



## Lab 3. Revision Control with Git

By the end of this lab exercise, you should be able to:

- Configure a Git client.
- Create and work with repositories.
- Understand the basics of revision control operations.
- Understand basic branching and merging.
- Work with remotes.
- Resolve conflicts.
- Leverage Docker for local development.

### Configuring Git Client

Before you start revision control, you need to configure Git. There are three different places you can configure Git, as explained here.

Git configs:

Type	Scope	Config Path
:-----	:-----	:-----
LOCAL	Repo	.git/config
GLOBAL	User	/home/user/.gitconfig
SYSTEM	System	/etc/gitconfig

You will create configurations with user scope, valid for all repositories created by the current user:

```
git config --global user.name "username"
```

```
git config --global user.email "user@gmail.com"
```

```
git config -l
```

```
cat ~/.gitconfig
```

You can also create additional configurations, e.g. to switch the default editor to a terminal text editor called *Vim*:

```
git config --global core.editor vim
```

To see the change, run the following command:

```
git config -l
```

To edit the configuration file, use the `-e` flag as follows:

```
git config -e --global
```

This will open the file in Vim. To exit Vim type the following:

```
:q
```

This will exit Vim and bring you back to the terminal.

Vim will now be the default editor for Git. *Nano* and *Emacs* are two other command line text editors that can be used. Pick your favorite. Nano is the most intuitive.

## Creating a Sample Application

We need an application to use version control on.

You will now be walked through how to create a simple `hello-world` application using the Java-based **spring-boot** framework.

To create the application visit <https://start.spring.io/>.

Be sure to select **Maven Project**. Change your artifact name to `hellocd`. This can be anything you want, but we will change the artifact name to `hellocd`. Select **Java 17**. Click **Generate** to download your application.

The screenshot shows the Spring Initializr web application. The 'Project' section has 'Maven' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '3.0.0' selected. The 'Project Metadata' section has 'Artifact' set to 'hellocd', 'Name' set to 'hellocd', and 'Description' set to 'Demo project for Spring Boot'. The 'Packaging' section has 'Jar' selected, and the 'Java' section has '17' selected. The 'Dependencies' section is empty. At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

After downloading the application code, move the .zip file to your local working directory and unzip that application. You will use this application for revision control operations.

## Basics of Revision Control

Now that you have a `hellocd` application downloaded and unzipped, you will use it to learn revision control operations. Enter the `hellocd/` directory.

```
cd hellocd
```

We will tell Git to make this directory a repository by *initializing* it. We do this with the following command from inside `hellocd/`:

```
git init
```

Now run:

```
git status
```

**git status** shows the status of all the files and directories in the repository. Untracked files and directories appear in red. You need to add these files so that Git starts tracking them. Use the following commands to start tracking the files in the `hellocd` project with Git:

```
git add pom.xml
```

```
git status
```

---

```
git commit
git status
git add src
git status
git add *
git commit mvnw
git status
git commit -am "adding files created with spring boot generator"
git log
```

`git log --branches` shows you the commit history and where each branch is at, while `git status` shows untracked files in your directory. Create one more file in your working directory called `README.md`.

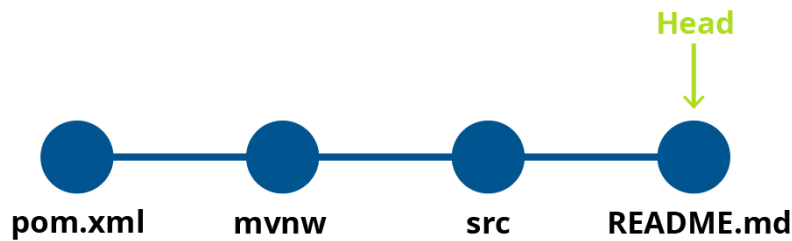
**Note:** the following command will start with whatever your chosen command line text editor was, nano, emacs, or vim:

```
vim README.md
```

Check your git status after that commit using **reset**:

```
git add README.md
git status
git reset README.md
git add README.md
git commit -am "adding README"
```

You have added and committed the files one by one, and this is the current state of your repository with each commit represented by a circle on the line:



## Using Docker in Development

After creating a Java project based on Maven, we need to build it and test it and then it can be deployed into an environment. To do this, you will need a build environment. Instead of setting up and then managing a build environment with Maven, Java, etc., you can leverage Docker's disposability feature.

