IT Club

Git & Github Workshop

What happens when I fork a repository? Is it like cloning but cooler?

How do we all work on the same code without stepping on each other's toes?

How on earth do teams manage complex projects with ease?

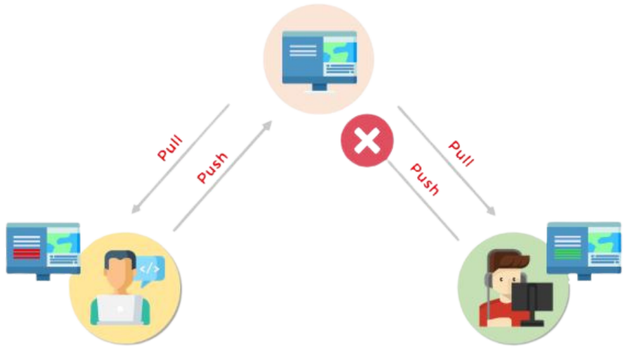
Day 3 - Collaborating & Utilizing Github Features

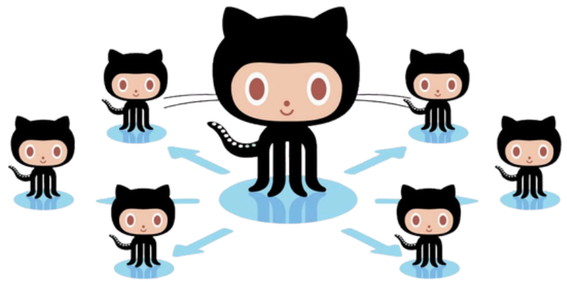
So far...

So far, we have learnt:

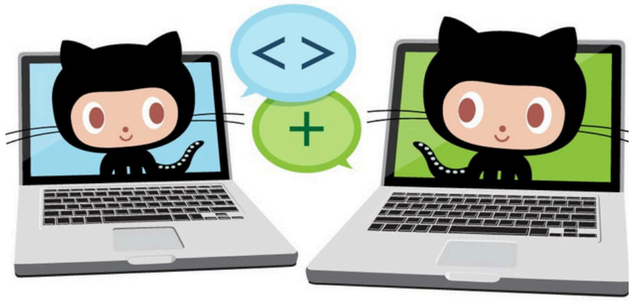
1. Basic git workflow
2. How to work with branches
3. How to merge branches
4. Connecting to Github

```
git add .
git commit -m "First commit @ItClub"
git push -u origin main
```

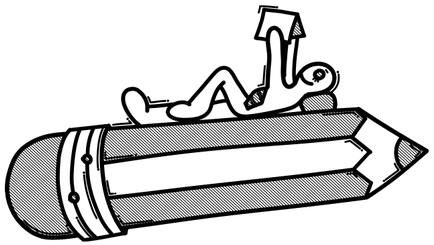




IT Club



Today's Agenda

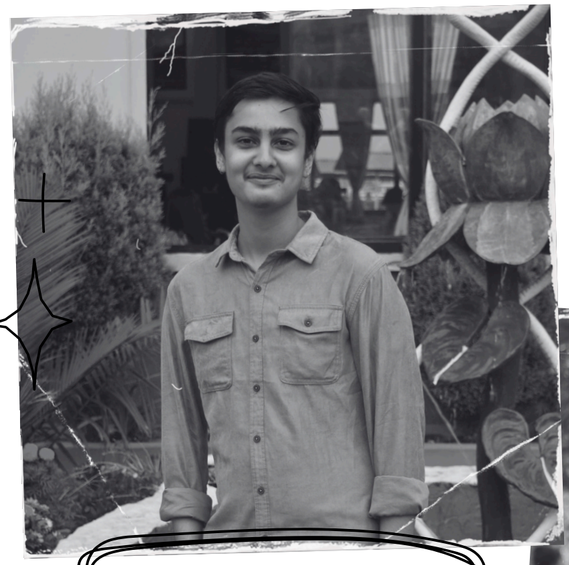


1. Working in Teams
2. Forks, Pull Requests & Merge Conflicts
3. Github Pages & Github Actions
4. Best Practices

Meet Your Instructors

Bishal & Sujan are both 3rd year Computer Engineering students in Pulchowk Campus. They will be your guide for today's session.

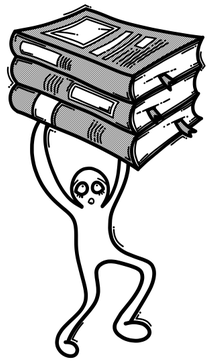
Ready to level up your workflow?



