Appendix G

Tracking Your Changes: Version Control Using Git And GitHub

(Note: While not required to complete this book, learning to use Git and GitHub is well worth the effort. At first you may use these tools only to backup and keep track of your work; over time you will learn to use and appreciate all of the functionally that these provide, and that will serve you both as a student and in the workplace. The material provided here, and in the companion tutorials on the textbook Web site, can be studied at any stage. Some students may prefer to defer this topic until after completing the course, while others may wish to take advantage of this software while working through the chapter exercises.)

Introduction

Perhaps the best way to introduce version control is to consider how you work on the exercises in this book. You may have quickly learned to save your code a number of times, using a slightly different file name each time, so you could retrieve a previous version if you got stuck. For example if you were working on an exercise file named **software-order.php**, you might have saved your first modifications as **software-order-1.php**, your second as **software-order-2.php**, and so on. You may have also learned to make a note in the comment section of each file to remind yourself of the changes that you made since the previous version. Once you completed the exercise, you might have then saved or renamed the final "production" version as **software-order.php**, and either deleted the previous versions or kept them as a history of your work.

This is a basic version control system: beginning with an initial version; developing, documenting, and saving different versions whenever you make significant changes, or complete different requirements, or simply stop for the day; and eventually integrating these changes into a final working version. Each version captures your work at a particular stage of development, so that you can refer back to it if needed, or even revert to a previous version if your current work is going nowhere.

Saving files with different names is better than nothing but is not a very efficient version control system. There is the problem of managing an increasingly large number of files, with similar names and often containing much of the same code. There is also the challenge of tracking and documenting your changes clearly. There is the difficulty of synchronizing changes made to multiple files (for example **include** files, **.css** files, image files, etc). You may have already experienced how confusing and overwhelming it can be to keep track of everything, even when developing small applications.

What is a Version Control System?

A **Version Control System** (VCS) will allow you to : track and synchronize changes made to multiple files (code, stylesheets, media, documents, data, etc); compare and retrieve different versions (even line by line) as needed; maintain a complete, documented, version history of a project; develop different requirements independently (merging these with the main development only when the work on each requirement is completed); collaborate with others at each stage (design, coding, testing, review, maintenance, etc).

This appendix introduces a widely used version control system known as **Git**, and a related online repository, **GitHub**. Used together, these tools are equally effective when managing small, one-person, applications (even textbook exercises!), and extremely complex development that involves large teams, sometimes thousands of professionals collaborating worldwide. The basics are easy to learn and master, and the textbook Web site provides **two hands-on tutorials**. The first tutorial demonstrates how to: install Git; create a sample Git repository on your local computer; and use Git to track changes while you simulate the development of a simple application. The second tutorial more or less repeats the same procedures as the first, but this time takes advantage of the resources available in GitHub. Together these two tutorials will introduce key terminology and procedures, and provide you with a working knowledge that you can put to immediate use in your work.

Git and GitHub

Git is a widely used, open source, and freely available VCS that runs on **Windows**, **macOS**, and **Linux** platforms (the Git system was created by Linus Torvalds in 2005 when he was engaged in developing the Linux kernel). **Git** tracks changes across your entire application development, taking a "snapshot" on request. These snapshots constitute a history of your work process, and among other things, allows you to return to any previous stage of your work whenever it suits your purpose. Once Git has been installed, you can define any work folder to function as a Git **repository** or **repo**, which allows you to apply Git commands to the files in the folder.

While Git functions very well as a standalone tool for local development, it is often used in conjunction with a cloud-based hosting and version control service called **GitHub**. GitHub is founded on Git tools, and facilitates code sharing, project planning

and management, version-based documentation, collaborative development, remote storage, and product distribution. It is very common practice to use Git and GitHub together; a standard approach is to maintain your application's "home" repository in GitHub while downloading (**cloning**) a version to your local git repository, which is the version that you work on. This practice ensures that your modifications are developed separately from the "production" version in GitHub. Your local work can be uploaded (**pushed**) back to GitHub at any stage, as often as needed, and, once in GitHub, may optionally be shared, reviewed and tested by others before being merged into the production version of the application. This production version can be public or private; you may have already used GitHub to download publicly available software, or other resources, provided by other developers.

