**What is version control?**

A version control system (VCS) is a program or set of programs that tracks changes to a collection of files. One goal of a VCS is to easily recall earlier versions of individual files or of the entire project. Another goal is to allow several team members to work on a project, even on the same files, at the same time without affecting each other's work.

Another name for a VCS is a software configuration management (SCM) system. The two terms often are used interchangeably—in fact, Git's official documentation is located at [git-scm.com](https://git-scm.com/). Technically, version control is just one of the practices involved in SCM. A VCS can be used for projects other than software, including books and online tutorials.

With a VCS, you can:

* See all the changes made to your project, when the changes were made, and who made them.
* Include a message with each change to explain the reasoning behind it.
* Retrieve past versions of the entire project or of individual files.
* Create *branches*, where changes can be made experimentally. This feature allows several different sets of changes (for example, features or bug fixes) to be worked on at the same time, possibly by different people, without affecting the main branch. Later, you can merge the changes you want to keep back into the main branch.
* Attach a tag to a version—for example, to mark a new release.

**Distributed version control**

Earlier instances of VCS’s, including CVS, Subversion (SVN), and Perforce, used a centralized server to store a project's history. This centralization meant that the one server was also potentially a single point of failure.

Git is *distributed*, which means that a project's complete history is stored both on the client *and* on the server. You can edit files without a network connection, check them in locally, and sync with the server when a connection becomes available. If a server goes down, you still have a local copy of the project. Technically, you don't even have to have a server. Changes could be passed around in e-mail or shared by using removable media, but no one uses Git this way in practice.

**Git terminology**

To understand Git, you have to understand the terminology. Here's a short list of terms that Git users frequently use. Don't be concerned about the details for now; all these terms will become familiar as you work your way through the exercises in this module.

* **Working tree**: The set of nested directories and files that contain the project that's being worked on.
* **Repository (repo)**: The directory, located at the top level of a working tree, where Git keeps all the history and metadata for a project. Repositories are almost always referred to as *repos*. A *bare repository* is one that isn't part of a working tree; it's used for sharing or backup. A bare repo is usually a directory with a name that ends in *.git*—for example, *project.git*.
* **Hash**: A number produced by a hash function that represents the contents of a file or another object as a fixed number of digits. Git uses hashes that are 160 bits long. One advantage to using hashes is that Git can tell whether a file has changed by hashing its contents and comparing the result to the previous hash. If the file time-and-date stamp is changed, but the file hash isn’t changed, Git knows the file contents aren’t changed.
* **Object**: A Git repo contains four types of *objects,* each uniquely identified by an SHA-1 hash. A *blob* object contains an ordinary file. A *tree* object represents a directory; it contains names, hashes, and permissions. A *commit* object represents a specific version of the working tree. A *tag* is a name attached to a commit.
* **Commit**: When used as a verb, *commit* means to make a commit object. This action takes its name from commits to a database. It means you are committing the changes you have made so that others can eventually see them, too.
* **Branch**: A branch is a named series of linked commits. The most recent commit on a branch is called the *head*. The default branch, which is created when you initialize a repository, is called main, often named master in Git. The head of the current branch is named HEAD. Branches are an incredibly useful feature of Git because they allow developers to work independently (or together) in branches and later merge their changes into the default branch.
* **Remote**: A remote is a named reference to another Git repository. When you create a repo, Git creates a remote named origin that is the default remote for push and pull operations.
* **Commands**, **subcommands**, and **options**: Git operations are performed by using commands like git push and git pull. git is the command, and push or pull is the subcommand. The subcommand specifies the operation you want Git to perform. Commands frequently are accompanied by options, which use hyphens (-) or double hyphens (--). For example, git reset --hard.

These terms and others, like push and pull, will make more sense shortly. But you have to start somewhere, and you might find it helpful to come back and review this glossary of terms after you finish the module.

**The Git command line**

Several different GUIs are available for Git, including GitHub Desktop. Many programming editors, like Microsoft [Visual Studio Code](https://code.visualstudio.com/), also have an interface to Git. They all work differently and they have different limitations. None of them implement *all* of Git's functionality.

