APPSUMO

Presents

Version Control - From Zero to Hero (with a focus on Git)

with

Alex Hillman

BeanstalkApp.com

* Transcript

Paul: Hi, everyone. I'm Paul Hontz from The Startup Foundry. Joining me today is Alex Hillman from Wildbit. Alex, thanks for being here.

Alex: Thanks for having me, Paul.

Paul: Now, what are we going to learn in this AppSumo action video?

Alex: Like you said, I'm from Wildbit, and we've got a couple of web products. One of them is Beanstalk, which is a hosted version control service. It supports a couple of popular version control systems, the two biggest ones being Subversion and Git. We're going to touch on Mercurial as well, and if you're interested in that, you can email me about a beta invite to our Mercurial support.

But the plan today is to give a brief overview on why you need version control, what options are available, some of the pros and cons of each system, and then we're going to jump into a practical example of how you would use Git to version control your own projects.

Paul: Awesome. Now, give us a little bit of your story here. Give us some background.

Alex: My story is I've been working with Wildbit for a little over a year, and my primary role is interacting with our customers and our partners to make sure that they get the best chance to know us and that we know the most about our customers and partners. I get to be out there in the field, learning from people, teaching people, and that's what makes this a really great opportunity for me today.

Paul: Great. Now, by the end of this tutorial action video, what are we going to be able to do?

Alex: You're going to understand why version control is important and why you should have been

using it all along. Once you start using it, you're not going to know how you lived without it. Then we're going to show you how Git gets installed. We're going to do some basic configuration. We're going to help you set up your first repository, teach you how to commit, teach you how to branch, teach you how to work with remotes, and a couple of other things. But all the basics you're going to need for day-to- day version control usage are going to be covered today.

Paul: Great. Now, let's get started. Why should we even use version control to begin with?

Alex: You're probably used to versioning files, like index.html, and you copy it into index-v12-old2.html. You know how messy that gets, right? There's got to be a better way. So, everyone's watching this, I'm presuming coming from some different areas of expertise. Some of you may already use some version control, like Subversion, but you want to learn more about Git. Some of you might not use version control at all. So, for that reason, we're going to start with basics, like I said. You're probably familiar with, like I said, the old way of giving file names stuff like imagev12old2.jpeg or whatever. You also commonly accidentally overwrite code or files, and you're not getting that stuff back, no matter how good your backups, once you've overwritten something. Making live updates directly on your production servers, you're probably guilty of this, right Paul?

Paul: I've done this one or two times. Not proud of it.

Alex: Yeah, everybody is. You're not alone. Don't be ashamed. We're going to help make things better today. Making live updates directly on your server not only could be risky for the people that are trying to visit the websites, but again, you might overwrite files not real-

izing what you did. If you're working with co- workers or a team, if they're updating files and you don't know about it, not only could you overwrite them, but they have to do extra work just to let you know that they made changes. If you're doing any sort of multi-server pushes to production, you've no idea what files are on which server because you're doing all these like FTP copies and whatnot. So, having version control be authoritative is going to help you. So, what I want to work on today is teaching you the new way.

Moving forward you're going to have consistent structure and naming conventions for all of your files. Your file names are going to be able to stay the same, no matter how many times you change them, and you're not going to have to lose any of the old versions. The other thing is that if you do rename a file, Git will keep track of all of those renamings. So, you're not going to get lost along the way.

You'll be able to make updates and sort of hack at things, with confidence, because you can easily revert to previous versions of any of the files that are in your repo. You'll be able to use source control as the authoritative source for all of your updates. Everyone's committing to the same place, which means you're deploying from that one place. Also, when it comes to collaborating with a team, before you'd have to like send out an email update at the end of the day, because you wouldn't want to do it after every little thing that you did, to let people know what you worked on. Now you're going to be able to use code as communication. People will be able to see the differences between each of the files that you keep track of in version control, and in your commit messages, you can even tell them why you did what you did. If you're doing deployments, like I said, using version control as your authoritative source, you always know the current version is on the right

server.

So, with version control, you can easily have multiple stages of your work. This is really cool and new to a lot of people. For instance, you can be doing local development, right? You could have one version there, and then you could have a staging server for your clients to take a look at and make sure your work is right, and that could be different from your development environment. Then, production could be the stuff that the clients have approved. All of that could be running out of the same repository. You never have to worry about not being sure which versions of which files are in which environment.

Ultimately, this leads to better habits when doing deployments and happier clients and less mistakes.

Paul: Sure. Now, I've just started to use some of this stuff, I don't know, maybe a month or two back, and this just absolutely blew my mind. Like, once you actually see how this works, just laid out, I can't imagine working without it. What we're learning today, you mentioned that we're going to be covering Git here. Now, will this stuff work with GitHub, as well? Explain that a little bit before we go further.

Alex: Yes, absolutely. So Git is an open source tool, which means that you'll be running the Git program on your machine. When we start talking about remotes, that's where GitHub and Beanstalk and some other tools like that come into play. All the stuff I show you today is going to work about the same on GitHub or Beanstalk or any other tool like that.

The fact of the matter is with Git, because it's decentralized, which we'll talk about in just a minute, you're going to do the majority of today's demo or I'm going to do the majority of today's demo without even connecting to Beanstalk.

So, you can download Git, get it installed, and once it's installed you could cut off your Internet connection and do version control without having GitHub or Beanstalk backing you up.

Paul: Awesome. Okay, now let's keep rolling here. There's a lot of new terminology here. Can you help us break this down?

Alex: Yes, totally. I want to explain the three main version control systems, but you're right, there are some basic terms that are common between all of them. The first one is the word repository, and repository is really just sort of a fancy name for a folder that keeps track of all of the changes. The repository could either be local or it could be remote on a server. We'll talk a little bit more about that in a bit.

You have the ability to add or update files. That's pretty much exactly what it sounds like. As needed, you can add or update files within the repository and then tell the repository that you need to do stuff with it.

The action of committing changes is going to be probably the most important thing you'll learn today, because what committing means is when you're ready, you can commit the changes that you've made, either since the last change, or if it's the first commit, all of your changes or rather all of your files, to the repository. So think of a commit like a snapshot, where it says here's all the things as they currently are, and then the next time you make a commit, it says here's all the things as they currently are.

A revision is sort of an identifier of the commit. So, in some schools, that's an incremented number - one, two, three, four, five. In Git and Mercurial, it's an ID that is good for reference, but it's not really human readable. We'll talk about that a little bit more too.

Reverting is what happens when you made a mistake. If you're committing often, you're able to actually roll back your changes to a previous commit or previous revision, if you will. You can even take a look at your current work against a previous version and cherry pick stuff out of that.

The last couple of concepts are branching and merging. We're not going to cover a whole lot of this. We're going to give you some basics. But the idea behind branching is it lets you work on experimental features. Basically, it lets you make a copy of the repository within the repository and work there, without touching the main files.

Then merging is the reverse of that, where you take those changes and you say, "I want to bring these back into the stable code." The version control systems give you tools to make that really easy to see what the differences are and what you changed and make it easy for you to get them in.

Finally, there's a diff, and a diff is just a difference between a couple of files or branches, sets of files.

You don't have to remember all of these things today. Day to day, you're going to do a lot with just knowing what repositories are, how to add and commit file changes. All the other stuff, you're going to hear me talk about, but you won't have to do every single day.

Paul: Okay. You said that there were different version control options. Can you get into that a little bit?

Alex: Sure. So right about now the three most popular options can be broken down into two categories. The first category is centralized version control. The key thing to know about centralized version control is that it's dependent on a client server relationship. So in this case, the repository resides in one place, usually a server where other

people can access it. If you're in a company, that might be a server in your network. Depending on your situation, you may have a server that's connected to the Internet. Beanstalk does Subversion hosting, so Beanstalk could be your centralized Subversion server. Changes are made locally, but then you commit them to that server, where other people can check out or update the latest version.

Like I mentioned, Subversion is the most popular version of this today. The biggest benefits to Subversion is that every revision number is an increment, like I said, one, two, three, four, five, up into the hundreds and thousands, which means it's really easy to understand sort of where you are in the lineage of your revisions.

The centralized model offers more control over users and access, which is good for some companies. It's been around for a while, so there are a lot of options when it comes to GUIs and IDEs, and if you're just getting started that can be popular. Again, the revision number being incremented means it's simpler to learn, easier to get started. It also happens to be free in open source, which means you can use it for free. Tools like Beanstalk, you pay for that, but we give you benefits on top of the open source version of the software.