You need to get the absolute path to the `hellocd` directory. From inside `hellocd` run:

```
pwd
```

The output will be the absolute path to `hellocd`. Be sure to use it in the following command in place of `/absolute/path/to/hellocd`.

Use the following command to run the container:

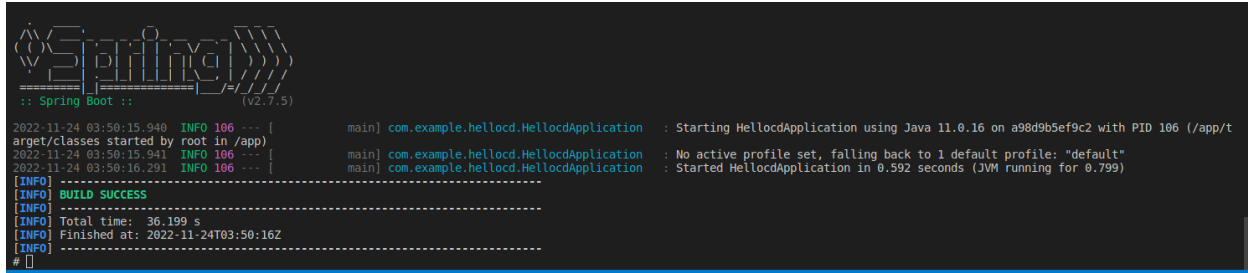
```
docker container run --rm -it -v /absolute/path/to/hellocd:/app  
maven:3.8-openjdk-17-slim sh
```

Running the above command gets you:

- A disposable container environment with Docker
- Created with the image `maven:3.8-openjdk-17-slim`
- This container has build tools necessary to run a Java application, including Maven, Java, etc.
- Exiting this container will delete it, as we are using the `--rm` option.

You will see that your terminal prompt has switched to a `#` which indicates that you have a terminal inside the container. While inside the container, run the project with:

```
# mvn spring-boot:run -f app/pom.xml
```



```

:: Spring Boot ::
(V2.7.5)

2022-11-24 03:50:15.940 INFO 106 --- [main] com.example.hellocd.HellocdApplication : Starting HellocdApplication using Java 11.0.16 on a98d9b5ef9c2 with PID 106 (/app/t
target/classes started by root in /app)
2022-11-24 03:50:15.941 INFO 106 --- [main] com.example.hellocd.HellocdApplication : No active profile set, falling back to 1 default profile: "default"
2022-11-24 03:50:16.291 INFO 106 --- [main] com.example.hellocd.HellocdApplication : Started HellocdApplication in 0.592 seconds (JVM running for 0.799)

[INFO] BUILD SUCCESS
[INFO] Total time: 36.199 s
[INFO] Finished at: 2022-11-24T03:50:16Z
[INFO]
#

```

You can see your libraries are stored in the `/root/.m2/` directory.

```
# ls /root/.m2/repository
```

```
ch com commons-codec commons-io io jakarta javax net org
```

Type **exit** to return to your machine's terminal:

```
# exit
```

The container is immediately destroyed. You can reduce the time to download libraries by creating a volume which can then be mounted at this path.

```
docker volume create m2
```

```
m2
```

Run the following command:

```
docker container run --rm -it -v m2:/root/.m2 -v /path/to/hellocd:/app
maven:3.8-openjdk-17-slim mvn spring-boot:run -f /app/pom.xml
```

Run the same command again. You should notice the build is faster due to the Maven package being cached with the newly mounted volume.

## Branching and Merging

Create a branch using the `git branch` command and switch to that branch using `git checkout`. The default branch is master. Whenever we need to work on another branch, we use `git checkout` so let's run the following commands to see our branches and create and move to the `webapp` branch:

```
git branch --list
```

```
git log
```

```
git branch webapp
```

```
git branch --list
```

```
git checkout webapp
```

```
git branch --list
```

Instead of using a two-step process, you could have created and switched to the **webapp** branch with the following single command:

```
git checkout -b webapp
```

You can delete your branch using `git branch -d`. Before deleting the branch, you should switch to another:

```
git checkout master
```

```
git branch -d webapp
```

Now `git branch` will show that you only have the **master** branch. Since we want the **webapp** branch, we will create it and switch to it one more time:

```
git checkout -b webapp
```

## Adding a New Feature by Branching Out

Now you are going to add new features to your application:

- For source code changes visit the [lfs 261 devops-repo](#) and go to the **hellocd** directory. Copy `HellocdApplication.java.v1` code and replace the code in `HellocdApplication.java` which you will find in your `src/main/java/com/example/hellocd` directory.
- Copy `pom.xml.snippet1` from <https://github.com/lfttraining/devops-repo/blob/master/hellocd/pom.xml.snippet1> and paste it into the `pom.xml` file under the dependencies section between `<dependencies>` and `</dependencies>` before *or* after the dependencies you see present there.

