DATA INVARIANT LAYER

git version control guideline

September 29, 2017

Contents

[1. Basic Git Workflow 3](#_Toc494463988)

[1.1 Hello Git 3](#_Toc494463989)

[1.2 Git init 3](#_Toc494463990)

[1.3 Git Workflow 3](#_Toc494463991)

[1.4 Git Status 4](#_Toc494463992)

[1.5 Git Add 4](#_Toc494463993)

[1.6 Git Diff 4](#_Toc494463994)

[1.7 Git Commit 5](#_Toc494463995)

[1.8 Git Log 6](#_Toc494463996)

[1.9 Generalizations 6](#_Toc494463997)

[2. Backtrack in Git 7](#_Toc494463998)

[2.1 Backtracking Introduction 7](#_Toc494463999)

[2.2 Head Commit 7](#_Toc494464000)

[2.3 Git Checkout 8](#_Toc494464001)

[2.4 More Git Add 9](#_Toc494464002)

[2.5 Git Reset I 9](#_Toc494464003)

[2.6 Git Reset II 10](#_Toc494464004)

[2.7 Git Reset Review 11](#_Toc494464005)

[2.8 Generalizations 11](#_Toc494464006)

[3. Git Branching 12](#_Toc494464007)

[3.1 Git Branch I 12](#_Toc494464009)

[3.2 Branching Overview 12](#_Toc494464010)

[3.3 Git Branch II 13](#_Toc494464011)

[3.4 Git Checkout 13](#_Toc494464012)

[3.5 Commit on a New Branch 14](#_Toc494464013)

[3.6 Git Merge 15](#_Toc494464014)

[3.7 Merge Conflict I 15](#_Toc494464015)

[3.8 Merge Conflict II 16](#_Toc494464016)

[3.9 Delete Branch 17](#_Toc494464017)

[3.10 Generalization 17](#_Toc494464018)

[4. Git Teamwork 17](#_Toc494464019)

[4.1 Overview 17](#_Toc494464021)

[4.2 Git Clone 18](#_Toc494464022)

[4.3 Git Remote-V 18](#_Toc494464023)

[4.4 Git Fetch 19](#_Toc494464024)

[4.5 Git Merge 19](#_Toc494464025)

[4.6 Git Workflow 20](#_Toc494464026)

[4.7 Git Push 21](#_Toc494464027)

[4.8 Generalization 21](#_Toc494464028)

[5. Basic Bitbucket Cloud 22](#_Toc494464029)

[5.1. Create a Git Repository 22](#_Toc494464030)

[5.2. Copy Git Repository and Add Files 23](#_Toc494464031)

[5.3. Pull Changes from Git repository on Bitbucket Cloud 28](#_Toc494464032)

[5.4. Use a Git Branch to Merge a File 30](#_Toc494464033)

[6. Code Review in Bitbucket Cloud 34](#_Toc494464034)

[6.1. Fork a Teammate's Repository 35](#_Toc494464035)

[6.2. Copy Fork and Make a Change to the Repository 35](#_Toc494464036)

[6.3. Create a Pull Request 38](#_Toc494464037)

[7. Branching with Bitbucket Cloud 40](#_Toc494464038)

[7.1. Create a Branch and Change using the Branching Workflow 40](#_Toc494464039)

[7.2. Create, Fetch, and Checkout a Remote Branch 42](#_Toc494464040)

[7.3. Push Change and Create a Pull Request 43](#_Toc494464041)

[7.4. Delete a Branch and Pull Master into Local Working Branch 44](#_Toc494464042)

# Basic Git Workflow

## Hello Git

Git is a software that allows you to keep track of changes made to a project over time. Git works by recording the changes you make to a project, storing those changes, then allowing you to reference them as needed.

**Instructions:**

**We’ll get started by taking a look at the screenplay project.** In **scene-1.txt**, add this text:

Harry Programmer and the Sorcerer’s Code: Scene 1

Then press **enter** to create a new empty line. Once you've created the new line

## Git init

Now that we have started working on the screenplay, let’s turn the sorcerers-code directory into a Git project. We do this with:

**Input:**

$ git init

The word init means initialize. The command sets up all the tools Git needs to begin tracking changes made to the project.

In the terminal, initialize a new Git project.

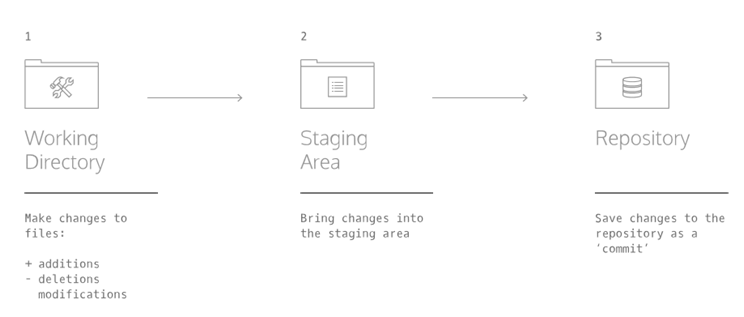
**Output:**

Initalized an empty git repository in /home/ccuser/workspace/sorcerers-code/.git/

The Git project was created.

## Git Workflow

Now we have a Git project. A Git project can be thought of as having three parts:



1. A *Working Directory*: where you'll be doing all the work: creating, editing, deleting and organizing files
2. A *Staging Area*: where you'll list changes you make to the working directory
3. A *Repository*: where Git permanently stores those changes as different *versions* of the project

* The Git workflow consists of editing files in the working directory, adding files to the staging area, and saving changes to a Git repository. In Git, we save changes with a *commit*, which we will learn more about in this lesson.

## Git Status

As you write the screenplay, you will be changing the contents of the working directory. You can check the status of those changes with:

From the terminal, check the status of the sorcerers-code project. In the output, notice the file in red under untracked files. Untracked means that Git sees the file but has not started tracking changes yet.

**Input:**

$ git status

**Output:**

Untracked files:

(use “git add <file> …” to include in what will be committed)

Init\_test.rb

Scene-1.txt

Nothing added to commit but untracked files present (use “git add” to track)

## Git Add

In order for Git to start tracking **scene-1.txt**, the file needs to be added to the staging area.

We can add a file to the staging area with:

$ git add filename

The word filename here refers to the name of the file you are editing, such as **scene-1.txt**.

**Instructions**

1. Add **scene-1.txt** to the staging area in Git. Recall that you will need to identify the file by its name.
2. Check the status of the project in Git. In the output, notice that Git indicates the changes to be committed with "new file: scene-1.txt" in green text. Here Git tells us the file was added to the staging area.

$ git add scene-1.txt

$ git status

On branch master

Initial commit

Changes to be committed:

(use “git rm –cached <file>…” to unstage)

New file: scene-1.txt

## Git Diff

Now you know how to add a file to the staging area.

Imagine that we type another line in **scene-1.txt**. Since the file is tracked, we can check the differences between the working directory and the staging area with:

$ git diff filename

Here, filename is the actual name of the file. If the name of my file was **changes.txt** the command would be

$ git diff changes.txt

**Instructions:**

1. In the code editor, add this text to scene-1.txt:

Dumblediff: I should've known you would be here, Professor McGonagit.

Click Run.

1. From the terminal, use the new command to check the difference between the working directory and the staging area. Notice the output:

* "Harry Programmer and the Sorcerer's Code: Scene 1" is in the staging area, as indicated in white.
* Changes to the file are marked with a + and are indicated in green.

$ git diff scene-1.txt

Diff –git a/scene-1.txt b/scene-1.txt

index c33ce4c..258fda4 100644

--- a/scene-1.txt

+++ b/scene-1.txt

@@ -1 +1,2 @@

- Harry Programmer and the Sorcerer<E2><80><99>s Code: Scene 1

+Harry Programmer and the Sourcerers Code: Scene 1

+Dumblediff: I should've known you would be here, Professor McGonagit.

IMPORTANT: press q on your keyboard to exit diff mode.

1. Add the changes to the staging area in Git. Recall that you will need to identify the file by its name.

$ git add scene-1.txt

## Git Commit

A *commit* is the last step in our Git workflow. A commit permanently stores changes from the staging area inside the repository.

git commit is the command we'll do next. However, one more bit of code is needed for a commit: the *option* -m followed by a message. Here's an example:

$ git commit -m "Complete first line of dialogue"

Standard Conventions for Commit Messages:

* Must be in quotation marks
* Written in the present tense
* Should be brief (50 characters or less) when using -m

**Instructions:**

1. Make your first commit! From the terminal, type the command along with a commit message. The message should describe the point of the commit.

If you're having trouble thinking of a good commit message, reflect on how the project has changed since it began.

**Output:**

[master (root-commit) f22c7ad] Complete first line of dialogue

1 file changed, 2 insertions (+)

Create mode 10644 scene-1.txt

## Git Log

Often with Git, you'll need to refer back to an earlier version of a project. Commits are stored chronologically in the repository and can be viewed with:

$ git log

**Instructions:**

1. From the terminal, log a list of your commits.

A 40-character code, called a SHA, that uniquely identifies the commit. This appears in orange text.

The commit author

The date and time of the commit

The commit message

**Output:**

$ git log

Commit f22c7ad4d83fa1523d94badeb8a1cfe0161fb02c

Author: Sion.Hwang

Date: Fri Sep 15 03:52:10 2017 +0000

## Generalizations

You have now been introduced to the fundamental Git workflow. You learned a lot! Let's take a moment to generalize:

1. Git is the industry-standard version control system for web developers
2. Use Git commands to help keep track of changes made to a project:

* **git init** creates a new Git repository
* **git status** inspects the contents of the working directory and staging area
* **git add** adds files from the working directory to the staging area
* **git diff** shows the difference between the working directory and the staging area
* **git commit** permanently stores file changes from the staging area in the repository
* **git log** shows a list of all previous commits

# Backtrack in Git

## Backtracking Introduction

When working on a Git project, sometimes we make changes that we want to get rid of. Git offers a few eraser-like features that allow us to undo mistakes during project creation. In this lesson, we'll learn some of these features.

**Instructions:**

1. You are in a Git project titled **hamlet-prince-of-denmark**. In the code editor, you'll be working on **scene-5.txt**. Here, Hamlet encounters the ghost of his father. Add this text to the file:

Ghost:

My hour is almost come,

When I to sulphurous and tormenting flames

Must render up myself.

1. From the terminal, add **scene-5.txt** to the staging area.

$ git add scene-5.txt

1. Commit the changes to the repository with a good commit message.

$ git commit -m “Complete dialogue”

**Output:**

$ git add scene-5.txt

$ git commit -m “Complete dialogue”

[master e262889] complete dialogue

1 file changed, 4 insertions (+)

## Head Commit

In Git, the commit you are currently on is known as the **HEAD commit**. In many cases, the most recently made commit is the HEAD commit.

To see the HEAD commit, enter:

$ git show HEAD

The output of this command will display everything the [**git log** command](https://www.codecademy.com/en/courses/learn-git/lessons/git-workflow/exercises/git-log) displays for the HEAD commit, plus all the file changes that were committed.

**Instructions:**

1. Enter the command to show the HEAD commit. Notice the output. The Ghost's most recently added line is in green text.

$ git show HEAD

Commit e2628898bfb8b9120bd2a98d5185ada8578028e9

Author: Sion.Hwang

Date: Fri Sep 15 03:52:10 2017 +0000

Complete dialogue

Diff –git a/scene-5.txt b/scene-5.txt

Index b12dd97...1e0c19e 100644

--- a/scene-5.txt

+++ b/scene-5.txt

@@ -11,4 +11,8 @@ mark me.

Helmet:

I will.

+Ghost:

+My hour is almost come,

+When I to sulphurous and tormenting flames

+Must render up myself.

## Git Checkout

What if you decide to change the ghost's line in the working directory, but then decide you wanted to discard that change?

You could rewrite the line how it was originally, but what if you forgot the exact wording? The command

$ git checkout HEAD filename

will restore the file in your working directory to look exactly as it did when you last made a commit.

Here, filename again is the actual name of the file. If the file is named changes.txt, the command would be

$ git checkout HEAD changes.txt

**Instructions:**

1. Change the ghost's words in some way.

Ghost:

My hour is almost come,

When I to sulphurous and tormenting balloons

Must render up myself.

1. From the terminal, use **git diff** to see the difference between **scene-5.txt** as it appears in the working directory vs. how it appears in your last commit. You may need to press q on your keyboard to restore the terminal.

**Output:**

$ git diff scene-5.txt

Diff –git a/scene-5.txt b/scene-5.txt

index c33ce4c..258fda4 100644

--- a/scene-5.txt

+++ b/scene-5.txt

@@ -13,6 +13,6 @@ I will.

Ghost:

My hour is almost come,

- When I to sulphuruous and tormenting flames

+When I to sulphurous and tormenting balloons

Must render up myself.

1. Use the new Git command to restore the file in your working directory to look as it did when you last made a commit. Notice that the changes you made to the ghost's line have been discarded.