The disadvantages, though, to centralized version control is that it's dependent on that centralized server being available, which means it's dependent on your network or your Internet connection being available. So if you're on a plane or your Internet goes down or the server goes down, you're screwed. You can't work. Also, managing servers is a pain in the neck, of any kind, but especially version control servers. You're setting up users and repos. It's a big pain in the neck. Of course, this is where you're keeping track of all of your

code and the revisions, so your backup systems need to be really good. That's something else you need to be worried about. Because everything happens over the wire, commits and other commands can be super, super slow, especially merging and branching, and Subversion is known for being painful because it's all done over the connection of the centralized server. That's really the biggest downside to Subversion is the fact that everything you're doing is talking to the server. So while that can be a benefit, it's also a detriment.

Paul: Okay. Now, will most people who are just getting started with version control use a centralized server?

Alex: It's sort of up to personal preference. The biggest benefits to Subversion as a centralized server have very little to do with the fact that it's centralized. It's more about the way Subversion is architected. If you run a company or you work for a company where tight access control is really important, then Subversion might be a better option, because decentralized or distributed version control is a bit different. We can talk about that now, if you want.

Paul: Yeah, just maybe give us a quick snapshot of that.

Alex: Sure. With decentralized or distributed version control, all your changes are committed locally and then pushed or pulled to other instances of that entire repository. So the most important attribute of this is that each person that's using the distributed version control has their own copy that contains the entire repository history. There is no single server. Instead it's sort of like a big network of repositories talking to each other. The two most popular tools that are decentralized are Git and Mercurial. Once we get through this intro, we'll be focusing more on Git, and they share a lot of pros and cons, even com-

mands and attributes.

The benefit to decentralized version control is that you can commit more often, because you don't have to connect to that remote server. Remember, when you commit to a remote server, that means somebody else is going to get your change right away. So, if you're working on experimental stuff or you're working on things you're not ready for other people to see, you've no way to commit. So you're often waiting until your work is done before you make a commit. With Git and Mercurial, you can commit as often as you want, because the other people on your team don't get that code until you push it. This means you end up with much more powerful and detailed tracking of your changes. It also means that it's faster. You don't have to do that stuff over the wire. Everything is being written locally to disk, and you could even make commits when you're on an airplane, which I like working on airplanes.

Merging and branching are way more powerful in both of these tools. When I say powerful, that means that there's less manual conflict resolution. We're going to do a little bit of an intro on conflict resolution near the end of the demo, but what it basically means is when two changes are made in the same place, you have to determine which version of the change you want or manually merge them. Git and Mercurial do a lot of that automatically, that Subversion can't.

As I mentioned, it's fast. It's also free and open source, both Mercurial and Git are free and open source, a lot like Subversion.

The downside is that these tools are more complicated. The commands are a little bit more like incantations. You feel like you're casting spells. I'm going to try and make that not so painful. Once you

learn the commands, there are a number of GUI and ID clients, but it's limited. That's changing very quickly. There are a lot more tools. People are building good tools. Tower is really popular. GitHub released an actual GitHub Client. It's a really simplified version of a local GitClient. It doesn't do all the things, and they simplified some things beyond what you might understand once you've actually taken this workshop.

But worst of all, I would say, is that these tools don't do as good of a job of protecting yourself, protecting you from yourself, I should say. This means it's possible to make mistakes and lose your work while you're learning. That doesn't mean you should be afraid. It just means you should be aware that it might happen. Everybody makes these mistakes. Most people only make them once. So try to only make them once.

Paul: All right. In this video, we'll learn how to hopefully avoid those mistakes.

Alex: Absolutely.

Paul: Okay. Why don't we get started here. Let's jump right in. How do we actually get Git installed on our computer?

Alex: If you're on a Mac, which I'm going to assume you are, it's really easy. You can just go to this Bitly link. It's bitly/downloadgit, and it'll take you right to download from Google code, where you can just get the DMG, open it up, double click the installer. It will ask you a couple of questions, and at the end, you'll be able to run Git commands.

If you're on another platform, like Windows or Linux, you have to go to get-scm.com, find your OS, find your version, and download and install it from the directions there. Their documentation's a little bit scary, but it's not too hard to find

the right version for yourself.

Paul: Yeah. Then, once you're up and running here, it's the same across platforms.

Alex: Right, exactly. So one other assumption, before we jump into actual using of Git, is that we're going to assume that you're on a Mac, like I said. The nice thing about that is even if the terminal commands are a little bit different, the Git commands are going to stay the same. Even if you're on Windows, you can run the same Git commands. The way you change directories, work with directories and files might be a little bit different, but the core Git commands are exactly the same.

Paul: Okay.

Alex: I just mentioned terminal. So in the event somehow you're not familiar with terminal, it's that scary black screen with the green letters that nerds type into. If you don't even know where it is, if you're on a Mac, like I said, it's in the applications/utilities folder. You can also dig up terminal and spotlight, or your favorite launcher. It can be a little bit scary. Like I said, it can be a little bit like incantations or casting spells. We're just going to focus on a handful of commands today so you can get started using Git. Once you get the hang of things, you can leave the terminal behind, not for everything, but for a lot of things, where basically a GUI client is just going to give you a button so you don't have to remember a fancy command with a bunch of extra stuff at the end of it.

Paul: Right. But it's good to start with the terminal just so you actually know what's happening.

Alex: Right. That's how you started, right?

Paul: Yep. That's exactly how I started. At first I tried to just jump right in with the GUI client, and

it worked. I got it working, but I had no idea what I was doing. And starting with the terminal, like you just gave me a much greater understanding of what was actually going on. So I'd highly recommend going with the terminal first.

Alex: Yeah, I think that's the best advice. Many people have said that's some of the best advice we've given them. Now, we were going to assume that you have Git installed. You've downloaded it. You've installed it. There are a couple of things that you have to do before we can actually start running Git commands on our code. They're basically configurations.

The first thing that you're going to do is the Git global config. I've got a couple of slides up right now that'll show you these commands, to save us the time of having to type them in again. But basically, you're going to set a user name and a user email address in your global Git config. You only have to do this once. You don't have to do it on each repository, and the reason this is important is, as you'll see in the logs later on, and if you're pushing to remote services like Beanstalk, this is where who you are is going to be associated with the changes that you make. That's a really important concept in version control. Every change that's tracked is associated with you as a person, so when there's trouble down the road, people can figure out who made the changes.

Paul: Okay. And again, you would be setting this up even if you're just running this locally on your computer and you're not actually connecting to the Internet?

Alex: Yes. At this point, for the next few sections, basically until we start talking about remotes, you don't need to worry about connecting to the Internet at all.

Paul: All these changes are on your local machine?

Alex: That is right, which also means that if you have multiple machines, you need to set this up multiple times. This is per development environment.

Paul: Okay.

Alex: You can do the "Git config--list", and it will list out the things that you just put in, your user name and your email address, so you can verify that it actually went it. The problem with a bunch of these commands, and you will notice this as you start using the Git command line more, is not everything gives you a response. Sometimes you need to type another command to make sure that what you did actually worked. So that is what the Git config--list lets you do. It lets you make sure that the user name and the user email were actually configured.

A couple of bonuses in global configs, these are not required, but they do make things a whole lot easier. Basically, you will see in the demos that my commit statements, my commit responses from Git are often colored green and red, and that will make more sense when we get there. If you set these two commands, again it is one per line, that is how you turn on those colors.

The last super bonus option, it allows you to override the editor. When you are writing your commit messages, which you will do every time you make a commit, by default Git opens up something like nano or vim, which means you have got yet another set of random key strokes to memorize. You do not have to do that. If you type this line in, and you have got TextMate installed, every time you commit, it will actually launch a TextMate window that is pre-populated with a bunch of information to help you write a good commit message. You can save it, and when you close that window doing command W, it

will automatically dump you back into the command line where you left off, and it will keep moving along nice and smooth.

Paul: Okay. I think some other text editors have this as well, if you prefer to work in other development environments.

Alex: Exactly. All you need to do is do a little bit of research and figure out what the command line command to launch your text editor is, and put that in where the Mate-W is between the quotes.

Paul: Okay. They can just google that and figure that out specific to their environment.