Bishal

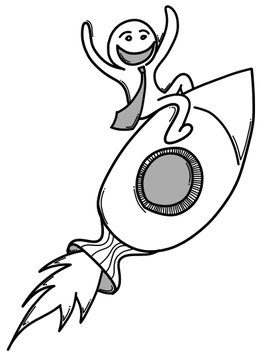
Sujan

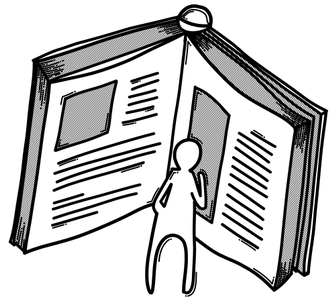
IT Club



How do we all work on the same thing without stepping on each other's toes?

Working in Teams





Example Scenario

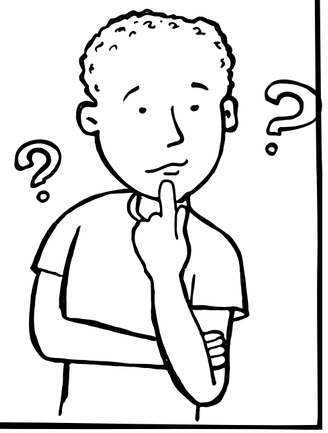
Imagine you and your friends are tasked with building a house. Each person has a specific role: one build up the foundation, another lays the walls, another installs electrical system and another installs the plumbing.

PROBLEM I

How do we ensure that everyone's work fits together seamlessly?

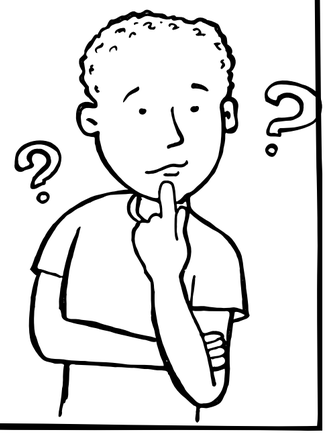
PROBLEM II

How do we handle conflicts when two people want to work on the same part of the project simultaneously (like installing plumbing and electrical systems in the same area)?

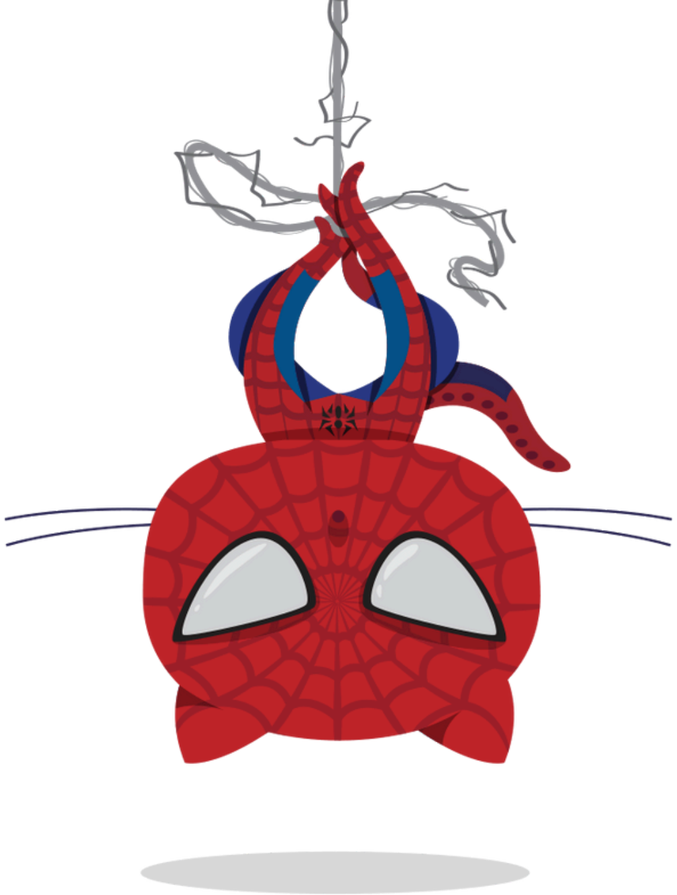


PROBLEM III

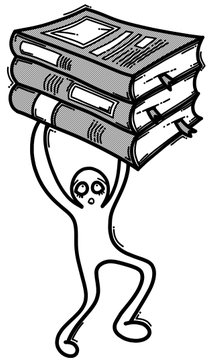
What if we need to roll back changes to a previous stage of development due to unforeseen issues (like discovering a flaw in plumbing)?



Luckily, We ain't Civil Engineers!

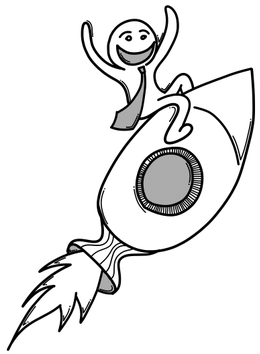


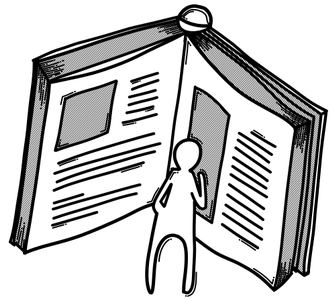
Git & Github to the rescue!



