

# 2
# Merge Conflicts Written by Jawwad Ahmad & Chris Belanger

The reality of development is that it's a messy business; on the surface, it's simply a linear progression of logic, a smattering of frameworks, a bit of testing — and you're done. If you're a solo developer, then this may very well be your reality. But for the rest of us who work on code that's been touched by several, if not hundreds, if not thousands of other hands, it's inevitable that you'll eventually want to change the same bit of code that someone else has recently changed.

Imagine that your team's project contains the following bit of HTML:

```
<p>Head over to the following link to learn how to get started with Git:</p>
<a href="http://guides.github.com/activities/hello-world/">link</a>
```

You've been tasked with updating all of the text of the links to something more descriptive, while your teammate has been tasked with changing HTTP URLs in this particular project to HTTPS.

At 9:00 a.m., your teammate pushes the following change to the piece of code to the project repository, to update `http` to `https`:

```
<p>Head over to the following link to learn how to get started with Git:</p>
<a href="https://guides.github.com/activities/hello-world/">link</a>
```

At 9:01 a.m. (because you were a little farther back in the coffee lineup that morning), you attempt to push the following change to the repository:

```
<p>Head over to the following link to learn how to get started with Git:</p>
<a href="http://guides.github.com/activities/hello-world/">GitHub's Hello World project</a>
```

But, instead of Git committing your changes to the repository, you receive the following message instead:

```
 ! [rejected]        main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/supersites/git-er-done.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

That's something you've probably seen before. The remote has your teammate's changes that you just haven't yet pulled down to your local system. "Easy fix," you think to yourself, so you execute `git pull` as suggested, and...

```
From https://github.com/supersites/git-er-done
   7588a5f..328aa94  main     -> origin/main
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Well, that didn't go as planned. You were expecting Git to be smart to merge the contents of the remote, which contains the commits from your teammate, with your local changes. But, in this case, you and your teammate have changed the **same line**. And since Git, by design, doesn't know anything about the language you're working with, it doesn't know that your changes won't impact your teammate's changes — and vice-versa. So Git plays it safe and bails, and asks *you* to do the work to merge the two files manually.

Welcome to the wonderful world of **merge conflicts**.

## What is a merge conflict?

As a human, it's fairly easy to see how two people modifying the same line of code in two separate branches could result in a conflict, and you could even argue that a halfway intelligent developer could easily work around that situation with a minimum of fuss. But Git can't reason about these things in a rational manner as you or I would. Instead, Git uses an algorithm to determine what bits of a file have changed and if any of those bits could represent a conflict.

For simple text files, Git uses an approach known as the **longest common subsequence** algorithm to perform merges and to detect merge conflicts. In its simplest form, Git finds the longest set of lines in common between your changed file and the common ancestor. It then finds the longest set of lines in common between your teammate's changed file and its common ancestor.

Git aligns each pair of files along its longest common subsequence and then asks, for each pair of files, "What has changed between the common ancestor and this new file?" Git then takes those differences, looks again, and asks, "Now, of those changes in each pair of files, are there any sets of lines that have changed *differently* between each pair?" And if the answer is "Yes," then you have a merge conflict.

To see this in action, you'll start working through the sample project for this section of the book, and you'll merge in some of your team's branches in order to see that resolving merge conflicts isn't *quite* as scary or frustrating as it looks on the surface.

## Handling your first merge conflict

To get started, you'll need to clone the magicSquareJS repository that's used in this section of the book.

You can do this by way of the `git clone` command:

```
git clone --branch main https://github.com/raywenderlich/magicSquareJS.git
```

**Note**: The first edition of this book used the `master` branch as the default branch. However GitHub has since transitioned to using `main` as the default branch for new repositories. See https://github.com/github/renaming for more information on this.

The https://github.com/raywenderlich/magicSquareJS will also be transitioning to using the `main` branch as the default branch. This will be done soon after the release of the second edition. The additional `--branch main` option in the above clone command ensures that you checkout the `main` branch even if the default branch is still `master`.

Once that's done, navigate into the directory into which you cloned it.

```
cd magicSquareJS
```

Now, here's the situation: Zach has been working on the front-end HTML of the magic square application to make it work with the back-end JavaScript. Zach isn't a designer, so Yasmin has offered to lend her design skills to the project UI and style the front end so that it looks presentable.

Here is a bird's-eye view of all of the branches in the repository:


View of all branches in the repository
**Note**: The view of all branches above is from GitUp, available at https://gitup.co. If you'd like to replicate this view, make sure to checkmark the **Stale Branch Tips** and **Remote Branch Tips** options in the **Show** menu dropdown.

As the project lead, you're responsible for merging the various bits together and testing out the project. So, at this point, you'd like to verify that Zach's HTML works properly with Yasmin's UI. To do this, you'll have to merge Zach's work with Yasmin's work, and then test the project locally.