Alex: Exactly.

Paul: Okay.

Alex: The last thing that we need to go over before we get into Git stuff is a rundown of some basic command line navigation tools. You are going to see me do them at the command line. I just want you to understand what I am doing. I am not going to spend a whole lot of time, to save time during this workshop, but you will at least know what I am up to.

PWD tells you the name and the path to the directory you are working in on the terminal. It says this is the folder I am working in, and it spells the whole thing out, all the way from the base folder.

CD is change directories. You type "CD" and then the full path to that directory and it is like jumping into that folder.

CD .., with a space between the "CD" and the ".." moves you up one level of folders. It is like moving one level up the tree.

The command LS lists everything in the current folder, and you may even see me do LS -A, and that lists all in the folder, even if the files are

hidden. You will see why that is important in a minute.

The command MKDIR makes a directory.

CP copies files.

MV moves files.

That is really all you need to know for making sure that you understand, for the most part, what I am up to as my keys are flying at the terminal demo.

Paul: All right. Alex, talk to us a little bit about repositories.

Alex: Sure. Like I said, a repository is a fancy folder. What happens is you turn a folder and its contents into a repository. Then you are able to start running commands on that repository to start tracking your changes. We are going to go over to the terminal on my screen and show you how that works.

On my screen, you can see I already have a terminal up, and I am going to just show you when I am real quickly. PWD shows you that I am in my home directory. The folder or directory that I want to turn into a repository is this one over here on My Desktop that says "AppSumo Workshop Demo." Type "CD DESKTOP" and then "AppSumo Workshop Demo." You will notice sometimes I move unusually fast. It does not look like I am typing out the rest of the command. The way I am doing that is I start typing it and I hit "Tab," and if there are not any other options that it can be, it will auto-complete, which is a handy little trick.

If I LS, then you can see inside this directory I have got a little web project, an index.html with some style sheets, images and things like that.

All it takes to turn this folder and the files inside of it into a get repository is "GET INIT" and "Enter."

You'll see it says "Initialized empty Git repository in" and then the folder/.git. That .git, I don't know if you've seen this before, but dot folders and dot files are hidden by default. So, if were to do an LS, then nothing changed. But if I do an LS -A, there's the .git folder in there. I can actually CD into that, and you'll see there are all kinds of goodies in here. We're not going to touch that today and you really shouldn't touch that very often. It can be useful in diagnostics with remotes and things like that, but we're going to get out of there as quickly as we can.

Paul: That's where all the Git magic is actually being stored?

Alex: Right, exactly. That's where it's storing the configuration specific to that repository. It's actually where's it stores all of the revision contents itself. If you've ever used a version, you might know about it putting .svn folders in every single directory through the entire project. It's a mess. One of the really cool things about Git is that that .git folder contain all of the magic. No matter how many layers of folders and files you have, everything is right there at the root of the project. So it's really easy to keep track of things.

Paul: Okay.

Alex: Let's go back to my slides real quick. If you're in a repository, you've got this command called "git status," and what this does is it tells you about the status of the repository. Let's go back to the terminal again real quick. If I run git status, it'll tell me a bunch of things right here. The first thing it says is that I'm on the master branch. We'll learn more about branches later. Then it says that I have untracked files. It says the directory of images, the index.html as well as the directory of Style Sheets.

We've created a repository, but the files that Git recognizes as in that

repository haven't been added to the list of files to track yet. If you look at the bottom of the message that came back, it says, "'Nothing added to commit, but untracked files present.'" It suggests you might want to use git add to start tracking them. That's exactly what we're going to do. You'll also note that the files are currently in red. That means that they won't have any changes tracked to them. So long as you've made that color configuration from earlier, only green files get committed when you run a commit command.

Paul: Okay.

Alex: Makes sense so far?

Paul: Yeah. This is basically telling Git, "Hey, watch these files for changes"?

Alex: Exactly. In order to do that, we need to run the git add command.

Paul: Okay.

Alex: What the git add command does is it takes the files that Git knows exist but doesn't know what to do with and puts them on the stage. The stage is a place for you to put the files that you want to commit. There are different phases that the files will go through. Files that are untracked are not on the stage. Files can be tracked but not on the stage. Files can be on the stage. It's only when they're on the stage that they're ready to be committed.

Just because a file has changed doesn't mean that Git can automatically commit it when you run the commit command, like I said. This is confusing I'm sure, but you'll get more comfortable with it. What will happen is you'll realize that this means that you're able to choose which files are included in each commit. Remember we talked about making sure your commits are really granular and simple and

you can do them very often. Even if you've got multiple files that have changed, you can choose to only commit the ones that you want.

I'm going to head over to the terminal and add all of the untracked files to the stage. That's really easy to do. I can type "git add" and then dot or a period. If I run that and there's no response from the command, it ran successfully and you just get back to a command prompt. The dot there after the add command means add everything that's here. Here is the present working directory and any of its sub-directories. That means it's not just going to add the index file as well as the folders, but all of the files and folders that sort of recurse down the tree.

Paul: Okay.

Alex: If you didn't want to add everything, you can run git add on specific files, specific folders, and things like that. You can find more about that in the Git documentation. What's important here is that if you do a git status after running the git add, all my files are green. You'll also notice that this time it's picked up the actual image files and the style sheet from inside the images and style sheets folder. So, git add successfully added everything to the stage. That means that when you run the commit command any changes that have been made to those files, which in this case is the file's entire contents, get committed.

Paul: Okay. Real world example, let's say you're working on the front end of the website and you add a new style sheet and you want Git to start tracking the changes. You'd have to actually run git add and then either the path to the file or git add with a period.

Alex: Right. Git add period, like I said, it's atomic. It's going to add everything. So you need to make sure that all of the things that are

currently listed as untracked or not on the stage you're okay with adding to the stage. Make sense?

Paul: Yes.

Alex: There's a shortcut when it comes to committing where you can sort of sidestep the add. We'll cover that in a little bit.

We've been talking about committing, but I haven't really explained what it is or how to do it. We're going to spend a good amount of time on commits now because there are a few different ways to do them and one you're going to use the most by far. Now that Git knows about the files, it still needs to create the initial snapshot of the files so it can begin tracking changes as you move forward. That snapshot, like we said, is called a commit and the command is pretty simple.

Let me show you what that looks like on the terminal. This is actually kind of cool. I wrote "git stats" by accident, and Git said, "This isn't a Git command. Here's the help. Did you mean this?" It offered "status" as an option. Remember I said it doesn't protect you from yourself? It does sometimes give you helpful things. "Git status" and all my files are green, which means if I do "git commit" and hit "enter," you'll notice that on my screen TextMate just opened up. Everything with a pound sign to the left of it is going to be ignored. It's not going to be a part of the commit message. It's just there for reference. Git gives you a bunch of references to help you write really good commits. I'm coming in here and looking and I see all my files from the project and I go, "Oh that's right. This is the first time I committed it." So I'm going to write something like "my first commit." Then when I hit "command S," it will save it, and when I hit "command W," it will close it. It drops me back into Git and it lists out what Git actually did at the end of that commit message. Apart

from the files that it'll tell you what it did with, it will also give you some counts. In this case, it says 15 files changed, 406 insertions, and zero deletions. We're going to talk about insertions and deletions in just a second.

Apart from the commit message, you can also run the "git status" command now, and it'll tell you that there's nothing left to commit and the working directory is clean. That's a great state to be in. At this point, we've taken our first snapshot, that's our first commit. Not that hard, just a couple of commands to get there. We can move ahead with making changes and tracking those changes.

I mentioned insertions and deletions. Git needs to be aware of the files in your project, but what it really looks at is the changes within the content within those files. This is kind of wacky and it's sort of like the under-the-hood architectural stuff, but it's important and it'll become more important when it comes to merging and conflict resolution to understand what happens. Imagine if Git took all the files in your project and stacked them one on top of each other, so it was one super long document. What Git is actually doing is paying attention to changes on a line-by-line basis.

What you'll soon see is that it's interested in new lines being added, which are called insertions, and lines being removed, which are called deletions. You'll notice there's no such thing as a change. The reason for that is a change to an existing line Git actually sees this as a deletion of the original line and the creation of a new one in its place. All you get with Git is files, insertions, and deletions. It's pretty cool.

