# Lab: (Sourcetree) Commit to a Local Repository

Estimated time: 10 minutes

> Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.
>
> Note: This lab assumes that you have created a local repository named `projecta`. This was done in the previous lab.

In this lab, you will:

1. View the file status of an untracked file.
2. Stage a file.
3. Create a commit.
4. View your commit history.

## 1: View the file status of an untracked file.

1. **(Mac)** From the main Sourcetree window, double-click on the `projecta` repository to view its details.
   **(Windows)** View the tab with the `projecta` repository.

2. You should see that the `File status` tab is highlighted, and that there are no files in the working tree. You should see the message "Nothing to commit".

3. Using whatever method you choose, create an empty file named `fileA.txt` in your `projecta` directory. This will be the first file in the working tree of the repository.

4. **(Mac)** Click on the Sourcetree window to refresh it (if necessary). You should now see that the `File status` tab has a 1 next to it. This indicates that Git has recognized a change related to one file. You should see your `fileA.txt` file as untracked, indicated with a question mark icon.

   **(Windows)** Click on the Sourcetree window to refresh it (if necessary). Under `Unstaged files`, you should see your `fileA.txt` file as untracked, indicated with a question mark icon.

   > Congratulations, you have viewed an untracked file in your working tree.

## 2: Stage a file.

1. **(Mac)** Select the three dots next to your `fileA.txt` file. This brings up a menu. Select **Stage file**.

   **(Windows)** Under `Unstaged files`, select `fileA.txt` and click **Stage Selected**.

2. You have added `fileA.txt` to the staging area. Notice that `fileA.txt` now shows as a staged file, as indicated by a green plus icon. This means that the next commit that you make will include this file.

   > Congratulations, you have staged a file.

## 3: Create a commit.

1. **(Mac)** Click **Commit** to begin the process of creating your first commit.

   **(Windows)** Optionally click the Commit button at the top of the screen. You usually can just enter a commit message without having to do this.

2. In the lower part of the screen, enter a commit message of "add fileA.txt".

3. Click **Commit**. You should again see the status of "Nothing to commit".

   > Congratulations, you have created a commit.

## 4: View your commit history.

1. **(Mac)** Click the **History** tab on the left. You should see your commit and your commit message.

   **(Windows)** Click the **Log/History** tab at the bottom. You could also click on the `master` branch label at the left. You should see your commit and your commit message.

2. You will not use the `projecta` repository in future labs. Feel free to delete it using the procedure below.
   **(Mac)** In the main Sourcetree window, right-click on the repository name and select `Delete`.

   **(Windows)** In a new tab, right-click on the repository name and select `Delete`. Then select `Delete Repositories on Disk`. This removes the repository from Sourcetree and

deletes the project folder.

> Congratulations. You have staged a file and created a commit using Sourcetree.

# Lab: (Command Line) Commit to a Local Repository

Estimated time: 10 minutes

> Note: This lab assumes that you are using a command line. If you would prefer to use Sourcetree, there are separate instructions.
>
> Note: This lab assumes that you have created a local repository named `projecta`. This was done in the previous lab.

In this lab, you will:

1. View file status using `git status`.
2. Stage content using `git add`.
3. Commit content using `git commit`.
4. View the commit history using `git log`.

## 1: View file status using `git status`.

1. In a command line, navigate to your `projecta` directory and execute `git status`. You should see the message "Nothing to commit".

2. Create an empty file named `fileA.txt` in the `projecta` directory using `touch fileA.txt`. This is the first file in your working tree.

3. Execute `git status`. You should see that Git notices the `fileA.txt` file and identifies it as untracked.

> Congratulations, you have viewed an untracked file in your working tree.

## 2: Stage content using `git add`.

1. Add `fileA.txt` to the staging area using `git add fileA.txt`.

2. Execute `git status` again. You should see that Git adds `fileA.txt` under "Changes to be committed".

> Congratulations, you have staged a file.

## 3: Commit content using `git commit`.

1. Execute `git commit –m "add fileA.txt"` to create a commit. The -m option adds a

commit message.

> Congratulations, you have created a commit.

## 4: View the commit history using `git log`.

1. Execute `git log`. You should see your commit details, including your commit message.
2. Execute `git log --oneline`. A concise version of the history is displayed.
3. You will not use the `projecta` repository in future labs. Feel free to delete it by simply deleting the `projecta` directory.

> Congratulations. You have staged a file and created a commit.

## Lab: Create a Remote Repository

Estimated time: 5 minutes

> *Note: This lab does not use the command line or Sourcetree. It applies to all users.*

> *Note: This lab assumes that you have or will create a free Atlassian account. Though not recommended, you can skip labs involving a remote repository if you would like.*

In this lab, you will:

1. Create a remote repository on Bitbucket.

**1: Create a remote repository on Bitbucket.**

1. Use bitbucket.org/product/coursera to get a free account on Bitbucket Cloud. Bitbucket is an online Git hosting provider.
2. (If necessary) Sign up for an account. Bitbucket is free for teams of up to 5 people. This includes private repositories.
3. Log into Bitbucket.
4. Click the plus icon and select **Repository**.
5. For Project, select the **existing project** or create a new one. Projects are a way to categorize or group repositories in Bitbucket.
6. For Repository name, enter **projectb**.
7. For Include a README?, select **No**.
8. Click **Create repository**.

> *Congratulations. You have created a remote repository.*

### Lab: (Sourcetree) Push to a Remote Repository

Estimated time: 10 minutes

> *Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.*

> *Note: This lab assumes that you have created a remote repository named `projectb`. This was done in the previous lab.*

In this lab, you will:

1. Clone your Bitbucket repository.
2. Create a commit in the local repository.
3. Push the commit to the remote repository.

**1: Clone your Bitbucket repository.**

1. Log into bitbucket.org.
2. Navigate to the `projectb` repository. Click the plus sign on the left and select **Clone this repository**.
3. **(Option 1)** Click **Clone in Sourcetree**. Sourcetree will open with a "Clone new" window. Accept the default information and click **Clone**.

   **(Option 2)** Copy the repository's URL from Bitbucket, then open a new tab in Sourcetree and select **Clone**. Accept the default values.
4. You should now see that you have a local repository named `projectb`. This is a clone of your remote repository.
5. Under the `Remotes` tab, you should see a remote named `origin` associated with your local repository. This is a shortcut name for your `projectb` repository on Bitbucket.
6. Using your operating system's file manager, view the `projectb` project directory in your `repos` directory.

   > *Congratulations, you have cloned your remote repository.*

**2: Create a commit in the local repository.**

1. Using your method of choice, create a `README.md` file in your `projectb` directory. Add "# projectb's README" as the content of this file. This content will show on the home page for the repository on Bitbucket.
2. In Sourcetree, you should see your `README.md` file as untracked. Click to refresh the window if you do not see it. **Stage** the file.
3. Click **Commit** to begin the process of committing the file. Enter "add README.md" as your commit message. Click **Commit**.

   > *Congratulations, you have created a commit in the local repository.*

**3: Push the commit to the remote repository.**

1. Click **Push** to begin the process of pushing your commit to the remote repository.
2. Select the **master** branch and ensure that the `track` checkbox is checked. Because we have not created any branches, our initial commit was on the default branch named `master`. The `tracked` checkbox

associates the master branches on the local and remote repositories.

3. Click **Push** or **Ok** to push the commit to Bitbucket.
4. In your browser, navigate to Bitbucket. Navigate to your `projectb` repository. Click on the **Source** tab. You should see your commit. You have synchronized your local and remote repositories.
5. Click the README.md link to view the file that you created.

> *Congratulations. You have cloned and committed to your remote repository.*

### Lab: (Command Line) Push to a Remote Repository

Estimated time: 10 minutes

> *Note: This lab assumes that you are using a command line. If you would prefer to use Sourcetree, there are separate instructions.*

> *Note: This lab assumes that you have created a remote repository named `projectb`. This was done in the previous lab.*

In this lab, you will:

1. Clone your Bitbucket repository.
2. Create a commit in the local repository.
3. Push the commit to the remote repository.

**1: Clone your Bitbucket repository.**

1. Log into bitbucket.org.
2. Navigate to the `projectb` repository. Click the plus sign on the left and select **Clone this repository**.
3. Copy the `git clone` command to your clipboard.
4. In a command line window, navigate to your `repos` directory.
5. **Paste** the `git clone` command that you copied. Execute the command. You have cloned an empty repository.
6. Change to the `projectb` directory using `cd projectb`.
7. View the contents of `projectb` using `ls -a`. You should see an empty working tree and a `.git` directory.
8. Execute `git remote -v`. You should see the URL to your remote repository, and see that the shortcut name of this remote repository is `origin`.

> *Congratulations, you have cloned your remote repository.*

**2: Create a commit in the local repository.**

1. Execute `echo "# projectb's README" > README.md` in your projectb directory. This creates a README file in your working tree. This content will show on the home page for the repository on Bitbucket.
2. Execute `git add README.md` to add the README file to the staging area.
3. Execute `git commit -m "add README.md"` to commit to the local repository.

> *Congratulations, you have created a commit in the local repository.*

**3: Push the commit to the remote repository.**

1. Execute `git push -u origin master`. The `-u` flag sets up the local and remote branches as tracking branches. `origin` is a shortcut name for the remote repository. `master` is the branch to push. We are pushing the default branch named `master`.
2. Execute `git status` and verify that your branch is up to date and there is nothing to commit.

3. In your browser, navigate to Bitbucket. Navigate to your `projectb` repository. Click on the **Source** tab. You should see your commit. You have synchronized your local and remote repositories.
4. Click the README.md link to view the file that you created.