What’s the secret to smooth collaboration with Git and GitHub?

Collaborating in Teams with Git & Github





IT Club

Forking -
Cloning But Cooler!

Forking

Forking

Imagine you're working on the electrical system of our house project. But what if you need to experiment with a new type of wiring without affecting the main project?

Forking a repository is like **creating a replica of the house for you to play with**. You can make all the changes you want without affecting the main project!

And if they work out, you can suggest your improvements to the main project!

Wait, how does it differ from git clone?

Cloning is to copy a remote repository in your local machine.

On the other hand, **Forking** is essentially making a copy of someone else's repository into your own account. It is not exactly stealing the repo cause you will end up with a copy linked to the original one.

Why do we fork?

If you clone someone else's repository, you don't have right access to push any changes. And there is good reason for this! You don't want anyone pushing stuff to your repository.

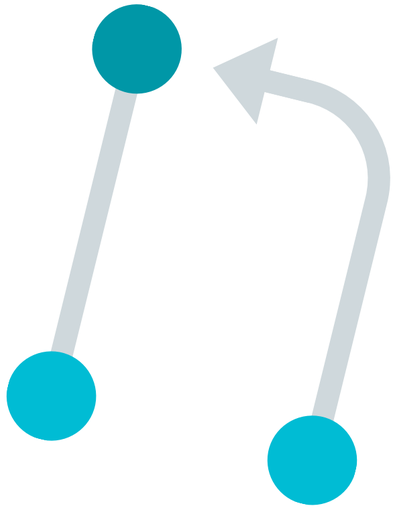
When you fork that repository, you create your own copy you can do all the changes you want into your own copy and then create a **pull request** to ask if your changes can be merged back into the original repository.

How do we fork?

Easy! Go to the repository you want to fork on Github and press the Fork button on top-right section.

PRs -
Just fancy requests
to merge code

Pull Requests (PRs)



Pull Requests

Now that you have made your changes to the wiring in your "forked house", how do you propose these changes to the main project?

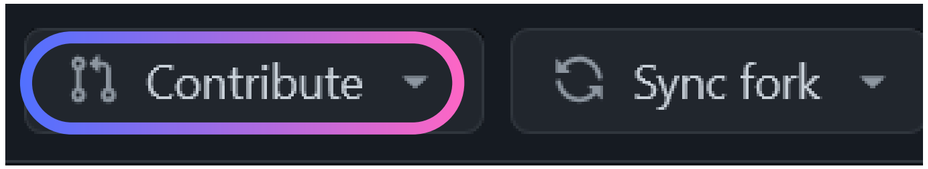
This is where Pull Requests come in.

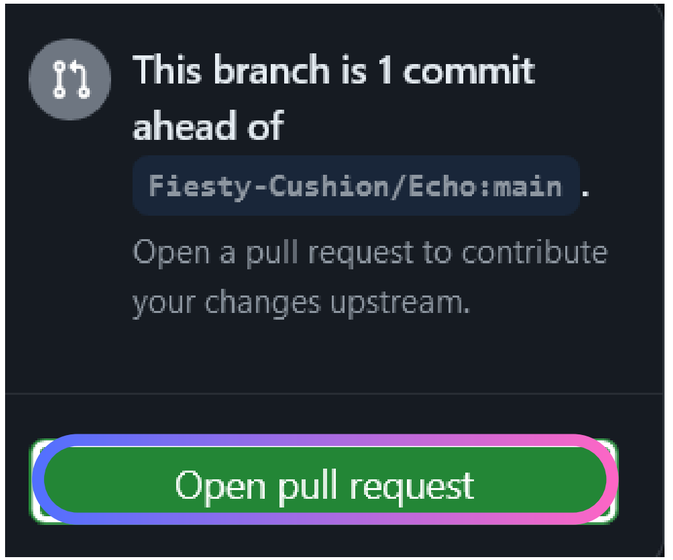
Pull Requests

Think of a PR as your formal proposal to the team to integrate your changes into the main house project. Just like in a proposal you should explain why your changes are beneficial and how they enhance the project.

Creating a Pull Request

1. First, ensure your changes are pushed to your forked repository.
2. Navigate to your repository and press the Contribute and Open Pull Request.