## Merging from another branch

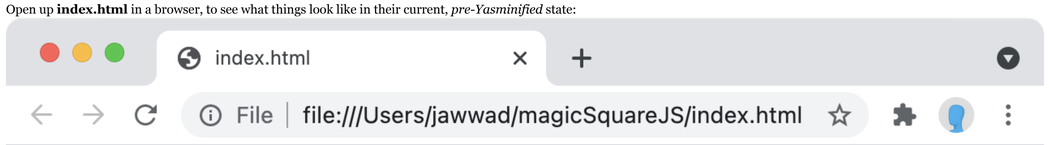
Zach has been doing his work in the **zIntegration** branch, while Yasmin has been working in the **yUI** branch. Your job is to merge Yasmin's branch with Zach's branch and resolve any conflicts.
Checkout Yasmin's **yUI** branch so you have a local, tracked copy of the branch:

```
git checkout yUI
```

Next, checkout Zach's **zIntegration** branch:

```
git checkout zIntegration
```

Open up **index.html** in a browser, to see what things look like in their current, *pre-Yasminified* state:




Well, it's clear that Zach is no designer. Good thing we have Yasmin.
Now you need to merge in Yasmin's **yUI** branch:

```
git merge yUI
```

It appears that Zach and Yasmin's work wasn't *completely* decoupled, though, since Git indicates you have a merge conflict:

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Helpfully, Git tells you above what file or files contain the merge conflicts. Open up **index.html** in an editor and find the following section:

```
  <body>
    <h1>magicSquareJS</h1>
<<<<<<< HEAD
    <section>
      <input type="text" placeholder="Size" id="magic-square-size" />
      <a href="#" id="magic-square-generate-button">Generate Magic Square</a>
      <pre id="magic-square-display">
=======
    <section class="box">
      <input type="text" class="flex-item" placeholder="Size"/>
      <a href="#" class="flex-item btn" >Generate Magic Square</a>
      <pre class="flex-item" >
>>>>>>> yUI
      </pre>
      <div id="validation" class="flex-item" ></div>
```

OK, you admit that your HTML is a *little* rusty, but you're pretty sure that `<<<<<<< HEAD` stuff isn't valid HTML. What on earth did Git do to your file?

## Understanding Git conflict markers

What you're seeing here is Git's representation of the conflict in your working copy. Git compared Yasmin's file to the common ancestor, then compared Zach's file to the common ancestor and found this block of code that had changed differently in each case.
In this case, Git is telling you that the HEAD revision (i.e., the latest commit on Zach's branch) looks like the block between the `<<< HEAD` marker, and the `===` marker. The latest revision on Yasmin's branch is the block contained between the `===` line and the `>>> yUI` marker.
Git puts both revisions into the file in your working copy, since it expects you to do the work yourself to resolve this conflict. If you were intimately familiar with the code in question, you might know exactly how to combine Zach's and Yasmin's code to get the desired result. But you skipped a few too many project design meetings, didn't you?
No matter; you can ask Git to give you a few more clues as to what's happened here. Remember that a merge in Git is a three-way merge, but by default Git only shows you the two child revisions in a merge conflict; in this case, Yasmin and Zach's changes. It would be quite instructional to see the common ancestor for both of these child revisions, to figure out the intent behind each change.

## Resolving merge conflicts

First, you need to return to the previous state of your working environment. Right now, you're mid-merge, and you only have two choices at this point: Either go forward and resolve the merge, or roll back and start over. Since you want to look at this merge conflict from a different angle, you'll roll back this merge and start over.
Reset your working environment with the following command:

```
git reset --hard HEAD
```

This reverts your working environment back to match HEAD, which, in this case, is the latest commit of your current branch, `zIntegration`.

### A better way to view merge conflicts

Now, you can configure Git to show you the three-way merge data with the following command:

```
git config merge.conflictstyle diff3
```

**Note**: If you ever wanted to change back to the default merge conflict tagging, simply execute `git config --unset merge.conflictstyle` to get rid of the custom setting.
To see the difference in the merge conflict output, run the merge again:

```
git merge yUI
```

Git explains patiently that yes, there's still a conflict. In fact, this is a good time to see what Git's view of your working tree looks like, before you go in and fix everything up. Execute the **`git status`** command, and Git shows you its understanding of the current state of the merge:

```
On branch zIntegration
Your branch is up to date with 'origin/zIntegration'.

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)
```

```
Changes to be committed:
        new file:   css/main.css

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   index.html
```

Most of that output makes sense, but the last bit is rather odd: `both modified: index.html`. But there's only one **index.html**, isn't there? Why does Git think there is more than one?