The exercises in this module use the Git command line—specifically, Git commands executed in Azure Cloud Shell. However, Git's command-line interface works the same, no matter what operating system you're using. Plus, the command line lets you tap into *all* of Git's functionality. Developers who see Git only through a GUI sometimes find themselves confronted with error messages they can't resolve, and they have to resort to the command line to get going again.

**Git and GitHub**

As you work with Git, you might wonder about differences between the features it offers and the features offered on [GitHub](https://github.com/).

As mentioned earlier, Git is a distributed version control system (DVCS) that multiple developers and other contributors can use to work on a project. It provides a way to work with one or more local branches and then push them to a remote repository.

GitHub is a cloud platform that uses Git as its core technology. GitHub simplifies the process of collaborating on projects and provides a website, more command-line tools, and overall flow that developers and users can use to work together. GitHub acts as the remote repository mentioned earlier.

Key features provided by GitHub include:

* Issues
* Discussions
* Pull requests
* Notifications
* Labels
* Actions
* Forks
* Projects

CONFIGURE GIT:  
git –version

Set your name in Cloud Shell with the following command. Replace <USER\_NAME>,<USER\_ EMAIL> with the user name, mail you want to use.  
git config --global user.name "<USER\_NAME>"

git config --global user.email "<USER\_EMAIL>"

Run the following command to check that your changes worked:

git config –list

 initialize your new repository and set the name of the default branch to main:

If you're running Git version 2.28.0 or later, use the following command:

git init --initial-branch=main

or

git init -b main

Now, use a git status command to show the status of the working tree:

git status

**Basic Git commands**:

Git works by remembering the changes to your files as if it's taking snapshots of your file system.

We'll cover a few basic commands to start tracking files in your repo. Then, you'll save your first "snapshot" for Git to compare against.

**git status**

The first and most commonly used Git command is git status. You've already used it once, in the preceding exercise, to see that you had initialized your Git repo properly.

git status displays the state of the working tree (and of the staging area—we'll talk more about the staging area soon). It lets you see which changes are currently being tracked by Git, so you can decide whether you want to ask Git to take another snapshot.

**git add**

git add is the command you use to tell Git to start keeping track of changes in certain files.

The technical term is *staging* these changes. You'll use git add to stage changes to prepare for a commit. All changes in files that have been added but not yet committed are stored in the *staging area*.

**git commit:**

After you've staged some changes for commit, you can save your work to a snapshot by invoking the git commit command.

*Commit* is both a verb and a noun. It has essentially the same meaning as when you commit to a plan or commit a change to a database. As a verb, committing changes means you put a copy (of the file, directory, or other "stuff") in the repository as a new version. As a noun, a commit is the small chunk of data that gives the changes you committed a unique identity. The data that's saved in a commit includes the author's name and e-mail address, the date, comments about what you did (and why), an optional digital signature, and the unique identifier of the preceding commit.