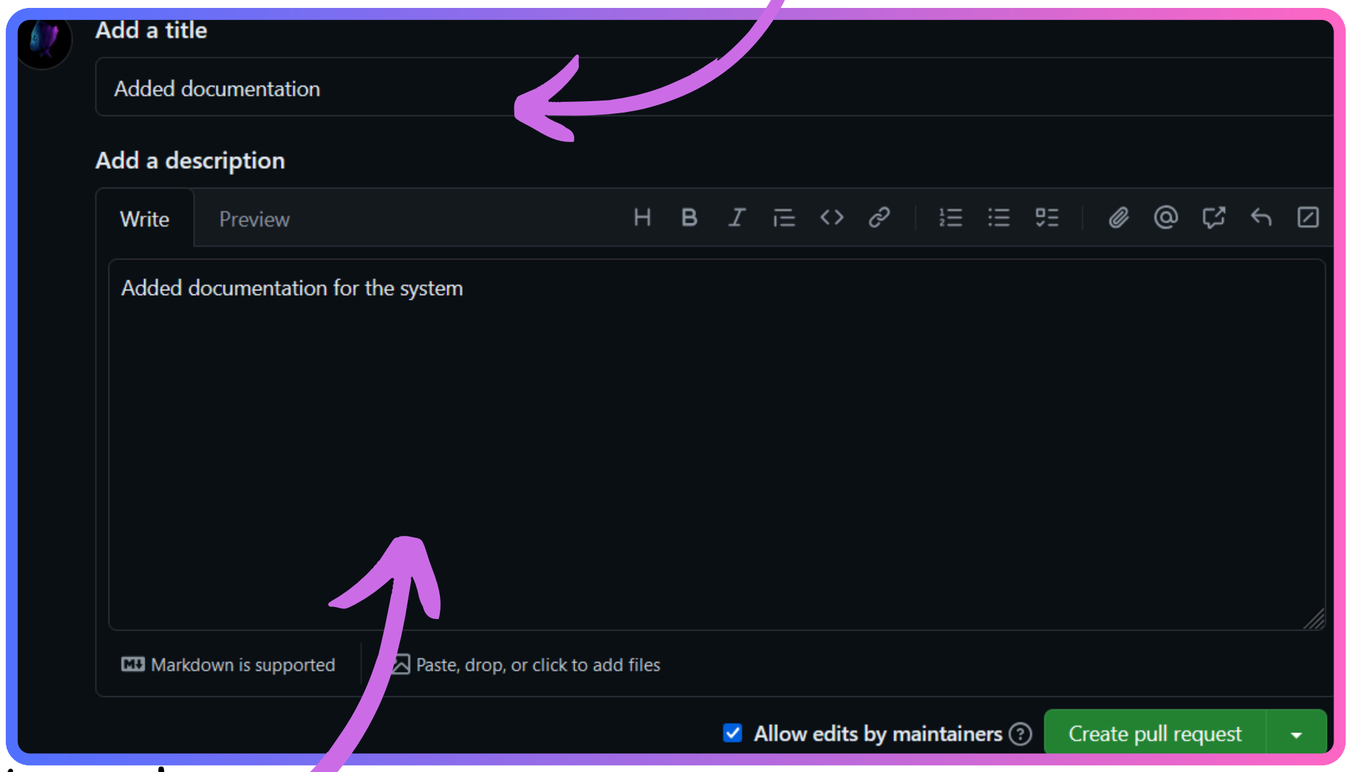




Creating a Pull Request...

Think of a PR as your formal proposal to the team to integrate your changes into the main house project. Just like in a proposal you should explain why your changes are beneficial and how they enhance the project.

