Tips & Tricks

This module presents a broad survey of useful Git utilities. We'll take a step back from the theoretical aspects of Git and focus on common tasks like preparing a project for release and backing up a repository. While working through this module, your goal shouldn't be to master all of these miscellaneous tools, but rather to understand why they were created and when they might come in handy.

Download the repositories for this module

If you've been following along from the previous module, you already have everything you need. Otherwise, download the zipped Git repositories from the above link, uncompress them, and you're good to go.

Archive The Repository

First, let's export our repository into a ZIP archive. Run the following command in your local copy of `my-git-repo`.

```
git archive master --format=zip --output=../website-12-10-2012.zip
```

Or, for Unix users that would prefer a tarball:

```
git archive master --format=tar --output=../website-12-10-2012.tar
```

This takes the current `master` branch and places all of its files into a ZIP archive (or a tarball), omitting the `.git` directory. Removing the `.git` directory removes all version control information, and you're left with a single snapshot of your project.

You can send the resulting archive to a client for review, even if they don't have Git installed on their machine. This is also an easy way to create Git-independent backups of important revisions, which is always a good idea.

Bundle the Repository

Similar to the `git archive` command, `git bundle` turns a repository into a single file. However, in this case, the file retains the versioning information of the entire project. Try running the following command.

```
git bundle create ../repo.bundle master
```

It's like we just pushed our `master` branch to a remote, except it's contained in a file instead of a remote repository. We can even clone it using the same `git clone` command:

```
cd ..
git clone repo.bundle repo-copy -b master
cd repo-copy
git log
cd ../my-git-repo
```

The log output should show you the entire history of our `master` branch, and `repo.bundle` is also the `origin` remote for the new repository. This is the exact behavior we encountered when cloning a "normal" Git repository.

Bundles are a great way to backup entire Git repositories (not just an isolated snapshot like `git archive`). They also let you share changes without a network connection. For example, if you didn't want to configure the SSH accounts for a private Git server, you could bundle up the repository, put it on a jump drive, and walk it over to your co-worker's computer. Of course, this could become a bit tiresome for active projects.

We won't be needing the `repo.bundle` file and `repo-copy` folder, so go ahead and delete them now.

Ignore a File

Remember that Git doesn't automatically track files because we don't want to record generated files like C binaries or compiled Python modules. But, seeing these files under the "Untracked files" list in `git status` can get confusing for large projects, so Git lets us ignore content using a special text file called `.gitignore`. Each file or directory stored in `.gitignore` will be invisible to Git.

Let's see how this works by creating a file called `notes.txt` to store some personal (private) comments about the project. Add some text to it and save it, then run the following.

```
git status
```

As expected, this will show `notes.txt` in the "Untracked files" section. Next, create a file called `.gitignore` in the `my-git-repo` folder and add the following text to it. Windows users can create a file that starts with a period by executing the `touch .gitignore` command in Git Bash (you should also make sure hidden files are visible in your file browser).

```
notes.txt
```

Run another `git status` and you'll see that the notes file no longer appears under "Untracked files", but `.gitignore` does. This is a common file for Git-based projects, so let's add it to the repository.

```
git add .gitignore
git commit -m "Add .gitignore file"
git status
```

You can also specify entire directories in `.gitignore` or use the `*` wildcard to ignore files with a particular extension. For example, the following is a typical `.gitignore` file for a simple C project. It tells Git to overlook all `.o`, `.out`, and `.exe` files in the repository.

```
*.o
```

```
*.out
*.exe
```

Stash Uncommitted Changes

Next, we'll take a brief look at **stashing**, which conveniently "stashes" away uncommitted changes. Open up `style.css` and change the `h1` element to:

```
h1 {
  font-size: 32px;
  font-family: "Times New Roman", serif;
}
```

Now let's say we had to make an emergency fix to our project. We don't want to commit an unfinished feature, and we also don't want to lose our current CSS addition. The solution is to temporarily remove these changes with the `git stash` command:

```
git status
git stash
git status
```

Before the stash, `style.css` was listed as "Changed by not updated." The `git stash` command hid these changes, giving us a clean working directory. We're now able to switch to a new hotfix branch to make our important updates—without having to commit a meaningless snapshot just to save our current state.

Let's pretend we've completed our emergency update and we're ready to continue working on our CSS changes. We can retrieve our stashed content with the following command.

```
git stash apply
```

The `style.css` file now looks the same as it did before the stash, and we can continue development as if we were never interrupted. Whereas patches represent

a committed snapshot, a stash represents a set of *un*committed changes. It takes the uncommitted modifications, stores them internally, then does a `git reset --hard` to give us a clean working directory. This also means that stashes can be applied to *any* branch, not just the one from which it was created.