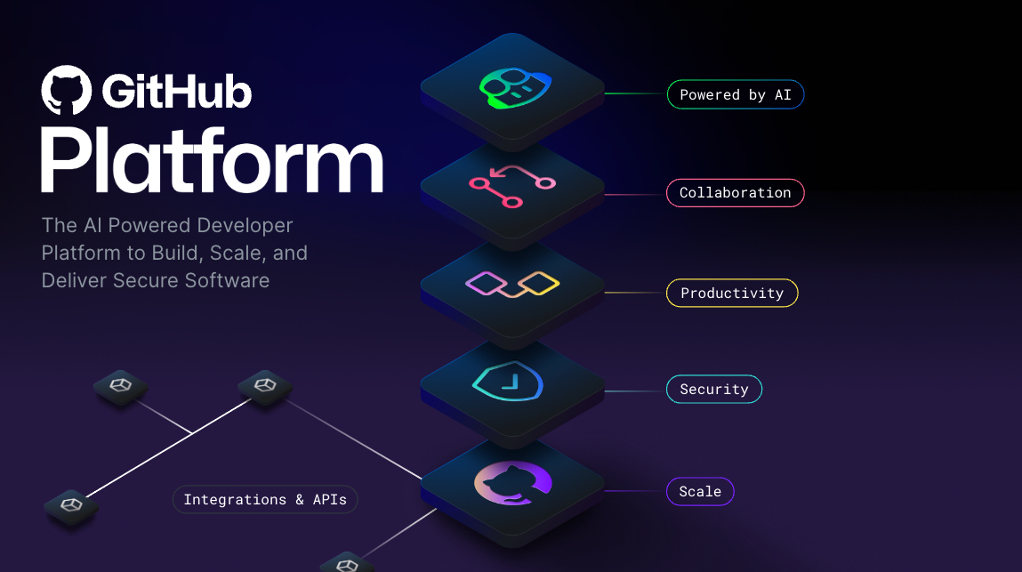
**git log**

The git log command allows you to see information about previous commits. Each commit has a message attached to it (a commit message), and the git log command prints information about the most recent commits, like their time stamp, the author, and a commit message. This command helps you keep track of what you've been doing and what changes have been saved.

**git help**

You've already tried out the git help command, but it's worth reminding you about. Use this command to easily get information about all the commands you've learned so far, and more.

Remember, each command comes with its *own* help page, too. You can find these help pages by typing git <command> --help. For example, git commit --help brings up a page that tells you more about the git commit command and how to use it.



**GitHub** is a cloud-based platform that uses Git, a distributed version control system, at its core. The GitHub platform simplifies the process of collaborating on projects and provides a website, command-line tools, and overall flow that allows developers and users to work together.

As we learned earlier, GitHub provides an AI powered developer platform to build, scale, and deliver secure software. Let’s break down each one of the core pillars of the GitHub Enterprise platform, AI, Collaboration, Productivity, Security, and Scale.

**AI**

Generative AI is dramatically transforming software development as we speak. The GitHub Enterprise platform is enhancing collaboration through AI-powered pull requests and issues, productivity through Copilot, and security by automating security checks faster.

**Collaboration**

Collaboration is at the core of everything GitHub does. We know inefficient collaboration results in wasted time and money. We counteract that with a suite of seamless tools that allow collaboration to happen effortlessly.

Repositories, Issues, Pull Requests, and other tools help to enable developers, project managers, operation leaders, and others at the same company. It enables them to work faster together, cut down approval times, and ship more quickly.

**Productivity**

Productivity is accelerated with automation that the GitHub Enterprise Platform provides. With built-in CI/CD (Continuous Integration and Continuous Delivery) tools directly integrated into the workflow, the platform gives users the ability to set tasks and forget them, taking care of routine administration and speeding up day-to-day work. This gives your developers more time to focus on what matters most, creating innovative solutions.

**Security**

GitHub focuses on integrating security directly into the development process from the start. GitHub Enterprise platform includes native, first-party security features that minimize security risk with a built-in security solution. Plus, your code remains private within your organization. At the same time, you're able to take advantage of security overview and Dependabot.

GitHub has continued to make investments to ensure that our features are enterprise-ready. Microsoft and highly regulated industries trust GitHub, and we meet global compliance requirements.

**Scale**

GitHub is the largest developer community of its kind with real-time data on over 100M+ developers, 330M+ repositories, and countless deployments. We’ve been able to understand the shifting needs of developers and make changes to our product to match.

This has translated into an incredible scale that is unmatched and unparalleled by any other company on the planet. Everyday we're gaining more insights from this impressive community and evolving the platform to meet their needs.

In essence, the GitHub Enterprise Platform focuses on the developer experience. It has the scale to provide industry-changing insights, collaboration capabilities for transformative efficiency, the tools for increased productivity, security at every step, and AI to power it all to new heights in a single, integrated platform.

Now let’s get into the backbone of GitHub, repositories.

**Introduction to repositories**

Let’s first review:

* What is a repository?
* How to create a repository
* Adding files to a repository
* How to search for repositories
* Introduction to gists, wikis, and GitHub pages

**What is a repository?**

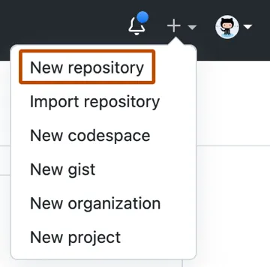
A repository contains all of your project's files and each file's revision history. It's one of the essential parts that helps you collaborate with people. You can use repositories to manage your work, track changes, store revision history, and work with others. Before we dive too deep, let’s first start with how to create a repository.