## More Git Add

The **hamlet** repository we are working on contains five files. In Git, it's common to change many files, add those files to the staging area, and commit them to a repository in a single commit.

For example, say you want to change the character "LARRY" to "LAERTES" in the script. The name currently appears in two files. After you change the name in both files, you could add the changed files to the staging area with:

$ git add filename\_1 filename\_2

Note the word filename above refers to the name of the file you are adding to the staging area, such as scene-3.txt.

**Instructions:**

1. The code editor is open to **scene-3.txt** and **scene-7.txt**. In **scene-3.txt**, everywhere you see the name "LARRY" change it to "LAERTES."
2. Now change all instances of "LARRY" to "LAERTES" in **scene-7.txt**.
3. Add the files to the staging area together using a single git command.

$ git add scene-3.txt scene-7.txt

## Git Reset I

The files you've added to the staging area belong in the same commit.

What if, before you commit, you accidentally delete an important line from scene-2.txt? Unthinkingly, you add scene-2.txt to the staging area. The file change is unrelated to the Larry/Laertes swap and you don't want to include it in the commit.

We can unstage that file from the staging area using

$ git reset HEAD filename

This command resets the file in the staging area to be the same as the HEAD commit. It does not discard file changes from the working directory, it just removes them from the staging area.

**Instructions:**

1. To try out the new command, let's make a mistake on purpose. The code editor is open to **scene-2.txt**. Delete any line from the file and click Run.
2. From the terminal, add **scene-2.txt** to the Git staging area.

$ git add scene-2.txt

1. Now check the **status** of the Git project. In the output, notice **scene-2.txt** under “Changes to be committed”.

$ git status

1. Use the new Git command to unstage scene-2.txt from the staging area. M is short for "modification". Notice in the output, "Unstaged changes after reset":

**Input:**

$ git reset HEAD scene-2.txt

**Output:**

M scene-2.txt

## Git Reset II

Creating a project is like hiking in a forest. Sometimes you take a wrong turn and find yourself lost.

Just like retracing your steps on that hike, Git enables you to rewind to the part before you made the wrong turn. You can do this with:

$ git reset commit\_SHA

This command works by using the first 7 characters of the SHA of a previous commit. For example, if the SHA of the previous commit is **5d692065cf51a2f50ea8e7b19b5a7ae512f633ba**, use:

$ git reset 5d69206

**HEAD** is now set to that previous commit.

**Instructions:**

1. From the terminal, print out your Git commit log.
2. From the terminal, enter the command to reset to a previous commit, using the first 7 characters of one of the past commit SHAs in your Git log. Next, print the Git commit log again.
3. The commits that came after the one you reset to are gone. The HEAD commit has been reassigned. You just changed history.

## Git Reset Review

To better understand git reset **commit\_SHA**, notice the diagram below. Each circle represents a commit.

**Before reset:**

HEAD is at the most recent commit

**After resetting:**

HEAD goes to a previously made commit of your choice

The gray commits are no longer part of your project

You have re-wound the project's history

## Generalizations

You can use these skills to undo changes made to your Git project.

Let's take a moment to review the new commands:

* **git checkout** HEAD filename: Discards changes in the working directory.
* **git reset** HEAD filename: Unstages file changes in the staging area.
* **git reset** commit\_SHA: Resets to a previous commit in your commit history.

Additionally, you learned a way to add multiple files to the staging area with a single command:

$ git add filename\_1 filename\_2

# Git Branching



## Git Branch I

Up to this point, you've worked in a single Git branch called master. Git allows us to create branches to experiment with versions of a project. Imagine you want to create version of a story with a happy ending. You can create a new branch and make the happy ending changes to that branch only. It will have no effect on the master branch until you're ready to merge the happy ending to the master branch.

In this lesson, we'll be using Git branching to develop multiple versions of a resumé.

You can use the command below to answer the question: “which branch am I on?”

$ git branch

**Instructions:**

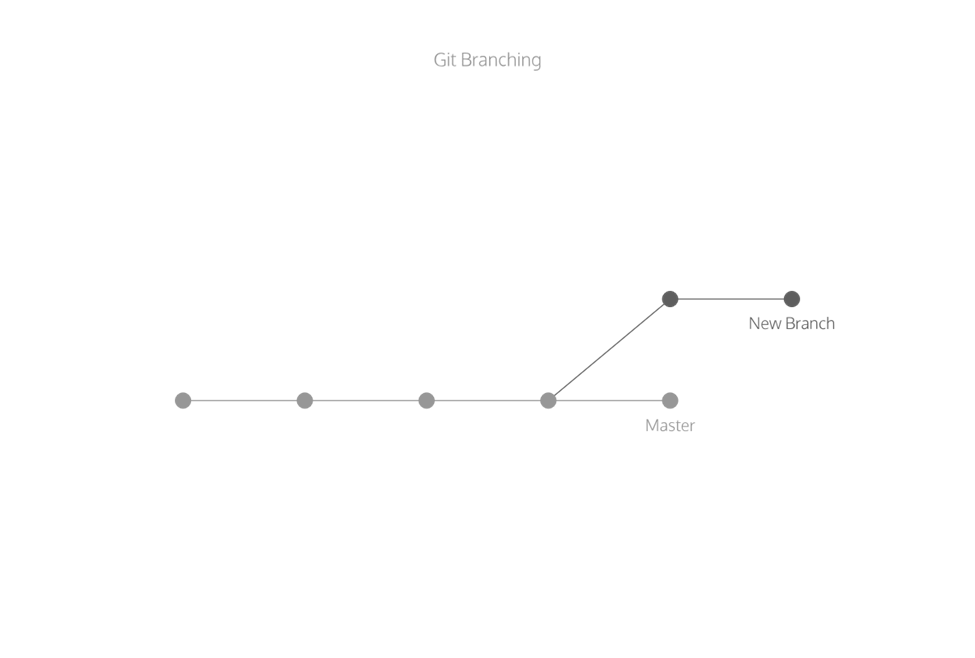
1. Check what branch you are currently on.

**Output:**

\* master

In the output, the \* (asterisk) is showing you what branch you’re on. The project only has one branch at this time.

## Branching Overview



The diagram above illustrates branching.

The circles are commits, and together form the Git project's commit history.

New Branch is a different version of the Git project. It contains commits from Master but also has commits that Master does not have.

## Git Branch II

Right now, the Git project has only one branch: **master**.

To create a new branch, use:

$ git branch new\_branch

Here **new\_branch** would be the name of the new branch you create, like **photos** or **blurb**. Be sure to name your branch something that describes the purpose of the branch. Also, branch names can’t contain whitespaces: **new-branch** and **new\_branch** are valid branch names, but **new branch** is not.

**Instructions:**

1. Let's create a new version of a resumé to apply for a fencing instructor role.

Create a new branch called fencing.

Remember to spell the word "fencing" correctly.

Next, view your branches as you did in the previous exercise.

Notice in the output there now appear two branches: master and fencing.

**Input:**

$ git branch fencing

$ git branch

**Output:**

fencing

\* master

## Git Checkout

The master and fencing branches are identical: they share the same exact commit history. You can switch to the new branch with

$ git checkout branch\_name

Here, branch\_name is the name of the branch. If the branch's name is skill

$ git checkout skill

Once you switch branch, be now able to make commits on the branch that have no impact on master.

You can continue your workflow, while master stays intact!

Instructions:

1. Switch to the **fencing** branch from the **master** branch.

**Input:**

$ git checkout fencing

**Output:**

Switched to branch ‘fencing’

1. Use git branch to verify that you have switched branches. In the output, notice the \* is now over the fencing branch

**Input:**

$ git branch

**Output:**

\* fencing

master

## Commit on a New Branch

All the commands you do on master, you can also do on this branch.

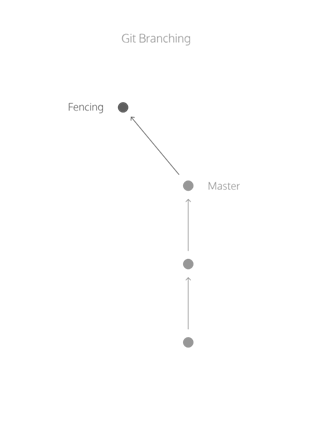
For example, to add files to the staging area, use:

$ git add filename

And to commit, use:

$ git commit -m “Commit message”

In a moment, you will make a commit on the fencing branch. The below diagram shows what will happen to the Git project.



Instructions:

1. Print the Git commit log.

Notice the output:

* The commits you see were all made in the master branch. fencing inherited them.
* This means that every commit master has, fencing also has.

1. In resume.txt, replace your skill at scheming against Hook with your experience in sword-fights.

Delete this line:

Scheme against Captain Hook

and type this line in its place:

Engage in swordfights with pirates

1. Add **resume.txt** into the staging area.

$ git add resume.txt

1. Commit the changes to the repository with a commit message.

$ git commit -m “Change Implemented”

## Git Merge

What if you wanted include all the changes made to the **fencing** branch on the **master** branch? We can easily accomplish this by merging the branch into master with:

$ git merge branch\_name

For example, if I wanted to merge the skills branch to master, I would enter

$ git merge skills

In a moment, you'll merge branches. Keep in mind:

* Your goal is to update master with changes you made to fencing.
* fencing is the giver branch, since it provides the changes.
* master is the receiver branch, since it accepts those changes.

**Instructions:**

1. You are currently on the **fencing** branch. Switch over to the **master** branch.

$ git checkout master

$ git branch

**Output:**

$ git branch

fencing

\* master

1. Your sword-fighting experience is so impressive that it belongs on the master version of your resumé.

From the terminal, merge the fencing branch into the master branch.

Notice the output: The merge is a "fast forward" because Git recognizes that fencing contains the most recent commit. Git *fast forwards* master to be up to date with fencing.

$ git merge fencing

## Merge Conflict I

The merge was successful because **master** had not changed since we made a commit on **fencing**. Git knew to simply update **master** with changes on **fencing**.

What would happen if you made a commit on **master** *before* you merged the two branches? Furthermore, what if the commit you made on **master** altered the same exact text you worked on in **fencing**? When you switch back to **master** and ask Git to merge the two branches, Git doesn't know which changes you want to keep. This is called a *merge conflict*.

**Instructions:**

1. You are on the **master** branch. In the code editor, where you have written:

-Engage in swordfights with pirates

Add the word "professional", so the text reads:

-Engage in swordfights with professional pirates

1. Add resume.txt to the staging area.

$ git add resume.txt

1. Commit the changes to the repository with a commit message.

$ git commit -m “Update fencing”

1. Imagine a few weeks have passed, and you'd like to develop your fencing resumé some more. Switch back to the fencing branch.

$ git checkout fencing

1. From fencing, change the line so it reads:

- Engage in swordfights with professional pirates such as Smee.

1. Once again, add resume.txt to the staging area.

$ git add resume.txt

1. Commit the changes to the repository with a commit message.

## Merge Conflict II

Let's say you decide you'd like to merge the changes from fencing into master.

Here's where the trouble begins!

You've made commits on separate branches that alter the same line in conflicting ways. Now, when you try to merge fencing into master, Git will not know which version of the file to keep.

**Instructions:**

1. Switch to the master branch.

$ git checkout master

1. From the terminal, enter the command below:

$ git merge fencing

This will try to merge fencing into master. In the output, notice the lines:

CONFLICT (content): Merge conflict in resumé.txt

Automatic merge failed; fix conflicts and then commit the result.

1. We must fix the merge conflict.

In the code editor, look at resume.txt. Git uses markings to indicate the **HEAD** (master) version of the file and the **fencing** version of the file, like this:

<<<<<<< HEAD

master version of line

=======

fencing version of line

>>>>>>> fencing

a

Git asks us which version of the file to keep: the version on master or the version on fencing. You decide you want the fencing version.

From the code editor:

Delete the content of the line as it appears in the master branch

Delete all of Git's special markings including the words HEAD and fencing. If any of Git's markings remain, for example, >>>>>>> and =======, the conflict remains.

1. Add **resume.txt** to the staging area.

$ git add resume.txt

1. Now, make a commit. For your commit message, type "Resolve merge conflict" to indicate the purpose of the commit.

$ git commit -m “Resolve merge conflict”

## Delete Branch

In Git, branches are usually a means to an end. You create them to work on a new project feature, but the end goal is to merge that feature into the **master** branch. After the branch has been integrated into **master**, it has served its purpose and can be deleted.

The command below will delete the specified branch from your Git project.

$ git branch -d branch\_name

Now that master contains all the file changes that were in fencing, let's delete fencing.

**Instructions:**

1. Delete the fencing branch. Now, verify that you have indeed deleted fencing by listing all your project's branches on the terminal. Notice in the output that only one branch, master, is shown.

## Generalization

Git branching allows users to experiment with different versions of a project by checking out separate branches to work on.

The following commands are useful in the Git branch workflow.

* **git branch**: Lists all a Git project's branches.
* **git branch branch\_name**: Creates a new branch.
* **git checkout branch\_name**: Used to switch from one branch to another.
* **git merge branch\_name**: Used to join file changes from one branch to another.
* **git branch -d branch\_name**: Deletes the branch specified.

