`)

Some Key Directories and Files:

```
.
|-- _quarto.yml                      # Quarto project configuration file
|-- data/                            # Data files
|   |-- raw/                         # Original
|   |-- processed/                   # Cleaned
|-- src/                             # R scripts
|   |-- data/                        # Data cleaning/processing scripts
|   |   |-- clean_mpg_data.R
|   |-- figs/                        # Plotting scripts
|   |   |-- plot_cty_hwy_mpg_point.R
|-- figs/                            # Saved plots
|   |   |-- cty_hwy_mpg_point.png
|-- pages/                           # Quarto website pages
|   |-- team_test_page.qmd
|-- docs/                            # Rendered website files
```

Part 1: Check Branches and `main` Upstream

In your RStudio Terminal, check what branches exist on the GitHub repository by typing:

```
git branch
```

If there are multiple branches, an * will appear next to the active branch.

Ensure you are on the `main` branch using *switch*.

```
git switch main
```

Pull changes to that your local `main` branch has the recent changes from the remote (`origin`).

```
git pull origin main
```

You could use `git pull` but `git pull origin main` makes the command explicit. As long as you know you are on `main`, `git pull` works just fine.

Test that you can *push* to the `main` on `origin`.

```
git push
```

If you cannot push, set the upstream once while on `main`. You will not need to do this again.

```
git branch --set-upstream-to=origin/main
```

Part 2: Managing Branches

Step 1: Create a Branch for Testing

In your RStudio Terminal, create and checkout your own branch using `switch -c`. Name your branch using your name in lowercase characters, no spaces, for example, `git switch -c <my-branch-name>`.

Switch and *c*reate a `test` branch using the `-c` flag. Once created, you can switch using `git switch test`.

```
git switch -c test
```

Although you have not made any changes to the local `test` branch, you want GitHub to recognize it and you want Git to allow you to push and pull to and from the associated remote branch. Pushing using the `-u` flag should allow you do this.

```
git push -u origin test
```

Notes:

- `-origin` = the name of your remote on GitHub
- `-u` sets the upstream (later, `git push` or `git pull` should work without specifying origin/branch).
- `git pull origin test` and `git push origin test` are more foolproof as they require you to set the remote and branch explicitly, leaving no ambiguity.

Step 2: Create/Switch to your Personal Feature Branch

Each team member works on their own branch to avoid inadvertently overwriting files on the main branch or on another branch. **Be very vigilant of the branch you are working on**. If you work on a team-member's branch and try to push changes to the remote repo, you will likely have to troubleshoot a bunch of issues. This experience will be educational but will likely not be fun for you.

In your RStudio Terminal, create and checkout your own branch using `switch -c`. Name your branch using your name in lowercase characters, no spaces, for example, `git switch -c <my-branch-name>`.

```
git switch -c <my-branch-name>
```

```
git push -u origin <my-branch-name>
```

Part 3: Coding Contribution to the Project

Now that you are sure your are on your personal branch, start your file edits. Remember, you can always use `git branch` to verify your active branch.

Student 1: Reading, Cleaning, and Writing Data

In RStudio, modify `scr/data/clean_mpg_data.R` in order to:

- Read the raw data file `data/raw/mpg.Rds`
- Clean the raw data using `{dplyr}`
- Write the cleaned file `data/processed/cleaned_mpg.Rds`
- Ensure that your data file is present
- Save your `.R` script

Student 2: Plotting Data and Saving Plots

For this exercise, the cleaned data from Student #1's coding is already available to you so that you can work on your part. This scenario will **not** be the case when working on a real team project as your teammates will need to make those available on the `test` branch.

In RStudio, modify `scr/figs/plot_cty_hwy_mpg_point.R` in order to:

- Read the *cleaned* data file `data/processed/cleaned_mpg.Rds` (based on Student #1's efforts)
- Plot the data using `{ggplot1}`
- Save the plot `figs/cty_hwy_mpg_point.png`
- Ensure that your plot file is present
- Save your `.R` script

Student 3: Editing a Quarto Webpage File

For this exercise, the cleaned data and plot created by Students #1 and #2 is available to you. This scenario will **not** be the case when working on a real team project as your teammates will need to make those available on the `test` branch.