> *Congratulations. You have cloned and committed to your remote repository.*

# Lab: (Sourcetree) Installation and Getting Started

Estimated time: 10 minutes

> Note: This lab assumes that you want to use the Sourcetree graphical application for this course. If you would prefer to use a command line interface, there are separate instructions.
>
> Note: Using Sourcetree requires creating a free Atlassian account. This account is also necessary in later labs when using Bitbucket to host your remote repositories. If you would prefer not to create an Atlassian account, you can use the command line version of this course and skip the labs related to Bitbucket.

In this lab, you will:

1. Install Sourcetree (if necessary).
2. Configure your name and email address.
3. Create a folder for your local repositories.

## 1: Install Sourcetree (if necessary).

1. If you have not installed Sourcetree, please download and install it for free from www.sourcetreeapp.com/coursera. If you do not have an Atlassian account, you will need to create one (see note above).

   > Congratulations, you have installed Sourcetree.

## 2: Configure your name and email address.

> Note: Your name and email address will be included with every commit (save) you make to a repository. This becomes part of the project history. It is important that this information is correct.

1. Open Sourcetree preferences.

   - (Mac users) You can do this by selecting **Sourcetree > Preferences…** or by clicking on the **gear icon** on the main Sourcetree page and selecting **Accounts…**.
   - (Windows users) **Tools > Options**.

   > Note: The Mac and Windows versions of Sourcetree have different user interfaces. Differences are identified in the labs when necessary. The videos show the Mac version of Sourcetree, but in general the same concepts apply to Windows users.

2. Select the **General** tab.

3. Verify that your name and email address are correct.

> Congratulations, you have configured your name and email address.

## 3: Create a folder for your local repositories.

> Note: It is recommended that you keep all of your local Git repositories in a single folder/directory on your computer. This keeps things nicely organized. A suggested location is ~/repos, where ~ represents your user folder. Another option is to place a repos directory in your documents folder.

1. Using your operating system's file manager, create a /repos folder. You can create this in your user folder (~), in your documents directory or anywhere that you would like.

2. Configure Sourcetree to use the /repos folder as your default location for local repositories. You do this in **Preferences/Options** below the location where you just specified your name and email address. Under **project folder**, browse to your /repos folder.

> Congratulations, you have created a /repos folder and configured Sourcetree to use it as the default location for local repositories.

> Congratulations. You have installed Sourcetree, configured your name and email address, and configured your default folder for local Git repositories. You have completed this lab.

# Lab: (Command Line) Installation and Getting Started

Estimated time: 10 minutes

> Note: This lab assumes that you want to use a Command Line Interface (CLI) for this course. If you would prefer to use the Sourcetree graphical client, there are separate instructions.
>
> Note: If you are a Windows user, the command line instructions assume you are using **Git bash**.

In this lab, you will:

1. Install the Git command line interface (if necessary).
2. Verify your Git version.
3. Explore Git help.
4. Configure your user name, email address and default Git editor.

## 1: Install the Git command line interface (if necessary).

1. If you have not installed Git, please use these instructions. https://www.atlassian.com/git/tutorials/install-git.

> Congratulations, you have installed the Git CLI.

## 2: Verify your Git version.

1. Open a command line and verify that Git is installed.

```
git --version
```

2. If version information is not returned, install Git using task 1 of this lab.

> Congratulations, you have verified your Git version.

## 3: Explore Git help.

1. View overall Git help by typing `git` at the command line.

2. Choose any Git command that you see and view the help on it using `git help`

`<command>`. For example, `git help init`.

3. View concise help on a command using the `-h` option. For example, `git init -h`.

> Congratulations, you have explored Git help.

## 4: Configure your user name, email address and default Git editor.

1. View your current setting for your user name with `git config user.name`.

2. If you would like to change your user name, use `git config --global user.name "Your Name"`.

3. View your current setting for your email address with `git config user.email`.

4. If you would like to change your email address, use `git config --global user.email "your@email"`.

5. View your current setting for default Git editor with `git config core.editor`.

6. If you would like to change your default Git editor, use `git config --global core.editor your_preferred_editor`. nano seems to be a good editor for people who
have not used vi or vim. Other options are Notepad++ (Windows) or Atom. Make sure that you install your editor before configuring it here.

> Congratulations, you have configured your user name, email address and default editor.

> Congratulations. You have completed this lab.

# Lab: (Sourcetree) Create a Local Repository

Estimated time: 5 minutes

> Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.

In this lab, you will:

1. Create a local repository using Sourcetree.

## 1: Create a local repository using Sourcetree.

1. **(Mac)** From the main Sourcetree window, select **New… > Create Local Repository**. **(Windows)** From a new tab, click the **Create** icon.

2. **(Mac)** In the create a local repository window, you should see a Destination Path section. Your /repos folder should be shown. You configured this default location in the previous lab. **Enter** `projecta` at the end of the Destination path. This will be your project directory. You should see that the `Name` textbox also fills in with `projecta`.

   **(Windows)** In the create a repository window, click **Browse**. You should see your \repos folder. You set this preference in the previous lab. Click **Select Folder** to keep the \repos folder as the project's parent folder. In Destination Path:, append `\projecta` to the repos folder. You should see that the `Name` textbox also fills in with `projecta`.

3. Click **Create** to create the local repository named `projecta`.

4. **(Mac)** In the main Sourcetree window, you should now see your `projecta` repository. **(Windows)** You should see your `projecta` repository in its own tab.

5. Using your operating system's file manager, verify that you can see a `projecta` directory inside of your `repos` directory. This is the project directory for `projecta`. If you have made hidden directories visible, you should also see a `.git` directory inside of the `projecta` directory.

> Congratulations. You have used Sourcetree to create a local repository.

# Lab: (Sourcetree) References

Estimated time: 10 minutes

> Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.
>
> Note: This lab assumes that you have created a local and remote repository named `projectb`. This was done in the previous two labs. If you do not have these, either create them now or go through the previous labs first.

In this lab, you will:

1. Create, push and view references of a commit.
2. Tag the commit.

## 1: Create, push and view references of a commit.

1. In your `projectb` working tree, create a file named `fileA.txt` containing the simple string "feature 1" (without the quotes).

2. Using Sourcetree, add `fileA.txt` to the staging area.

3. **Commit** this file to the local repository. Specify a commit message of "add feature 1". Notice that you have the option of pushing this commit to the remote repository by checking the "Push changes to origin/master" checkbox. You can check this if you want to, but often you might want to push the commit later, because in general you don't want to share commits unless you are sure that you won't want to change them.

4. If you did not select the "Push changes to origin/master" checkbox previously, **push** the commit to the remote repository.

5. View the commit on Bitbucket.

6. In Sourcetree, view the commit history. Notice that there are labels/references associated with the commit(s). Labels starting with "origin" are related to the remote repository. We will discuss those more later.

7. Click on the most recent commit in the history. View the commit details.

8. View the SHA-1 of the commit in the commit details. Notice that it is a 40 character hexadecimal string. View the same commit next to the commit graph. Notice that only the

first seven characters of the commit's SHA-1 are listed.

> Congratulations, you have created, pushed and viewed the references of a commit.

## 2: Tag the commit.

1. Let's say that we want to "permanently" attach a version label to the commit that implements feature 1. In Sourcetree, right-click on the latest commit and select **Tag…**. In the window that opens, your latest commit should be selected. Because we want to tag the latest commit, you could also select the "Working copy parent" option, since that represents the latest commit too.

2. Name the tag **v0.1**. You can select the "Push tag" checkbox if you want. This will push the tag to the remote repository when the tag is created. You also can push the tag later as a separate step.

3. Click **Add** to tag the commit. You should now see the v0.1 tag on the commit.

4. If you did not select the "Push tag" checkbox when creating the tag, push the tag now by right-clicking on your tag under the TAGS tab and selecting **Push to > origin**.

5. View the commit on Bitbucket. Notice that it has the v0.1 tag associated with it.

6. You will not use the `projectb` repository in future labs. Feel free to delete it using the procedure below. You can delete the remote and local repositories.
   **(Bitbucket)** Delete the remote repository in Bitbucket by selecting `Settings`, `Delete Repository` and `Delete`.
   **(Sourcetree- Mac)** In the main Sourcetree window, right-click on the repository name and select `Delete`.
   **(Sourcetree- Windows)** In a new tab, right-click on the repository name and select `Delete`. Then select `Delete Repositories on Disk`. This removes the repository from Sourcetree and deletes the project folder.

> Congratulations, you have created a tag and pushed it to the remote repository.

# Lab: (command line) References

Estimated time: 10 minutes

> Note: This lab assumes that you are using the command line. If you would prefer to use Sourcetree, there are separate instructions.
>
> Note: This lab assumes that you have created a local and remote repository named `projectb`. This was done in the previous two labs. If you do not have these, either create them now or go through the previous labs first.

In this lab, you will:

1. Create, push and view references of a commit.
2. Tag the commit.

## 1: Create, push and view references of a commit.

1. In your `projectb` working tree, create a file named `fileA.txt` containing the simple string "feature 1" (without the quotes).

   ```
   repos$ cd projectb
   projectb$ echo "feature 1" > fileA.txt
   ```

2. Add `fileA.txt` to the staging area. (As a reminder, you can do this with the `git add` command.)

3. **Commit** this file to the local repository. Specify a commit message of "add feature 1". (As a reminder, you use the `git commit` command with the -m option.)

4. **Push** the commit to the remote repository.

   ```
   projectb$ git push origin master
   ```

5. View the commit on Bitbucket.

6. At the command line, use `git log --oneline` to view your local repository's commit history. Notice that the SHA-1 values of the commit objects have been shortened. Also notice that there are labels/references associated with the commit(s). Labels starting with "origin" are related to the remote repository. We will discuss those more later.