# Git Teamwork



## Overview

So far, we've learned how to work on Git as a single user. Git offers a suite of collaboration tools to make working with others on a project easier.

Imagine that you're a science teacher, developing some quizzes with Sally, another teacher in the school. You are using Git to manage the project.

In order to collaborate, you and Sally need:

* A complete replica of the project on your own computers
* A way to keep track of and review each other's work
* Access to a definitive project version

You can accomplish all of this by using remotes. A remote is a shared Git repository that allows multiple collaborators to work on the same Git project from different locations. Collaborators work on the project independently, and merge changes together when they are ready to do so.

## Git Clone

Sally has created the remote repository, science-quizzes in the directory curriculum, which teachers on the school's shared network have access to. In order to get your own replica of science-quizzes, you'll need to clone it with:

$ git clone remote\_location clone\_name

In this command:

* **remote\_location** tells Git where to go to find the remote. This could be a web address, or a filepath, such as:

/Users/teachers/Documents/some-remote

* **clone\_name** is the name you give to the directory in which Git will clone the repository.

**Instructions:**

The Git remote Sally started is called:

science-quizzes

Enter the command to clone this remote. Name your clone:

my-quizzes

**Input:**

$ git clone science-quizzez my-quizzes

**Output:**

cloning into 'my-quizzes'...

done.

Git informs us that it's copying everything from science-quizzes into the my-quizzes directory.

my-quizzes is your local copy of the science-quizzes Git project. If you commit changes to the project here, Sally will not know about them.

## Git Remote-V

We have a clone of Sally's remote on our computer. One thing that Git does behind the scenes when you clone **science-quizzes** is give the remote address the name ***origin***, so that you can refer to it more conveniently. In this case, Sally's remote is ***origin***.

You can see a list of a Git project's remotes with the command:

$ git remote -v

**Instructions:**

1. Using the file navigator, examine the contents of the cloned Git project. There are a few quiz files here, which we will be working with during this lesson. Open a file of your choice in the code editor.
2. Change directories into the **my-quizzes** directory, enter this command on the terminal:

cd my-quizzes

Enter **git remote -v** to list the remotes.

Notice the output:

origin /home/ccuser/workspace/curriculum/science-quizzes (fetch)

origin /home/ccuser/workspace/curriculum/science-quizzes (push)

Git lists the name of the remote, **origin**, as well as its location.

Git automatically names this remote **origin**, because it refers to the remote repository of origin. However, it is possible to safely change its name.

The remote is listed twice: once for (**fetch**) and once for (**push**).

## Git Fetch

After you cloned **science-quizzes**, you had to run off to teach a class. Now that you're back at your computer, there's a problem: what if, while you were teaching, Sally changed the **science-quizzes** Git project in some way. If so, your clone will no longer be up-to-date.

An easy way to see if changes have been made to the remote and bring the changes down to your local copy is with:

$ git fetch

This command will not merge changes from the remote into your local repository. It brings those changes onto what's called a remote branch.

**Instructions:**

1. Enter this command:

cd my-quizzes

to go into the my-quizzes directory.

1. Fetch any new changes Sally may have made to the remote.

$ git fetch

## Git Merge

Even though Sally's new commits have been fetched to your local copy of the Git project, those commits are on the **origin/master** branch. Your local **master** branch has not been updated yet, so you can't view or make changes to any of the work she has added.

Now we'll use the **git merge** command to integrate **origin/master** into your local **master** branch.

$ git merge origin/master

**Instructions:**

1. Enter this command:

to go into the my-quizzes directory.

cd my-quizzes

1. You are on your local **master** branch. In your commit history, the commit message of the **HEAD** commit is:

Add first question to Physics quiz

From the terminal, merge with origin/master, where Sally's most recent commits are.

**Output:**

Updating a2ba090..bc87a1a

Fast-forward

biology.txt | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

* Git has performed a "fast-forward" merge, bringing your local **master** branch up to speed with Sally's most recent commit on the remote.

1. Print the commit history. In the output, notice that the **HEAD** commit has changed. The commit message now reads:

Add heading and comment to biology quiz

## Git Workflow

Now that you've merged **origin/master** into your local **master** branch, you're ready to contribute some work of your own. The workflow for Git collaborations typically follows this order:

1. Fetch and merge changes from the remote
2. Create a branch to work on a new project feature
3. Develop the feature on your branch and commit your work
4. Fetch and merge from the remote again (in case new commits were made while you were working)
5. Push your branch up to the remote for review

Steps 1 and 4 are a safeguard against merge conflicts, which occur when two branches contain file changes that cannot be merged with the git merge command.

**Instructions:**

1. Enter this command:

to go into the my-quizzes directory.

cd my-quizzes

1. Enter the Git command:

$ git branch <branch\_name>

to create a branch to develop questions for the biology quiz. Name the branch bio-questions.

1. Switch to your new branch with the command:

$ git checkout <branch\_name>

1. On your branch, open **biology.txt** in the code editor and make changes.
2. Add **biology.txt** to the staging area.

$ git add biology.txt

1. Commit the work to the repository with a commit message.

$ git commit -m "Update questions”

## Git Push

Now it's time to share our work with Sally.

The command:

$ git push origin your\_branch\_name

will push your branch up to the remote, origin. From there, Sally can review your branch and merge your work into the master branch, making it part of the definitive project version.

1. Enter this command:

to go into the my-quizzes directory.

cd my-quizzes

1. Push your branch up to the remote.

In the output, notice the line:

To /home/ccuser/workspace/curriculum/science-quizzes

\* [new branch] bio-questions -> bio-questions

Git informs us that the branch **bio-questions** was pushed up to the remote. Sally can now review your new work and can merge it into the remote's master branch.

## Generalization

A remote is a Git repository that lives outside your Git project folder. Remotes can live on the web, on a shared network or even in a separate folder on your local computer.

The Git Collaborative Workflow are steps that enable smooth project development when multiple collaborators are working on the same Git project.

We also learned the following commands

* git clone: Creates a local copy of a remote.
* git remote -v: Lists a Git project's remotes.
* git fetch: Fetches work from the remote into the local copy.
* git merge origin/master: Merges origin/master into your local branch.
* git push origin <branch\_name>: Pushes a local branch to the origin remote.

Git projects are usually managed on Github, a website that hosts Git projects for millions of users. With Github you can access your projects from anywhere in the world by using the basic workflow you learned here

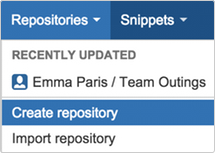
# Basic Bitbucket Cloud

## Create a Git Repository

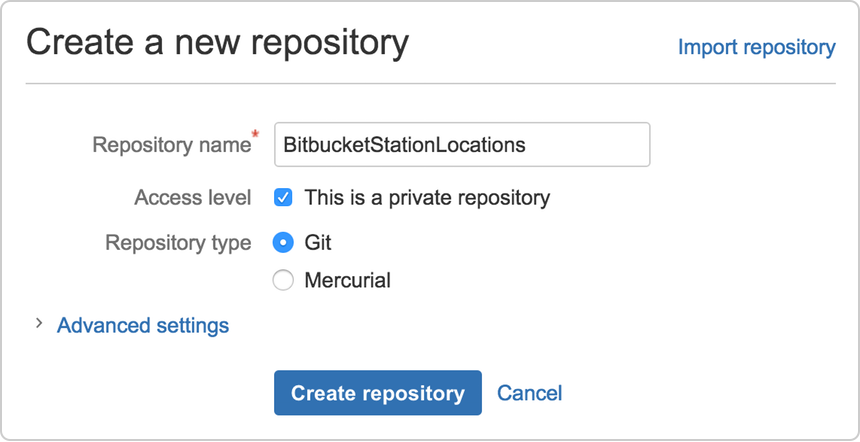
Initially, the repository you create in Bitbucket is going to be empty without any code in it. That's okay because you will start adding some files to it soon. This Bitbucket repository will be the central repository for your files, which means that others can access that repository if you give them permission. After creating a repository, you'll copy a version to your local system—that way you can update it from one repo, then transfer those changes to the other.

Do the following to create your repository:

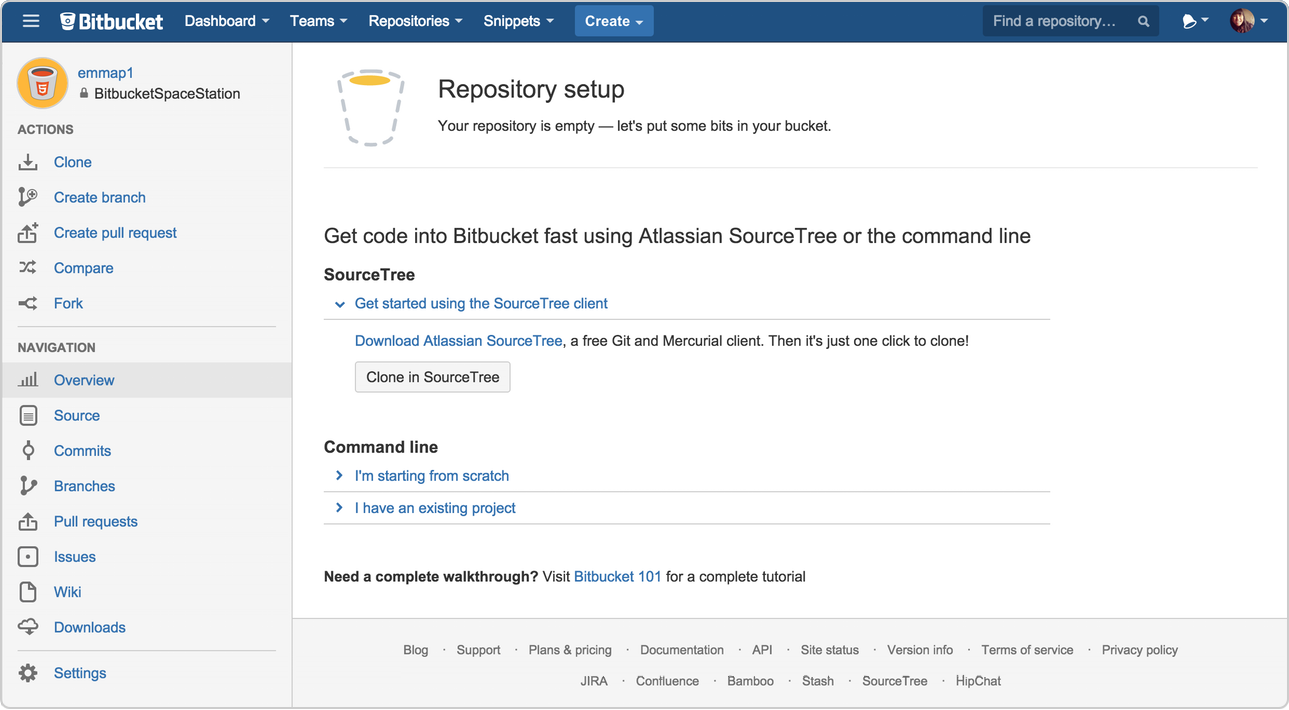
1. From Bitbucket, click **Repositories > Create repository** button at the top of the page.



The system displays the **Create a new repository** page. Take some time to review the dialog's contents. With the exception of the Repository type, everything you enter on this page you can later change.



1. Enter BitbucketStationLocations for the **Name** field. Bitbucket uses this **Name** in the URL of the repository. For example, if the user the\_best has a repository called awesome\_repo, the URL for that repository would be https://bitbucket.org/the\_best/awesome\_repo.
2. For **Access level**, leave the **This is a private repository box checked**. A private repository is only visible to you and those you give access to. If this box is unchecked, everyone can see your repository.
3. Pick **Git** for the **Repository type**. Keep in mind that you can't change the repository type after you click **Create repository**.
4. Click **Create repository**. Bitbucket creates your repository and displays its **Overview** page.

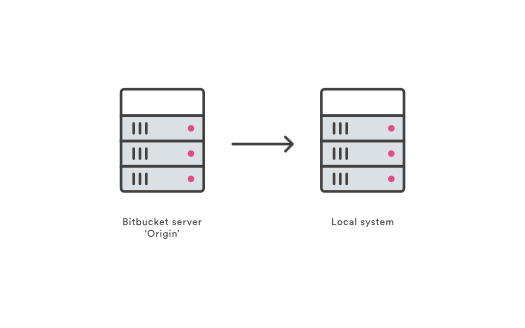
You should be on the repository's Overview page:

Click items on the Navigation menu to see what is behind each one. To view the shortcuts available to navigate these items, press the **“?”** key on your keyboard.

When you click the **Commits** option on the menu bar, you find that you have no commits because you have not created any content for your repository. Your repository is private and you have not invited anyone to the repository, so the only person who can create or edit the repository's content right now is you, the repository owner.