Title

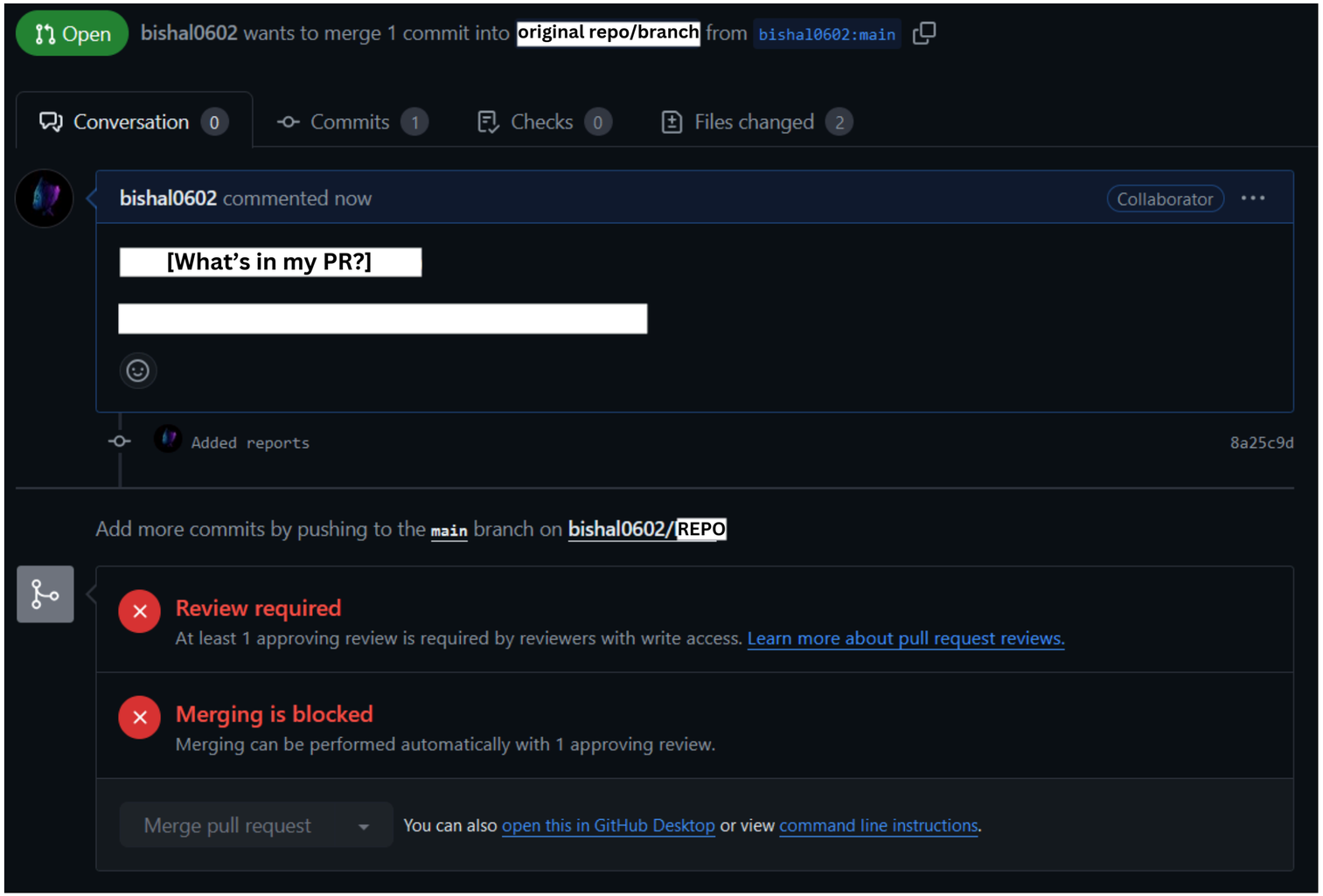


Explain what and why of your PR

Someone sent a PR!

Now, lets put you on the other side, Say someone proposed changes to your repository by sending a Pull Request.

NOW WHAT?!



Reviewing a PR

First of all you need to review the changes carefully, cause if the creator of the PR did something wrong, it might affect your entire code base(or in our analogy, cause damage to our entire house)!

Here is what you must do:

- Review the code for quality and consistency
- If possible, test the changes locally
- Provide feedback if something needs to be updated

Merging a PR

Now, after you made sure everything is good and approved, it's time to merge!

To merge choose a merge method based on your workflow:

- Merge Commit to preserve all commits
- Squash and Merge to combine commits into one
- Rebase and Merge to reapply commits cleanly.

Time to ~~break branches~~ stop breaking branches

Handling Merge Conflicts

Sometimes, two team members might be working on the electrical and plumbing systems simultaneously on the same wall, causing overlaps and conflicts. Merge conflicts in GitHub are similar.

When changes from different sources interfere, we need to manually resolve these conflicts.