**Question 1** What local branch are you on? (answers at end of file)

**Question 2** What does `HEAD -> master` mean?

7. Execute the `git show` command, adding an argument of the latest commit. This will show the contents of the latest commit object, as well as some information on what has changed (we will discuss "diff" later). All of these commands should produce the same result:

```
$ git show (first four characters of SHA-1)
$ git show (complete SHA-1)
$ git show HEAD
$ git show master
```

8. (If you have multiple commits) Execute `git show HEAD~` to view the details of the parent of the latest commit. Execute `git show HEAD^` to see the same details. As we learned in the video, ~ and ^ coincidently show the same results here, but in general they refer to different things. For example, ~2 and ^2 have different meanings. ~2, ~~ or ^^ refer to the grandparent. ^2 refers to the second parent in a merge commit.

> Congratulations, you have created, pushed and viewed the references of a commit.

## 2: Tag the commit.

1. Execute `git tag` to view the tags in your local repository. There should be no tags.

2. Let's say that you want to "permanently" attach a version label to the commit that implements feature 1. Use the `git tag` command for this. Create an annotated tag by specifying the -a option. Use the -m option to specify a tag message of "includes feature 1". Since the most recent commit implements feature 1, you do not need to specify a commit, because the command defaults to HEAD. Name the tag "v0.1".

```
$ git tag -a -m "includes feature 1" v0.1
```

3. Execute `git tag` again to view the tags. You should see your "v0.1" tag.

4. Execute `git show v0.1`. You should see your tag information, as well as information on its associated commit object.

5. Push the tag to the remote repository.

```
$ git push origin v0.1
```

6. View the commit on Bitbucket. Notice that it has the v0.1 tag associated with it.

7. (If you have multiple commits) Tag the parent of the current commit with a "temp" tag. You can use the `HEAD~` reference to refer to this commit.

   ```
   $ git tag -a -m "just a temp tag" temp HEAD~
   ```

   Delete this tag using the -d option.

   ```
   $ git tag -d temp
   ```

8. You will not use the `projectb` repository in future labs. You can delete the remote and local repositories. Delete the local repository by deleting the `projectb` directory. Delete the remote repository in Bitbucket by selecting `Settings`, `Delete Repository` and `Delete`.

> Congratulations, you have created a tag and pushed it to the remote repository.

**Answers to questions**

**Question 1** What local branch are you on? `master`

**Question 2** What does `HEAD -> master` mean? `HEAD` is a reference to the current commit. The content of that commit is in your working tree. `master` is the name of the branch, and this commit is the tip of the branch, so this is the `master` branch label. The arrow represents that the `HEAD` reference points to the `master` branch label, which points to the SHA-1 of the commit.

# Lab: (Sourcetree) Branches

Estimated time: 20 minutes

> Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.

In this lab, you will:

1. Create and checkout a branch.
2. Create commits on the branch.
3. Checkout an old commit.
4. Delete a branch.

## 1: Create and checkout a branch.

1. Using Sourcetree, create a local repository named `projectc`. If you need help, please refer to the previous labs.

2. Create a commit with a `fileA.txt` file containing a string "feature 1". The commit message should be "add feature 1". This commit should be made on the `master` branch.

3. **(Mac)** Create a branch off of the latest master commit named "featureX". In Sourcetree's History tab, make sure that the latest commit is highlighted and its details show that it is the current commit ( `HEAD -> master`). Its commit message should say "add feature 1". **Click** the Branch icon to start creating the branch.

   **(Windows)** Create a branch off of the latest master commit named "featureX". Select Sourcetree's Log/History tab (at the bottom) and make sure that the latest commit has a hollow circle next to it. This means that this is the currently checked out commit. Its commit message should say "add feature 1". **Click** the Branch icon to start creating the branch.

4. In the window that appears, make sure that **New Branch** is selected at the top. **Name** the branch `featureX`. Under Commit, "Working copy parent" should be selected. This means that you will be branching off of the current commit. Make sure the "Checkout new branch" checkbox is **selected**. **Click Create Branch** to create the `featureX` branch.

5. Under the BRANCHES tab on the left, you should now see your `featureX` branch. There is a circle next to the branch name, indicating that it is checked out. This means that working tree contains the files from this version of the project, and the next commit that you make will be to this branch.

6. Notice that the latest commit now has both the `master` and the `featureX` branch labels.

   **(Mac)** In the commit details, you should see `HEAD -> featureX`, indicating that the next commit you make will be to the `featureX` branch. (Note, you might need to click on a different commit and click back to refresh the details.)

   > Congratulations, you have created and checked out a branch.

## 2: Create commits on the branch.

1. Now that you have created and checked out the `featureX` branch, you can do some work on the project without affecting the `master` branch. In your local repository, **create a commit** on the `featureX` branch with the following:

   - modify `fileA.txt`, adding "feature mistake" directly under the line "feature 1"
   - add a commit message of "add feature mistake"
     (As a reminder, you can use any tool that you want to modify the `fileA.txt` file. You can usually double-click on the file in Sourcetree to edit it. Once you have edited and saved `fileA.txt`, click on the "File status" tab. `fileA.txt` should show as unstaged. Stage the file. Click Commit (if necessary) and add the message "add feature mistake".)

2. Click on the History (or Log/History for Windows users) tab and view your commit graph. You should see a straight line, with your `featureX` branch label and "add feature mistake" commit message on the most recent commit. You should see that the `featureX` branch is checked out (under BRANCHES).

3. Under BRANCHES, double-click the `master` branch to check it out. Your working tree will be updated with the older version of `fileA.txt`. View the contents of that file and verify that you do not see your "feature mistake" content. The `master` branch is unaware of the work that you did on the `featureX` branch. Notice in the commit graph that the second commit is the current commit, as indicated by the hollow circle. If you changed the working tree and committed right now, the commit would be to the `master` branch.

4. **Change back** to the `featureX` branch by checking it out (double-click under BRANCHES).

5. **Create another commit** on the `featureX` branch with the following:

   - in `fileA.txt`, under "feature 1", change the line "feature mistake" to "feature bigger mistake"
   - add a commit message of "add feature bigger mistake"

6. Click on the History (or Log/History) tab and view your commit graph. You should again see a straight line, with two commits on the `featureX` branch.

> Congratulations, you have created commits on the `featureX` branch.

## 3: Checkout an old commit.

1. Let's say you want to view the first change that we made on the `featureX` branch. **Checkout** the first commit you made on the `featureX` branch ("add feature mistake"). Do this by double-clicking on the commit in the commit graph. You will be shown a confirmation dialog about entering a detached HEAD state. This means that there is no branch label associated with this commit, and that your HEAD reference will point directly to the commit. **Click OK** to checkout the commit.

2. Verify that you are seeing the older version of `fileA.txt` ("feature mistake") in your working tree. Also notice that the current commit has a HEAD tag with no branch label. You are in a detached HEAD state. We are only viewing the old commit, so we are OK. If we wanted to create new commits based on this commit, we should create a branch right now. We don't need to do that though.

3. **Checkout** the `master` branch to get out of the detached HEAD state.

> Congratulations, you have checked out an old commit.

## 4: Delete a branch.

> Well, our great featureX idea might not have been so great after all. We went from "big mistake" to "bigger mistake". We will delete the `featureX` branch without merging it into `master` (you will learn about merging later).

1. Under BRANCHES, right-click on the `featureX` branch and select **Delete > featureX**. A confirmation dialog appears. Try deleting the branch now without selecting Force delete- **click OK**. Sourcetree opens a dialog, showing the response from Git as if you were on a command line. You will see that Git won't let you delete this branch, because it has not been merged. Your two commits on the `featureX` branch would become "dangling commits" and would eventually be garbage-collected by Git. In the Git message, it says that if you are sure that you want to delete the branch, use the `-D` option with the `git branch` command. You don't have to do that in Sourcetree, as we will see. Click **Close** to dismiss the Git message.

2. **Delete** the `featureX` branch again, but this time select the **Force delete** checkbox. Behind the scenes, Sourcetree uses the `-D` option with the `git branch`, as suggested

by Git. The `featureX` branch is deleted.

3. View the commit graph and verify that you are back to having only a `master` branch.

4. **(Mac)** (If you are interested) Want to "undo" the deleting of the `featureX` branch? In Sourcetree, select View > Command History. Expand the entry for Deleting branch…. You should see something like "Deleted branch featureX (was 345263e)". That SHA-1 is the commit ID of the commit that the `featureX` branch label pointed to. **Copy** that SHA-1. Click on **Branch**, select **New Branch**. Name the branch `featureX` and paste the SHA-1 into the textbox related to "Specified commit". You should now see your `featureX` branch again! Git doesn't really delete commits right away. Another way to obtain that commit's SHA-1 is to use the `git reflog` command at the command line. Please delete the `featureX` branch again.

   **(Windows)** (If you are interested) Want to "undo" the deleting of the `featureX` branch? Git doesn't really delete commits right away. You can obtain the `featureX` branch tip's SHA-1 using the `git reflog` command at the command line. You could then checkout that commit and create another `featureX` branch.

> Congratulations, you have deleted a branch and completed this lab.

# Lab: (Command Line) Branches

Estimated time: 20 minutes

> Note: This lab assumes that you are using a command line. If you would prefer to use Sourcetree, there are separate instructions.

In this lab, you will:

1. Create and checkout a branch.
2. Create commits on the branch.
3. Checkout an old commit.
4. Delete a branch.

## 1: Create and checkout a branch.

1. Create a local repository named `projectc`. If you need help, please refer to the previous labs.

2. Create a commit with a `fileA.txt` file containing a string "feature 1". The commit message should be "add feature 1". This commit should be made on the `master` branch.

3. Use `git branch` to verify that you have a single branch in your local repository, and its name is `master`. Use `git log --oneline --graph` to verify that you are currently on the most recent commit. You should see `HEAD -> master` on the most recent commit.