The other thing that's interesting about this is every revision, all it's keeping track of are those actual insertions and deletions, which

means your repository doesn't get huge. You're probably worried about the hard drive space as you're going through and making all these commits. Each commit isn't saving an entire copy of the file. All it's saving is the line-by-line insertions and deletions, and that saves you a lot of space and also lets it be really fast. Because it's saving space, you have no reason to not be committing super often. You should be committing as often as you can. More commits equals more communication for you to your future you, or if you're on team, to your other team members.

Paul: Sure. If you commit something five times, the file size of the folder isn't going to be five times bigger?

Alex: Exactly. All it's going to save is the tiny little bit that changed. Each commit is saving those deletions and creations. Commits are fast and small. Let's do one more change in one of the files. I want to show you another way to commit.

Let's go back to my files. I'm going to go into the index file here in TextMate. What am I going to change here? I'm going to just take out this H1. It must not be important. I'll save it. When I go back to the terminal, I run "git status." It says that we're on the master branch. There are files that are changes not staged for commit, and the index.html was modified. You'll notice that it's red. Didn't I say that red files needed to be added before they can be committed? But we already added index.html once before. So why is it asking us to do that again? Again, Git track changes, not files. So, even though Git knows about the file, you'll notice that it says the file is modified. What it really needs you to do is stage that modification, and we do that again with the "git add" command. So, we do "git add index.html", enter. Now if I run "git status," the file is still modified, but

it's green. You've made the change to your file. It's ready for commit.

This is a tiny little commit. So maybe you don't want to fire up TextMate for it. There's a short cut here. I'm going to show you what's called in-line commits. Because this change was small and you remember all the things you did, you can write the commit message in-line. It's all one shot.

You're going to do Git commit. This time you are going to do the M flag. So it's Git commit dash M, and then in quotes write, "Removed that stupid header. I didn't like it." And then end quote, you have to end quote. When you hit enter, you are going to see something that should look familiar. It'll tell you that a file changed. There were zero insertions, and in this case there were two deletions because I deleted some white space as well.

This technique of typing your commit log message without opening a text editor can save you a bunch of time and key strokes, which is usually what leads to people writing better commit messages. I like to teach this to people to make sure they're not skipping over tiny commits. Being able to do it quickly means you're going to be able to do it more often.

Paul: Typically, when I have been working, in-line commits is what I use just because a lot of times I'm committing so frequently that those changes are really small, and I can easily fit that in one line.

Alex: Yeah. I'm the same way, knowing that you could write a really complex commit. In fact, there are some really interesting best practices on writing commits where the first line is sort of a general description of what's contained. Then, if you're writing a really good documentation, you could be writing multiple lines of commit explaining why you did what you did. Like the guys who

use Git to manage Linux kernel, which is sort of why Git was invented, write really verbose messages because they're keeping track of an important decision history, and they do all of that right in the commit message. So you can do the same thing.

Paul: Sure.

Alex: There's another, it's like a pseudo commit. It's called stash. Once in a while, while you're working on something, you've dirtied up your working branch with changes that you're not totally sure you want to commit. But maybe somebody calls you and says, "I need you to fix this bug." You could branch and commit, and then go back to your working directory and fix your bug and then merge your branch in. But that's a bunch of work.

So, Git gives you a super handy tool for small sets of changes that maybe you've been interrupted, and it's basically some routine to commit, which is permanent, and a clipboard, which you know is less permanent. It's actually really easy to use. I use it for one-offs all the time.

So we're going to go back into one of the files and make a change real quick. Let's see here. I'm going to go up in the title here, and I'm going to change it to Beanstalk Version Control for Awesome Designers. I'm going to make it the best designers in the world. When I save it, I do a "git status." As we would expect, index.html is now marked as modified.

Let's say I got a phone call from one of my team members, and they're like, "Hey, your page is breaking Internet Explorer." I go, "Well, shoot, I don't want to fix it and then push this title live until I check it with my boss." So, I need to stash the change for later. All I have to do is type "git stash" and hit enter, and it'll tell me that it saved the

working directory in its index state off to a stash. It'll tell me that head, which is sort of like where we are right now, is at the commit and it actually gives you the ID of the commit, as well as the last description. So you know exactly where you left off. I can say, "Okay, I can work forward from here.

What's neat is, if I go back to the file, oh, look at that. It actually got rid of the change that I made. So I can go in and do some more work and fix the bug, and push it up. Then when I'm ready, I can come back to this file and I can do "git stash apply," hit enter, and it puts things back exactly like they were. It still hasn't committed that change, but it basically pulled that little pseudo commit off of a clipboard, put it back in place, and said you can keep moving from here.

Paul: Got you. So these are some more advanced things that people who are just starting out might not use right of the bat, but it's definitely handy things to know.

Alex: Actually, I use "git stash" quite a bit, and now that you know that it's there, I imagine you're going to find opportunities to use it. It also comes in handy when we talk about remotes. There's a situation where when you're pulling down changes from a remote, sometimes you haven't committed your working changes yet and you don't want to. But Git won't let you pull until you've committed changes. So, stash is really handy for when you need to pull, but you're not ready to commit, because you can stash your changes, complete the pull, and then reapply your changes. Does that make sense?

Paul: Yeah, that does.

Alex: Cool.

Paul: Sorry, I was a little unclear with even using stashes before. I'm still a Git new myself.

Alex: No, that's cool. Like I said, now that you know it's there, I imagine you'll find opportunities to use it. I can't wait to hear you tell me what that is. Cool.

I mentioned this a couple of times in terms of reasons that version control can improve your code quality. But the real key here is that commits are meant to be communication. So, committing your code is for more than just saving your revisions. It's also for leaving notes for yourself or your collaborators, like I said. The nice thing is those notes are typed directly to the work. Because of this, it's a good idea to commit often with discrete tasks. I mentioned this before, unlike Subversion, committing doesn't affect your other collaborators until you decide to push to a remote. You can leave information about what you did in the commit, as well as other ideas, like reference links, or even like issue or task tickets.

Beanstalk has this really cool feature where you can basically put an angle in like a square bracket. You can put in your issue numbers, and if you've integrated your Beanstalk account with like a Lighthouse or FogBugz or Sifter, we can automatically link your commit to that ticket. You can even close tickets right from inside of a commit. So, it's another thing to keep you really efficient, keep a clean line of communication between the work you're doing and other systems and other people.

Paul: Sure. You were talking about it more in the realm of working in a small team, correct?

Alex: Yeah.

Paul: So this would be stuff that would be online.

Alex: Yes, it would. That's correct.

Paul: Okay. All right. Let's keep rolling here.

Alex: Sweet. The log, Git you can think of as a log, but Git actually provides you with quite specifically a log of all of your commits, who they were done by, but also all of the commits that everyone has done, if you're working on a team. It keeps them in chronological order. We need a way to look at this history, of course, and that's where this next command is going to come in handy. We're going back to the command line real quick.

On the command line, if I type "git log," enter, the simple command shows the list of commits and some details about them. It's ordered from newest commits at the top. At this point, we've only made two commits. The output is pretty simple. But I want to point out a couple of pieces of information in this, because they're going to become more important later. The first thing is the commit ID. That's a unique ID to every single commit. It's crazy. In my thing here, one of them is 64C8AE0 blah, blah, blah. Basically, that is a hash, and what a hash is a unique ID that's actually generated from the contents of the commit. So, the reason that every single ID is unique is because it's created by the contents of the commit. The reason this is important is the way Git keeps track of the order of the commits is, rather than having them sequential, like Subversion, the newest commits are going to refer to the previous commit that they originated from. So it creates sort of like a family tree, and Git's able to keep track of things by knowing each of these unique IDs. Because it's a hash, it's able to verify that the contents and the ID are actually accurate because they're related to each other.

The author tag is that stuff that we put in earlier. The date is the date. Then, the actual commit message. There are a couple of different flags that you can add to the git log command that we're not going to cover today. You can look those up

in help manuals. All they really do is help format the log information. But I wanted you to know what was in here. I wanted you to know what it looks like, because we are going to refer to this in a couple of other places in the walk-through.

Paul: Let's talk a little bit more about remotes. We got into it maybe this deep earlier, but explain this, why should we either bother with a remote server?

Alex: Sure. So far you've been working completely on your own machine, or I have for the purpose of the demo. Remember Git is distributed version control, like I mentioned in the intro, which means that you can use it completely independent of other servers, other computers. But what happens when you want to share your repository with another person or even just backup your work? There are also times where maybe you want to make it really easy for you to deploy your changes to a server. Git lets you use what it calls remotes to handle all of these situations.