Now you can check your git status. The result will be that those files you have changed are unstaged. We will add them, but first you can review the changes made using the following:

```
git diff
```

Continuously press **Enter** or hold **Enter** down to scroll to the bottom to see all changes. Press **q** to exit.

You can commit after `git add` or else directly commit by using `git commit -a`. The `-a` option, if you recall, means 'add all files with changes to the staging area'. We will also use the

`-m` option to add a message from the command line:

```
git commit -am "committing changes to pom.xml and  
HellocdApplication.java"
```

```
git status
```

Now you can see the changes are in the new branch. The master still does not have any of those incorporated yet. This is the safer way of bringing your code to master. In the next subsection, you will learn how to bring these changes into the master.

## Merging with Commit History

Previously you have made some changes and committed to a new branch `webapp`. It's now time to test the application.

Again, run the same `spring-boot` container command with the port mapped to 8080 on host:

```
docker container run --rm -it -v m2:/root/.m2 -v /path/to/hellocd:/app  
-p 8080:8080 maven:3.8-openjdk-17-slim mvn spring-boot:run -f  
/app/pom.xml
```

Once the container is running, you can visit the application on `http://IPADDRESS:8080`, in most cases this will be `http://localhost:8080`. When you visit your `webapp`, right now you will get an error due to some missing code in `HellocdApplication.java`. To fix this issue, copy the code inside [HellocdApplication.java.v2](#) and use that code to replace *all* the code inside your own

`hellocd/src/main/java/com/example/hellocd/HellocdApplication.java` file.

After replacing the snippet, run the following command to see your changes:

```
git diff
```

You will see that Git has made the added lines green. Continuously press `Enter` or hold `Enter` down to scroll to the bottom to see all changes. Press `q` to exit.

```
git commit -am "added missing annotations"
```

Again, run your `spring-boot` container and visit `localhost:8080`. You should now see `Hello World!` in your browser.

Change "Hello World!" to "Hello Continuous Delivery!" in `HellocdApplication.java`.

Commit the change, then run your container. You will see `Hello Continuous Delivery!` in the browser.



```
1  package com.example.hellocd;
2
3  import org.springframework.boot.*;
4  import org.springframework.boot.autoconfigure.*;
5  import org.springframework.web.bind.annotation.*;
6  import org.springframework.boot.SpringApplication;
7  import org.springframework.boot.autoconfigure.SpringBootApplication;
8
9  @SpringBootApplication
10 @RestController
11 @EnableAutoConfiguration
12 public class HellocdApplication {
13
14
15     @RequestMapping("/")
16     String home() {
17         return "Hello Continuous Delivery!";
18     }
19
20     public static void main(String[] args) {
21         SpringApplication.run(HellocdApplication.class, args);
22     }
23
24 }
```

Now that the feature is added, it's time to incorporate these changes into master by using `git merge`. Before you do so, make sure you switch to the master branch first:

```
git checkout master
```

```
git merge webapp
```

```
git log --branches
```

We can now delete our `webapp` branch using `git branch -d`:

```
git branch -d webapp
```