4. Create and checkout a branch off of the latest master commit named "featureX". You can do this with two command or one command:

```
# two command approach
$ git branch featureX
$ git checkout featureX

# one command approach
$ git checkout -b featureX
```

5. Execute `git branch` to verify that you have created a `featureX` branch, and that it is the currently checked out branch. Execute `git log --oneline --graph` to verify that the `featureX` branch is the current branch- you should see `HEAD -> featureX`. Notice that the latest commit now has both the `master` and the `featureX` branch labels. Because `featureX` is the current branch, the next commit you make will be to this branch.

> Congratulations, you have created and checked out a branch.

## 2: Create commits on the branch.

1. Now that you have created and checked out the `featureX` branch, you can do some work on the project without affecting the `master` branch. In your local repository, **create a commit** on the `featureX` branch with the following:

    - modify `fileA.txt`, adding "feature mistake" directly under the line "feature 1"
    - add a commit message of "add feature mistake"
      (If you need a refresher on how to use `git add` and `git commit` to create a commit, see the previous labs.)

2. Execute `git log --oneline --graph` and view your commit graph (the asterisks). You should see a straight line, with your `featureX` branch label and "add feature mistake" commit message on the most recent commit. You should see that the `featureX` branch is checked out ( `HEAD -> featureX` ).

3. Execute `git checkout master` to checkout the master branch. Your working tree will be updated with the older version of `fileA.txt`. View the contents of that file and verify that you do not see your "feature mistake" content. The master branch is unaware of the work that you did on the `featureX` branch.

4. Execute `git log --oneline --graph`. Notice that only information about the current branch is listed. Also notice that the current branch is the `master` branch `HEAD -> master`. If you changed the working tree and committed right now, the commit would be to the `master` branch.

5. Execute `git log --oneline --graph --all`. Add `--all` shows all of the local branches. Now you can see your `featureX` branch, and the `featureX` branch has a commit more current than the commit at the tip of the `master` branch.

6. **Change back** to the `featureX` branch by checking it out.

7. **Create another commit** on the `featureX` branch with the following:

    - modify `fileA.txt`, under "feature 1", change the line "feature mistake" to "feature bigger mistake"
    - add a commit message of "add feature bigger mistake"

8. Execute `git log --oneline --graph --all` and view your commit graph. You should again see a straight line, with two commits on the `featureX` branch.

> Congratulations, you have created commits on the `featureX` branch.

## 3: Checkout an old commit.

1. Let's say you want to view the first change that we made on the `featureX` branch. **Checkout** the first commit you made on the `featureX` branch ("add feature mistake"). Do this by executing `git checkout HEAD~`. The appended ~ means "parent of the commit". Git will warn you that you are entering a detached HEAD state. This is because your HEAD reference points directly at the SHA-1 of a commit, instead of to a branch label. Read Git's message, it is informative.

2. Verify that you are seeing the older version of `fileA.txt` ("feature mistake") in your working tree. Execute `git log --oneline --graph --all` and notice that the current commit has a HEAD tag with no branch label. You are in a detached HEAD state. We are only viewing the old commit, so we are OK. If we wanted to create new commits based on this commit, we should create a branch right now. We don't need to do that though.

3. **Checkout** the `master` branch to get out of the detached HEAD state.

> Congratulations, you have checked out an old commit.

## 4: Delete a branch.

> Well, our great featureX idea might not have been so great after all. We went from "big mistake" to "bigger mistake". We will delete the `featureX` branch without merging it into `master` (you will learn about merging later).

1. Try to delete the `featureX` branch using `git branch -d featureX`. You will see that Git won't let you delete this branch, because it has not been merged. Your two commits on the `featureX` branch would become "dangling commits" and would eventually be garbage-collected by Git. In the Git message, notice that it says that if you are sure that you want to delete the branch, use the `-D` option with the `git branch` command.

2. **Delete** the `featureX` branch again, but this time use the `-D` option. The `featureX` branch is deleted.

3. View the commit graph and verify that you are back to having only a `master` branch.

4. (If you are interested) Want to "undo" the deleting of the `featureX` branch? Execute `git reflog`. This shows the local history of HEAD references. Since Git doesn't immediately delete commits, you can find the SHA-1 of your most recent `featureX` branch there. **Copy** the SHA-1 of the "add feature bigger mistake" commit. Execute `git checkout -b featureX [SHA-1 YOU COPIED]`. View your commit graph and verify that your `featureX`

branch has returned. **Delete** the `featureX` branch again.

> Congratulations, you have deleted a branch and completed this lab.

# Lab: (Sourcetree) Merging

Estimated time: 10 minutes

> Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.
>
> Note: This lab assumes that you have created a local repository named `projectc`. This was done in the previous lab.

In this lab, you will:

1. Perform a fast-forward merge.
2. Perform a merge with a merge commit.

## 1: Perform a fast-forward merge.

1. In a previous lab, you should have created a commit in your `projectc` repository with a `fileA.txt` file containing a string "feature 1". The commit message should be "add feature 1". This commit should be on the `master` branch. If you do not have this commit, create it now.

2. Create and checkout a branch off of the latest master commit named "feature2". Use the same process that you used in the previous lab.

3. In your local repository, **create a commit** on the `feature2` branch with the following:

    - modify `fileA.txt`, adding "feature 2" directly under the line "feature 1"
    - add a commit message of "add feature 2"

4. View your commit graph. You should see a straight line, with your `feature2` branch label and "add feature 2" commit message on the most recent commit. You should see that the `feature2` branch is checked out (under BRANCHES).

5. Let's assume that feature 2 is ready to be merged into the `master` branch. Start by **checking out** the `master` branch.

6. Click the **Merge** button to start the merge process.

7. Under `Pick a commit to merge into your tree` (Windows: `Pick a commit to merge into your current branch`), select the tip of the `feature2` branch.

8. Read the checkboxes under options. One says `Create a commit even if merge resolved via fast-forward` (Windows: `Create a new commit even if fast-forward is possible`). Leave that checkbox **unchecked**. If it is unchecked, Git will perform a fast-forward merge if possible.

9. Click OK to perform the fast-forward merge. You should now see a linear history with the `master` and `feature2` labels on the most recent commit. The fast-forward merge simply moved the `master` branch label to the latest commit.

10. Delete the `feature2` branch label.

> Congratulations, you have performed a fast-forward merge.

## 2: Perform a merge with a merge commit.

1. Repeat the process above to create a `feature3` branch and commit.

2. This time, when you are ready to merge in the `feature3` branch, check the `Create a commit even if merge resolved via fast-forward` checkbox (Windows: `Create a new commit even if fast-forward is possible`). This will create a merge commit, resulting in a non-linear history. Verify that a merge commit was created.

3. Delete the `feature3` branch label.

4. You will not use the `projectc` repository in future labs. You can delete it.

> Congratulations, you have performed a merge with a merge commit and completed this lab.

# Lab: (Command Line) Merging

Estimated time: 10 minutes

> Note: This lab assumes that you are using a command line. If you would prefer to use Sourcetree, there are separate instructions.
>
> Note: This lab assumes that you have created a local repository named `projectc`. This was done in the previous lab.

In this lab, you will:

1. Perform a fast-forward merge.
2. Perform a merge with a merge commit.

## 1: Perform a fast-forward merge.

1. In a previous lab, you should have created a commit in your `projectc` repository with a `fileA.txt` file containing a string "feature 1". The commit message should be "add feature 1". This commit should be on the `master` branch. If you do not have this commit, create it now.

2. Create and checkout a branch off of the latest master commit named "feature2". Use the same process that you used in the previous lab.

3. In your local repository, **create a commit** on the `feature2` branch with the following:

    - modify `fileA.txt`, adding "feature 2" directly under the line "feature 1"
    - add a commit message of "add feature 2"

4. Use `git log --oneline --graph --all` view your commit graph. You should see a straight line, with your `feature2` branch label and "add feature 2" commit message on the most recent commit. You should see `HEAD -> feature2`.

5. Let's assume that feature 2 is ready to be merged into the `master` branch. Start by **checking out** the `master` branch.

6. Execute `git merge feature2`. By default, this command will perform a fast-forward merge if possible.

7. You should now see a linear history with the `master` and `feature2` labels on the most recent commit. The fast-forward merge simply moved the `master` branch label to the

latest commit.

8. Delete the `feature2` branch label.

> Congratulations, you have performed a fast-forward merge.

## 2: Perform a merge with a merge commit.

1. Repeat the process above to create a `feature3` branch and commit.

2. This time, when you are ready to merge in the `feature3` branch, execute `git merge --no-ff feature3`. The `--no-ff` option will create a merge commit, resulting in a non-linear history. Verify that a merge commit was created.

3. Delete the `feature3` branch label.

4. You will not use the `projectc` repository in future labs. You can delete it.

> Congratulations, you have performed a merge with a merge commit and completed this lab.

# Lab: (Sourcetree) Resolving Merge Conflicts

Estimated time: 10 minutes

> Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.

In this lab, you will:

1. Create branches that contain a merge conflict.
2. Merge the branches, resolving the merge conflict.

## 1: Create branches that contain a merge conflict.

1. Use Sourcetree to create a local repository named `projectd`.

2. Create a commit in your `projectd` repository with a `fileA.txt` file containing a string "feature 1". The commit message should be "add feature 1". This commit should be on the `master` branch.

3. Create and checkout a branch off of the latest master commit named "feature2".

4. In your local repository, **create a commit** on the `feature2` branch with the following:

    - modify `fileA.txt`, adding "feature 2" directly under the line "feature 1"
    - add a commit message of "add feature 2"

5. Checkout the `master` branch.

6. Create a commit on the `master` branch with the following:

    - modify `fileA.txt`, adding "feature 3" directly under the line "feature 1"
    - add a commit message of "add feature 3"

> Congratulations, you have created branches that contain a merge conflict. The `master` branch and the `feature2` branch have modified the same hunk of `fileA.txt` in different ways.