In addition to temporarily storing uncommitted changes, this makes stashing a simple way to transfer modifications between branches. So, for example, if you ever found yourself developing on the wrong branch, you could stash all your changes, checkout the correct branch, then run a `git stash apply`.

Let's undo these CSS updates before moving on.

```
git reset --hard
```

Hook into Git's Internals

Arguably, Git's most useful configuration options are its **hooks**. A hook is a script that Git executes every time a particular event occurs in a repository. In this section, we'll take a hands-on look at Git hooks by automatically publishing our website every time someone pushes to the `central-repo.git` repository.

In the `central-repo.git` directory, open the `hooks` directory and rename the file `post-update.sample` to `post-update`. After removing the `.sample` extension, this script will be executed whenever *any* branch gets pushed to `central-repo.git`. Replace the default contents of `post-update` with the following.

```
#!/bin/sh

# Output a friendly message
echo "Publishing master branch!" >&2

# Remove the old `my-website` directory (if necessary)
rm -rf ../my-website

# Create a new `my-website` directory
mkdir ../my-website

# Archive the `master` branch
```

```
git archive master --format=tar --output=../my-website.tar

# Uncompress the archive into the `my-website` directory
tar -xf ../my-website.tar -C ../my-website

exit 0
```

While shell scripts are outside the scope of this tutorial, the majority of commands in the above code listing should be familiar to you. In short, this new `post-update` script creates an archive of the `master` branch, then exports it into a directory called `my-website`. This is our "live" website.

We can see the script in action by pushing a branch to the `central-repo.git` repository.

```
git push ../central-repo.git master
```

After the central repository receives the new `master` branch, our `post-update` script is executed. You should see the `Publishing master branch!` message echoed from the script, along with a new `my-website` folder in the same directory as `my-git-repo`. You can open `index.html` in a web browser to verify that it contains all the files from our `master` branch, and you can also see the intermediate `.tar` archive produced by the hook.

This is a simple, unoptimized example, but Git hooks are infinitely versatile. Each of the `.sample` scripts in the `hooks` directory represents a different event that you can listen for, and each of them can do anything from automatically creating and publishing releases to enforcing a commit policy, making sure a project compiles, and of course, publishing websites (that means no more clunky FTP uploads). Hooks are even configured on a per-repository basis, which means you can run different scripts in your local repository than your central repository.

For a detailed description of the available hooks, please consult the official Git documentation.

View Diffs Between Commits

Up until now, we've been using `git log --stat` to view the changes introduced by

new commits. However, this doesn't show us which lines have been changed in any given file. For this level of detail, we need the `git diff` command. Let's take a look at the updates from the `Add a pink block of color` commit:

```
git diff HEAD~2..HEAD~1
```

This will output the diff between the `Add a pink block of color` commit (`HEAD~1`) and the one before it (`HEAD~2`):

```
index 828dd1a..2713b10 100644
--- a/pink.html
+++ b/pink.html
@@ -4,10 +4,17 @@
   <title>The Pink Page</title>
   <link rel="stylesheet" href="style.css" />
   <meta charset="utf-8" />
+  <style>
+    div {
+      width: 300px;
+      height: 50px;
+    }
+  </style>
 </head>
 <body>
   <h1 style="color: #F0F">The Pink Page</h1>
   <p>Only <span style="color: #F0F">real men</span> wear pink!</p>
+  <div style="background-color: #F0F"></div>

   <p><a href="index.html">Return to home page</a></p>
 </body>
```

This diff looks nearly identical to the patches we created in the previous module, and it shows exactly what was added to get from `HEAD~2` to `HEAD~1`. The `git diff` command is incredibly useful for pinpointing contributions from other developers. For example, we could have used the following to view the differences between John's `pink-page` branch and our `master` before merging it into the project in Distributed Workflows.

```
git diff master..john/pink-page
```

This flexible command can also generate a detailed view of our uncommitted changes. Open up `blue.html`, make any kind of change, and save the file. Then, run `git diff` without any arguments:

```
git diff
```

And, just in case we forgot what was added to the staging area, we can use the `--cached` flag to generate a diff between the staged snapshot and the most recent commit:

```
git add blue.html
git diff --cached
```

A plain old `git diff` won't output anything after `blue.html` is added to the staging area, but the changes are now visible through the `--cached` flag. These are the three main configurations of the `git diff` command.