## Copy Git Repository and Add Files

Now that you have a place to add and share your space station files, you need a way to get to it from your local system. To set that up, you want to copy the Bitbucket repository to your system. Git refers to copying a repository as "cloning" it. When you clone a repository, you create a connection between the Bitbucket server (which Git knows as origin) and your local system.



**Step 1. Clone your repository to your local system**

Open a browser and a terminal window from your desktop. After opening the terminal window, do the following:

1. Navigate to your home (~) directory.

$ cd~

As you use Bitbucket more, you will probably work in multiple repositories. For that reason, it's a good idea to create a directory to contain all those repositories.

1. Create a directory to contain your repositories.

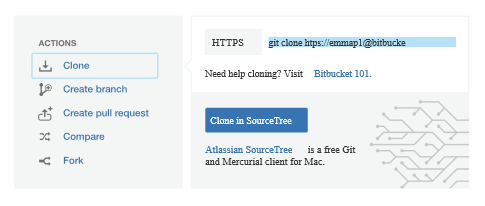
$ mkdir repos

1. From the terminal, update the directory you want to work in to your new repos directory.

$ cd ~/repos

1. From Bitbucket, go to your **BitbucketStationLocations** repository.
2. Click **Clone**.

The system displays a pop-up clone dialog. By default, the clone dialog sets the protocol to HTTPS or SSH, depending on your settings. For the purposes of this tutorial, don't change your default protocol.



1. Copy the highlighted clone command.
2. From your terminal window, paste the command you copied from Bitbucket and press **Return**.
3. Enter your Bitbucket password when the terminal asks for it. If you created an account by linking to Google, use your password for that account.

If you experience a **Windows password error**:

* In some versions of Microsoft Windows operating system and Git you might see an error similar to the one in the following example.

**Windows clone password error example**

$ git clone

https://emmap1@bitbucket.org/emmap1/bitbucketstationlocations.git

Cloning into 'bitbucketspacestation'...

fatal: could not read

Password for 'https://emmap1@bitbucket.org': No such file or directory

If you get this error, enter the following at the command line:

$ git config --global core.askpass

Then go back to step 4 and repeat the clone process. The bash agent should now prompt you for your password. You should only have to do this once. At this point, your terminal window should look like this:

$ cd ~/repos

$ git clone https://emmap1@bitbucket.org/emmap1/bitbucketstationlocations.git

Cloning into 'bitbucketstationlocations'...

Password

warning: You appear to have cloned an empty repository.

1. List the contents of your repos directory and you should see your bitbucketstationlocations directory in it.

$ ls

**Step 2. Add a file to your local repository and put it on Bitbucket**

With the repository on your local system, it's time to get to work. You want to start keeping track of all your space station locations. To do so, let's create a file about all your locations.

1. Go to your terminal window and navigate to the top level of your local repository.

$ cd ~/repos/bitbucketstationlocations/

1. Enter the following line into your terminal window to create a new file with content.

$ echo "Earth's Moon" >> locations.txt

If the command line doesn't return anything, it means you created the file correctly!

1. Get the status of your local repository. The [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-status) command tells you about how your project is progressing in comparison to your Bitbucket repository.

At this point, Git is aware that you created a new file, and you'll see something like this:

$ git status

On branch master

Initial commit

Untracked files:

(use "git add <file>..." to include in what will be committed)

locations.txt

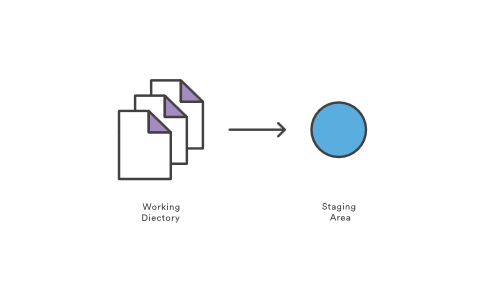
nothing added to commit but untracked files present (use "git add" to track)

The file is untracked, meaning that Git sees a file not part of a previous commit. The status output also shows you the next step: adding the file.

1. Tell Git to track your new locations.txt file using the [git add](https://www.atlassian.com/git/tutorials/saving-changes/git-add) command. Just like when you created a file, the git add command doesn't return anything when you enter it correctly.

$ git add locations.txt

The git add command moves changes from the working directory to the Git staging area. The staging area is where you prepare a snapshot of a set of changes before committing them to the official history.



1. Check the status of the file.

$ git status

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: locations.txt

Now you can see the new file has been added (staged) and you can commit it when you are ready. The git status command displays the state of the working directory and the staged snapshot.

1. Issue the [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) command with a commit message, as shown on the next line. The -m indicates that a commit message follows.

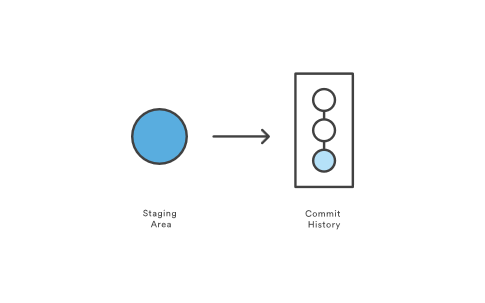
$ git commit -m 'Initial commit'

[master (root-commit) fedc3d3] Initial commit

1 file changed, 1 insertion(+)

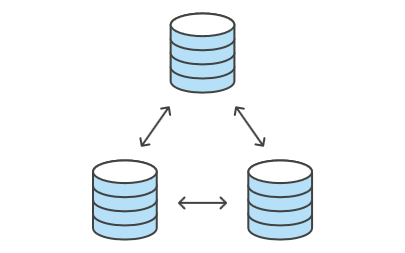
Create mode 100644 locations.txt

The git commit takes the staged snapshot and commits it to the project history. Combined with git add, this process defines the basic workflow for all Git users.



Up until this point, everything you have done is on your local system and invisible to your Bitbucket repository until you push those changes.

* + *Learn a bit more about Git and remote repositories*
    - Git's ability to communicate with remote repositories (in your case, Bitbucket is the remote repository) is the foundation of every Git-based collaboration workflow.
    - Git's collaboration model gives every developer their own copy of the repository, complete with its own local history and branch structure. Users typically need to share a series of commits rather than a single changeset. Instead of committing a changeset from a working copy to the central repository, Git lets you share entire branches between repositories.



* + - You manage connections with other repositories and publish local history by "pushing" branches to other repositories. You see what others have contributed by "pulling" branches into your local repository.

1. Go back to your local terminal window and send your committed changes to Bitbucket using [git push origin master](https://www.atlassian.com/git/tutorials/syncing/git-push). This command specifies that you are pushing to the master branch (the branch on Bitbucket) on origin (the Bitbucket server).

You should see something like the following response:

$ git push origin master

Couting objects: 3, done.

Writing objects: 100% (3/3), 253 bytes | 0 bytes/s, done.

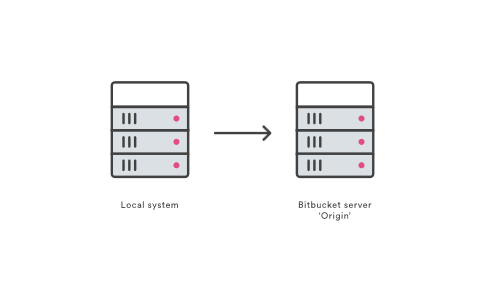
Total 3 (delta 0), reused 0 (delta 0) To

https://emmap1@bitbucket.org/emmap1/bitbucketstationlocations.git

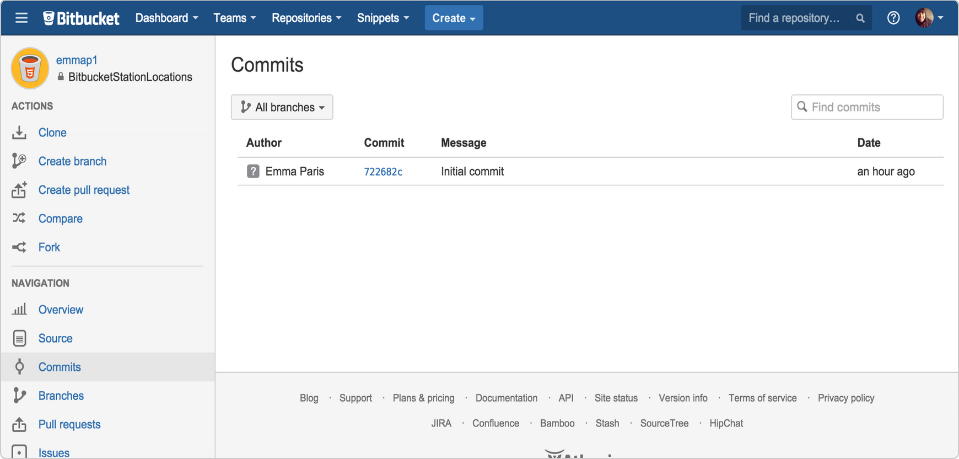
\* [new branch] master -> master

Branch master set up to track remote branch master from origin.

Your commits are now on the remote repository (origin).

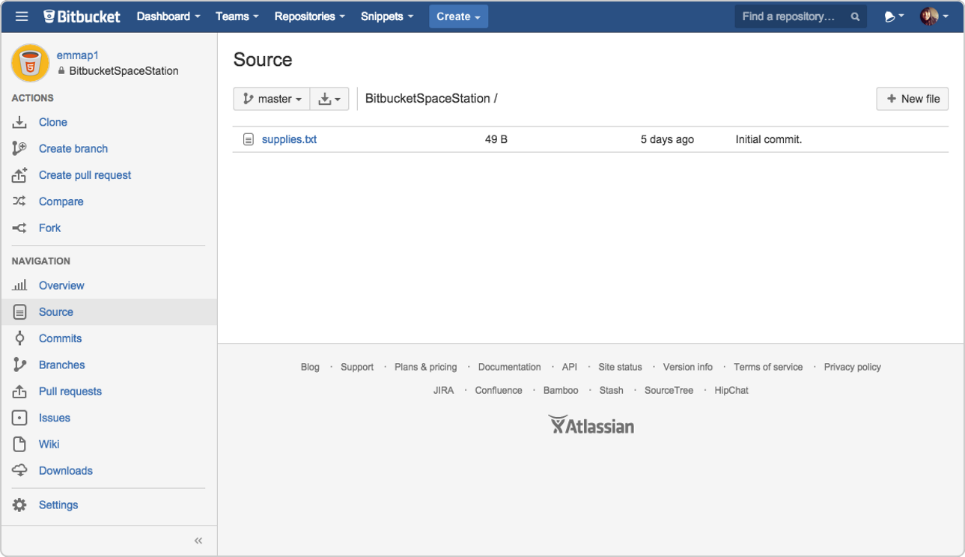


1. Go to your BitbucketStationLocations repository on Bitbucket and click the Commits item on the menu bar.
2. You should see a single commit on your repository. Bitbucket combines all the things you just did into that commit and shows it to you. You can see that the Author column shows the value you used when you configured the Git global file ( ~/.gitconfig).



1. Click the Source option.

You should see that you have a single source file in your repository, the locations.txt file you just added



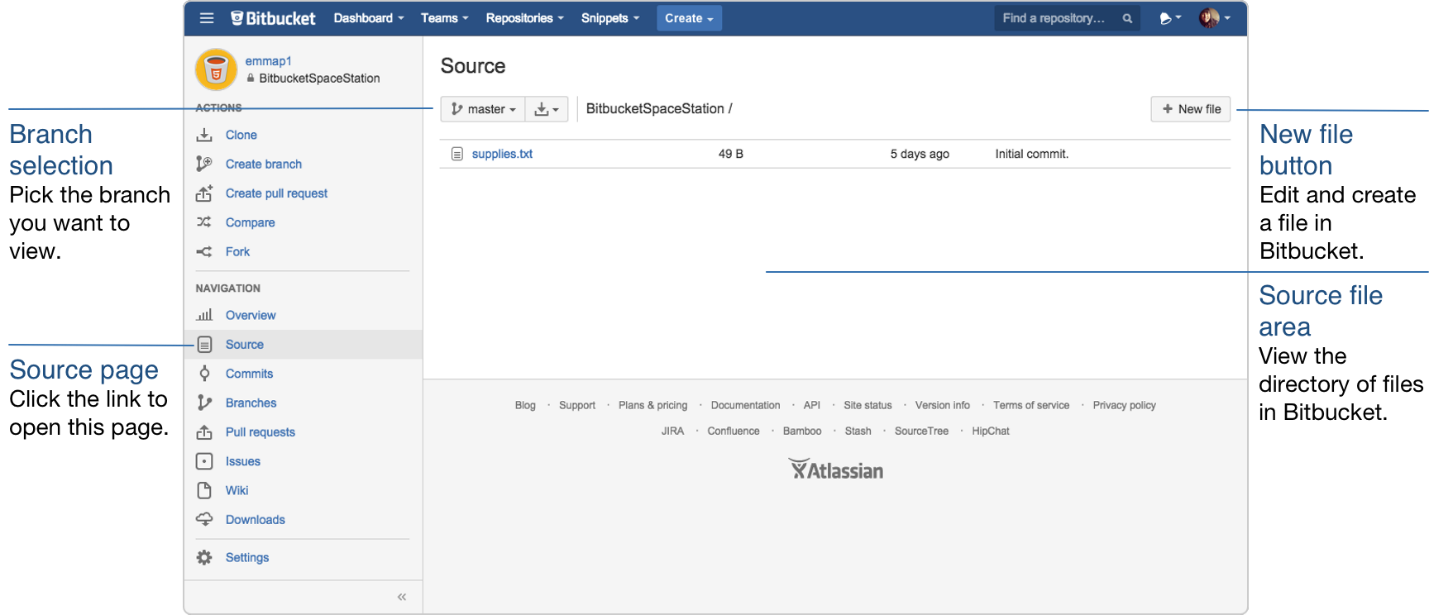
## Pull Changes from Git repository on Bitbucket Cloud

Next on your list of space station administrator activities, you need a file with more details about your locations. Since you don't have many locations at the moment, you are going to add them right from Bitbucket.