While GitHub is a proprietary product (owned by Microsoft), and offers a number of priced plans, you can sign up for a free account that provides a comprehensive set of features at no cost. GitHub is extremely popular among developers: at the time of writing GitHub has over 40 million users, and is the largest host of source code in the world.

Using the Tutorials

No previous programming experience is required to use the tutorials, except that you will work from the command line and will need to change directories (a quick read of Appendix C will provide everything you need to know). You can work through the tutorials at any point while working through the textbook, or simply download them for future reference. You should probably allocate about four hours to complete the tutorials and absorb the information.

These and other tutorials can be found on the textbook Web site, under the **Students** menu:

```
https://www.mikeokane.com/textbooks/wbip/support.php
```

The remainder of this appendix describes basic Git procedures and terminology, and explains some common commands (this material is also included in the tutorials).

General Procedure and Terminology

Once a folder has been initialized as a Git **repository**, or **repo**, you can issue Git commands to track and manage changes made to the files within the repository (this includes changes within files, creation of new files and deletion of files). These files may contain code, documentation, media, data files, style sheets, notes and references; all that matters is that the files are stored within the repository (which includes any subfolders). The Git **init** command must be used to initiate a folder as a Git repository.

The Git version control process is based on a series of "snapshots" of your application development: each snapshot records the changes you have made since the previous

snapshot. These changes may include: code additions, modifications, and deletions; file additions, deletions, and renaming; media development; documentation and research; data acquisition, etc. When you wish to preserve the latest changes you have made to your repository you simply take a new snapshot. These snapshots are called **commits** in Git terminology, because each snapshot commits your most recent changes to the repository.

Since you are likely to be working on multiple files between commits, and since you may not necessarily want **all** changes to **all** files to be included in the next commit, Git provides a **two-step process**. Before issuing a commit, you must first **add** file names to a "staging area", to indicate that the modifications made to these files since the last commit are to be included in the next commit. Once you have listed all of the files to the stage with changes that you wish to be recorded, you can issue a **commit** command. Git will then compare the content of these files at the time the file name was added to the staging area with their content the previous time they were committed, and any changes will be recorded in the commit.

This process of committing versions as you work allows you to easily maintain a clear history of your development, and the staging requirement means that you can commit versions that reflect changes made for a specific purpose, such as adding a new feature, or completing a task. A commit can also simply capture all of the changes you have made before heading off to lunch, or finishing work for the day, just like making a backup for peace of mind. Git makes it easy to review your history of commits, compare different versions, and go back to a previous version if needed.

(It is not unusual to add a file to the stage and then realize it needs some more work before it's ready for the next commit. That's not a problem: you can add the **same** file to the stage multiple times, in which case only the changes that appear in the latest version will be included in the next commit.)

A unique ID is generated to identify each commit, and you are invited to provide a description as documentation (which can be a few words or more extensive). Once a commit is executed the stage is cleared so you can start adding files for your next commit.

Imagine a sequence of these commits, forming a version history of changes to your application. The entire sequence is called a **branch**, and every repository includes a **main** branch that tracks changes that have been applied to your core application. A pointer called the head points to the latest version (commit) on a branch. If you are not happy with your most recent commit(s) and want to revert to a previous commit in the branch you can **reset** the head to point to an earlier commit.

A single repository can contain multiple branches. A **main** branch may be all you need for a small application, but it is often useful to develop different components, or experiment with your code, on separate "side" branches. You can commit changes on each branch, and then **merge** these changes into the main version if and when you are ready to do that.