Every time your webpage is built, you will need to ensure the most up-to-date and accurate file is used to render the page files. This means that reading and cleaning data as well as plotting and saving plots is performed at this final step.

In RStudio, modify `pages/team_test_page.qmd` in order to:

- Call/source Student #1's cleaning script, `scr/data/clean_mpg_data.R`
- Call/source Student #2's plotting script, `src/figs/plot_cty_hwy_mpg_point.R`
- Display the most recent plot `.png` file, `figs/cty_hwy_mpg_point.png`
- Save your `.qmd` file

Part 4: Version Control with Git and GitHub

Now that you have saved your file edits, you will make these changes accessible on the remote repo to other team members. All of the files need to be accessible to all team members in order to work together and for testing.

Key Files:

- Student 1 modified: `src/data/clean_mpg_data.R`
- Student 2 modified: `src/figs/plot_cty_hwy_mpg_point.R`
- Student 3 modified: `pages/team_test_page.qmd`

Rather than clutter your personal branch with files that are not yours, testing will be performed on the `test` branch. This branch will be able to see all file changes across all branches not including your own. To perform the testing, the files need to be merged with the remote `test` branch.

- Add, commit, and push your local personal branch to the remote
- Switch to the local `test` branch and fetch and merge those changes
- Merge your work into `test` branch

Step 1: Stage/Add and Commit Push File Changes

Important: Only *add* and *commit* files that you have created or modified on your personal branch. Do not commit other students' files. Your authorship in your scripts will make this clear but you may need a way to distinguish file names. If you are unable to remember who is working on what, a simple approach would be to add your initials as a prefix for all code scripts you write.

Add files in a targeted manner using the path and file name (don't use `git add .`). This approach will prevent you from staging files you do not intend to stage.

```
git add <path-to-file-and-filename.R>
```

Commit your change with a message using `-m` and your message in quotes:

```
git commit -m "xyz script for ..."
```

Step 2: Push your Branch to the Remote (`origin`)

Push your local branch to the remote repository on GitHub. `origin` is the default name given to the remote repository stored on GitHub. This step will ensure that your remote branch is up-to-date with your local branch.

```
git push origin <my-branch-name>
```

Clarification:

First, *push* your feature branch to GitHub so your changes are saved remotely. Then, *switch* to the local `test` branch to *merge* your changes from your feature branch for integration.

Step 3: Fetch and Merge the `test` Branch

Now that you have pushed your changes to your remote personal branch, switch to the shared `test` branch and merge the changes by you and your collaborators.

Switch to the local `test` branch so that you can merge changes from the remote `test` to the local `test` branch.

```
git switch test
```

Importantly, before merging files, ensure your local `test` branch is up to date with any recent changes by *fetching* them from the remote using `fetch`.

```
git fetch origin
```

Note: `git fetch origin` will bring in all branches from the remote, including all other branches. There is no need for you to ever jump on a teammates personal branch, please don't do it. Only interact with other students' work via the shared `test` branch but even still, you should not be touching the files/work created by others.

Now *merge* those fetched changes using `merge origin/test`.

```
git merge origin/test
```

This step merges the updates on the remote `test` with your local `test` and is not a reflection of that remote branch. Any previous changes that you made to the remote `test` will be on your local `test`. Similarly, if your collaborators have pushed their work to the remote `test` (e.g., `origin/test`), those changes will be here too.

Now that you know you have the most recent version of `test`, you still want to merge your new file changes.