## A consolidated git status

Remember that Git doesn't always think about files, per se. In this case, Git is talking about both *branches* that are modified. To see this in a bit more detail, you can add the `-s` (`--short`) and `-b` (`--branch`) options to `git status` to get a consolidated view of the situation:

```
git status -sb
```

Git responds with the following:

```
## zIntegration...origin/zIntegration
A  css/main.css
UU index.html
```

The first two columns (showing `A` and `UU`) represent the "ours" versus "theirs" view of the code. The left column is your local branch, which currently is the mid-merge state of the original `zIntegration` branch mixed with the changes from the `yUI` branch. The right column is the remote branch. So this abbreviated `git status` command shows the following:
You have one file added (`A`) on your local branch; this is **css/main.css** that Yasmin must have added in her work. But it's not in conflict with your work.
On the other hand, you have not one, but *two* revisions of a file that are unmerged (`U`) in your branch. This is the original **index.html** from the `zIntegration` branch, and the **index.html** from the `yUI` branch.
These files are considered unmerged because Git has halted partway through a merge, and put the onus on you to fix things up. Once you've fixed them up, committing those changes will continue the merge.

# Editing conflicts

Open up **index.html** and have a look at the conflicted block of code now, with the new `diff3` conflict style:

```
  <body>
    <h1>magicSquareJS</h1>
<<<<<<< HEAD
    <section>
      <input type="text" placeholder="Size" id="magic-square-size" />
      <a href="#" id="magic-square-generate-button">Generate Magic Square</a>
      <pre id="magic-square-display">
||||||| 69670e7
    <section>
      <input type="text" placeholder="Size"/>
      <a href="#">Generate Magic Square</a>
      <pre>
=======
    <section class="box">
      <input type="text" class="flex-item" placeholder="Size"/>
      <a href="#" class="flex-item btn" >Generate Magic Square</a>
      <pre class="flex-item" >
>>>>>>> yUI
      </pre>
      <div id="validation" class="flex-item" ></div>
```

There's a new section in there: `||||||| 69670e7`. This shows you the hash of the common ancestor of both Yasmin and Zach's changes; that is, what the code looked like before each created their own branch.
A visual comparison of HEAD (which is Zach's branch) against the common ancestry shows the following changes:
Added `id="magic-square-size"` to the `input` tag
Added `id="magic-square-generate-button"` to the `a` (anchor) tag
Added `id="magic-square-display"` to the `pre` tag
A quick visual comparison of Yasmin's changes against the common ancestor shows the following changes:
Added `class="box"` to the `section` tag
Added `class="flex-item"` to the `input` tag
Added `class="flex-item btn"` to the `a` tag
Added `class="flex-item"` to the `pre` tag
It looks like it will be less work to migrate Zach's changes into Yasmin's code. So edit **index.html** by hand, moving Zach's new `id` attributes, from the first block of code in the conflicted section, into the third block in the conflicted section, which is Yasmin's code.
When you've moved those three `id` attributes into Yasmin's code, you can now delete the entire first two blocks from the conflicted section, from `<<< HEAD` all the way to `===`. Then, delete the `>>> yUI` line as well. When you're done, this section of code should look like the following:

```
  <body>
    <h1>magicSquareJS</h1>
    <section class="box">
      <input type="text" id="magic-square-size" class="flex-item" placeholder="Size"/>
      <a href="#" id="magic-square-generate-button" class="flex-item btn" >Generate Magic Square</a>
      <pre id="magic-square-display" class="flex-item" >
      </pre>
      <div id="validation" class="flex-item" ></div>
```

Save your work and return to the command line.

# Completing the merge operation

You've finished resolving the conflict, so you can stage your changes with the following:

```
git add index.html
```

Execute **`git status -sb`** to see what Git thinks about your merge attempt:

```
## zIntegration...origin/zIntegration
A  css/main.css
M  index.html
```

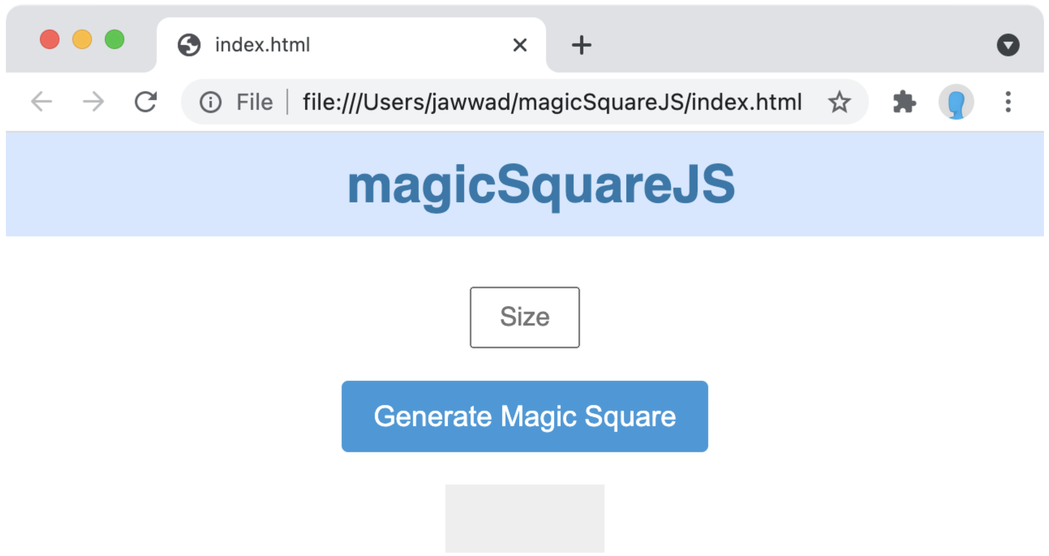
There you are; one new file and one modified file. Git's noticed that you've resolved the outstanding conflicts, so all that's left to do to complete the merge is to commit your staged changes.
Commit those changes now, this time letting Git provide the merge message via Vim:

```
git commit
```