**Step 1. Create a file in Bitbucket**

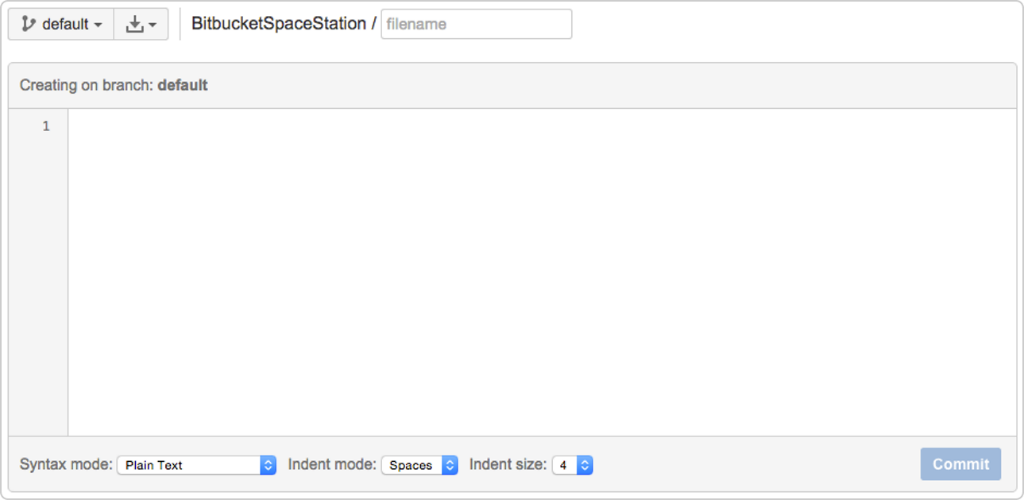
To add your new locations file, do the following:

1. From your **BitbucketStationLocations** repository, click **Source** to open the source directory. Notice you only have one file, locations.txt, in your directory.



1. From the Source page, click New file in the top right corner. This button only appears after you have added at least one file to the repository.

A page for creating the new file opens, as shown in the following image.



1. Enter stationlocations in the **filename** field.
2. Select **HTML** from the **Syntax** **mode** list.
3. Add the following HTML code into the text box:

<p>Bitbucket has the following space stations:</p>

<p>

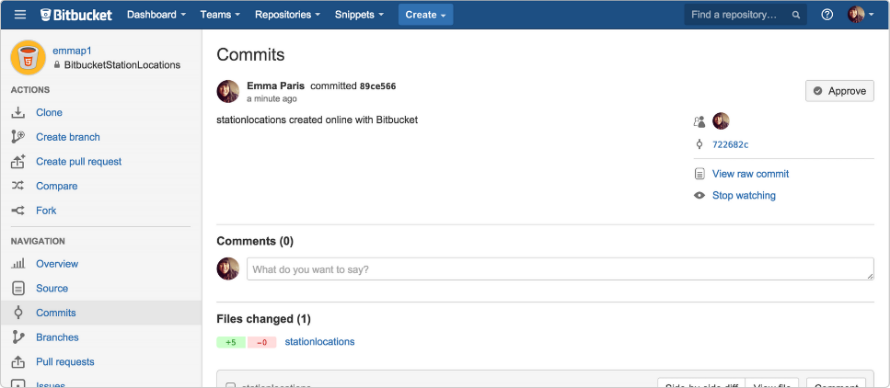
<b>Earth's Moon</b><br>

Headquarters

</p>

1. Click Commit. The Commit message field appears with the message: stationlocations created online with Bitbucket.
2. Click Commit under the message field.

You now have a new file in Bitbucket! You are taken to a page with details of the commit, where you can see the change you just made:



**Step 2. Pull changes from a remote repository**

Now we need to get that new file into your local repository. The process is straight forward, basically just the reverse of the push you used to get the locations.txt file into Bitbucket.

To pull the file into your local repository, do the following:

1. Open your terminal window and navigate to the top level of your local repository.

$ cd ~/repos/bitbucketstationlocations/

1. Enter the git pull --all command to pull all the changes from Bitbucket. (In more complex branching workflows, pulling and merging all changes might not be appropriate.) Enter your Bitbucket password when asked for it. Your terminal should look similar to the following:

$ git pull --all

Fetching origin

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 3 (delta 0), reused 0 (delta 0)

Unpacking objects: 100% (3/3), done.

From https://bitbucket.org/emmap1/bitbucketstationlocations

fe5a280..fcbeeb0 master -> origin/master

Updating fe5a280..fcbeeb0

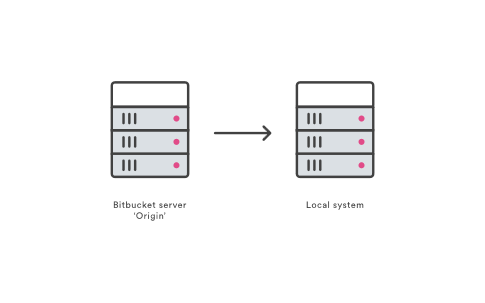
Fast-forward

stationlocations | 5 ++++++++++++++

1 file changed, 5 insertions(+)

create mode 100644 stationlocations

The [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) command merges the file from your remote repository (Bitbucket) into your local repository with a single command.

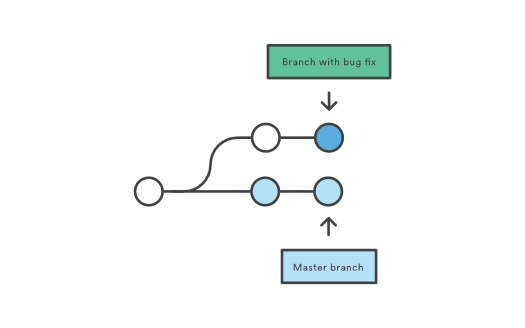


1. Navigate to your repository folder on your local system and you'll see the file you just added.

## Use a Git Branch to Merge a File

Learning branches will allow you to update your files and only share the information when you're ready. Branches are most powerful when you're working on a team. You can work on your own part of a project from your own branch, pull updates from Bitbucket, and then merge all your work into the main branch when it's ready.

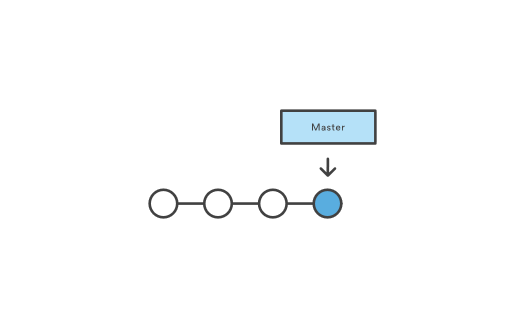
A branch represents an independent line of development for your repository. Think of it as a brand-new working directory, staging area, and project history. Before you create any new branches, you automatically start out with the main branch (called master). For a visual example, this diagram shows the master branch and the other branch with a bug fix update.



**Step 1. Create a branch and make a change**

Create a branch where you can add future plans for the space station that you aren't ready to commit. When you are ready to make those plans known to all, you can merge the changes into your Bitbucket repository and then delete the no-longer-needed branch.

It's important to understand that branches are just pointers to commits. When you create a branch, all Git needs to do is create a new pointer—it doesn’t create a whole new set of files or folders. Before you begin, your repository looks like this:



To create a branch, do the following:

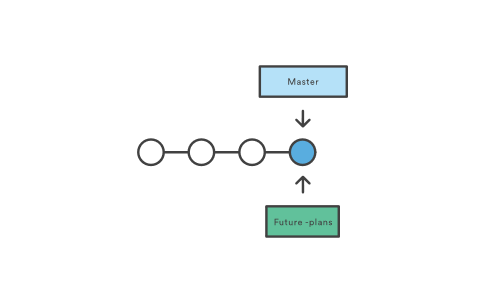
1. Go to your terminal window and navigate to the top level of your local repository using the following command:

$ cd ~/repos/bitbucketstationlocations/

1. Create a branch from your terminal window.

$ git branch future-plans

This command creates a branch but does not switch you to that branch, so your repository looks something like this:



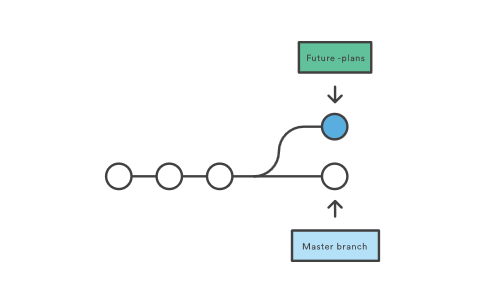
The repository history remains unchanged. All you get is a new pointer to the current branch. To begin working on the new branch, you have to check out the branch you want to use.

1. Checkout the new branch you just created to start using it.

$ git checkout future-plans

Switched to branch 'future-plans'

The git checkout command works hand-in-hand with git branch. Because you are creating a branch to work on something new, every time you create a new branch (with git branch), you want to make sure to check it out (with git checkout) if you're going to use it. Now that you’ve checked out the new branch, your Git workflow looks something like this:



1. Search for the bitbucketstationlocations folder on your local system and open it. You will notice there are no extra files or folders in the directory as a result of the new branch.
2. Open the stationlocations file using a text editor.
3. Make a change to the file by adding another station location:

<p>Bitbucket has the following space stations:</p>

<p>

<b>Earth's Moon</b><br>

Headquarters

</p>

<p>

<b>Mars</b><br>

Recreation Department

</p>

1. Save and close the file.
2. Enter git status in the terminal window. You will see something like this:

$ git status

On branch future-plans

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: stationlocations

no changes added to commit (use "git add" and/or "git commit -a")

Notice the On branch future-plans line? If you entered git status previously, the line was on branch master because you only had the one master branch. Before you stage or commit a change, always check this line to make sure the branch where you want to add the change is checked out.

1. Stage your file.

$ git add stationlocations

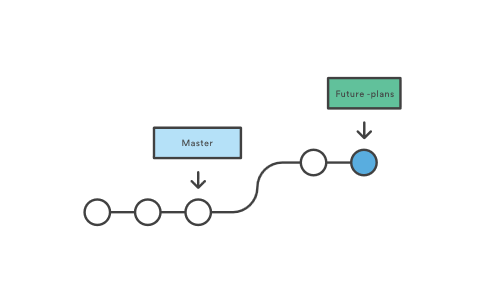
1. Enter the git commit command in the terminal window, as shown with the following:

$ git commit stationlocations -m 'making a change in a branch'

[future-plans e3b7732] making a change in a branch

1 file changed, 4 insertions(+)

With this recent commit, your repository looks something like this:

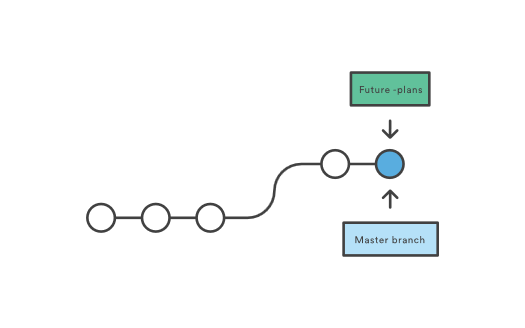


Now it's time to merge the change that you just made back into the master branch.

**Step 2. Merge your branch: fast-forward merging**

Now you can merge your future-plans branch into the main branch on your local system. Because you created only one branch and made one change, use the fast-forward branch method to merge. You can do a fast-forward merge because you have a linear path from the current branch tip to the target branch.

Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast-forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one.



This branch workflow is common for short-lived topic branches with smaller changes and are not as common for longer-running features.

To complete a fast-forward merge do the following:

1. Go to your terminal window and navigate to the top level of your local repository.

$ cd ~/repos/bitbucketstationlocations/

1. Enter the git status command to be sure you have all your changes committed and find out what branch you have checked out.

$ git status

On branch future-plans

nothing to commit, working directory clean

1. Switch to the master branch.

$ git checkout master

Switched to branch 'master'

Your branch is up-to-date with 'origin/master'.