## 2: Merge the branches, resolving the merge conflict.

1. Verify that the `master` branch is checked out.

2. Click **Merge** and attempt to merge in the `feature2` branch. You should be warned that

there is a merge conflict. Click to dismiss this message.

3. View the uncommitted `fileA.txt` file that Git has placed in your working tree. Notice the conflict markers in the file. That is the part of the merge that Git couldn't automatically resolve.

4. **(Mac)** Rather than fix the conflict right now, abort the merge process by selecting **Reset…** under the **Repository** menu of Sourcetree. Click both **Reset all** buttons to abort the merge.

   **(Windows)** Rather than fix the conflict right now, abort the merge process by selecting **Discard**. Select the **Reset All** tab, then click **Reset All**.

5. Verify that you are back to the state before the merge attempt, with no uncommitted files in the working tree.

6. This time, let's resolve the merge conflict. Click **Merge** again and attempt to merge in the `feature2` branch. Dismiss the merge conflict message.

7. **Edit** `fileA.txt` to resolve the merge conflict. Remove the conflict markers and make sure the file contains three lines of text: "feature 1", "feature 2" and "feature 3".

8. **Add** `fileA.txt` to the staging area so that the fixed version of the file is part of the merge commit.

9. **Commit** the merge. Accept the default merge commit message.

10. **Delete** the `feature2` branch label.

11. Verify that you have a commit graph with a merge commit containing all three features.

12. You will not use the `projectd` repository in future labs. You can delete it.

> Congratulations, you have resolved a merge conflict and completed this lab.

# Lab: (Command Line) Resolving Merge Conflicts

Estimated time: 10 minutes

> Note: This lab assumes that you are using a command line. If you would prefer to use Sourcetree, there are separate instructions.

In this lab, you will:

1. Create branches that contain a merge conflict.
2. Merge the branches, resolving the merge conflict.

## 1: Create branches that contain a merge conflict.

1. Create a local repository named `projectd`.

2. Create a commit in your `projectd` repository with a `fileA.txt` file containing a string "feature 1". The commit message should be "add feature 1". This commit should be on the `master` branch.

3. Create and checkout a branch off of the latest master commit named "feature2".

4. In your local repository, **create a commit** on the `feature2` branch with the following:

    - modify `fileA.txt`, adding "feature 2" directly under the line "feature 1"
    - add a commit message of "add feature 2"

5. Checkout the `master` branch.

6. Create a commit on the `master` branch with the following:

    - modify `fileA.txt`, adding "feature 3" directly under the line "feature 1"
    - add a commit message of "add feature 3"

> Congratulations, you have created branches that contain a merge conflict. The `master` branch and the `feature2` branch have modified the same hunk of `fileA.txt` in different ways.

## 2: Merge the branches, resolving the merge conflict.

1. Verify that the `master` branch is checked out.

2. Execute `git merge feature2` and attempt to merge in the `feature2` branch. You should

see that there is a merge conflict.

3. Execute `git status` to see that Git has modified `fileA.txt`.

4. View the `fileA.txt` file. Notice the conflict markers in the file. That is the part of the merge that Git couldn't automatically resolve.

5. Rather than fix the conflict right now, abort the merge process by executing `git merge --abort`.

6. Verify that you are back to the state before the merge attempt, with no uncommitted files in the working tree.

7. This time, let's resolve the merge conflict. Repeat the merge attempt. You should again see a merge conflict.

8. **Edit** `fileA.txt` to resolve the merge conflict. Remove the conflict markers and make sure the file contains three lines of text: "feature 1", "feature 2" and "feature 3".

9. **Add** `fileA.txt` to the staging area so that the fixed version of the file is part of the merge commit.

10. **Commit** the merge. Accept the default merge commit message.

11. **Delete** the `feature2` branch label.

12. Verify that you have a commit graph with a merge commit containing all three features.

13. You will not use the `projectd` repository in future labs. You can delete it.

> Congratulations, you have resolved a merge conflict and completed this lab.

# Lab: (Sourcetree) Tracking Branches

Estimated time: 10 minutes

> Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.

In this lab, you will:

1. View tracking branches.
2. Create a state with the local branch one ahead of the tracking branch.

## 1: View tracking branches.

1. Use bitbucket.org to create a remote repository named `projecte`. You can do this by clicking on the plus sign at the left and then `Create a new > Repository`.

2. Use Bitbucket to create the first commit.

    - Click **Create a README**.
    - Modify the text to contain only the line `# PROJECTE README #`.
    - Click **Commit**. Change the commit message to `add README.md`. Click Commit to create the commit.

    > We created the first commit so that we don't clone an empty repository. When you clone, a tracking branch is set up automatically if you have at least one commit.

3. Clone the `projecte` repository.

    - In Bitbucket, select the + sign on the left. Click `Clone this repository`.

    **(Option 1)** Click **Clone in Sourcetree**. Sourcetree will open with a "Clone new" window. Accept the default information and click **Clone**.

    **(Option 2)** Copy the repository's URL from Bitbucket, then open a new tab in Sourcetree and select **Clone**. Accept the default values.

4. View the tracking branches.

    - In Sourcetree, click on the `History` tab (Windows: `Log/History`). You should see the `add README.md` commit that you just made in Bitbucket.
    - View the branch labels on the commit. Please answer these questions: (answers at end of lab below)

QUESTION A. What does the `master` label represent?

QUESTION B. What does the `origin/master` label represent?

QUESTION C. What does the `origin/HEAD` label represent?

- On the `BRANCHES` tab, notice that only the local branch `master` is represented.
- Expand the `REMOTES > origin` tabs. You should see `HEAD` and `master`. This `master` branch is the tracking branch.

> Congratulations, you have viewed tracking branches.

## 2: Create a state with the local branch one ahead of the tracking branch.

QUESTION D. How do you create a state with the local branch one ahead of the tracking branch?

1. Modify your local README.md. Append the line "Fun with tracking branches." to the file. Use Sourcetree to **add and commit** this file with the commit message "add fun line to README.md". Make sure that the `Push changes immediately to origin/master` checkbox is **not** selected.

2. View your commit history. You should see that the `master` branch label is on the latest commit, but the `origin/master` and `origin/HEAD` labels have stayed on the original commit. You should also see a `1 ahead` message (or 1 with an up arrow) on the current commit.

3. Notice that the `Push` button has a `1` message. This means that you have made 1 commit locally that is not on the remote repository. **Click the Push button** to push the commit to the remote repository. You should now see the three branch labels on the latest commit. The local and remote repositories are again synchronized.

4. Verify that the second commit has been pushed to Bitbucket.

5. You will not use the `projecte` repository in future labs. You can delete it (locally and on Bitbucket).

> Congratulations, you have created a state with the local branch one ahead of the tracking branch. You have also recovered from this state by pushing the commit to the remote repository.

**ANSWERS TO QUESTIONS**

A) The `master` label represents the tip of the local `master` branch.

B) The `origin/master` label represents the tip of the tracking branch that tracks the `master` branch on the remote repository.

C) The `origin/HEAD` label represents the tip of the default branch on the remote repository. The default branch on the remote repository is the `master` branch.

D) If you make a commit to the local repository without pushing it to the remote repository, the local branch becomes ahead of the tracking branch. The tracking branch only knows what the remote repository knows.

# Lab: (Command Line) Tracking Branches

Estimated time: 10 minutes

> Note: This lab assumes that you are using a command line. If you would prefer to use Sourcetree, there are separate instructions.

In this lab, you will:

1. View tracking branches.
2. Create a state with the local branch one ahead of the tracking branch.

## 1: View tracking branches.

1. Use bitbucket.org to create a remote repository named `projecte`. You can do this by clicking on the plus sign at the left and then `Create a new > Repository`.

2. Use Bitbucket to create the first commit.

    - Click **Create a README**.
    - Modify the text to contain only the line `# PROJECTE README #`.
    - Click **Commit**. Change the commit message to `add README.md`. Click Commit to create the commit.

    > We created the first commit so that we don't clone an empty repository. When you clone, a tracking branch is set up automatically if you have at least one commit.

3. Clone the `projecte` repository.

    - In Bitbucket, select the + sign on the left. Click `Clone this repository`.
    - Paste the `git clone` command into your command line and change to the project directory.

4. Use `git branch --all` to view the local and tracking branch names.

5. Use `git log --all --oneline --graph` to view the combined log of all local and tracking branches.

    - View the branch labels on the commit. Please answer these questions: (answers at end of lab below)

    QUESTION A. What does the `master` label represent?

QUESTION B. What does the `origin/master` label represent?

QUESTION C. What does the `origin/HEAD` label represent?

> Congratulations, you have viewed tracking branches.

## 2: Create a state with the local branch one ahead of the tracking branch.

QUESTION D. How do you create a state with the local branch one ahead of the tracking branch?

1. Modify your local README.md. Append the line "Fun with tracking branches." to the file. Add and commit the file with the commit message of "add fun line to README.md".

2. View the commit graph, again using the `--all` option. You should see that the `master` branch label is on the latest commit, but the `origin/master` and `origin/HEAD` labels have stayed on the original commit.

3. Execute `git status`. Notice the message say your branch is ahead of the tracking branch by one commit. This means that you have made one commit locally that is not on the remote repository.

4. Execute `git push` to push the commit to the remote repository. Execute the `git log` and `git status` commands again. You should now see the three branch labels on the latest commit. The local and remote repositories are again synchronized.

5. You will not use the `projecte` repository in future labs. You can delete it.

> Congratulations, you have created a state with the local branch one ahead of the tracking branch. You have also recovered from this state by pushing the commit to the remote repository.