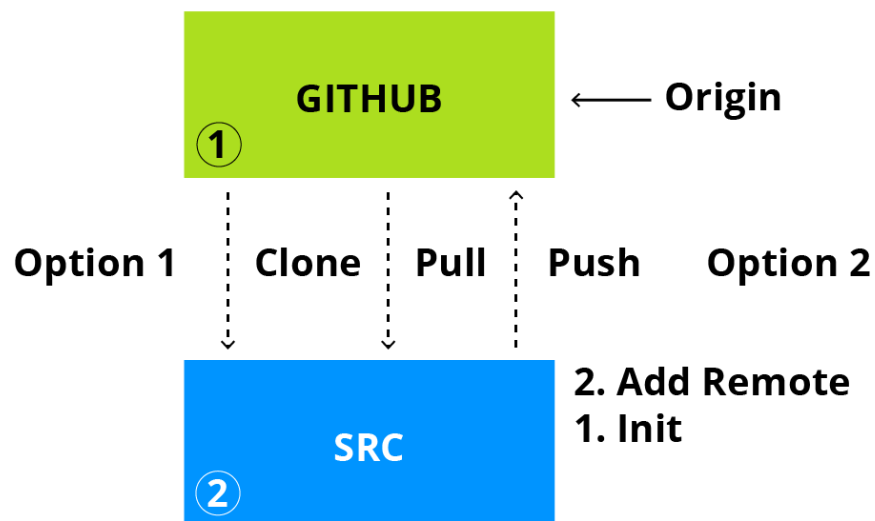


## Working with Remotes

You can collaborate and work with different branches, users or central repositories by using *remotes*. Here, your central repository is GitHub.

There are two ways to go about this:

1. Create the repository on GitHub first.
  - a. Clone the repository on your local machine.
2. Create the repository on your machine first.
  - a. Create the repository on GitHub
  - b. Tell your local repository to make the remote repository your GitHub repository



Before you start, you need to create an account on GitHub. Visit [GitHub](https://github.com) to create your account. While creating your account, use the free account option.

Once you create an account, create a new `hellocd` public repository. You have two ways to work with a repository. You could use an initialized repository or clone the repository from GitHub and make changes and push to your repository.

Here you have already initialized a repository. Now you need to add the newly created repository as origin by using the following command and push your application to that origin master:

```
git remote add origin https://github.com/xxxxx/hellocd.git
```

[Replace `xxxxx` with your GitHub user/org name, as per your repository URI]

```
git remote show
```

```
git remote show origin
```

```
git push origin master
```

Once you push to the origin master, you can see your code in the repository with the commit that you have changed.

## Resolving Conflicts

When you are collaborating with others, conflicts can happen. Git provides you with various options to resolve them.

Add the line `server.port=80` in `src/main/resource/application.properties` while you are on the master branch, commit and push:

```
git diff
```

```
git commit -am "added server port 80"
```

```
git push origin master
```

## Creating a Conflict

To resolve a conflict, first we will have to create one. To do that, you will create two branches and update the same file with different content in each branch. The following sequence of commands will help you do that.

You have an option to work in pairs. The commands are differentiated with `user1` and `user2`. Make sure you are working with the same repository in case you are working in pairs. If you

cannot find a partner, go ahead and run both blocks of commands from the same workspace.

*user1*: Use the following code:

```
git checkout -b user1
```

Set the value of `server.port` to 8080 in

`src/main/resource/application.properties`:

```
git diff
```

```
git commit -am "this should run on port 8080"
```

```
git push origin user1
```

*user2*: Use the following code:

```
git checkout -b user2
```

Set the value of `server.port` to 8081 in

`src/main/resource/application.properties`:

```
git diff
```

```
git commit -am "this should run on port 8081"
```

```
git push origin user2
```

Once you do the above, switch to the master branch and try to merge the branches you just updated, simultaneously. Each will run into a conflict.

*user1*:

```
git checkout master
```

```
git pull origin user2
```

```
git log --branches
```

```
git push origin master
```

*user2*:

```
git checkout master
```

```
git pull origin user1
```

```
git log --branches
```

```
git push origin master
```

---

You should get into a conflict here. Use `git log` to check the details.

## Manually Resolving a Conflict

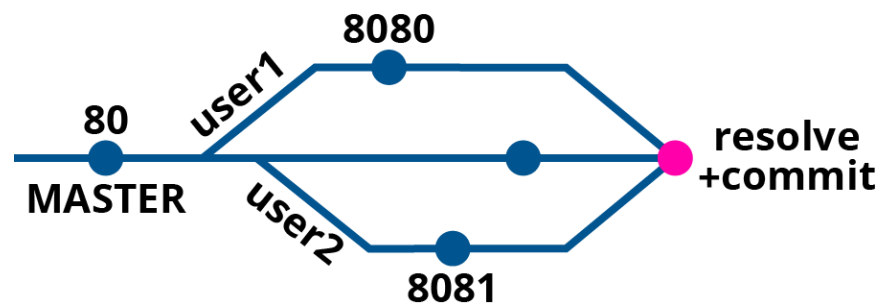
You can resolve a conflict manually by editing the file in conflict. In this example, you can pick one value for `server.port` and keep that line in the file. Remove everything else including the lines starting with `<<<<<<<` and `>>>>>>>`. Once you do so, commit the changes:

```
git commit -am "resolved merge conflict"
```

```
git push origin master
```

Validate by examining the commit history:

```
git log --branches
```



Once you complete this, delete your `user1` and `user2` branches by using the following commands:

```
git branch -d user1
```

```
git branch -d user2
```

```
git push origin --delete user1
```

```
git push origin --delete user2
```

## Learning to Undo

It's common to add changes to the commit history and later wish you could have taken those back. At times, there are mistakes or bugs you detect in your feature branches that do not need to be reflected in the commit history. There are often times when you may wish to unstage a file. Git offers the `reset` command with its options to achieve this.