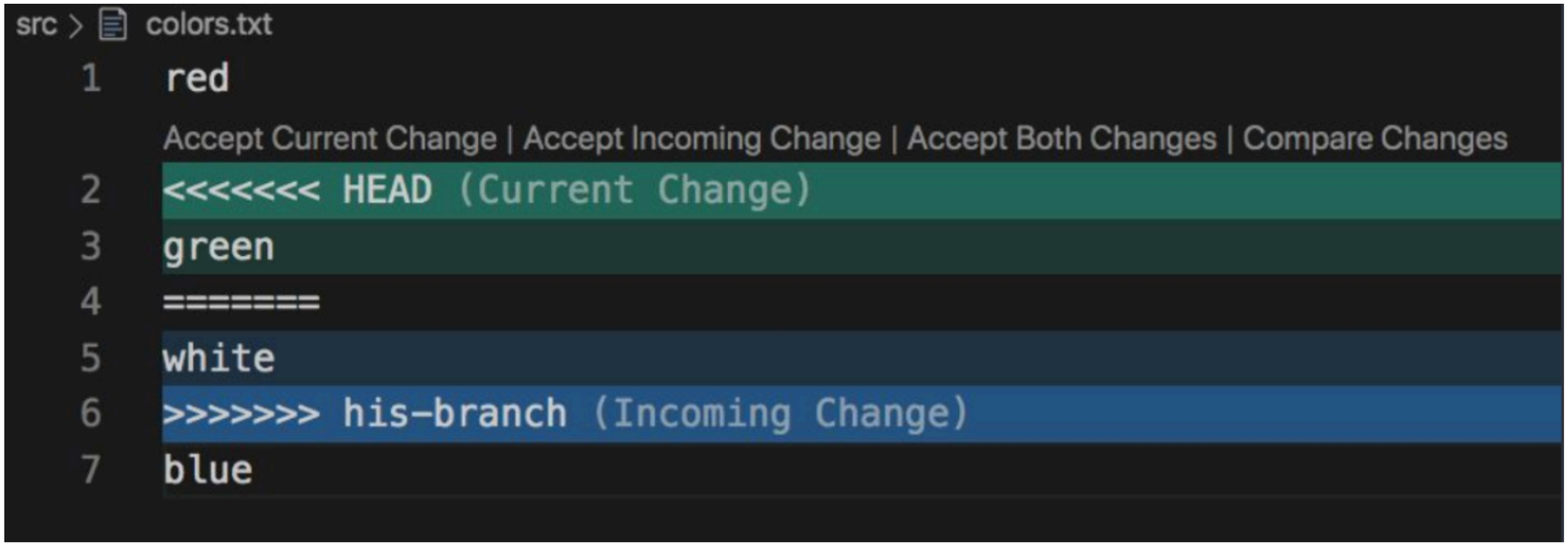
But, HOW?

To resolve merge conflicts you can apply the following strategy:

1. Identify conflicting files.
2. Manually resolve conflicts.
3. Test the integrated changes.

Tip: To resolve conflicts, use Git along with Modern IDEs. They will clearly mark your conflicts and help you resolve them with their built in Merge Conflict Editor.

But, HOW?...



What awesome features does GitHub offer to supercharge my projects?

Utilizing Github Features

I. Github Pages

GitHub Pages is a static site hosting service that takes HTML, CSS, and JavaScript files directly from a repository on GitHub, runs the files through a build process, and publishes a website for you! Free!

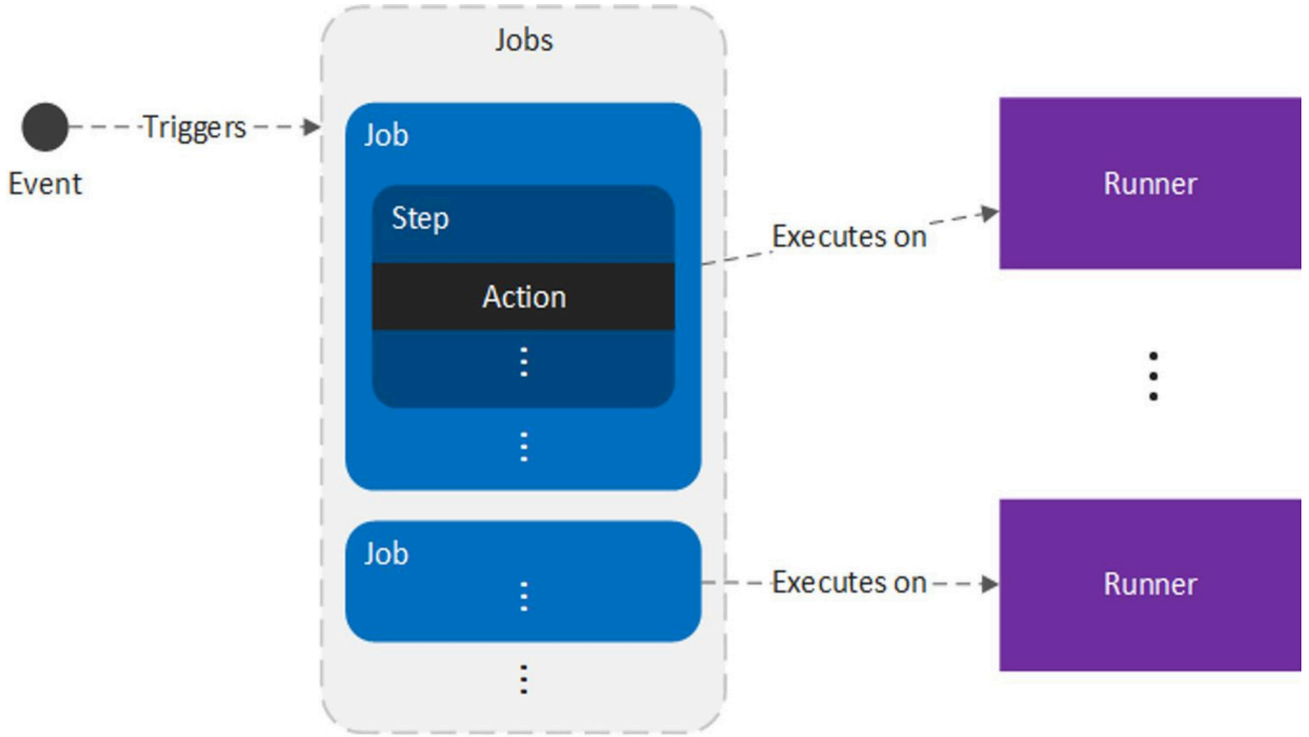
II. Github Actions

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline.