**How to create a repository**

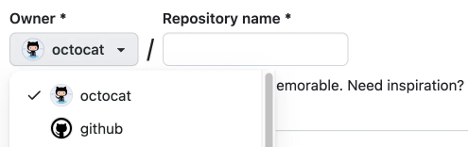
You can create a new repository on your personal account or any organization where you have sufficient permissions.

Let’s tackle creating a repository from github.com.

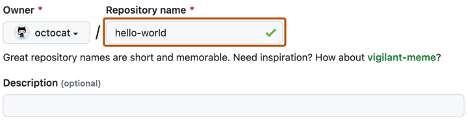
1. In the upper-right corner of any page, use the drop-down menu, and select **New repository**.



1. Use the **Owner** drop-down menu to select the account you want to own the repository.



1. Type a name for your repository, and an optional description.



1. Choose a repository visibility.
   * **Public repositories** are accessible to everyone on the internet.
   * **Private repositories** are only accessible to you, people you explicitly share access with, and, for organization repositories, certain organization members.
2. Select **Create repository** and congratulations! You just created a repository!

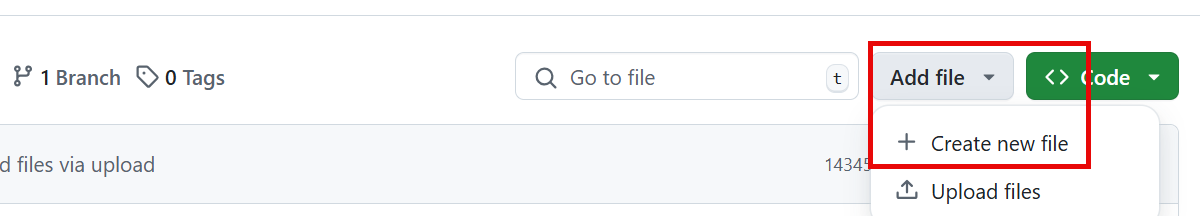
Next up, let’s review how to add files to your repository.

**How to add a file to your repository**

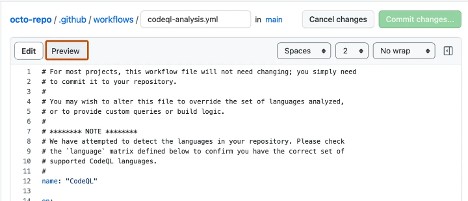
Files in GitHub can do a handful of things, but the main purpose of files is to store data and information about your project. It's worth knowing in order to add a file to a repository that you must first have minimum **Write** access within the repository you want to add a file.

Let’s review how to add a file to your repository.

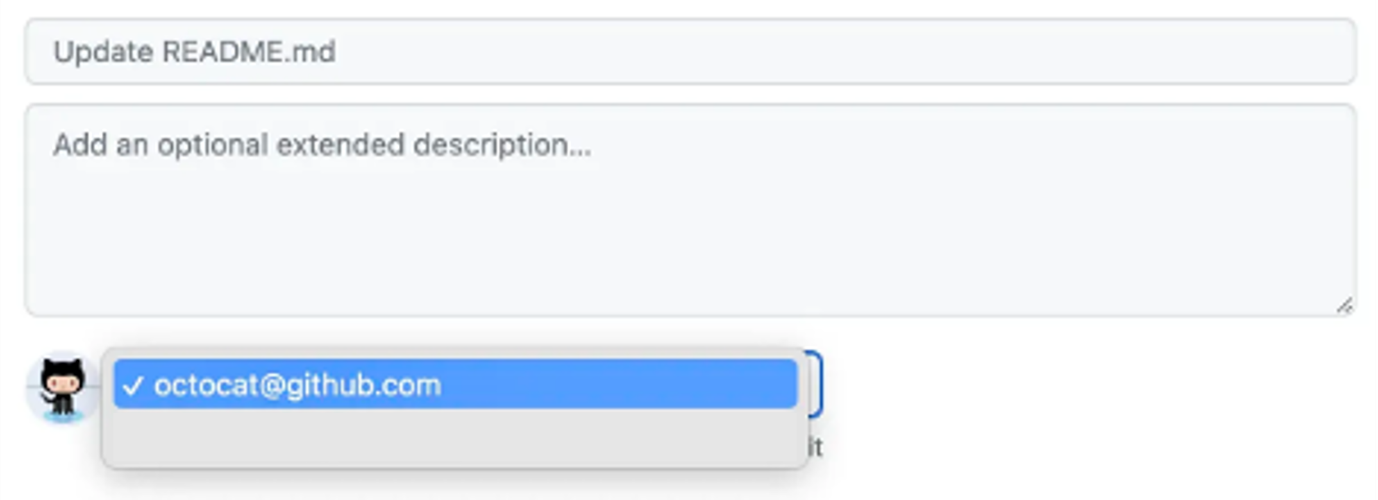
1. On GitHub.com, navigate to the main page of the repository.
2. In your repository, browse to the folder where you want to create a file by selecting the **creating a new file** link or **uploading an existing file**.
3. Once added, above the list of files select the **Add file ᐁ** drop-down menu. Then select **Create new file**.