**ANSWERS TO QUESTIONS**
A) The `master` label represents the tip of the local `master` branch.

B) The `origin/master` label represents the tip of the tracking branch that tracks the `master` branch on the remote repository.

C) The `origin/HEAD` label represents the tip of the default branch on the remote repository. The default branch on the remote repository is the `master` branch.

D) If you make a commit to the local repository without pushing it to the remote repository, the

local branch becomes ahead of the tracking branch. The tracking branch only knows what the remote repository knows.

# Lab: (Sourcetree) Fetch, Pull and Push

Estimated time: 10 minutes

> Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.

In this lab, you will:

1. Fetch the latest commits from the remote repository.
2. Execute a pull with a fast-forward merge.
3. Execute a pull with a merge commit.
4. Push commits to the remote repository.

## 1: Fetch the latest commits from the remote repository.

1. Use Bitbucket (bitbucket.org) to create a remote repository named `projectf`.

2. Use Bitbucket to create the first commit.

    - Click **Create a README**.
    - Modify the text to contain only the line `# PROJECTF README #`.
    - Click **Commit**. Change the commit message to `add README.md`. Click Commit to create the commit.

3. Clone the `projectf` repository.

4. Using Bitbucket, modify and commit the README.md file.

    - Click on the **Source** tab.
    - Click on README.md.
    - Click Edit.
    - Append the line "Fun with network commands.".
    - Click Commit and specify a commit message "append line to README.md".

5. Because you have created a commit on the remote repository after cloning the repository, your local `master` branch is behind. View your repository in Sourcetree. At this point, it may or may not be aware of the new commit on the remote repository. This depends on your Sourcetree settings and on timing.

6. Click on the Settings icon. Click on the Advanced tab. If the `Refresh remote status in the background` checkbox is checked, Sourcetree will periodically check for remote

updates and fetch them.

7. View your commit graph. If Sourcetree has done a fetch, you should see your second commit, and that the tracking branch is ahead of your `master` branch. It is safe to click `Fetch` at any time. It will have no impact if Sourcetree has already fetched. Otherwise, it will retrieve the latest remote objects. Click **Fetch** if you don't see the second commit.

> Congratulations, you have fetched the latest commits from the remote repository.

## 2: Execute a pull with a fast-forward merge.

1. Because we have not added any commits to our local `master` branch, we can perform a pull with a fast-forward merge. Click on **Pull** to begin the merge process.

2. Pull from the remote's `master` branch into your local `master` branch. Notice that the `Create new commit even if fast-forward...` checkbox is **unchecked**. Because this merge is fast-forwardable, no merge commit will be created.

3. Click OK to perform the fast-forward merge. Your `master` branch label should move to the latest commit. The `master` branch is synchronized.

> Congratulations, you have executed a pull with a fast-forward merge.

## 3: Execute a pull with a merge commit.

1. In your local repository, create an empty `fileA.txt` file. Add and commit the file, specifying a commit message of "add fileA.txt". **Do not Push the commit**.

2. In Bitbucket, make a minor edit to the `README.md` file. **Commit the change**.

3. In Sourcetree, click `Fetch` to retrieve the latest remote commit. You should now see that your local `master` branch is 1 ahead and 1 behind the tracking branch. This is because you made commits locally and in the remote repository.

4. Notice that the Pull and Push icons have a number 1 associated with them. **Attempt to Push**. You will receive a message saying that the updates were rejected, because the tip of your current branch is behind the tracking branch. The message suggests to do a pull.

5. Click `Pull` to start the local merge process.

6. Click OK to perform the merge. Notice that a merge commit was created, combining the work of your local commit and the commit that you made on the remote repository. Also notice that the tracking branch is now two commits behind. At this point, the remote

repository doesn't know about your "add fileA.txt" commit or about the local merge commit.

> Congratulations, you have executed a pull with a merge commit.

## 4: Push commits to the remote repository.

1. In Sourcetree, click `Push` to add the two local commits to the remote repository. You should now see that the local and tracking branches are synchronized.

2. In Bitbucket, click on `Commits` and verify that the commit graph matches your local commit graph.

3. You will not use the `projectf` repository in future labs. You can delete it.

> Congratulations, you have pushed commits to the remote repository and completed this lab.

# Lab: (Command Line) Fetch, Pull and Push

Estimated time: 10 minutes

> Note: This lab assumes that you are using a command line. If you would prefer to use Sourcetree, there are separate instructions.

In this lab, you will:

1. Fetch the latest commits from the remote repository.
2. Execute a pull with a fast-forward merge.
3. Execute a pull with a merge commit.
4. Push commits to the remote repository.

## 1: Fetch the latest commits from the remote repository.

1. Use Bitbucket (bitbucket.org) to create a remote repository named `projectf`.

2. Use Bitbucket to create the first commit.

    - Click **Create a README**.
    - Modify the text to contain only the line `# PROJECTF README #`.
    - Click **Commit**. Change the commit message to `add README.md`. Click Commit to create the commit.

3. Clone the `projectf` repository to create a local repository.

4. Using Bitbucket, modify and commit the README.md file.

    - Click on the **Source** tab.
    - Click on README.md.
    - Click Edit.
    - Append the line "Fun with network commands.".
    - Click Commit and specify a commit message "append line to README.md".

5. Because you have created a commit on the remote repository after cloning the repository, your local master branch is behind. View your commit history. Git is not aware of the new remote commit, because you have not executed any network commands like `git fetch`.

6. Execute `git fetch`. View your commit graph. You should see your second commit, and that the tracking branch is ahead of your `master` branch.

> Congratulations, you have fetched the latest commits from the remote repository.

## 2: Execute a pull with a fast-forward merge.

1. Because we have not added any commits to our local `master` branch, we can perform a pull with a fast-forward merge. Execute `git pull`. Because this merge is fast-forwardable, no merge commit will be created.

2. View your commit graph. Your 'master' branch label should move to the latest commit. The `master` branch is synchronized.

> Congratulations, you have executed a pull with a fast-forward merge.

## 3: Execute a pull with a merge commit.

1. In your local repository, create an empty `fileA.txt` file. Add and commit the file, specifying a commit message of "add fileA.txt". **Do not Push the commit**.

2. In Bitbucket, make a minor edit to the README.md file. **Commit the change**.

3. In the command line, execute `git fetch`. You should now see that your local `master` branch is 1 ahead and 1 behind the tracking branch. This is because you made commits locally and in the remote repository.

4. Attempt a `git push`. You will receive a message saying that the updates were rejected, because the tip of your current branch is behind the tracking branch. The message suggests to do a pull.

5. Execute `git pull`. Notice that a merge commit was created, combining the work of your local commit and the commit that you made on the remote repository. Also notice that the tracking branch is now two commits behind. At this point, the remote repository doesn't know about your fileA.txt commit or about the local merge commit.

> Congratulations, you have executed a pull with a merge commit.

## 4: Push commits to the remote repository.

1. Execute `git push` to add the two local commits to the remote repository. You should now see that the local and tracking branches are synchronized.

2. In Bitbucket, click on Commits and verify that the commit graph matches your local commit graph.

3. You will not use the `projectf` repository in future labs. You can delete it.

> Congratulations, you have pushed commits to the remote repository and completed this lab.

# Lab: (Sourcetree) Rebasing

Estimated time: 10 minutes

> Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.

In this lab, you will:

1. Rebase a topic branch.
2. Rebase and resolve a merge conflict.

## 1: Rebase a topic branch.

1. Use Sourcetree to create a local repository named `projectg`.
2. Create an initial commit that adds an empty `fileA.txt` file. The commit message should be "add fileA.txt".
3. Create a `feature1` branch.
4. Create a commit on the `feature1` branch. Add the string "feature 1 wip" to `fileA.txt`. The commit message should be "add feature 1 wip".
5. Checkout the `master` branch. Create a commit that adds an empty `fileB.txt` file. The commit message should be "add fileB.txt".
6. View your commit graph and notice that it is no longer linear.
7. Rebase the commit on the `feature1` branch to the tip of the `master` branch. Do this by first checking out the `feature1` branch. Right-click on the tip of the `master` branch and select `Rebase...`. Click OK to confirm the rebase.
8. View the commit graph. It should now be linear. Verify that you can see `fileB.txt` in your working tree of `feature1`.

> Congratulations, you have rebased a topic branch.

## 2: Rebase and resolve a merge conflict.

1. Checkout the `master` branch.

2. Create a commit on the `master` branch. Add the string "feature 2" to `fileA.txt`. The commit message should be "add feature 2".

3. Rebase the `feature1` branch onto the tip of the `master` branch. This will involve resolving a merge conflict, because the `master` branch and the `feature1` branch both modified the same part of `fileA.txt` in different ways.

    - Checkout the `feature1` branch.
    - Right-click on the tip of the `master` branch and select `Rebase...`.
    - Click OK to confirm the rebase.
    - View and close the merge conflict message.
    - Fix the content of `fileA.txt` so that "feature 1 wip" is before "feature 2".
    - Stage the fixed file.
    - Right-click on the tip of the `feature1` branch and select `Rebase...`.
    - Select `Continue Rebase`.

4. View your commit graph. Verify that you again have a linear graph.

5. You will not use the `projectg` repository in future labs. You can delete it.

> Congratulations, you have performed a rebase involving a merge conflict and have completed this lab.

# Lab: (Command Line) Rebasing

Estimated time: 10 minutes

> Note: This lab assumes that you are using a command line. If you would prefer to use Sourcetree, there are separate instructions.

In this lab, you will:

1. Rebase a topic branch.
2. Rebase and resolve a merge conflict.

## 1: Rebase a topic branch.

1. Create a local repository named `projectg`.

2. Create an initial commit that adds an empty `fileA.txt` file. The commit message should be "add fileA.txt".

3. Create a `feature1` branch.

4. Create a commit on the `feature1` branch. Add the string "feature 1 wip" to `fileA.txt`. The commit message should be "add feature 1 wip".