We're really going to focus on one particular workflow today that allows you to push your changes to a code hosting tool, like Beanstalk or GitHub. Those tools are designed to help you collaborate with other people, keep track of your changes, make comments, look at activity, so on and so forth.

So a remote is really just another Git repository. That's actually really important to know. Think of a remote as sort of like a blank canvas repository, and what you're going to do is you're going to push the entire contents of your repository, not just the files, but all the stuff that was in that docket folder, that gets shoved up there too.

Paul: So all the changes, all the tracking, everything.

Alex: Yes, the entire history that

you've made, which in this case is only two commits, but you could have been working for months and then finally decide to get a Beanstalk account. When you push it up there, Beanstalk is going to receive everything that you've ever done. There are a lot of ways to do this, some more complicated than others. Like I said, the easiest way to show you how remotes work is to use a hosted version control system.

For today's demonstration, we're going to use Beanstalk. I'm a little bit biased, but like I said, all of these techniques, there's going to be something comparable in any other tool, whether it be Beanstalk or GitHub or Unfuddle or any of the others that are also really great.

This is the part where you sign up for Beanstalk. That's my shameless plug. If you want to follow along, you want to use our tools, you can go in and grab yourself a Beanstalk account. There's a free trial for one user and 100 mgs of storage. So for you to just try it out, it's totally free. Then if you decide you really like it, there are upgrade opportunities.

Once you've signed up for our free trial, it's going to be really tempting to start by setting up your new repository, because there's a button that says "Create Your Repository", right dead center. But first we need to help your computer be allowed to talk to Beanstalk. This process can seem a little bit complicated, but it's only required once per computer, to talk to Beanstalk. The same thing with GitHub.

When you're logged into Beanstalk, you're going to go to your profile and setting section in the top right corner, and then you're going to click on the key in the sub-navigation, like you can see on my screen. If you've never used an SSH key pair before, which is entirely likely if we're having this conversation, you're going to need to generate one, which means, we're going to

head back to the terminal.

You're going to start by making sure that you're in your home directory. You do that by typing CD, tilde, slash. That just makes sure that you're in the right place to generate your key.

Then you're going to run the command, it's SSH dash key gen dash TRSA. You don't have to worry about what this means. It's something you can quite literally copy and paste or type in. Just make sure it's exactly as you see it on my screen. It's going to ask you a couple of questions. You can take all of the defaults. That includes the default location, which should be .ssh/IDRSA.pub. Like I said, you don't have to worry about what this is. Just keep hitting enter and go.

When it asks for a pass phrase, though, you should choose something strong and memorable. This is optional. You don't need to create a pass phrase. What a pass phrase does is it's just one more layer of security that makes sure any time you push and pull, it's going to ask you for . . . it's basically a password.

But basically what keys do is keys allow your computer and the remote server or the remote, whether it's Beanstalk or GitHub or even another person's computer, to communicate to each other over SSH. That's the primary protocol that Git communicates with.

The last part is, now that file, that id_rsa.pub file has been created, that's your public key. That's a file that you can actually give away. SSH does a bunch of checking to make sure that that file is compatible with another one that it stores, but the contents of that .pub file is a bunch of random characters.

You're going to need to copy and

paste that into your Beanstalk or GitHub account. If you're on a Mac, you can type the command that's on the screen right now, which is "cat" and then the "sshidrsa.pub" and then a pipe and then "ppbcopy" and that actually pipes the contents of the idrsa file directly onto your clipboard, which is going to make it really easy for you to paste into your tool.

Paul: Okay.

Alex: If we head back to Beanstalk now, and we go into the keys screen again, GitHub has a comparable one, you're going to paste the clipboard contents in the text area, and it should look something like what's on my screen. It'll start with "ssh-rsa." It will have four or so lines of what looks like random characters, and then probably the name of your computer at the end.

When you click save, you're all set. That's configured your computer to be able to talk to Beanstalk as a Beanstalk user, as your Beanstalk user specifically. It's not just allowing it, but it's tying the user that you're logged into Beanstalk with directly to your machine so you can communicate seamlessly. Beanstalk knows who you are, where you're allowed to push things to, and things like that, which is nice because as you're doing your work, you don't have to enter a username and password every time. It will just work automatically.

The nice thing is once you set this up once, you never have to do it again. This has kind of been a pain in the neck, I know, but once you've done it on your computer with your remote host Beanstalk, GitHub, whatever it is, you don't have to worry about it ever again. If you've got more than one computer, you can do this, you can add more than one key to your remote as well. Just repeat the steps that we went through, paste the additional keys in below the first one.

It's really that simple.

Paul: Yeah. This can definitely be intimidating to get started with, but again it's a one time thing.

Alex: It's a one time thing. You can literally follow the direct steps. Also in Beanstalk when you set up, we've got a bunch of really easy to use guides. Actually, if you go to guides.beanstalk.app.com, we've got a bunch of really beautifully designed, very easy, written for humans, not other computers or ubernerds, walk-throughs on a lot of these things. There is a guide specifically to getting ssh set up on Mac or on Windows, depending on which system you're using, We've made it as easy as possible to learn. All you really have to do is read through it, follow a couple of commands. It's all really just copy and paste.

Paul: Okay. Cool.

Alex: Since we've already set up the git repo on our computer, the one that I've been working on so far, now we need to add a new Beanstalk repo as a remote to the local repository. Following so far?

Paul: Okay. Yeah. Clarify that just a touch.

Alex: Sure. Remember that Beanstalk has a repo, but it's technically empty right now. We have a repo, and it's got changes in it. What we're going to do is I'm going to give you a couple of commands that you can run on your local repository, that basically point it to the Beanstalk repository and say these two are allowed to talk to each other. We're even going to create a couple of shortcuts that are going to make it less things you need to type every time you push and pull. Make more sense?

Paul: Yep. Got it.

Alex: Excellent. Now we can get back to creating our repository.

Beanstalk makes all this super easy. To save time, I've already done that part. If you go over here, you can actually see on my Beanstalk account, I've got an AppSumo workshop demo repository set up. I've already done all my ssh key work. You don't have to worry about that right now. The important part is I'm going to grab this clone URL using the copy button here. That's going to put that URL on my clipboard, makes it easy cause it's kind of big and clunky. Then what we're going to do is add that, as a remote, to our local repository.

What that means is we've got our entire self-contained local repository, all the work we've been doing, and we've got this empty repository on Beanstalk servers. What we need to do is tell our local repository that that empty repository on Beanstalk's servers is a remote. The way we're going to do that is first we're going to make sure that we're in the present working directory of the project, which is easy to do by typing "pwd." We are. If you weren't, you could "CD" your way in there. You might not be. If you were just doing your ssh key creation, you'd need to make sure that you're in the right project.

Another way to check that is to "git status" and make sure that you're actually in the right repository. Once you've verified that and you've got the remote clone URL from Beanstalk, all we need to do is type a couple of commands to add them. So, it's going to be like this. I'm going to "git remote add beanstalk," and then I'm going to paste in that URL.

We're going to step through this real quick. The "git remote add" part should be pretty self-explanatory because we're telling Git to add a remote. The word Beanstalk is an alias, and it could really be anything descriptive. If you're pushing to a different Git hosting company, like GitHub or Unfuddle, you could replace Beanstalk with a

more appropriate descriptive alias. You can actually push directly to things like production and staging, if you wanted to set things up that way and you could set up your alias there as well. So that's all that is, and again, for the sake of describing where we're pushing to, aliases help us out.

That last part might look a little bit different depending on who your host is, but it contains all the necessary information. The "git app" part in the beginning might begin with an https instead, but most hosts default to the "git app" prefix because that's how Git connect over SSH.

When I hit enter, in this case it's telling me that Beanstalk, the remote already exists because I added it just a couple of minutes ago before our video died. But normally it would come back and give you a simple no response.

Now, you're ready to do your first push to your remote.

Paul: All right. So what this push is going to do is take everything that's on your local machine and push it up to this other repository.

Alex: Correct. We haven't really talked about branches, but sort of built into every Git repository is one branch. It's the master branch. So, what we're really going to do is push the master branch from our local repository to the empty repository at Beanstalk. We're going to be using the command "git push beanstalk master."