1. Merge changes from the future-plans branch into the master branch. It will look something like this:

$ git merge future-plans

Updating fcbeeb0..e3b7732

Fast-forward

stationlocations | 4 ++++

1 file changed, 4 insertions(+)

You've essentially moved the pointer for the master branch forward to the current head and your repository looks something like the fast forward merge above.

1. Because you don't plan on using future-plans anymore, you can delete the branch.

$ git branch -d future-plans

Deleted branch future-plans (was e3b7732).

When you delete future-plans, you can still access the branch from master using a commit id. For example, if you want to undo the changes added from future-plans, use the commit id you just received to go back to that branch.

1. Enter git status to see the results of your merge, which show that your local repository is one ahead of your remote repository. It will look something like this:

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

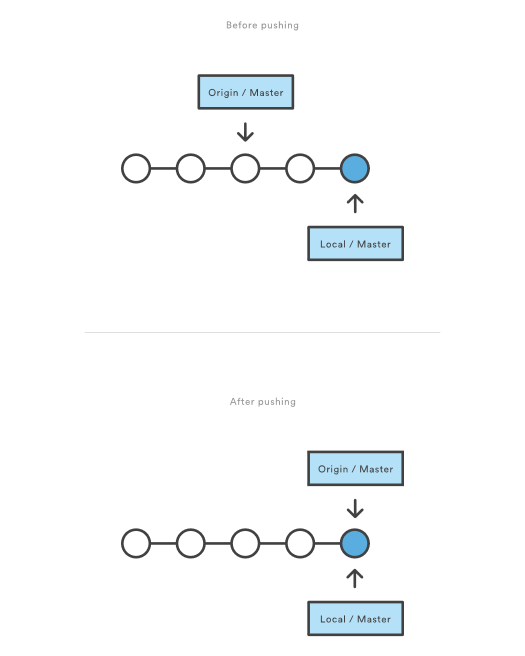
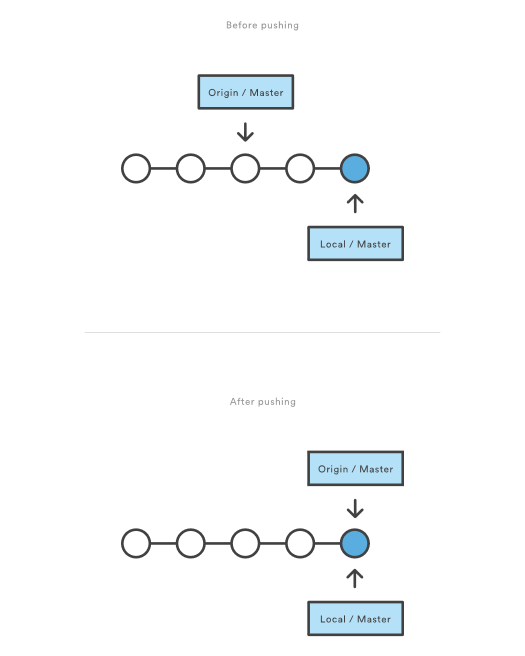
(use "git push" to publish your local commits)

nothing to commit, working directory clean

**Step 3. Push your change to Bitbucket**

You want to make it possible for everyone else to see the location of the new space station. To do so, you can push the current state of your local repository to Bitbucket.

This diagram shows what happens when your local repository has changes that the central repository does not have and you push those changes to Bitbucket.



Here's how to push your change to the remote repository:

1. From the repository directory in your terminal window, enter git push origin master to push the changes. It will result in something like this:

$ git push origin master

Counting objects: 3, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 401 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://emmap1@bitbucket.org/emmap1/bitbucketstationlocations.git

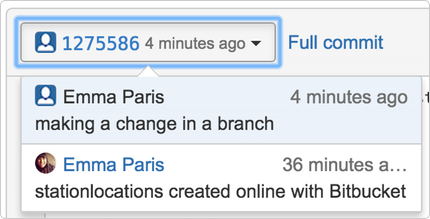
fcbeeb0..e3b7732 master -> master

1. Click the **Overview** page of your Bitbucket repository, and notice you can see your push in the **Recent** **Activity** **stream**.
2. Click **Commits** and you can see the commit you made on your local system. Notice that the change keeps the same commit id as it had on your local system.



You can also see that the line to the left of the commits list has a straight-forward path and shows no branches. That’s because the future-plans branch never interacted with the remote repository, only the change we created and committed.

1. Click **Branches** and notice that the page has no record of the branch either.
2. Click **Source**, and then click the stationlocations file. You can see the last change to the file has the commit id you just pushed.
3. Click the file history list to see the changes committed for this file, which will look similar to the following figure.



# Code Review in Bitbucket Cloud

When you work with another user's public Bitbucket repository, typically you have read access to the code but not write access. This is where the concept of forking comes in. Here's how it works:

1. Fork the repository to copy it to your own account.
2. Clone the forked repository from Bitbucket to your local system.
3. Make changes to the local repository.
4. Push the changes to your forked repository on Bitbucket.
5. Create a pull request from the original repository you forked to add the changes you made.
6. Wait for the repository owner to accept or reject your changes.

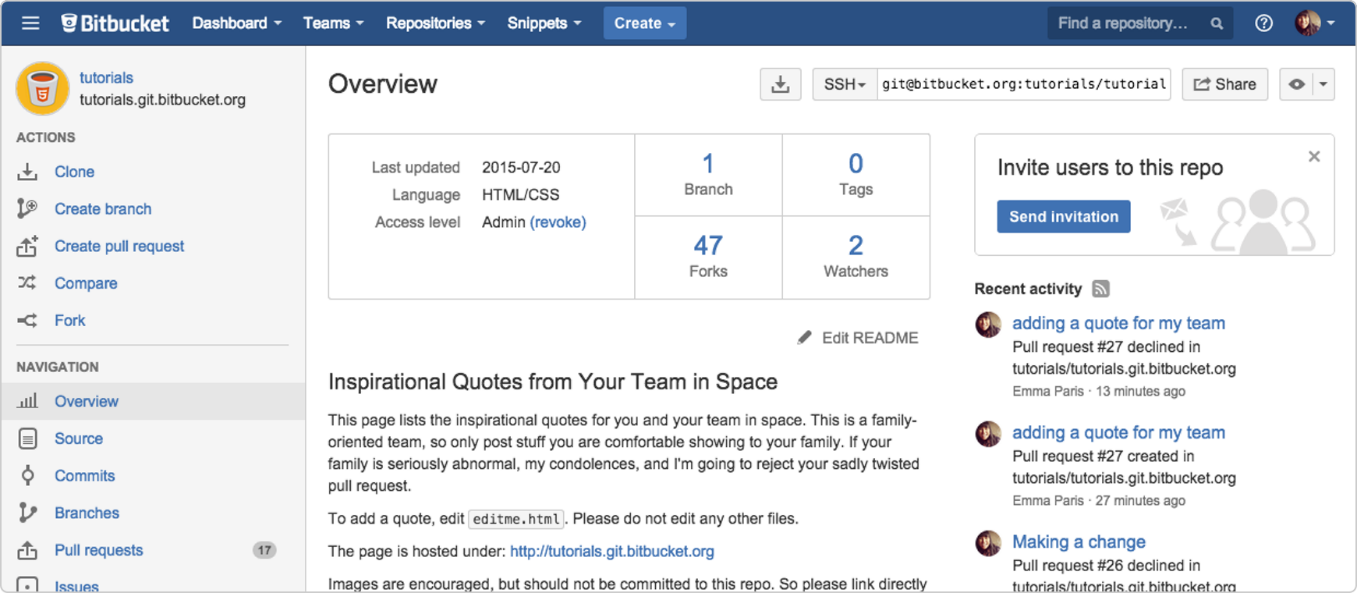
If a repository owner accepts the pull request, Bitbucket merges your code changes into the original repository. It is recommended that you work with forks and pull requests, even if the repository owner gives you write access to a public repository. While a pull is part of the Git and Mercurial workflow, pull requests and forks are concepts used only by repository hosting services — like Bitbucket.

*Fork* is another way of saving a clone or copy. The term fork (in programming) derives from a Unix system call that creates a copy of an existing process. So, unlike a branch, a fork is independent from the original repository. If the original repository is deleted, the fork remains. If you fork a repository, you get that repository and all of its branches.

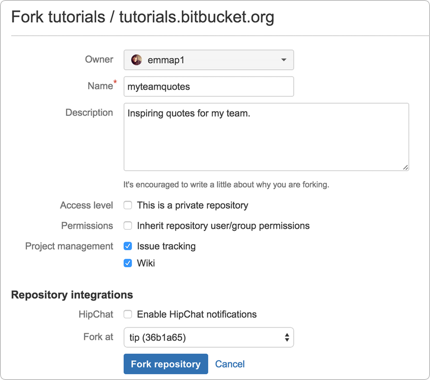
## Fork a Teammate's Repository

In this example, you'll fork a public repository belonging to a user called tutorials.

1. Depending on which DVCS tool you are using, use the search field in the top right corner of Bitbucket to find one of the repository.



1. From the repository you open, click **Fork** on the left side of the screen. The system displays the fork page.



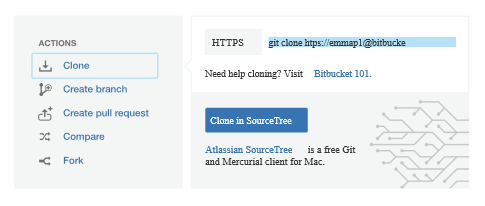
1. Change the **Name** for example, to myteamquotes.
2. In the **Description** field, enter Inspiring quotes for my team.
3. Uncheck **Inherit repository user/group permissions**.
4. Press **Fork** **repository**.

## Copy Fork and Make a Change to the Repository

Next step is to get the copy of the forked repository to your local system so that you can add it.

**Step 1. Clone your fork to your local system**

1. From the **Overview** page of the forked repository in Bitbucket, click **Clone** on the left side. The system displays a pop-up clone dialog. By default, the clone dialog sets the protocol to **HTTPS** unless you have already set up SSH for Git.



1. From the pop-up clone dialog, copy the highlighted clone command.
2. From your terminal window, change the directory to your repositories directory.

$ cd ~/repos

1. Paste the command you copied from Bitbucket onto the command line and press **Return**.
2. Enter your Bitbucket password when the terminal asks for it. This is the password you entered when you created your Bitbucket account. If you created an account by linking to Google or Facebook, use your password for that account. At this point, your terminal window should look similar to this:

$ cd ~/repos

$ git clone https://emmap1@bitbucket.org/emmap1/myteamquotes.git

Cloning into 'myteamquotes'...

Password

remote: Counting objects: 15, done.

remote: Compressing objects: 100% (15/15), done.

remote: Total 15 (delta 4), reused 0 (delta 0)

Unpacking objects: 100% (15/15), done.

Checking connectivity... done.

You now have the forked repository on your local system.

**Step 2. Make a change to the repository source**

This repository contains a website that has an editme.html file. This file lists inspirational quotes for your space teammates. Now, it is your turn to record an inspirational quote... or just a favorite quote. Do the following to contribute to this repository:

1. Use Google or some other search engine to locate your favorite quote.
2. Navigate to the directory folder containing your repository files (something like ~/repos/myteamquotes).
3. Open the editme.html file with a text editor.
4. Go ahead and add a quote of your choosing. You can add an image link to your quote if you like, just place it above the <blockquote> tag.

*Here is a sample of what an addition will look like:*

<!-- Please don't edit above or below this line.

To add a quote, copy the div below and paste it at the bottom of the file.

Then populate it with your quote and attribution.

<div class="quote">

<blockquote>Your quote goes here</blockquote>

<cite>Attribution</cite>

</div>

You can add an optional image tag. The image must be linked from an external

site and not be in the repository.

<img src="http://your\_image\_link.jpg">

<!--Please don't edit above this line.>

<div class="quote">

<img src="http://upload.wikimedia.org/wikipedia/commons/thumb/b/b9/Steve\_Jobs\_Headshot\_2010-CROP.jpg/245px-Steve\_Jobs\_Headshot\_2010-CROP.jpg">

<blockquote>Put a dent in the universe.</blockquote>

<cite>Steve Jobs</cite>

</div>

If you are not sure what to do, you can copy the example tags at the top of the file, paste them just below the last quote on the page, and modify them with your quote, as shown in the preceding example.

1. Save and close the file.
2. From your terminal window, change the directory to myteamquotes.

$ cd ~/repos/myteamquotes

1. Display the status of the repository with git status.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: editme.html

no changes added to commit (use "git add" and/or "git commit -a")

If you added an image file, you will see that file as well.

1. Add your changes locally with git add:

$ git add editme.html

1. Commit your changes locally with git commit:

$ git commit -m "Making a change"

[master 83bc100] Making a change

1 file changed, 10 insertions(+), 5 deletions(-)

1. Enter git push to push the changes to your Bitbucket fork, and enter your password to finish pushing changes.

$ git push origin master

Counting objects: 3, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.