5. Checkout the `master` branch. Create a commit that adds an empty `fileB.txt` file. The commit message should be "add fileB.txt".

6. View your commit graph and notice that it is no longer linear. Be sure to use the `--all` option with `git log`.

7. Rebase the commit on the `feature1` branch to the tip of the `master branch`. Do this by first checking out the `feature1` branch. Then execute `git rebase master`.

8. View the commit graph. It should now be linear. Verify that you can see `fileB.txt` in your working tree of `feature1`.

> Congratulations, you have rebased a topic branch.

## 2: Rebase and resolve a merge conflict.

1. Checkout the `master` branch.

2. Create a commit on the `master` branch. Add the string "feature 2" to `fileA.txt`. The commit message should be "add feature 2".

3. Rebase the `feature1` branch onto the tip of the `master` branch. This will involve resolving a merge conflict, because the `master` branch and the `feature1` branch both modified the same part of `fileA.txt` in different ways.

    - Checkout the `feature1` branch.
    - Execute `git rebase master`. Notice that there are merge conflicts.
    - Execut `git status` to view the status of the rebase.
    - Fix the content of `fileA.txt` so that "feature 1 wip" is before "feature 2".
    - Stage the fixed file.
    - Execute `git rebase --continue`.

4. View your commit graph. Verify that you again have a linear graph.

5. You will not use the `projectg` repository in future labs. You can delete it.

> Congratulations, you have performed a rebase involving a merge conflict and have completed this lab.

# Lab: (Sourcetree) Rewriting History

Estimated time: 10 minutes

> Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.

In this lab, you will:

1. Amend a commit.
2. Use interactive rebase to remove a commit.

## 1: Amend a commit.

1. Use Sourcetree to create a local repository named `projecth`.

2. Create an initial commit that adds a `fileA.txt` file containing the text "mistake". The commit message should be "add fileA.txt with mistake". Note the first to characters of the commit's SHA-1.

3. Amend the commit so that `fileA.txt` is empty and the commit message simply reads "add fileA.txt".

    - Modify `fileA.txt` so that it is empty.
    - Add `fileA.txt` to the staging area.
    - **(Mac)** Select `Commit`. Under `Commit Options...`, select `Amend last commit`. Specify a commit message of "add fileA.txt". Click `Commit`.
    - **(Windows)** Under `Commit Options...` (lower right corner), select `Amend latest commit`. Specify a commit message of "add fileA.txt". Click `Commit`.

4. Notice the SHA-1 of your amended commit. It should be different than the one you noted prior to amending the commit.

> Congratulations, you have amended a commit.

## 2: Use interactive rebase to remove a commit.

1. Create a `feature1` branch.

2. Create a commit on the `feature1` branch. Add the string "feature 1 wip" to `fileA.txt`. The commit message should be "add feature 1 wip".

3. Create another commit on the `feature1` branch. The contents of `fileA.txt` should be "feature 1". The commit message should be "add feature 1".

4. Let's say that you think that the "feature 1 wip" commit adds little value to the commit history. Use interactive rebase to remove that commit by squashing it.

    - Checkout the `feature1` branch.
    - Right-click on the initial commit in the repository (which is also the first commit of the `feature1` branch) and select `Rebase children of (SHA-1) interactively...`.
    - Explore the interactive rebase form.
    - Remove the commit with the commit message of "feature 1 wip". Do this by selecting somewhere on the **most recent** commit and clicking `Squash with previous`. You should see that two commits will be squashed into one commit.
    - Click `Edit message` and change the commit message to simply read `add feature 1`.
    - Click OK to perform the interactive rebase.

5. View your commit graph. Verify that the commit with a message of "feature 1 wip" has been removed.

6. You will not use the `projecth` repository in future labs. You can delete it.

> Congratulations, you have performed an interactive rebase and have completed this lab.

# Lab: (Command Line) Rewriting History

Estimated time: 10 minutes

> Note: This lab assumes that you are using a command line. If you would prefer to use Sourcetree, there are separate instructions.

In this lab, you will:

1. Amend a commit.
2. Use interactive rebase to remove a commit.
3. Perform a squash merge.

## 1: Amend a commit.

1. Create a local repository named `projecth`.

2. Create an initial commit that adds a `fileA.txt` file containing the text "mistake". The commit message should be "add fileA.txt with mistake". Note the first to characters of the commit's SHA-1.

3. Amend the commit so that `fileA.txt` is empty and the commit message simply reads "add fileA.txt".

    - Modify `fileA.txt` so that it is empty.
    - Add `fileA.txt` to the staging area.
    - Amend the previous commit by executing `git commit --amend -m "add fileA.txt"`.

4. Notice the SHA-1 of your amended commit. It should be different than the one you noted prior to amending the commit.

    > Congratulations, you have amended a commit.

## 2: Use interactive rebase to remove a commit.

1. Create a `feature1` branch.

2. Create a commit on the `feature1` branch. Add the string "feature 1 wip" to `fileA.txt`. The commit message should be "add feature 1 wip".

3. Create another commit on the `feature1` branch. The contents of `fileA.txt` should be "feature 1". The commit message should be "add feature 1".

4. Let's say that you think that the "feature 1 wip" commit adds little value to the commit history. Use interactive rebase to remove that commit by squashing it.

    - Checkout the `feature1` branch.
    - View the commit graph of the `feature1` branch.
    - Execute `git rebase -i <SHA-1>`, where is for the initial commit of the repository (which is also the first commit of the `feature1` branch).
    - Explore the interactive rebase editor.
    - Remove the commit with the commit message of "feature 1 wip". Do this by changing the `pick` command to `squash` for the commit with a commit message of "add feature 1". Save the file.
    - An editor then opens for the commit message of the squashed commit. Change it to simply read `add feature 1`.

5. View your commit graph. Verify that the commit with a message of "feature 1 wip" has been removed.

6. Merge the `feature1` branch and delete its branch label.

    > Congratulations, you have performed an interactive rebase.

## 3: Perform a squash merge.

1. Create a `feature2` branch off of the `master` branch.

2. Create a commit on the `feature2` branch. Add the string "feature 2 wip" to `fileA.txt`. The commit message should be "add feature 2 wip".

3. Create another commit on the `feature2` branch. In `fileA.txt`, change the line "feature 2 wip" to "feature 2". The commit message should be "add feature 2".

4. Perform a squash merge of the `feature2` branch into the `master` branch.

    - Checkout the `master` branch.
    - Execute `git merge --squash feature2`.
    - Execute `git commit`. An editor opens. Specify a commit message of "add feature 2".

5. View your commit graph. Notice that the commit with a message of "feature 2 wip" is not part of the `master` branch.

6. Delete the `feature2` branch label. This will create two "dangling" commits that will be garbage collected.

7. You will not use the `projecth` repository in future labs. You can delete it.

> Congratulations, you have performed a squash merge and have completed this lab.

### Lab: (Sourcetree) Pull Requests I

Estimated time: 10 minutes

> *Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.*

In this lab, you will:

1. Prepare for a single repository pull request.
2. Create a pull request.
3. Merge a pull request

**1: Prepare for a single repository pull request.**

1. Log into Bitbucket and create a remote repository named `projecti`. You can have Bitbucket create the first commit here if you would like.
2. If you didn't create commit above, create and commit a `README.md` file containing the text `# PROJECTI README #` with a commit message of "initial commit".
3. **Clone** your repository into Sourcetree.
4. Using Sourcetree, create a branch named `feature1` off of the `master` branch. This will be the branch that is part of the single repository pull request.
5. Create a commit on the `feature1` branch containing a file named `fileA.txt` with the line "feature 1" as the content of the file.
6. Click `Push`. In the `Track` column, check the boxes next to each branch so that you will see a visual indicator when the local and remote branches are not synchronized. Select the `Push` checkbox associated with the `feature1` branch, because we just created a local commit on that branch. Click `OK` to push the `feature` branch to the remote `projecti` repository. Your local and remote branches should now be synchronized (for example, the `feature1` and `origin/feature1` labels should be pointing to the same commit).

> *Congratulations, you have prepared for a single repository pull request.*

**2: Create a pull request.**

1. In Bitbucket, navigate to the `projecti` repository. Click on `Pull requests`. Click on `Create pull request`.
2. In the `Create a pull request` window, select the `feature1` branch on the left and the `master` branch on the right. Modify the information if you would like and select `Create pull request`. You should see that your pull request was created.

> *Congratulations, you have created a single repository pull request.*

**3: Merge a pull request.**

1. (If necessary) In Bitbucket, navigate to the `projecti` repository. Click on `Pull requests`. You should see your pull request. Click on the link to view the pull request.
2. Click the `Merge` button associated with your pull request. Accept the default commit message and merge strategy (merge commit). Click `Merge`. You should see that the `feature1` branch is now merged. Click on `Commits` and verify that the work of feature 1 is now in your commit graph.

3. In Sourcetree, checkout the `feature1` branch and peform a `pull` of the `feature1` branch. Perform a `push` if necessary. Checkout the `master` branch and perform another `Pull` on the `master` branch. Perform a `push` if necessary. All of the local are remote (origin) branch labels should now be on the latest commit.

4. Checkout the `master` branch. Delete the `feature1` branch label by right-clicking the `feature1` branch name in the sidebar and selecting `Delete feature1`. Select the `Delete remote branch` checkbox to also delete the `feature1` branch label on `origin`.