I'll show you what that looks like. If everything goes well, it will tell you it's counting some stuff up. It will talk about delta compression. It will do a bunch of whiz bang things, and if everything goes according to plan, what I just saw on my screen will be what you see on your screen.

Sometimes you'll get something

about the remote not being able to be connected to or this weird, it says "unexpected hangup." Nerds write the weirdest error messages. But the point is the odds are something was wrong with your public key configuration, and you're going to need to go back and check that out. If you happen to be using Beanstalk, just drop us a line in support. We can help you out. We do it all the time.

If you go back to Beanstalk, this is the cool part, and I hit refresh, you'll see everything that was in our log now presented a little bit more pretty right on the activity screen.

What I want to do, I want to connect some dots. We're looking at the activity screen in Beanstalk, and this is, like I said, it's essentially a pretty interface for what you saw on Git log. You'll notice each commit has this eight character ID. That's the first eight characters of that hash that we were talking about when we were reviewing Git log. You've also got the author information. That's how it figures out who you are. It will tell you a little bit about the branch that you are working on and a list of the files that were contained in the commit.

Paul: Okay. So, this is just pulling everything together.

Alex: Yes, exactly. This is just giving you a prettier interface, a more usable interface for doing things like code review. If you were to actually click on the commit itself, that ID, or even the description, it will take you inside of the commit, and you'll get a visual presentation of the changes that were tracked in the commit, the insertions and the deletions.

We mark deletions as red with a little negative next to them and insertions as green with a little plus, to sort of follow the themes inside of Git.

Remember we described the diff as a differential, or the difference between two files. In this case, this is just showing the difference between this commit in the file and the state the file was in, in the previous commit.

Beanstalk gives you some handy tools to let you leave comments, leave yourself a to-do, leave a note for a team member, more about the contents of the file. You can subscribe to those kinds of updates and things like that.

Paul: Okay. Then other remote servers will have similar things.

Alex: Exactly. Yes, you're going to have tools that will let you view the files as they are or they'll let you view the differences between files in different useful ways. Git's also got tools like Blame, where you can actually view a file and it will show you, for each line or set of lines that were last touched by a certain person, it will tell you which one it is. So, if your website's breaking, if your web app is breaking and you're getting an error on a certain line, you can go back to the file and the line number and see who the last person that touched it is, and go talk to them about it. It's pretty useful.

Paul: Yeah, that makes a lot of sense. With small teams, like this is why my mind was just blown when I started to play with the whole Git thing. There's just so much data that you can pull out from this.

Alex: Yeah, it's really useful. A lot of teams use this part of the workflow for code review. So that activity screen that I was showing you is a great opportunity for other team members to jump in and look at your code, ask questions, and really up the quality of what you're working on. Also, it's good for your own code review, just going in and checking out what you worked on, because sometimes you forget.

Paul: Sure.

Alex: So the next part of our remotes, this is going to be a little bit difficult to demo because I'm only one person on only one computer with the remotes. We're going to try to simulate two people pushing and pulling between remotes. Before we even get there though, I want to talk about what the process of pulling is.

So what if you were just joining this repository that you and I have been working on for the first time, and you wanted to get all the files as well as a local copy of the entire commit logs, to see what I'd been working on. It wouldn't make sense for you to set up an empty repository locally, set up a remote and try to pull everything down. That would basically be the reversal of what we just did, right?

Luckily Git gives us a really simple tool called Clone that lets us set up a local repository, get all of the files and the history, and even configure the remote that you cloned from as a remote all in one shot. It's one command, and it does all of that at the same time.

So I'm going to show you what that looks like. I'm going to go back to Beanstalk, and we're going to go back to the activity screen, because we're going to need that clone URL again. So I'm going to copy that and have it ready in my clipboard.

Like I said, we're going to cheat in this demo. Instead of pulling to a new computer, we're going to simulate doing something totally crazy. I'm going to simulate a catastrophe. Yeah, you ready for this?

I'm actually going to take the entire folder that we've been working in. I'm going to drag it to my trash, and I'm going to empty my trash. Oops! I just deleted all of our work, all of our history, and I'm screwed!

If you, for whatever reason, are following along in the demo and you haven't been able to add your remote and do your first push, you're going to want to not try and replicate this part of the demo. We can do this because we've pushed everything up to Git and we don't have to worry about it. We don't recommend doing this as common practice, obviously. But for today's demo, it actually let's us demonstrate things and get a point across.

So if you see all the files and changes you committed earlier in Beanstalk, and you really want to try and delete a project, go ahead and get rid of it. Just like I'm no longer responsible for what you do.

The next step we're going to do is clone that location, or rather clone the remote that we pulled down before. So I'm going to make sure that in the terminal, I'm going to be on my desktop and I want to clone there, like I did before. So you're going to do "git clone," and then I'm going to paste in the clone URL from Beanstalk. I want to hit enter.

It's going to sort of do a reverse of that first push. It's going to say "cloning into." By default, it clones into a folder that's the same name as the repository you just cloned from. So in the case of when I set things up in Beanstalk, the repository was called AppSumo Workshop Demo, same as before. So when I cloned, I didn't have to tell it anything. It just automatically created that folder, copied down all the files, copied down all the revision history, and it also does all this super- compressed. If you've ever used Subversion, you know how slow it can take you to do file by file by file. Git's wicked fast when it's moving stuff over the wire, (a) because it's just passing deltas, but even when it's doing this initial checkout on really big repos, Git is orders of magnitude faster just because it compresses

everything before it transfers it.

So if I change directories into that directory, you'll see magically I've got all of the files that had been added in track before, which is pretty neat. But that's not the really cool part. If I run "git log," you'll see that the entire log from before is exactly where we. So a deletion be damned. We're exactly where we left off, which is pretty awesome.

Paul: Yeah, that is. That is really slick! I was thinking another time that this could be useful, let's say that there's an open source project that you want to hack away at. You could just go and grab that and pull it all down to your local machine as well.

Alex: Exactly. So GitHub is awesome for open source. If you're browsing GitHub and you find an open source project, one of the first things they have right up top is a clone URL. What that lets you do is the same thing. You copy that, and then you just go to the command line. You type "git clone" and then you past that in, and the next thing you know you've got not just the entire code from that project, but the entire history locally too. You can bang on it. You can work on it, do whatever you want. If you're running a fork, which is another really cool thing that GitHub lets you do, you can actually push it back up and have that open source project accept some of your changes.

Paul: Cool.

Alex: I said we were back where we left off. That's not entirely true. Remember when we created that alias and we called it Beanstalk? If I do "git remote," which is a command to tell Git to list out all the remotes that are configured, you'll notice that it just lists Origin. Origin is this silly default alias that it always creates when you do a clone. If you want to rename your clone to be something more descrip-

tive, it's really easy. You just type "git remote rename," and then the original one, which is Origin, and the new one, which is going to be Beanstalk. I hit enter, and if I do "git remote" and hit enter, it's now just listing Beanstalk. So, you're remote is Beanstalk again. Now we're exactly where we left off.

Paul: All right. Now, let's start talking about Git pull.

Alex: Sure. Technically, you're already done a pull, since it's part of cloning. It was just done for you automatically. It's sort of an opposite of Git push. The command, "git pull," asks the remote for all of the changes from the remote that your local doesn't currently have. In fact, if you're collaborating with somebody and the remote changes and they make changes on the remote that you don't have yet, Git will force a pull to your local repository before you're allowed to push to it. So, Git does do some things automatically, and it gives you easy to read messages to know. It says, "You can't push until you pull again." When you pull, it fetches any changes on the remote automatically and merges them in, all in one smooth motion. As you can tell, because I covered that really quickly, this is the potential to get very confusing. So we're going to keep it super super simple, because that's all you really need for today.

In order to demonstrate, instead of setting up another computer, I'm going to clone again into a different folder on my desktop. So, I'll show you how that works.

Paul: Okay.

Alex: Back in my terminal, I'm going to go to my desktop, where you'll see I've got my AppSumo Workshop Demo. I'm going to do another "git clone." So I'm going to do "git clone," and I'm going to paste the same URL I did before. This time to the end I'm going to add a specific

folder name. So, remember before, it cloned into AppSumo Workshop Demo. This time I'm going to clone it into Other Guys Copy. We'll actually call it Paul's Copy. Cool? You'll see now it's cloning into Paul's copy. Other than that, everything happens exactly the same way.