Type **:wq** inside of Vim to accept the preconfigured merge commit message, and Git dumps you back to the command line with a brief status, showing you that the merge succeeded:

```
[zIntegration b81a302] Merge branch 'yUI' into zIntegration
```

Now, open **index.html** in a browser to see the changes:




That looks quite good. It's not fully functional at the moment, but you can see that Yasmin's styling changes are working. You're free to delete her branch and merge this work into `main`.
First, delete the `yUI` branch:
```
git branch -d yUI
```

Switch to the `main` branch:
```
git checkout main
```

Now, attempt a merge of the `zIntegration` branch:
```
git merge zIntegration
```

Git takes you straight into Vim, which means the merge had no conflicts. Type **:wq** to save this commit message and complete the merge. Git responds with the results of the merge:
```
Merge made by the 'recursive' strategy.
 css/main.css | 268 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++...
 index.html   |  14 +++++-----
 js/main.js   |  85 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 3 files changed, 362 insertions(+), 5 deletions(-)
 create mode 100644 css/main.css
```

You're now able to delete the `zIntegration` branch, so do that now:
```
git branch -D zIntegration
```

**Note**: The `-D` switch forces the local deletion of the branch regardless of its status. If you used the normal `-d` switch here, you'd see a warning about the branches changes for `zIntegration` not having been pushed to the remote.

## Challenge: Resolve another merge conflict

The challenge for this chapter is straightforward: resolve another merge conflict.
**Note**: If you encountered any issues in completing the chapter, you may use the project in the challenge directory as a starter point for this challenge.
Xanthe has an old branch with some updates to the documentation; this work is in the `xReadmeUpdates` branch. You want to merge that work to `main`.
The steps are as follows:
Checkout the `xReadmeUpdates` branch and look at the README.md file to see Xanthe's version.
Checkout `main`, since this is the destination for your merge.
Resolve any merge conflicts by hand.
Stage your changes.
Commit your changes.
Delete the `xReadmeUpdates` branch.
If you get stuck, or want to check your solution, you can always find the answer to this challenge in the **02-challenge.md** file under the **challenge** folder for this chapter.

## Key points

Merge conflicts occur when you attempt to merge one set of changes with another, conflicting set of changes.
Git uses a simple three-way algorithm to detect merge conflicts.
If Git detects a conflict when merging, it halts the merge and asks for manual intervention to resolve the conflict.
`git config merge.conflictstyle diff3` provides a three-way view of the conflict, with the common ancestor, "their" change, and "our" change.
`git status -sb` gives a concise view of the state of your working tree.
To complete a merge that's been paused due to a conflict, you need to manually fix the conflict, add your changes, and then commit those changes to your branch.

## Where to go from here?

In practice, merge conflicts can get pretty messy. And it might seem that, with a bit of intelligence, Git could detect that adding HTML attributes to a tag is not really a conflict. And there are, in fact, lots of tools, such as IDEs and their plugins, that are language-aware and can resolve conflicts like this easily, without making you perform all the edits by hand. But no tool can ever replace the insight that you have as a developer, nor can it replace your intimate understanding of your code and its intent. So even though you may come across tools that seem to do most of the work of resolving merge conflicts for you, at some point you'll find that there is no other way to resolve a merge conflict except by manual code surgery, so learning this skill now will serve you well in the future.
Up to now, your workflow has been constrained to the "happy path": you can create commits, switch between branches, and generally get along quite well without being interrupted. But real life isn't like that; you'll more often than not be partway through working on a feature or a fix, when you want to switch your local branch to take a look at something else. But because Git works at the atomic level of the commit, it doesn't like leaving things in an uncommitted state. So you need to stash the current state of your work somewhere, before you switch branches. And `git stash`, covered in the next chapter, does just that for you.

3. Stashes 1. How Does Git Actually Work?