5. You will not use the `projecti` repository in future labs. You can delete it.

> *Congratulations, you have merged a pull request and completed this lab.*

### Lab: (Command Line) Pull Requests I

Estimated time: 10 minutes

> *Note: This lab assumes that you are using the command line. If you would prefer to use Sourcetree, there are separate instructions.*

In this lab, you will:

1. Prepare for a single repository pull request.
2. Create a pull request.
3. Merge a pull request

**1: Prepare for a single repository pull request.**

1. Log into Bitbucket and create a remote repository named `projecti`. You can have Bitbucket create the first commit here if you would like.
2. If you didn't create commit above, create and commit a `README.md` file containing the text `# PROJECTI README #` with a commit message of "initial commit".
3. **Clone** your repository locally.
4. Using the command line, create a branch named `feature1` off of the `master` branch. This will be the branch that is part of the single repository pull request.
5. Create a commit on the `feature1` branch containing a file named `fileA.txt` with the line "feature 1" as the content of the file.
6. Push the `feature1` branch to the remote `projecti` repository.

> *Congratulations, you have prepared for a single repository pull request.*

**2: Create a pull request.**

1. In Bitbucket, navigate to the `projecti` repository. Click on `Pull requests`. Click on `Create pull request`.
2. In the `Create a pull request` window, select the `feature1` branch on the left and the `master` branch on the right. Modify the information if you would like and select `Create pull request`. You should see that your pull request was created.

> *Congratulations, you have created a single repository pull request.*

**3: Merge a pull request.**

1. (If necessary) In Bitbucket, navigate to the `projecti` repository. Click on `Pull requests`. You should see your pull request. Click on the link to view the pull request.
2. Click the `Merge` button associated with your pull request. Accept the default commit message and merge strategy (merge commit). Click `Merge`. You should see that the `feature1` branch is now merged. Click on `Commits` and verify that the work of feature 1 is now in your commit graph.
3. In the command line, execute a pull to add the new merge commit to your local repository.
4. Delete the `feature1` branch label locally and on `origin`. To delete the `origin` (remote) branch, view the project in Bitbucket. Select `Branches`, then `All branches`. Click the `...` next to the branch

name, then `Delete`.

5. You will not use the `projecti` repository in future labs. You can delete it.

> *Congratulations, you have merged a pull request and completed this lab.*

## Lab: (Sourcetree) Pull Requests II

Estimated time: 25 minutes

> *Note: This lab assumes that you are using Sourcetree. If you would prefer to use a command line interface, there are separate instructions.*

In this lab, you will:

1. Fork a remote repository.
2. Synchronize a forked repository using Bitbucket.
3. Create a multi-repository pull request.
4. Merge a multi-repository pull request.

**1: Fork a remote repository.**

1. Create a remote repository that we will consider to be the "upstream" repository. Do this by logging into Bitbucket and creating a repository named `projectj`. You can have Bitbucket create the README commit if you would like.
2. If you didn't create commit above, create and commit a `README.md` file containing the text `# PROJECTJ README #` with a commit message of "Initial commit".
3. **Fork** your `projectj` repository. Do this by clicking the + and selecting `Fork this repository`. Name the fork `projectjfork`. After creating the fork, you should see `projectjfork` in your list of Bitbucket repositories.

> *Congratulations, you have forked a remote repository.*

**2: Synchronize a forked repository using Bitbucket.**

1. In the upstream repository (`projectj`), update `README.md` and create a commit. You can do this directly in Bitbucket or from your local client, pushing the changes to `projectj`.
2. In Bitbucket, navigate to `projectjfork`.
3. Select the **Source** tab. View the Repository details in the upper right. You may need to click on the icon in the upper right (or press ]) to expand the repository details, then click > to show the dropdown.
4. Click the **Sync (1 commit behind)** link. Accept the default merge message and click `Sync`.
5. Click the `Commits` link. Notice that a merge commit was created in your fork.

> *Congratulations, you have synchronized a repository using Bitbucket.*

**3: Create a multi-repository pull request.**

1. Using Sourcetree, clone `projectjfork` to create a local repository for the forked repository.
2. Create a branch named `feature1` off of the `master` branch. This will be the branch that is part of the pull request.
3. Create a commit on the `feature1` branch containing a file named `fileA.txt` with the line "feature 1" as the content of the file.
4. Push the `feature1` branch to the remote `projectjfork` repository.

5. In Bitbucket, navigate to the `projectjfork` repository. Click on `Commits` and `Branches` to verify that your `feature1` branch and commit are on the remote repository.
6. Click on `Pull requests`. Click `Create pull request`.
7. On the `Create a pull request` page, select `feature1` on the left and `master` on the right. Modify the information if you would like and select `Create pull request`. You should see that your pull request was created.

> *Congratulations, you have created a multi-repository pull request.*

**4: Merge a multi-repository pull request.**

1. In Bitbucket, navigate to `projectj`. This is the upstream repository. Click on `Pull requests`. You should see the pull request from your fork.
2. Click on the link to view the pull request.
3. Click the `Merge` button. Accept the default commit message and merge strategy (merge commit). Click `Merge`. You should see that the `feature1` branch is now merged. Click on `Commits` and verify that the work of feature 1 is now in your commit graph.
4. In Bitbucket, navigate to the `projectjfork` repository. Select the **Source** tab. Because of the merge commit upstream, this repository is 2 commits behind. Click the `Sync` button on the right (in `Repository details`). Accept the default commit message and click `Sync`.
5. Because the work of the `feature1` branch is merged, you can delete the `feature1` branch label in the forked remote repository. You should find the `feature1` branch under `Branches > Merged branches`.
6. In Sourcetree, delete the local `projectjfork` repository and create a new clone of the same name (you may need to delete the existing local folder first). View the commit graph and verify that the merge commit from upstream is present. Now all of your repositories should be synchronized.

> *Note: Deleting the local repository and recreating it using `clone` is a choice. Because we knew that all of our local work has been pushed to the remote repository, we can delete the local repository and start over using a clone. We are then assured that the remote and local repositories are synchronized.*

7. You will not use the `projectj` or `projectjfork` repositories in future labs. You can delete them.

> *Congratulations, you have merged a multi-repository pull request and completed this lab.*

### Lab: (Command Line) Pull Requests II

Estimated time: 25 minutes

> *Note: This lab assumes that you are using a command line. If you would prefer to use Sourcetree, there are separate instructions.*

In this lab, you will:

1. Fork a remote repository.
2. Synchronize a forked repository using Bitbucket.
3. Create a multi-repository pull request.
4. Merge a multi-repository pull request.

**1: Fork a remote repository.**

1. Create a remote repository that we will consider to be the "upstream" repository. Do this by logging into Bitbucket and creating a repository named `projectj`. You can have Bitbucket create the README commit if you would like.
2. If you didn't create commit above, create and commit a `README.md` file containing the text `# PROJECTJ README #` with a commit message of "Initial commit".
3. **Fork** your `projectj` repository. Do this by clicking the + and selecting `Fork this repository`. Name the fork `projectjfork`. After creating the fork, you should see `projectjfork` in your list of Bitbucket repositories.

> *Congratulations, you have forked a remote repository.*

**2: Synchronize a forked repository using Bitbucket.**

1. In the upstream repository (`projectj`), update `README.md` and create a commit. You can do this directly in Bitbucket or from your local client, pushing the changes to `projectj`.
2. In Bitbucket, navigate to `projectjfork`.
3. Select the **Source** tab. View the Repository details in the upper right. You may need to click on the icon in the upper right (or press ]) to expand the repository details, then click > to show the dropdown.
4. Click the **Sync (1 commit behind)** link. Accept the default merge message and click `Sync`.
5. Click the `Commits` link. Notice that a merge commit was created in your fork.

> *Congratulations, you have synchronized a repository using Bitbucket.*

**3: Create a multi-repository pull request.**

1. Using the command line, clone `projectjfork` to create a local repository for the forked repository.
2. Create a branch named `feature1` off of the `master` branch. This will be the branch that is part of the pull request.
3. Create a commit on the `feature1` branch containing a file named `fileA.txt` with the line "feature 1" as the content of the file.
4. Push the `feature1` branch to the remote `projectjfork` repository.

5. In Bitbucket, navigate to the `projectjfork` repository. Click on `Commits` and `Branches` to verify that your `feature1` branch and commit are on the remote repository.
6. Click on `Pull requests`. Click `Create pull request`.
7. On the `Create a pull request` page, select `feature1` on the left and `master` on the right. Modify the information if you would like and select `Create pull request`. You should see that your pull request was created.

> *Congratulations, you have created a multi-repository pull request.*

**4: Merge a multi-repository pull request.**

1. In Bitbucket, navigate to `projectj`. This is the upstream repository. Click on `Pull requests`. You should see the pull request from your fork.
2. Click on the link to view the pull request.
3. Click the `Merge` button. Accept the default commit message and merge strategy (merge commit). Click `Merge`. You should see that the `feature1` branch is now merged. Click on `Commits` and verify that the work of feature 1 is now in your commit graph.
4. In Bitbucket, navigate to the `projectjfork` repository. Select the **Source** tab. Because of the merge commit upstream, this repository is 2 commits behind. Click the `Sync` button on the right (in `Repository details`). Accept the default commit message and click `Sync`.
5. Because the work of the `feature1` branch is merged, you can delete the `feature1` branch label in the forked remote repository. You should find the `feature1` branch under `Branches > Merged branches`.
6. Using the command line, delete the local `projectjfork` repository/directory and create a new clone of the same name (you may need to delete the existing local folder first). View the commit graph and verify that the merge commit from upstream is present. Now all of your repositories should be synchronized.

> *Note: Deleting the local repository and recreating it using `clone` is a choice. Because we knew that all of our local work has been pushed to the remote repository, we can delete the local repository and start over using a clone. We are then assured that the remote and local repositories are synchronized.*

7. You will not use the `projectj` or `projectjfork` repositories in future labs. You can delete them.

> *Congratulations, you have merged a multi-repository pull request and completed this lab.*