Paul: Okay.

Alex: This can be handy if you're cloning from a remote and they've got like a wacky name for their repository. This gives you a really easy way to rename it. If you do decide to rename the folder that Git clones into, it doesn't break anything, because remember, all the Git data is stored in that .git folder, which is right inside of that.

Paul: Okay. That makes sense.

Alex: Cool? So, if I CD into your working copy, Paul's copy, and I'm going to open it up in TextMate. We're going to make a change. What should we change? I'm going to delete these three steps, rather you are Paul. You don't even know it yet. So, I'm going to delete those three steps, and I'm going to save the file. I'm going to do a "git commit-M, removed the steps. They were stupid." Oh, this is great. I tried to do a "git commit," and I did a git commit-M, with an inline message, but the problem is that the file that I just made a change in, I forgot to add it to the stage. Git reminded me of that. My file is modified, but it's red. Remember when it's red, you need to add it. So, "git add index." Now I can rerun my "git commit." Boom. One file changed, zero insertions, ten deletions. That's exactly what you did. You deleted a bunch of lines.

Now if I do a "git push," what it's going to do is take those changes and sling them up to Beanstalk. Remember, it knows where Beanstalk is, because we did a clone, and clone automatically sets up the remotes. Right?

Paul: Okay.

Alex: If I go over the Beanstalk and I hit refresh, you're going to see . . . this is not the best demo, because I'm the same user pretending to be two people. So it's going to show that the changes were made by me. But the point is that the changes were made and they were pushed up.

Paul: Right.

Alex: So I'm going to close Paul's copy, and instead I'm going to open up the AppSumo copy, and you'll see the three steps are still there.

Now check this out. If I, in the command line, go over here and I do a "git pull," it's going to come down. It's going to tell me that one file changed. There were ten deletions. It tells me what file it was, and Git even has these little meters where it shows you the relative deletions to insertions and stuff like that so you have an idea of what was done in that move.

Now if I switch back to TextMate, the changes are gone, exactly like I wanted them to be. So effectively what happened is we've made a simulation of you, Paul, making some changes in the repo, pushing them up to Beanstalk, and then I decided to pull the changes that you made down from Beanstalk, and they're automatically applied to my local repo. It's pretty cool.

Paul: Right.

Alex: At this point, we've explored all the basic operations of working with a remote on your Git repository. It can be complicated, but between adding remotes, pushing and pulling, that's really the main functionality.

Paul: Correct. That's actually the three things that I really focused on when I was first getting started with this.

Alex: Right.

Paul: Once you have that under your belt, it's easy to start stacking some of these other things on top.

Alex: Exactly. So that's why we're going to move on to branches.

I mentioned this before pretty quickly, but Git's sneaky, and without knowing it you've already been working with your first branch. I've mentioned master a bunch of times, and master is the default branch for every repository that Git creates. It contains the history for all the files it's tracking changes in.

So far everything you've learned you can use on other branches as well. Now you just need to know how to create them and switch between them.

So I have the doc up here. You can't see my screen, but you know what my slides are. I've got the doc up here, because a Git branch is like an alternative timeline for your work. If you need to diverge from the main path to tackle something like a new feature idea or a bug fix, you can create a branch and you can do the work there, because anything done in that branch won't touch the master branch. So you can keep it clean, continue deploying from it, and things like that until you decide you're ready to.

So branching with Git is actually really easy. So we're going to head over to the terminal. I'm going to close some of this stuff I have open here. The first thing I want to do is show you a list of the current branches, and that's easy with just "git branch." You'll see that there's just the master branch, and it happens to have a little asterisk next to it. If you ran those color commands from earlier in the workshop, it'll be marked as green as well.

Paul: Okay.

Alex: That's the branch you're on; the default master branch. The one with the asterisk that's green, that's the one you're working on.

So what we're going to do is we're going to create a new branch by issuing this command. We're going to do "git branch remove CSS link." Oops. I started typing it as different words. It needs to be hyphenated or ideally all one word. So we're going to do "removeCSSlink" all one word. Now, if I do "git branch," you'll see that there are two branches listed. The master's still the one that we're on. We're not just there yet.

Git has a command for changing the branches that might look familiar if you've used Subversion, and that tends to be pretty confusing for people. It's the checkout command. In Subversion, checkout is used to pull all the files out of a remote, out of the central repository. But with Git, checkout is used for switching between branches. I have no idea why they chose the same command. But once you know that, it's not that hard to forget.

So what we do is we run "git checkout removeCSS link," enter. It'll say "switch to branch," and then the one that we just typed in. Now if I do "git branch," you'll see that that removeCSSlink branch is the one with the asterisk next to it and it's green. At this point, the removeCSSlink branch is identical to the master branch, and it will be until we make any changes.

So I'm going to move into . . . we're working in the AppSumo Demo Workshop. I want to make sure we're working in the right project now that we've got a fake Paul remote. We're going to go into the index file here, and we're going to strip out, we're going to pull out this IE 7 style sheet because we're not going to support IE anymore, because why would you?

We're going to save it. We're going to close it. What I want to do real quick is switch back to master, "git checkout master." You can double check. It will tell you a little bit about what maybe happened. We're not really going to get into that today. But if I do "git branch," we can confirm that we're on master. If I go back to those files, the IE7 link is still gone, but I told you that the branch that we made was an alternative timeline. That's technically true, but Git works in snapshots called commits. So Git doesn't know which branch your CSS link change was made on until you make a commit.

So, let me show you how that works. If I go back to here, we'll switch back to the "git checkout removeCSSlink." I do "git commit." I'm going to do a little shortcut here, because remember before I accidentally did an inline commit-M, and it told me to add it? What I can actually do is "git commit-AM.' That'll add and do an inline message all in one shot. So, we're going to say, "Removed IE7 CSS." That's my commit. One file was changed, and now, obviously, we're on that branch, so that's how it is. But if I go to "git checkout master" again, you'll see that on the master branch the IE7 CSS is still in place. So, it took the commit to tell Git that the change we made was on the branch that we wanted it to be on.

Paul: Okay.

Alex: That's probably the most common mistake that people make when they're working with branches, is forgetting that the change isn't committed to a specific branch until you actually commit it while you're working on that branch.

Paul: Okay. I definitely missed those little things when I was first starting out.

Alex: Right. So, the idea is relatively

simple, though super powerful, because think about it this way. You can develop entire features, explore new ideas, and fix bugs this way. Imagine being able to produce an entire new page and related navigation while you're sort of redesigning, and not have to worry about commenting out stuff when you move files to the server to make sure that you're not bumping into your own work. Branches let you do exactly that.

Paul: Yeah, branches are just huge. They can save so much time.

Alex: Yeah, absolutely. First we're just going to show how the basics of branch integration works.

Paul: Okay.

Alex: If you work too long on a branch, it's possible for you to get behind on changes that were made to the master branch. Because of this, many people only use branches for small features. If you wanted to work on a big feature or something that was really twisted into your code, Git provides this tool called Rebase. Rebase basically lets you periodically sync changes from your master branch back into your branch work. As powerful as this can be, it's more complicated than we really have time to demonstrate today. But the quickest explanation that I can think of is imagine sort of rewinding your work, replaying the stuff, basically applying the master branch stuff to it, and then replaying your work over top of it. So it's sort of like working in reverse for a little while, as far back as you diverged from master, then reapplying master changes to your branch, and then fast forwarding and replaying all the work that you had done with the opportunities to do any conflict resolution along the way. Like I said, it's complicated. I've used Rebase twice in all the time I've been using Git. It's powerful, but you don't need it very often.

The other thing, since we're not covering a whole lot of it, is Scott, who's like the main Git evangelist over at GitHub, has a book called "Pro Git." There's a printed version, I think, from A Press, but there's a free version that's fully online at ProGit.org. He's got an awesome chapter on Rebase that explains it better than I ever could, and I highly recommend checking that out, if you're interested in how Rebase works.

Paul: Okay. Great.

Alex: So, besides rebasing, there are still a few things you may want to do with branches. The two that we're going to talk about today are merging and deleting. I'm going to show you how that works next.

So, like branching, merging with Git is typically pretty easy. We're going to show you how to merge the changes from that CSS change branch that we worked on just before, back into master. In order to do that, we're going to jump back to the terminal.