A new branch is created with the Git **branch** command and, once created, you can switch to the branch using the **checkout** command. Remember that the **main** branch contains the history of changes (commits) that have been made to the main version of your application, and that the head points to the latest commit. When you first create a new branch, its head will initially point to the same head as the main branch. Once you checkout the **new** branch, your future commits will be added to the new branch, until you checkout a different branch. You can checkout different branches as needed, just remember that your commits are always applied to the branch that you have currently checked out. This may sounds confusing; let's say you're working on a film ratings application, and you've added a branch to develop a new feature that allows users to add new films; you checkout the new feature branch and make commits on it as you work on this new feature; while you're working on this you notice a minor bug in the code, unrelated to the new feature; since you don't want this to be recorded as a commit in your new feature branch you checkout the main branch, fix the bug, commit the change on the main branch, then you checkout your new feature branch again to continue working on the new feature. When (and if) you are satisfied with your new feature, you can tell Git to **merge** the branch commits with the main branch. If any conflicts are found Git will report these to you so you can resolve them. Once the merge process has been successfully completed, you can delete the new feature branch since the commits that tracked this work are now merged into your main branch development.

Don't worry if this is a bit confusing when you first read it. The hands-on tutorial will clarify the process and demonstrate how easy it is to issue Git commands and work between branches, while developing your application.

Summary of Common Git Commands

Here is short summary of the Git commands mentioned above, and a few others that you will use in the tutorial:

git init to establish your project's working directory as a Git repository.

git add to add files to the Git staging area, in preparation for the next commit.

git status to see a list of the files that have already been added to the staging area, and also a list of files that have been modified but not yet added to the stage.

git commit to create a snapshot of all the changes made to the files that are listed in the staging area at the time that each file was added.

git branch to create a separate branch for your development work, allowing you to commit changes to the new branch instead of the main branch.

git checkout to switch to a different branch. When you do this, your future commits will be associated with this branch, until you checkout a different branch.

git merge to merge your commits on one branch into another branch (usually the main branch). If Git discovers a conflict (changes made to the same lines in both files) it will stop the merge and alert you of the conflicts so you can resolve them.

git diff command to compare differences of all kinds (for example, to view the changes you have made to a file since it was added to the stage, or to view differences between two different commits).

git reset to return to (restore) a previous version of your application. This command includes an option to remove all changes made in your application files since that previous version.

git log to view a history of your commits. The history includes the ID and description of each commit, which is useful since you may need this information when issuing other Git commands

Each of these commands have many options and arguments allowing a great deal of customization, and there are also a number of other Git commands available for your use. For a full list of Git commands and their options, see:

```
https://git-scm.com/docs/
```

Note also that a number of Git GUI clients are available, removing the need to work at the command line, and providing additional development support.

Introducing GitHub

You may have already used **GitHub** to download software or other materials. GitHub is a Web-based hosting platform designed to provide a global resource to support code development, version control, collaboration, code-sharing, and project management. Whereas Git is open source and freely available as a local resource, GitHub is a cloud-based Microsoft product and service. At the time of writing the basic GitHub service, including storage, is available at no cost, with additional services available at different fee levels. Anyone can create a GitHub account, and GitHub works very well in combination with local repositories. There are other online Git-based platforms, but we will focus on GitHub since it is currently the most popular world-wide.

Once you have a GitHub account you can create, maintain, update, and manage your repositories online, while continuing to modify and update the content in a local Git repo, working with a downloaded (**cloned**) copy. This work model is very common: the stable version (often production version) of an application evolves on the main branch of the GitHub repository, and whenever work is to be undertaken, a branch is created for that purpose. The branch is usually cloned to a local repository for development, and when the work is complete it is **pushed** back up the GitHub branch, and "pull requests" are issued, inviting review, testing, and modifications, until the work on the branch is approved for merging with the main branch. This model works just as well for solo development as for larger projects involving teams, even projects that are open to anyone who wishes to become involved. Access to a GitHub repo can be kept

private, shared with selected users, or made fully public. Collaboration is at the heart of modern application development and GitHub provides a range of collaboration and shared project management tools.

In Conclusion

Learning Git and GitHub is a sound investment, well worth the small amount of time and effort that is required. A good approach is to start simply, using these tools to keep track of your changes, and to back up your work, and then extend your use, and reach out to other developers, as you gain experience. It is a very good idea to list your experience with Git, and any involvement in collaborative work on GitHub, on your professional resume.

For more about Git and GitHub see:

https://git-scm.com/

https://github.com/