Meh, Too many buzzwords!

Lets simply it.

Simply put, Github Actions define sequence of tasks(called **Jobs**) that are triggered in response to certain events on your Github repository like pushing a commit, opening a pull request, or even triggered on a schedule or by an API request!



Some teminologies related to Github Actions:

Jobs

- Set of steps that is executed

Workflow

- Configurable automated process that will run one or more jobs

Events

- Specific activity in a repository that triggers a workflow run

Runners

- Server that runs your workflow

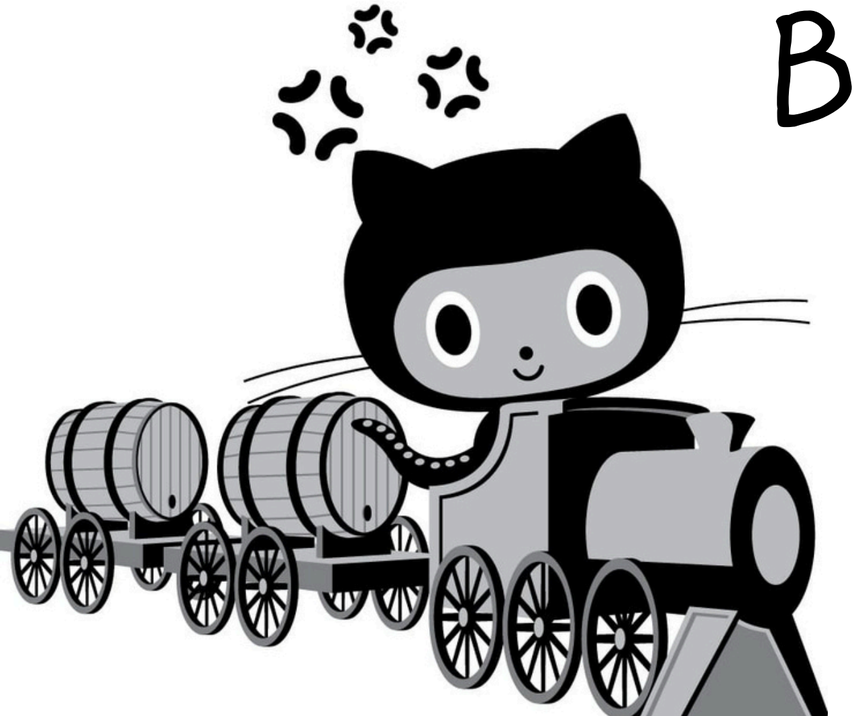
PRACTICAL TIME!

Cool! What else can I do with Github Actions?

- Automatically build and test your project on every commit.
- Deploy your application to different environments automatically.
- Automate the building and publishing of Docker images from your code base.
- Automatically scan for vulnerabilities and stay secure.
- Send alerts to Slack or email when a workflow completes.

What are the best practices to follow so my team does not hate working with me?

Git & Github Best Practices



ITClub

Keep your secrets secret!

Never commit sensitive information such as API keys, passwords, or secret tokens to your repository.

How?

- Use a .gitignore file to exclude sensitive files and directories.

Use Branches When Introducing New Features

Always create a new branch for each feature or bug fix to keep the main branch clean and stable.

Wanna see if a balcony will look good on your house?
First do:

```
git checkout -b feature/add-balcony
```

Keep Commits Small, Focused & Atomic

Each commit should represent a single logical change. This makes it easier to understand the history and revert changes if necessary.

Bad commits

```
git commit -m 'Updated dependencies, added error handling,
updated styling and refactored variable names'

```

Keep Commits Small, Focused & Atomic

Good commits

```
# Complete updating dependencies ...
git commit -m 'Update dependencies to latest versions'

# Complete adding error handling ...
git commit -m 'Add error handling for database connection'

# Complete updating styling ...
git commit -m 'Update styling for mobile responsiveness'

# Complete refactoring ...
git commit -m 'Refactor variable names for clarity'

```

Keep commit message meaningful

Commit messages should be clear and descriptive, providing context about the change. Cause if you use commit messages like these:

```
git commit -m 'Fixed stuff'
git commit -m 'Changes'
git commit -m 'Update'
git commit -m 'Bug fixes'
git commit -m 'Miscellaneous updates'
```

No one is gonna love you!