The first thing you need to do in order to merge is make sure that you're operating on the branch that you want to merge into. So in this case, we want to merge changes from our CSS change work branch into master. So, we're going to "git checkout master." Make sure we're operating on master. My screen is telling me I'm already on master, so we're good to go. All we need to do to merge . . . was is it removeCSS? I can't remember what we were doing. Get a branch. It was removeCSSlink was the name of the branch. So we just need to do "git merge removeCSSlink." We hit enter and Git will come back and it will tell you, basically, all of the things that it did.

We've only touched one file, so it's going to be really simple. But if

you were merging in lots of work, it would apply that to all the files. It would change all the files, and so long as the work didn't have any conflicts, which we'll get into in just a second, you'll just get a confirmation saying everything went well.

If we go back to the file and we look at it, even though we're working on the master branch, even though we're in the AppSumo Workshop Demo folder, that CSS link is gone. If we go into "git log," you can see that the log now has the moved IE CSS commit in it, from the master branch as well, and we're good to go.

Another good practice with version control, especially with branching, is to try and clean up your branches. You can sort of end up with this spidery mess. Once you've merged a branch in, it's a pretty good practice to delete that branch. So it's really easy to do that by typing "git branch-d" and then the branch name. So, that was removeCSSlink, hit enter. It will say "deleted branch removeCSSlink" and it will tell you what revision you merged in at.

If for some reason you were working on a branch and you decided to scrap that work and not merge it in, you need to use a capital "D" to sort of force a delete of a branch. Git won't let you delete a branch you haven't merged, unless you use the capital "D" to force a deletion, if that makes sense.

Paul: Okay. So it's just a safety precaution.

Alex: Exactly, and if we run a "git branch," the only thing left is my master branch. The other branch is totally gone; the chain has been merged in.

We've talked about conflict resolution a couple of times. Merging is extremely powerful but it's absolutely not perfect. Sometimes

you'll attempt a merge, either as part of a pull or a rebase, applying a stash, any of those sorts of things and automatic conflict resolution won't know what to do. The times this happens, remember we talked about insertions and deletions? Conflicts happen when an insertion or a deletion happened on the same line at the same time and Git's not sure which one to prefer.

Paul: Okay.

Alex: So, rather then matching up your files by guessing, Git decides to tell you that it wasn't comfortable doing the merge on it's own and it wants you to assist. You can experience these merge failures in two different kinds of ways.

The first one is when you didn't commit yet. I mentioned this before, but you were thinking of doing a pull before committing your latest work, and that's probably the most common thing you're going to run into.

Remember, doing a pull is effectively a fetch as well as a merge, all in one move. So, if you haven't made a commit, Git doesn't have the information it needs to apply the merge from the stuff it fetched on to your local repository. So it requires you to do a commit before you do a pull. Luckily, Git catches that for you, if you've got uncommitted changes, warns you and tells you what to do. You can choose to either make a commit, or this is a great opportunity to use a stash if you're not ready to make that change permanent.

Actually, in fact if I remember right, when you do that pull, it will tell you to commit. It also offers, says you might want to stash them before you merge. I've got that on my screen right here.

So, the other type of area you're going to run into is you did commit, but the changes you committed were too close. They were

made on the same lines, and it's not really sure what to do. This scenario is more likely the bigger your team gets and the more people are working in the same place, in the same files. But it also happens when you apply patches via Git, if you're working on an open source project. So it's a good idea to know how to handle it.

Once again, Git's tracking insertions and deletions. If two commits attempt to merge and they both have the same lines marked for insertion and deletion, Git doesn't know which one has priority and it turns to you. That's something I'm going to teach you how to deal with now.

Paul: All right.

Alex: So, you'll know this is the case when you have something like what's on my screen right now. It basically says there is a merged conflict in index.html. It'll tell you that the automatic merge failed, and it'll tell you to fix the conflicts and then commit the result. That's exactly what we're going to do.

I'm going to need to simulate that though. We're going to use the fact that we have these two versions of the repository. I'm going to go into Paul's copy real quick, and if I remember right, there's a Facebook link in here, and it says follow us on Facebook. But Facebook just made all these changes, and we don't like it anymore, so we're going to change it to Twitter. So, we're going to make it Twitter.com. I'll say, "Follow us on Twitter," and we're going to save that.

I'm going to make sure I'm in the correct working copy. I'm not. So I need to go into the Paul's copy version here. "Git status," and it shows me that I've got a file that was modified. "Git commit," and do the AM thing again to save us some time. Change Twitter to Facebook. Enter, go. I'm going to do a push real quick just to get that up there

onto Beanstalk.

Now, I'm going to take off my Paul hat, and I'm going to put on my Alex hat, and I'm going to go back to my working directory, the AppSumo one. I'll close Paul's copy. I open up mine. I'm going to go work on the same file because I didn't know that Facebook made these changes today, but what I saw was follow us on Facebook. That doesn't make any sense. You should like us on Facebook. So, I'm going to change follow to like and I'm going to save it.

You realize what happened here? I just made changes to the exact same line of code in the file that you made changes in. That's because we didn't get a chance to talk to each other about what was happening in the world of Facebook. So, what's going to happen is I'm going to commit this change. "Git commit," and change follow to like. We're going to do a "git push," and it's going to tell me, hey, you can't do that. There are changes on Beanstalk. So, you need to do a pull first.

We're going to do a "git pull," and then it's going to yell at me and say, there are merge conflicts in index.html. Your automatic merge failed. Fix conflicts and commit the result. Now, what the heck do I do?

The way conflict resolution works is you are actually working in the file that the conflict exists in. So, we're going to go into index.html, and you will see that it starts and ends with these angle brackets. The angle brackets are the beginning and the end of the conflict. In between them is a bunch of equal signs showing like a little bar dividing between the two options.

So, my job between them is to remove the conflict delimiters, is what they're called, and choose between them. So, I didn't know that we were going to Twitter, but I came over and talked to you, and

you said we are. I'm going to delete the Facebook version. But I also read on some blog that if write "you should follow us on Twitter," you'll get more followers. So, I'm going to make it you should follow us on Twitter.

This is a pretty complex resolution even though it's very simple. What we've done is we've taken two changes, we've merged them into one, and I made yet another change. So, now what I need to do is save this. You "git commit," and I'm going to say, "Merged my stuff with Paul's. He was right, my bad." Then, I can "git push" that. That goes up to Beanstalk. You're able to pull that down, and we're in synchronization again.

Paul: Got it. That's a common stumbling block that I kept tripping over when I was just getting started. It seems like when you're working on a team this can happen pretty frequently.

Alex: It can. It really depends on your work style, your team, and how much communication you have in between your coding sessions. Conflict resolution often is an opportunity for you to talk to a team member, which is a good thing as well, and make a smart decision about which side of the conflict to choose or to merge them together in an intelligent way.

Paul: Awesome. Alex, is there anything else that you have in this AppSumo action video for us?

Alex: I don't have a whole lot more. I'm pretty sure you've learned more than enough to get started using Git to manage your source and your projects, but we've only scratched the surface of version control as far as all the things you can learn. As your skills advance, you may want to continue learning it, and I hope you do. There are tons of awesome resources online. I mentioned Guides.beanstalk.com, and there are tons of other great

stuff.

One of my favorites is a site called "Getting Good with Git." You can pick that up. There's another one called "Git Immersion". If you Google for either of these, super, super tools that will help you learn more advanced techniques than we talked about today.

Paul: Awesome. I'll get those links from you, and we'll have it the video here.

Alex: That would be great. The other thing that I'll say is at this point you have enough knowledge now to experiment with some of the GUI clients out there. I really like Git Tower for Mac. There's also Tortoise Git on Windows and a number of others. You'll still likely end up back at the command line from time to time. Conflict resolution, adding remotes and things like that, those tools don't do very well, so be sure to practice your command line skills that you learn today so you don't get rusty.

Paul: Awesome. And then, Alex, how can people contact you just in case they would want to get in touch?

Alex: So, you can email me, alex@wildbit.com. You can send an at message to @BeanstalkApp. I'm usually the one responding to those. You can hit me up on Twitter as well. I'm AlexKnowsHTML. Actually, I've got that right here on my last slide.

Paul: Awesome.

Alex: That's easy. I don't have to sit here and spell it.

Paul: Alex, thanks again for your time.

Alex: My pleasure. Thanks, Paul.