Lets add a change that we will undo later:

```
git checkout -b config
```

```
git branch
```

Update `src/main/resource/application.properties` with `server.port=9000`:

```
git status
```

```
git add src/main/resource/application.properties
```

```
git status
```

The file has now been staged. To unstage the same:

```
git reset src/main/resource/application.properties
```

```
git status
```

```
git diff
```

```
git commit -am "changed port to 9000"
```

```
git log
```

```
git push origin config
```

You have not only added it to a commit history, but also pushed the change to the remote.

Let's say that you have realized that 9000 should really be 8080.

Try using the following command:

```
git reset HEAD~1
```

Note: that is HEAD *tilda* 1. Don't mistake the tilda for dash or it won't work. This should revert the last commit and point the **HEAD** (tip of the git repo) to the previous commit. You could use **HEAD~2**, **HEAD~3** to shift the **HEAD** back by two/three commits respectively. You could optionally use the commit id to shift **HEAD** to a specific commit.

What you did above was a mixed reset. This means, the undo was done for the commit history only. You would still see the files in the staging area, and the current working tree being unchanged.

You can undo it all the way (this is a destructive process) by doing a hard reset:

```
git reset --hard HEAD~1
```

Once you reset, if you try to push the changes to remote, it should give you an error. To undo changes to remote, use the **force** option:

```
git push origin config -f
```

## Revert vs Reset

**reset** is useful while working on feature/personal branches. However, when you have a collaborative workflow with common branches such as trunk/mainline/master, resetting changes on common branches could leave everyone else confused and in an inconsistent state.

This is because **reset** wipes out the commits to undo changes. A cleaner solution in such cases is using **git revert** instead, which undoes the changes by adding a new commit, which is then part of the commit history, leaving everyone in sync.

To learn how **revert** works, let's first update a file and commit it.

Use the **config2** branch and file and commit the changes in the feature branch, bring those changes to the master branch using following commands:

```
git checkout -b config2
```

Update **src/main/resource/application.properties** with **service.port=7000**:

```
git diff
```

```
git commit -am "run on port 7000"
```

```
git push origin config2
```

```
git checkout master
```

```
git pull origin config2
```

```
git push origin master
```

Now that you have incorporated these changes to the master branch, it's time to revert:

```
git log
```

```
git revert HEAD
```

or

```
git revert 5080dcc9d13ac920b5867891166a
```

Where,

- 5080dcc9d13ac920b5867891166a is the previous commit id that you would want to revert.

Once you revert, you need to sync the changes in the master branch with remote by using following command:

```
git log
```

```
git push origin master
```

Once you sync the changes, it will show the changes to your co-developer and delete your `config2` branch after `git revert`:

Go ahead and delete the branch:

```
git branch -d config2
```

```
git push origin :config2
```

## Working with GitHub

You have been tasked to create a new repository to keep a journal. On GitHub, create a repo and add a description and `init` with a `README.md` file with the license for your public repository.

Once you create the repository, clone the repository locally by using the following command and add your skills in a file, then push to your origin. Create a file inside the repo called `skills.md`:

```
git clone https://github.com/yourUsername/skills-journal.git
```

Create a file inside the repo called `skills.md` then commit it:

```
git commit -am "added skills"
```

```
git remote show
```

```
git remote show origin
```

```
git push origin master
```



If you like, you can create more files, commit them and push them to your remote repository on GitHub.

## Summary

In summary, we initialized a local git repository on our machine. We created a SpringBoot application in that repository. We made changes and kept track of those changes with Git. We ran a Java and Maven container so that we would not have to load Maven on our local machine. This container ran our SpringBoot application. We connected our local git repository to a remote git repository on GitHub. Finally, we learned how to handle conflicts when merging between branches both locally and remotely.