Three-Linked Repositories

Learn how to link three repositories so that you can retrieve changes from them to your local repository.

We'll cover the following

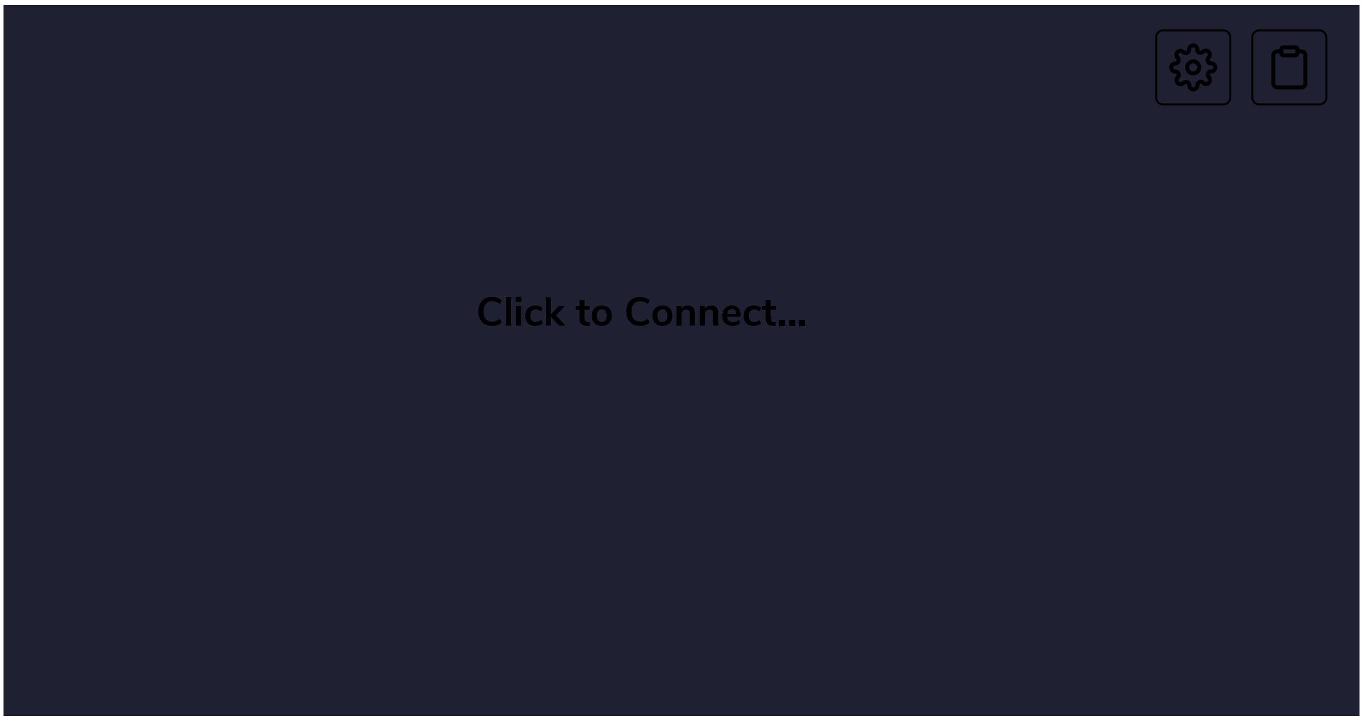
Now you are going to work with multiple repositories.

You're going to do the same as you did in the last chapter, but this time you will create *two* clones of the origin: `alice_cloned` and `bob_cloned`.

```
mkdir -p lgthw_remotes
cd lgthw_remotes
mkdir git_origin
cd git_origin
git init
echo 'first commit' > file1
git add file1
git commit -am file1
cd ..
git clone git_origin alice_cloned
git clone git_origin bob_cloned
```

Terminal 1

Terminal



Now `alice_cloned` and `bob_cloned` have `git_origin` as the origin remote:

```
cd alice_cloned
git remote -v
```

Alice makes a change in her `master` branch. Type the following three commands in the terminal above.

```
cd ../alice_cloned
echo alice_change >> file1
git commit -am 'alice change'
```

Current workflow

The workflow currently looks like this:

```
alice_cloned <=> git_origin <=> bob_cloned
```

Remote changes into your `master` branch

Let's see how you can make changes into your master branch without going to origin.

going to origin.

Problem

The question is: how does Bob get Alice's change into his `master` branch without going to the `origin`?

This is a common scenario in distributed teams. If you consider that Git was created for managing the codebase of the Linux operating system, it's easy to imagine the `git_origin` as Linus Torvalds' repository, Alice as a contributor, and Bob as a so-called lieutenant.

Solution

So here is how:

1. ADD Alice's repository as a remote to Bob's.
2. FETCH Alice's updated `master` branch.
3. MERGE Alice's `master` branch into Bob's local one.

As you have already seen, steps 2 and 3 can be collapsed into a `git pull`, but it is more instructive to keep these separate.

1. ADD Alice's repository as a remote to Bob's.

First, Bob needs to add Alice's repository as a remote.

```
cd ../bob_cloned
git remote add alice ../alice_cloned
git remote -v
```

Terminal 1

Terminal

You have now linked up your repository to Alice's and given it the name `alice`.

2. FETCH Alice's updated `master` branch.

```
git fetch alice master
```

Alice's `master` branch is now fetched to your local repository.

```
git branch -vv -a
```

3. MERGE Alice's `master` branch into Bob's local one:

```
git merge alice/master
cat file1
```

You may be wondering why you use `alice/master` and not `remotes/alice/master`, as the output of `git branch -vv -a` tells you. You can run:

```
git merge remotes/alice/master
```

It will do the same. Git assumes that the branch is a remote (presumably from seeing the `*/*` in the branch) and adds the `remotes` for you.

Git's intended purpose vs. actual use

This "Lieutenant's model" is one example of a Git workflow. Although it was the one Git was originally created for, it is still common for developers to use a traditional centralized model around a repository such as GitLab or BitBucket.

This is why people make jokes when GitHub is down. Git is designed to be a distributed source control tool, but the simplicity of depending on a central server is also a powerful and tempting model. In any case, Git can support both models.

← Back

Introduction: Working With Multiple R...

Next →

Challenge: Working With Multiple Rep...

✓ Mark as Completed