Note: `git pull` will perform both `git fetch` and `git merge`. I recommend using `git fetch` followed by `git merge` because this makes clear there are two actions. This is important for inexperienced but also experienced Git users. Specifically, `git fetch` downloads updates from the remote without changing your local files. Therefore, you can see what is new before merging something that could be problematic. Then `git merge` lets you explicitly incorporate those changes into your branch. This approach is safer for beginners, reduces the chance of accidentally overwriting work, and makes it easier to understand how remote changes affect your branch. Making small changes like a file or two also ensures fewer errors.

Now that you know you have the most recent version of `test`, you still want to merge your new file changes.

Step 4: Merge your Work into the `test` Branch

You should still be on the local `test` branch. Do now switch off of it yet.

Merge your changes on your local personal branch into the local `test` branch.

```
git merge <my-branch-name>
```

This step integrates your changes into the shared `test` branch but the local branch is out of date with the remote `test` branch.

Push the updated local `test` branch to the remote `test` branch.

```
git push origin test
```

Note: `git pull` will perform both `git fetch` and `git merge`. I recommend using `git fetch` followed by `git merge` because this makes clear there are two actions. This is important for inexperienced but also experienced Git users. Specifically, `git fetch` downloads updates from the remote without changing your local files. Therefore, you can see what is new before merging something that could be problematic. Then `git merge` lets you explicitly incorporate those changes into your branch. This approach is safer for beginners, reduces the chance of accidentally overwriting work, and makes it easier to understand how remote changes affect your branch. Making small changes like a file or two also ensures fewer errors.

????? But once everyone has merged their personal branch with the local `test` and pushed to the remote `test`, the remote `test` will be out of data with the local `test` because the changes pushed contains

?????

Part 5: Managing File Paths Appropriately

Step 1: Evaluating Path Management

In order to ensure that your project renders, your paths to all directories and files need to exist on the platform running the code. How did your team decide to manage and code file paths? Did all team members have different paths to files because they have different user names on their computers? Did you collaborate with people who have the same name to make things easy?

Did you hard-code the paths like this?:

- `"usr/Sally/desktop/this_project_dir/src/figs/plot_cty_hwy_mpg_point.R"` or `"C:/Users/Sally/desktop/this_project_dir/src/figs/plot_cty_hwy_mpg_point.R`?

Hard-coding file paths will not allow you to run the code without changing all of the file paths on each user's system. Moreover, what will happen if or when you move the project directory someplace on your computer? Will you need to change all of the paths again?

Step 2: Managing File Paths with `{here}`

Load the library:

```
library(here)
```

Test the `here::here()` or `here()` function:

```
here::here()
```

Notice the returned path is the root to the RStudio project directory, always and forever. But how do you build a path to a file located in a sub-directory of the project?

Well, read the function docs:

```
?here::here
```

You will see that you need to build a file path with character arguments:

```
here::here("dir", "anotherdir", "my_file.R")
```

You will see that a directory path is built with a `/` between each argument passed into the function.

But, check whether the path exists using `fs::file_exists()`:

```
fs::file_exists(here::here("dir", "anotherdir", "etc"))
```

Well of course, there is no such path if you type random characters.

Note: `{fs}` is a much better library for managing files on your *f*ile *s*ystem than the `{base}` library. You could have used `file.exists()` but I recommend using and becoming comfortable with `{fs}` as doing so will open more doors for you. For directories, the `{fs}` equivalent to ‘`dir.exists()` is `dir_exists()`.

Build a proper path to a file that actually exists and check whether it exists:

```
fs::file_exists(here::here("src", "figs", "plot_cty_hwy_mpg_point.R"))
```

No matter which system you are on, `{here}` will manage your file and directory paths **without hard coding**. This is the approach you will take for **all** project paths.

Step 3: Fixing File Paths with `{here}`

If you did not use `{here}` for path management, you will need to fix your file paths and repeat the steps in Part 4 (e.g., process of editing files, adding, committing, pushing, fetching, merging, etc.).

Remember

- Switch to your personal branch
-

Part 6: Render Website

Now that you have your files, you should all be able to render the website. In the RStudio toolbar, click **Build > Render Website**.