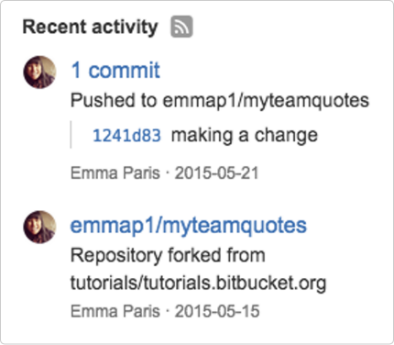
Total 3 (delta 2), reused 0 (delta 0)

To https://emmap1@bitbucket.org/emmap1/myteamquotes.git

7479b42..83bc100 master -> master

Branch master set up to track remote branch master from origin.

1. Log into your Bitbucket repository and notice you can see your push in the activity stream.



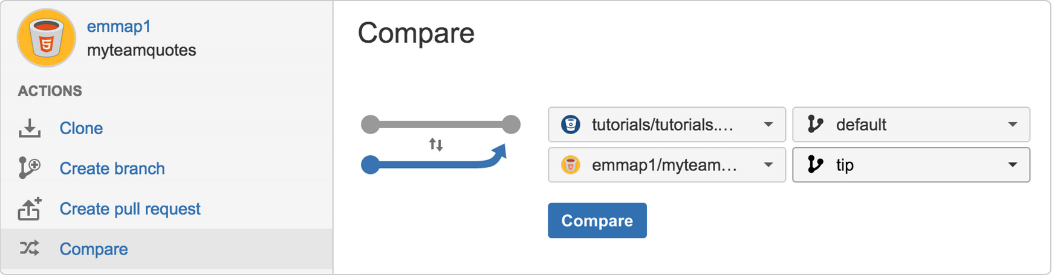
## Create a Pull Request

To request your change to get added to the original repository. To do that, you create a pull request.

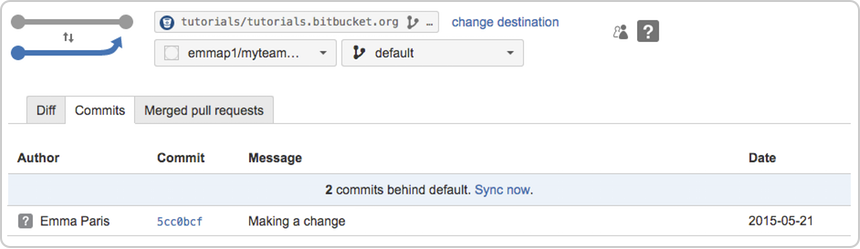
**Step 1. Compare your fork to the original**

While you were working on adding your quote, another teammate might have made updates to the original repository. At this point, you can check that and decide if you need to adjust your fork accordingly. Do the following to compare your changes with the repository:

1. Log into Bitbucket and navigate to your myteamquotes repository. Forked repositories have a special widget that lets you compare your fork work to the original.
2. Click the **Compare** link on the left side of the page. The **Compare** section has a specialized view that is available only in forked repositories.



1. Press the **Compare** button to compare your forked repository to the original repository.



1. If someone has made changes to the original repository since you forked it, you will see that your forked repository is one or more commits behind the original. If that is the case, to update your repository, click **Sync now** and then **Sync** on the popup that appears. If you want to see the differences between your current repository and the original repository, click the **Diff** tab to compare changes. If there are multiple commits, you see their cumulative changes by file in this section. Click the **Side-by-side diff** button to see changes displayed side-by-side. Or press the **View file** button to view the full file in Bitbucket.

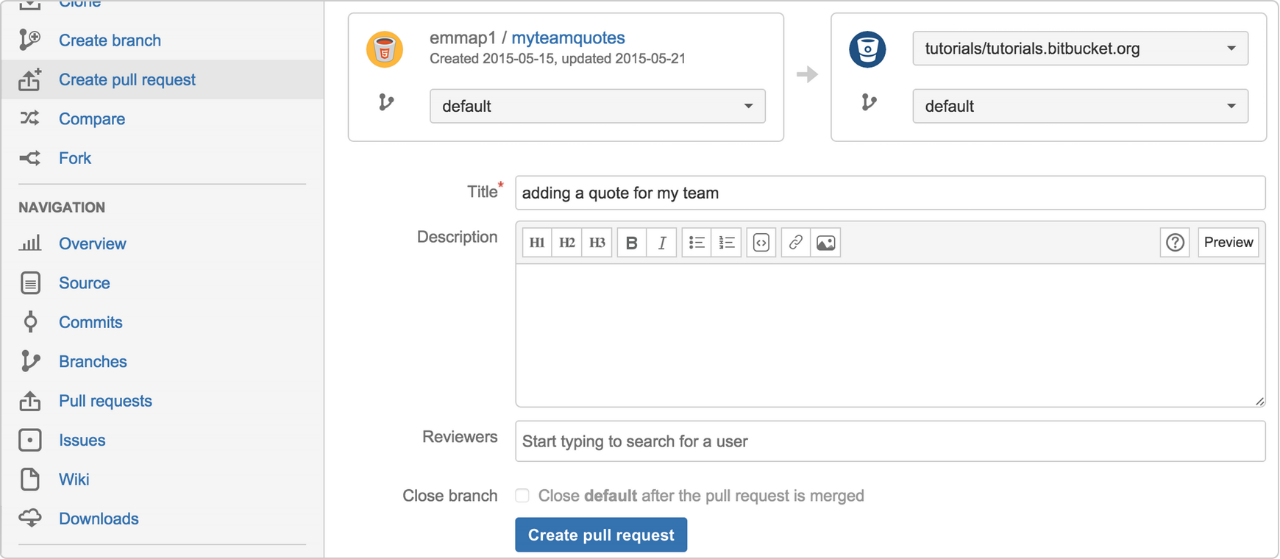


1. Switch back to the **Commits** tab to see the list of commits pushed from your local repository to the fork in Bitbucket. To see the contents of a specific commit in isolation, click a **Commit** link and Bitbucket takes you to the **Commits** page.

**Step 2. Create a pull request**

Now it's time to request that your quote get added to the original repository for all to see! From your myteamquotes repository in Bitbucket, do the following:

1. Press **Create Pull Request**. The system displays the request form.
2. Complete the form. For your purposes today, you only have to add a **Title**. When you are done, it will look something like this:

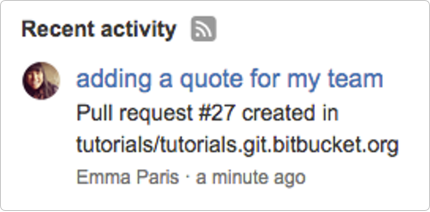


1. Press **Create** **pull** **request**. The system opens your latest request on the **Pull** **Request** page of the original repository in the tutorials account. To see the list of all the pull requests against this repo, click the **Pull** **Request** tab.

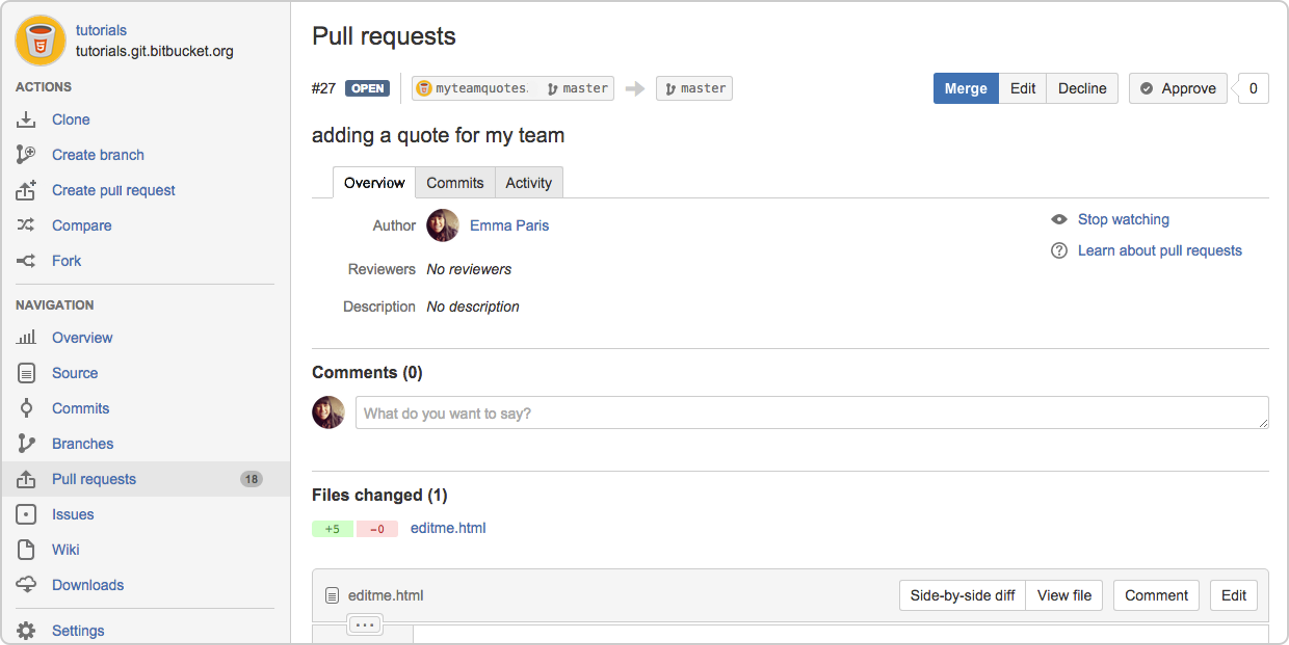
After you create a pull request, you can't delete it. If you delete your fork after you make a request, the receiver can only decline your request because the repository to pull from is gone.

**Step 3. Learn what happens to your pull request**

You have to wait for the repository owner to accept your pull request. When the original repository owner logs into Bitbucket, that user's newsfeed shows your pull request and your fork from a few days earlier, as shown in the following image.



When the repo owner clicks on your pull request, that user can **Merge**, **Edit**, or **Decline** it. In addition to owners, anyone with access to the repository can **Approve** the request, which means that person reviewed the changes in the pull request.



# Branching with Bitbucket Cloud

## Create a Branch and Change using the Branching Workflow

1. Create a branch using the git branch command.

$ git branch test-1

1. Check out the branch you just created using the git checkout command.

$ git checkout test-1

Switched to branch 'test-1'

1. List the branches you have locally using the git branch command.

$ git branch

  master

  \* test-1

1. Make an update to the editme.html file by adding a quote. You can use something like the following:

<div class="quote">

  <blockquote>This is a quote, and I like it.</blockquote>

  <cite>A quote: The Art of Quoting</cite>

</div>

1. Add that change.

git add editme.html

Note: your change isn't committed to the Git history yet it's in a "waiting" state. We learned about this in [Saving changes](https://www.atlassian.com/git/tutorials/saving-changes).

1. Commit the change with a descriptive commit message.

git commit editme.html -m'added a new quote'

[test-1 063b772] added a new quote

1 file changed, 3 insertions(+), 3 deletions(-)

Note: now the changes is part of the Git history as a single "commit" We learned about this in [Saving changes](https://www.atlassian.com/git/tutorials/saving-changes).

1. Push that change to Bitbucket using the git push command.

git push

fatal: The current branch test-1 has no upstream branch.

To push the current branch and set the remote as upstream, use

  git push --set-upstream origin test-1

You will see an error because the first time you push a new branch you created locally you have to designate that branch.

1. Push the branch and change using the git push branch command.

$ git push origin test-1

Counting objects: 3, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 363 bytes | 0 bytes/s, done.

Total 3 (delta 2), reused 0 (delta 0)

remote:

remote: Create pull request for test-1:

remote: https://bitbucket.org/dstevenstest/dans.git.bitbucket.org/pull-requests/new?source=test-1&t=1

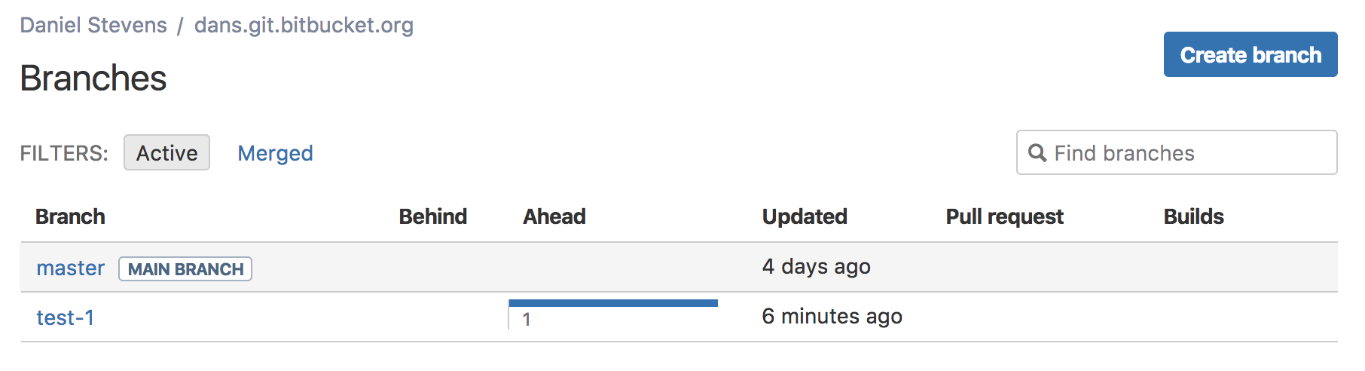
remote:

To https://bitbucket.org/dstevenstest/dans.git.bitbucket.org.git

\* [new branch] test-1 -> test-1

This tells the system that the origin repository is the destination of this new branch.