Use git diff before commiting

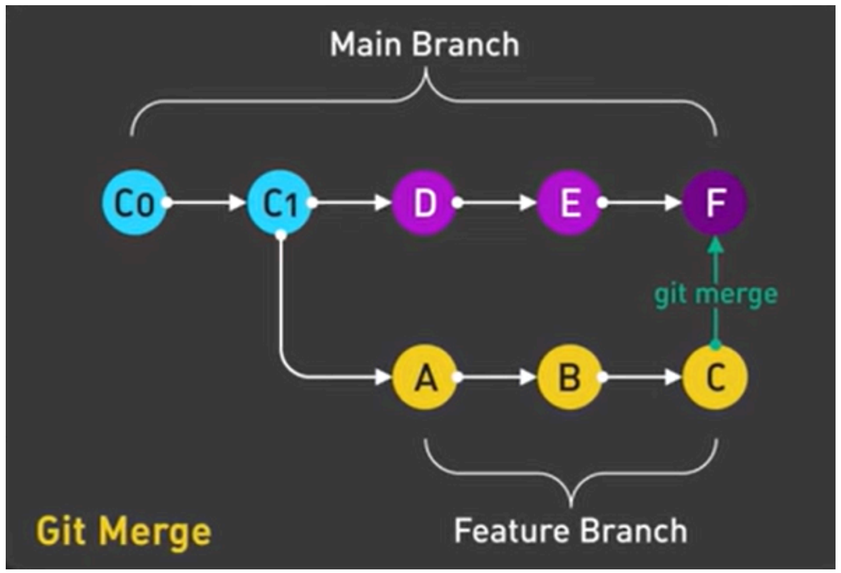
Always use git diff to review your changes before committing. This helps catch any mistakes or unintended changes early.

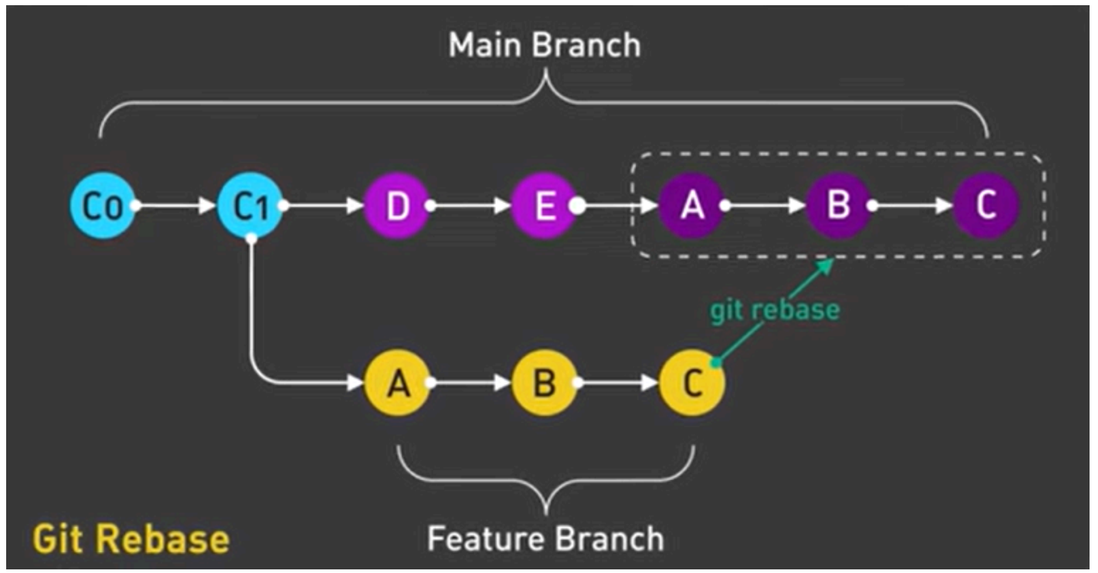
```
diff --git a/main.cpp b/main.cpp
index 6ef70fe..eca6e54 100644
--- a/main.cpp
+++ b/main.cpp
@@ -2,6 +2,6 @@

 int main()
 {
-        std::cout<<"Hello world";
+        std::cout<<"Hello universe";
         return 0;
 }
```

Keep a Clean and Linear Commit History with Rebase

git rebase allows you to keep your commit history clean and linear by applying your changes on top of another base tip.





And, once again remember to teach your kids how to use .gitignore!

Thank you! ☺

Today we learnt:

1. How to collaborate in team using Git & Github
2. How to utilize Github features
3. What should you do, so your team doesn't hate you

Join us at discord:

IT Club

bit.ly/itclub-discord