**Summary**

In this module, you learned how to collaborate with others by using Git and GitHub.

Andy and Mara are well on their way to implementing a system that will allow them to better collaborate as a team and help ensure that only quality code is merged to the master branch.

A build badge and dashboard widgets help the team and others understand the state of the latest build and track the build history over time.

Although Andy and Mara are the primary code contributors, setting up a rule on GitHub to require a review is a good way to enforce a disciplined code review practice. Even minor errors can cause a build to break. As an example, you watched a typing error make its way through the build and into the hands of the QA team.

**Learn more**

**Try GitHub Desktop**

In this module, you made basic changes to the *Space Game* website. In practice, your changes will likely be much more complex. Although you can do everything you need from the command line, GitHub Desktop  is a great way to visually compare and commit your changes.

**Learn more about Git**

Although you can learn Git with just a few commands, full mastery will enable you to perform more complex kinds of merges and understand the history of your code base.

git-scm.com  and the book Pro Git  are two excellent resources for learning more about Git.

**Define your workflow**

The workflow you used in this module is a standard way to get started. Your team can refine it to suit your needs. You'll find many resources and perspectives on the web from various teams with varying needs.

Understanding the GitHub flow  introduces an approach that's similar to what you saw in this module, and it includes tips that you can apply to your own workflow.

A successful Git branching model  proposes a more advanced branching and merging strategy.

How to Split Pull Requests  describes how to split large pull requests into smaller ones to help others more easily understand your changes.

How to Write a Git Commit Message  teaches you how to be a better collaborator by writing effective commit messages.

Build pipeline triggers  explains how triggers enable you to control which files or Git branches cause a build to occur.

**Appendix**

The following sections are optional. They provide additional information beyond what's covered in this module.

**Source control migration strategies**

Now that you're familiar with Git, let's talk briefly about how to migrate to Git from another source control system. Specifically, we'll discuss migrating from Team Foundation Version Control (TFVC) to Git.

Most teams wish they could reorganize their source control structure. Typically, the structure the team is using today was set up a decade ago and it's no longer suitable. Migrating to Git could be a good opportunity to restructure your repo.

This brings up the question of whether to migrate your history along with the code. Only migrating the code is called "migrating the tip." As the name indicates, you're not carrying forward any history from your TFVC repository. You're simply taking the last version of the code and setting that up in Git.

In general, migrating the tip is the better approach than migrating the history as well. TFVC and Git are fundamentally different. TFVC stores branches as individual folders and changes are stored as chain sets. Git works in a completely different way. You're relying on the migration to transform the history from one mechanism into a completely different mechanism that Git understands.

Also, if you're going to use the migration as an opportunity to restructure your repo, it probably doesn't make sense to migrate history since you're going to restructure the code (or break the code into multiple repos).

**Single Branch Import**

If you're on TFVC and you're in Azure DevOps, then you have the option of a simple single-branch import. Just click on **Import repository** from the Azure Repos top level drop-down menu to open the dialog. Then, enter the path to the branch you're migrating and if you want history or not (up to 180 days). Finally, name the repo and the import will be triggered. This is a very simple but effective way to do the migration.

**Git-tfs**

What if you need to migrate more than a single branch and retain branch relationships? Or you're going to take all the history with you? In that case, you're going to have to use Git-tfs . This is an open-source project that synchronizes Git and TFVC repos. Git-tfs can migrate multiple branches and will preserve the relationships so that you can merge branches in Git after you migrate. Be warned that it can take a while to do this conversion - especially for large repos or repos with long history. One approach is to do a dry-run migration locally, resolve any issues, and then do it for real.

**Implement branch strategies**

Another important aspect of Git is having a good branch strategy. A branch strategy gives structure to how features are developed, how hotfixes and patches are created, and to the whole development lifecycle. Three popular branch strategies that are Git-based are: the Feature Branch Workflow, the Gitflow Workflow, and the Forking Branch Workflow.

**Feature Branch Workflow**

The core idea behind the Feature Branch Workflow is that all feature development takes place in a dedicated branch instead of the master branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the master branch will never contain broken code, which is a huge advantage for continuous integration environments.