Reset and Checkout Files

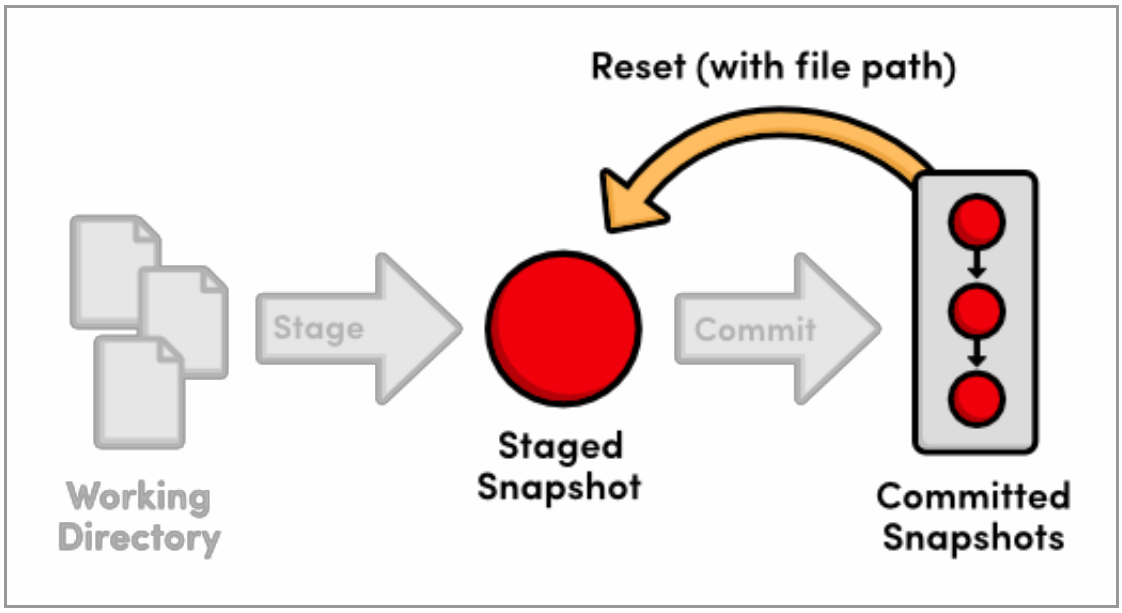
We've used `git reset` and `git checkout` many times throughout this tutorial; however, we've only seen them work with branches/commits. You can also reset and check out individual files, which slightly alters the behavior of both commands.

The `git reset` we're accustomed to moves the current branch to a new commit and optionally updates the working directory to match. But when we pass a file path, `git reset` updates the *staging area* to match the given commit instead of the working directory, and it doesn't move the current branch pointer. This means we can remove `blue.html` from the staged snapshot with the following command.

```
git reset HEAD blue.html
git status
```

This makes the `blue.html` in the staging area match the version stored in `HEAD`, but it

leaves the working directory and current branch alone. The result is a staging area that matches the most recent commit and a working directory that contains the modified `blue.html` file. In other words, this type of `git reset` can be used to unstage a file:

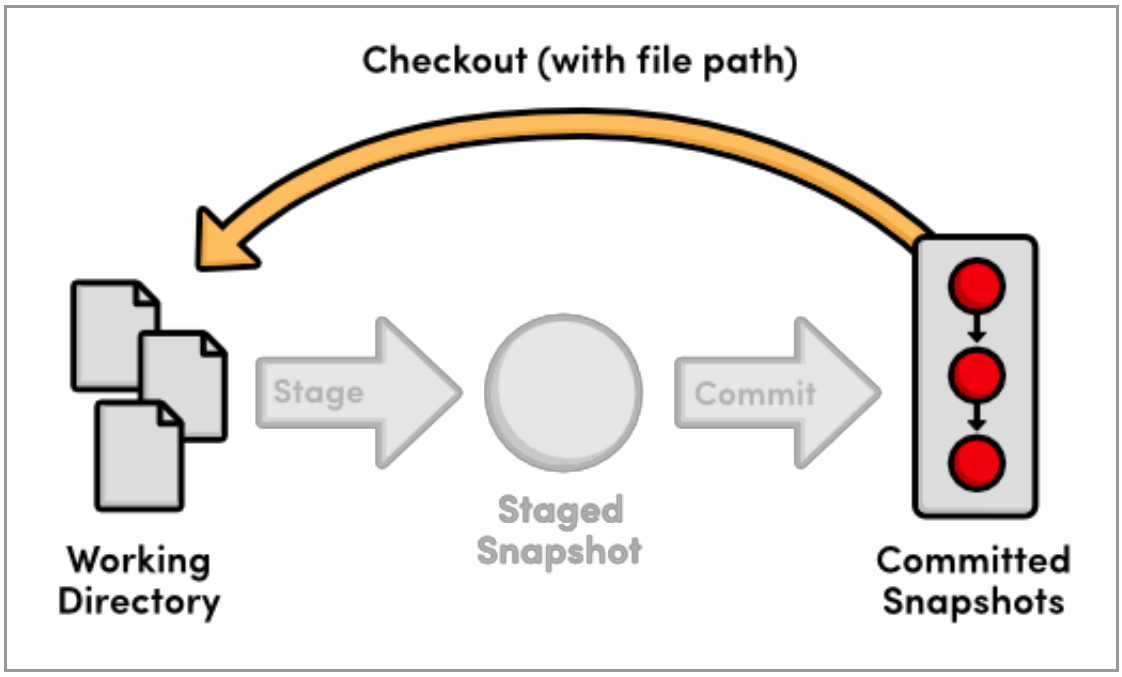


Using `git reset` *with a file path*

Let's take this one step further with `git checkout`. The `git checkout` we've been using updates the working directory *and* switches branches. If we add a file path to `git checkout`, it narrows its focus to only the specified file and does *not* update the branch pointer. This means that we can "check out" the most recent version of `blue.html` with the following command.

```
git checkout HEAD blue.html
git status
```

Our `blue.html` file now looks exactly like the version stored in `HEAD`, and we thus have a clean working directory. Passing a file path to `git checkout` reverts that file to the specified commit.



Using `git checkout` *with a file path*

To summarize the file-path behavior of `git reset` and `git checkout`, both take a committed snapshot as an reference point and make a file in the staging area or the working directory match that reference, respectively.

Aliases and Other Configurations

Typing `git checkout` every time you wanted to see a new branch over the last ten modules has been a bit verbose. Fortunately, Git lets you create **aliases**, which are shortcuts to other commands. Let's create a few aliases for our most common commands:

```
git config --global alias.co checkout
git config --global alias.ci commit
git config --global alias.br branch
```

Now, we can use `git co` instead of `git checkout`, `git ci` for committing, and `git br` for listing branches. We can even use `git br <branch-name>` for creating a new branch.

Git stores these aliases in a global config file, similar to the local config file we looked at in Mary's repository (`marys-repo/.git/config`). By default, global configurations reside in `~/.gitconfig`, where the `~` character represents your home

directory. This file should resemble the following.

```
[user]
    name = Ryan
    email = ryan.example@rypress.com
[alias]
    co = checkout
    ci = commit
    br = branch
```

Of course, your settings should reflect the name and email you entered in The Basics. As you can see, all of our new aliases are also stored in `.gitconfig`. Let's add a few more useful configurations by modifying this file directly. Append the following to `.gitconfig`.

```
[color]
    status = always
[core]
    editor = gvim
```

This makes sure Git colorizes the output of `git status` and that it uses the gVim text editor for creating commit messages. To use a different editor, simply change `gvim` to the command that opens your editor. For example, Emacs users would use `emacs`, and Notepad users would use `notepad.exe`.

Git includes a long list of configuration options, all of which can be found in the official manual. Note that storing your global configurations in a plaintext file makes it incredibly easy to transfer your settings to a new Git installation: just copy `~/.gitconfig` onto your new machine.

Conclusion

In this module, we learned how to export snapshots, backup repositories, ignore files, stash temporary changes, hook into Git's internals, generate diffs, reset individual files, and create shorter aliases for common commands. While it's impossible to cover all of Git's supporting features in a hands-on guide such as this, I hope that you now have a clearer picture of Git's numerous capabilities.

With all of these convenient features, it's easy to get so caught up in designing the perfect workflow that you lose sight of Git's underlying purpose. As you add new commands to your repertoire, remember that Git should always make it *easier* to develop a software project—never harder. If you ever find that Git is causing more harm than good, don't be scared to drop some of the advanced features and go back to the basics.

The final module will go a long way towards helping you realize the full potential of Git's version control model. We'll explore Git's internal database by manually inspecting and creating snapshots. Equipped with this low-level knowledge, you'll be more than ready to venture out into the reality of Git-based project management.

Quick Reference

`git archive <branch-name> --format=zip --output=<file>`
Export a single snapshot to a ZIP archive called `<file>`.

`git bundle create <file> <branch-name>`
Export an entire branch, complete with history, to the specified file.

`git clone repo.bundle <repo-dir> -b <branch-name>`
Re-create a project from a bundled repository and checkout `<branch-name>`.

`git stash`
Temporarily stash changes to create a clean working directory.

`git stash apply`
Re-apply stashed changes to the working directory.

`git diff <commit-id>..<commit-id>`
View the difference between two commits.

`git diff`
View the difference between the working directory and the staging area.

`git diff --cached`
View the difference between the staging area and the most recent commit.

`git reset HEAD <file>`
Unstage a file, but don't alter the working directory or move the current branch.

`git checkout <commit-id> <file>`
Revert an individual file to match the specified commit without switching branches.

```
git config --global alias.<alias-name> <git-command>
```

Create a shortcut for a command and store it in the global configuration file.

Continue to *Plumbing* ›