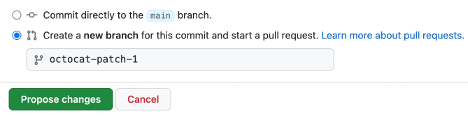
1. In the file name field, type the name and extension for the file. To create subdirectories, type the **/** directory separator.
2. In the file contents text box, type **content** for the file.
3. To review the new content, above the file contents, select **Preview**.



1. Select **Commit changes**.
2. In the **Commit message** field, type a short and meaningful commit message that describes the change you made to the file. You can attribute the commit to more than one author in the commit message.
3. If you have more than one email address associated with your account on GitHub.com, select the email address drop-down menu. Then select the email address to use as the Git author email address. Only verified email addresses appear in this drop-down menu. If you enabled email address privacy, then *[username]@users.noreply.github.com* is the default commit author email address.



1. Below the **Commit message** fields, decide whether to add your commit to the current branch or to a new branch. If your current branch is the default branch, you should choose to create a new branch for your commit, and then create a pull request.



1. Select **Commit changes** or **Propose changes**.

**What are gists?**

Now that we have a good understanding of repositories, we can review gists. Similarly to repositories, gists are a simplified way to share code snippets with others.

Every gist is a Git repository, which you can fork and clone and be made either public or secret. Public gists are displayed publicly where people can browse new ones as they’re created. Public gists are also searchable. Conversely, secret gists aren't searchable, but they aren’t entirely private. If you send the URL of a secret gist to a friend, they'll be able to see it.

To learn more about gists, see the linked article in our Resources section at the end of this module titled *Creating Gists*.

**What are wikis?**

Every repository on GitHub.com comes equipped with a section for hosting documentation, called a wiki. You can use your repository's wiki to share long-form content about your project, such as how to use it, how you designed it, or its core principles. While a README file quickly tells what your project can do, you can use a wiki to provide additional documentation.

It’s worth a reminder that if your repository is private, only people who have at least read access to your repository will have access to your wiki.

**Components of the GitHub flow:**

In this unit, we're reviewing the following components of the GitHub flow:

* Branches
* Commits
* Pull Requests
* The GitHub Flow

**What are branches:**

In the last section, we created a new file and a new branch in your repositories.

Branches are an essential part to the GitHub experience because they're where we can make changes without affecting the entire project we're working on.

Your branch is a safe place to experiment with new features or fixes. If you make a mistake, you can revert your changes or push more changes to fix the mistake. Your changes won't update on the default branch until you merge your branch.