The Feature Branch Workflow assumes a central repository, and master represents the official project history. Instead of committing directly on their local master branch, developers create a new branch every time they start work on a new feature. Feature branches should have descriptive names, like new-banner-images or bug-91. The idea is to give a clear, highly-focused purpose to each branch. Git makes no technical distinction between the master branch and feature branches, so developers can edit, stage, and commit changes to a feature branch.

**Gitflow Workflow**

The Gitflow Workflow defines a strict branching model designed around the project's release. It provides a robust framework for managing larger projects.

Gitflow is ideally suited for projects that have a scheduled release cycle. This workflow doesn't add any new concepts or commands beyond what's required for the Feature Branch Workflow. Instead, it assigns very specific roles to different branches and defines how and when they should interact. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases. Of course, you also get all the benefits of the Feature Branch Workflow: pull requests, isolated experiments, and more efficient collaboration.

Gitflow uses a central server model called origin. Instead of a single master branch, this workflow uses two branches to record the history of the project. The master branch stores the official release history, and the develop branch serves as an integration branch for features. Feature branches may branch off from develop branch and must merge back into develop. Release branches may branch off from the develop branch and must merge back into the develop and master branches. For hotfixes, you use the hotfix branch, which is cut from the master branch, and then merged back into the master and develop branches.

**Forking Branch Workflow**

The Forking Workflow is fundamentally different than other popular Git workflows. Instead of using a single server-side repository to act as the "central" codebase, it gives every developer their own server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one. The Forking Workflow is most often seen in public open source projects.

The main advantage of the Forking Workflow is that contributions can be integrated without the need for everybody to push to a single central repository. Developers push to their own server-side repositories, and only the project maintainer can push to the official repository. This allows the maintainer to accept commits from any developer without giving them write access to the official codebase.

The Forking Workflow typically follows a branching model based on the Gitflow Workflow. This means that complete feature branches will be merged into the original project maintainer's repository. The result is a distributed workflow that provides a flexible way for large, organic teams (including untrusted third-parties) to collaborate securely.

As in the other Git workflows, the Forking Workflow begins with an official public repository stored on a server. But when a new developer wants to start working on the project, they do not directly clone the official repository.

Instead, they fork the official repository to create a copy of it on the server. This new copy serves as their personal public repository. No other developers are allowed to push to it, but they can pull changes from it. After they have created their server-side copy, the developer performs a git clone to get a copy of it onto their local machine. This serves as their private development environment, just like in the other workflows.

When they're ready to publish a local commit, they push the commit to their own public repository—not the official one. Then, they file a pull request with the main repository, which lets the project maintainer know that an update is ready to be integrated.

**Use branch policies**

Branch policies help you to protect your important Git branches. Policies enforce your team's code quality and change management standards. You can configure branch policies from the Azure DevOps portal. Select **Repos** > **Branches** to open the **Branches** page. Locate the branch and select the **...** button. Select **Branch policies** from the context menu. Configure your policies in the **Policies** page.

Some examples of the policies you can configure include: the minimum number of reviewers, checking for linked work items, checking for comment resolution, enforcing a merge strategy, and checking for build validation.

**Pull request strategies**

Git is all about collaboration, and pull requests let you tell others about changes you've pushed to a GitHub repository. Once a pull request is sent, interested parties can review the set of changes, discuss potential modifications, and even push follow-up commits if necessary.

Pull requests are commonly used by teams and organizations that have a shared repository, where everyone shares a single repository and branches are used to develop features and isolate changes. Many open source projects on GitHub use pull requests to manage changes from contributors. They provide a way to notify project maintainers about changes one has made and in initiating code review and general discussion about a set of changes before merging them into the main branch.

Pull requests combine the review and merge of your code into a single collaborative process. Once you're done fixing a bug or new feature in a branch, create a new pull request. Add the members of the team to the pull request so they can review and vote on your changes. Use pull requests to review works in progress and get early feedback on changes. There's no commitment to merge the changes as the owner can abandon the pull request at any time.

There are a few critical branches in your repo, such as the master branch, that you may want to protect with some pull request policies. Pull requests should be mandatory if a developer wants to make changes on a critical branch. Developers pushing changes directly to these branches will have their pushes rejected.

Add additional conditions to your pull requests to enforce a higher level of code quality in your key branches. A clean build of the merged code and approval from multiple reviewers are some extra requirements you can set to protect your key branches.