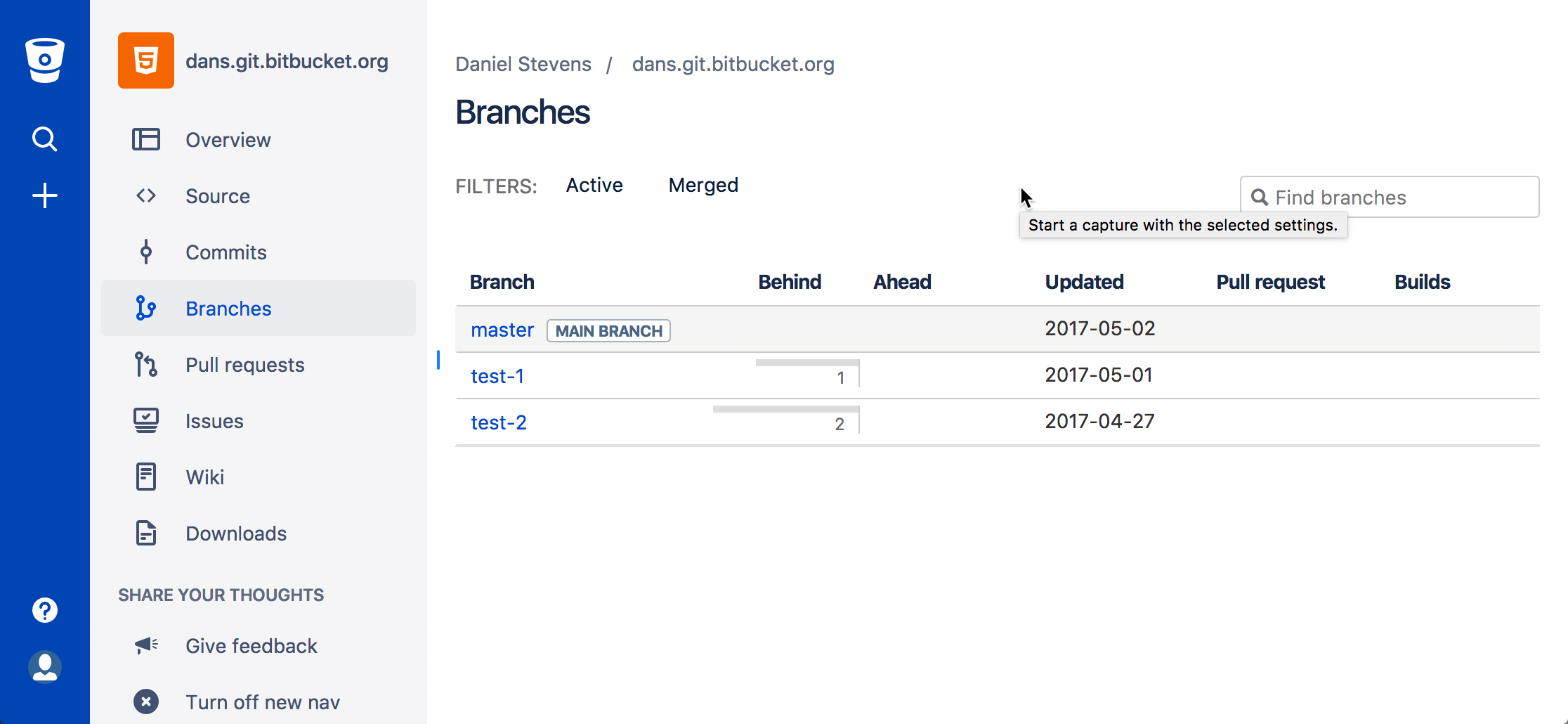
1. Open your tutorial repository and click Branches. You should now see both the master and the test-1 branches. It should look something like this:



## Create, Fetch, and Checkout a Remote Branch

When you're working in a team you'll likely have to pull or fetch branches which other team members create and push to Bitbucket. This example will give you some of the basics of creating and working with branches others create.

1. Go to your tutorial repository in Bitbucket and click **Branches**. You should see something like



1. Click Create branch, name the branch test-2, and click Create.
2. Copy the git fetch command in the check out your branch dialog. It will probably look something like this:

$ git fetch && git checkout test-2

From https://bitbucket.org/dstevenstest/dans.git.bitbucket.org

\* [new branch] test-2 -> origin/test-2

Branch test-2 set up to track remote branch test-2 from origin.

Switched to a new branch 'test-2'

1. Use the git branch command in your terminal. You should see a list of branches something like this:

$ git branch

  master

  test-1

\* test-2

The branch with the asterisk \* is the active branch. This is critical to remember when you are working in any branching workflow.

1. Use the git status command and you'll see something like this:

$ git status

On branch test-2

Your branch is up-to-date with 'origin/test-2'.

nothing to commit, working tree clean

You can see what branch you're on and that the branch is currently up to date with your remote (origin) branch.

1. Use the git checkout command to change the focus back to your other branch. The command will look something like this:

$ git checkout test-1

Switched to branch 'test-1'

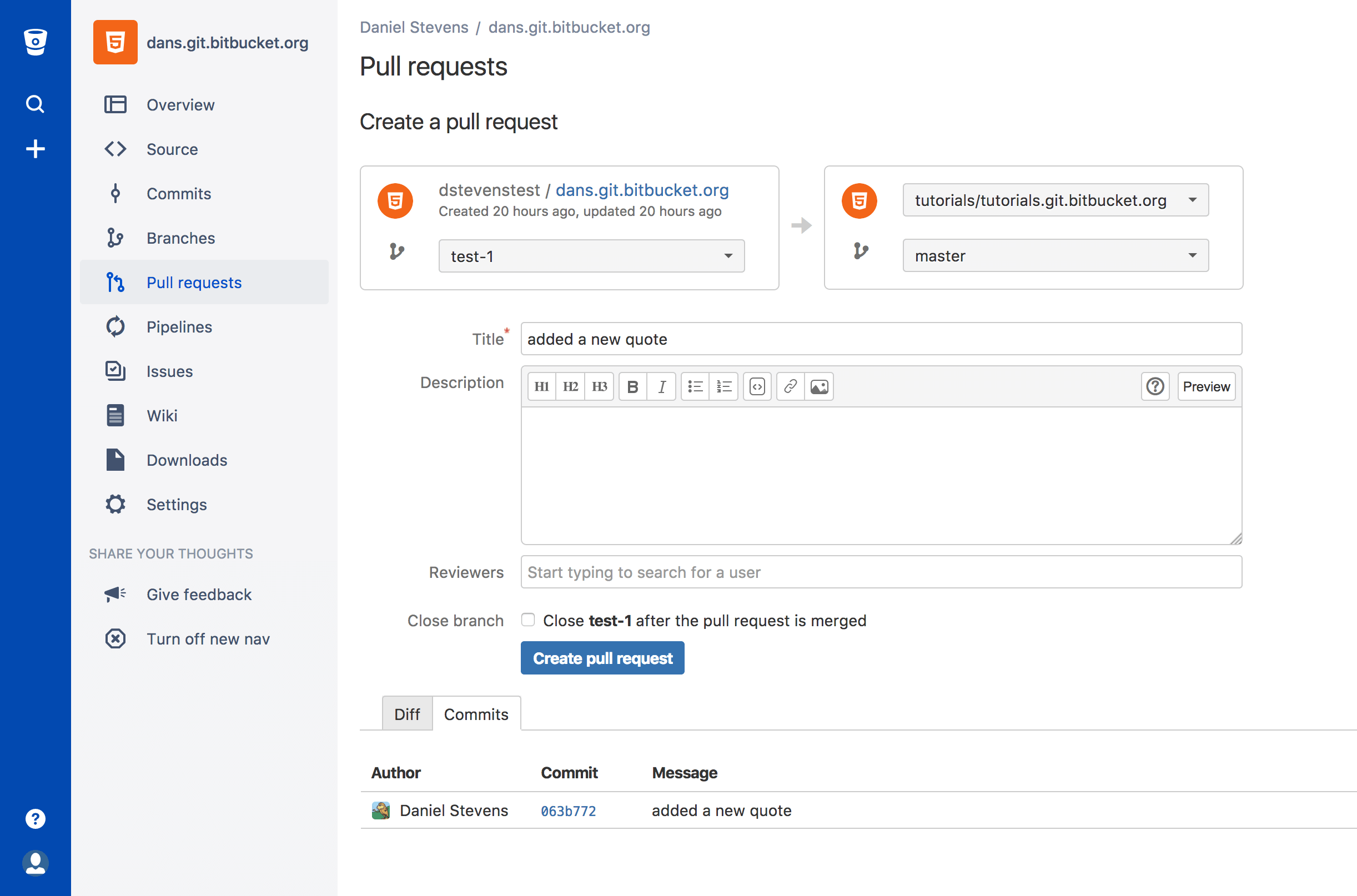
Your branch is ahead of 'origin/test-1' by 3 commits.

(use "git push" to publish your local commits)

One of the most important things to remember when working in branches is that you want to be sure the branch you're making changes to is the correct branch.

## Push Change and Create a Pull Request

1. Click **+> Create a pull request**. You can see your test-1 branch as the source branch and master in the destination branch.



1. Click **Create pull request**.
2. Make a comment in the pull request by selecting a line in the diff (the area displaying the change you made to the editme.html file).
3. Click **Approve** in the top left of the page. Of course, in a real pull request you'd have reviewers making comments
4. Click **Merge**.
5. (Optional) Update the **Commit message** with more details.
6. Select the **Merge Commit** Merge strategy from the two options:

* **Merge commit** — Keeps all commits from your source branch and makes them part of the destination branch. This option is the same as entering git merge --no-ff in the command line.
* **Squash** — Combines your commits when you merge the source branch into the destination branch. This option is the same as entering git merge --squash in the command line.

1. Click **Commits** and you will see how the branch you just merged fits into the larger scheme of changes.

## Delete a Branch and Pull Master into Local Working Branch

Now you've gone through the basic branching workflow and your change is in master. The last thing we'll learn is how to delete the branch you just merged, pull the updated master branch, and merge the updated master branch into your test-2 branch.

1. Open your terminal and run the git status command the result should look something like this:

$ git status

On branch test-1

nothing to commit, working tree clean

You can see you're on the branch you just used to make your change and that you don't have any changes. We're ready to get rid of that branch now that we've finished that work.

1. Switch to the master branch by running the git checkout master command. The result should look something like this:

git checkout master

Switched to branch 'master'

Your branch is up-to-date with 'origin/master'.

Notice that the message says you are up-to-date? This is only your local branch. We know this because we just merged a change into master and haven't pulled that change from the remote repository to our local system. That's what we'll do next.

1. Run the git pull command. The result should look something like this:

$ git pull

remote: Counting objects: 1, done.

remote: Total 1 (delta 0), reused 0 (delta 0)

Unpacking objects: 100% (1/1), done.

From https://bitbucket.org/dstevenstest/dans.git.bitbucket.org

2d4c0ab..dd424cb master -> origin/master

Updating 2d4c0ab..dd424cb

Fast-forward

editme.html | 6 +++---

1 file changed, 3 insertions(+), 3 deletions(-)

What happened is that when you pull the changes from the remote repository git runs a fast-forward merge to integrate the changes you made. It also lists how many files and lines in that file changed.

1. Run the git branch -d {branch\_name} command to remove the test-1 branch. The result will look something like this:

$ git branch -d test-1

Deleted branch test-1 (was 063b772)

You can see that it deleted the branch and what the last commit hash was for that branch. This is the safe way to delete a branch because git won't allow you to delete the branch if it has uncommitted changes. You should be aware however that this won't prevent deleting changes which are committed to the git history but not merged into another branch.

1. Switch to the test-2 branch using the git checkout command.

$ git checkout test-2

Switched to branch 'test-2'

Your branch is up-to-date with 'origin/test-2'.

1. Merge the master branch into your working branch using the git merge master test-2 command. The result will look something like this:

$ git merge master test-2

Updating 2d4c0ab..dd424cb

Fast-forward

editme.html | 6 +++---

1 file changed, 3 insertions(+), 3 deletions(-)

It's important to remember the following:

* The active branch matters. If you want to merge master into test-2 you want to have test-2 checked out (active). The same is true if you want to merge test-2 into master you need to have master checked out.
* To see what branch is active at any time use git branch and the active branch will have an asterisk or use git status and it will tell you want branch you are on and if there are pending local changes.