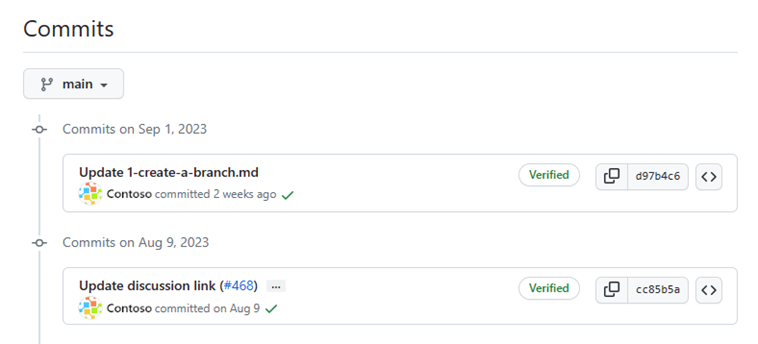
**Note**

Alternatively, you can create a new branch and check it out by using git in a terminal. The command would be git checkout -b newBranchName

**What are commits?**

In the previous unit, you added a new file into the repository by pushing a commit. Let’s briefly review what commits are.

A **commit** is a change to one or more files on a branch. Every time a commit is created, it's assigned a unique ID and tracked along with the time and contributor. Commits provide a clear audit trail for anyone reviewing the history of a file or linked item, such as an issue or pull request.



Within a git repository, a file can exist in several valid states as it goes through the version control process. The primary states for a file in a Git repository are **Untracked** and **Tracked**.

**Untracked:** An initial state of a file when it isn't yet part of the Git repository. Git is unaware of its existence.

**Tracked:** A tracked file is one that Git is actively monitoring. It can be in one of the following substates:

* **Unmodified:** The file is tracked, but it hasn't been modified since the last commit.
* **Modified:** The file has been changed since the last commit, but these changes aren't yet staged for the next commit.
* **Staged:** The file has been modified, and the changes have been added to the staging area (also known as the index). These changes are ready to be committed.
* **Committed:** The file is in the repository's database. It represents the latest committed version of the file.

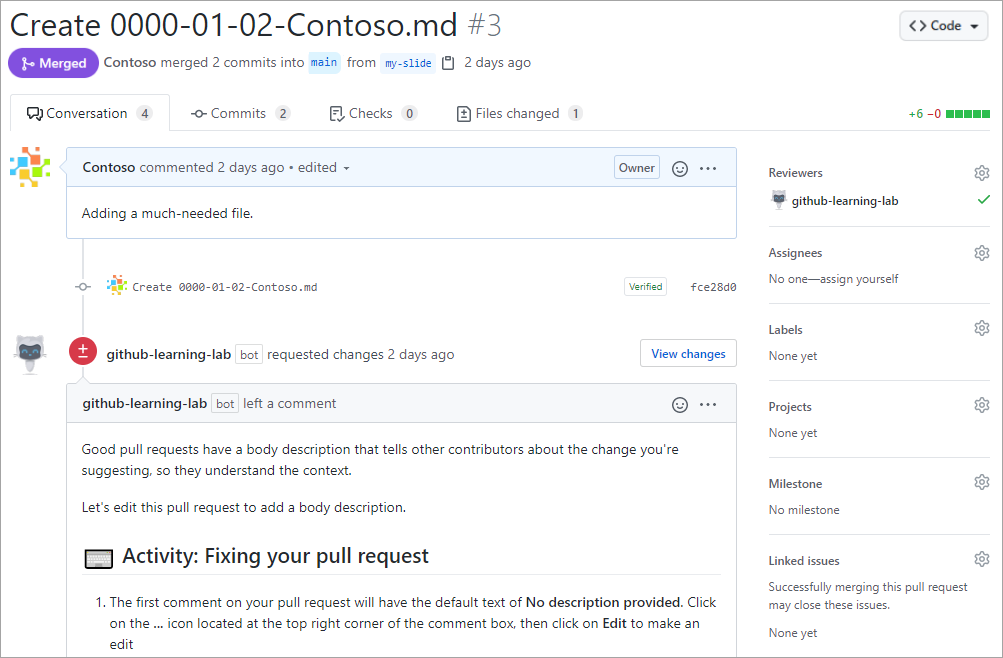
These states and substates are important to collaborating with your team to know where each and every commit is in the process of your project. Now let’s move on to pull requests.

**What are pull requests?**

A **pull request** is the mechanism used to signal that the commits from one branch are ready to be merged into another branch.

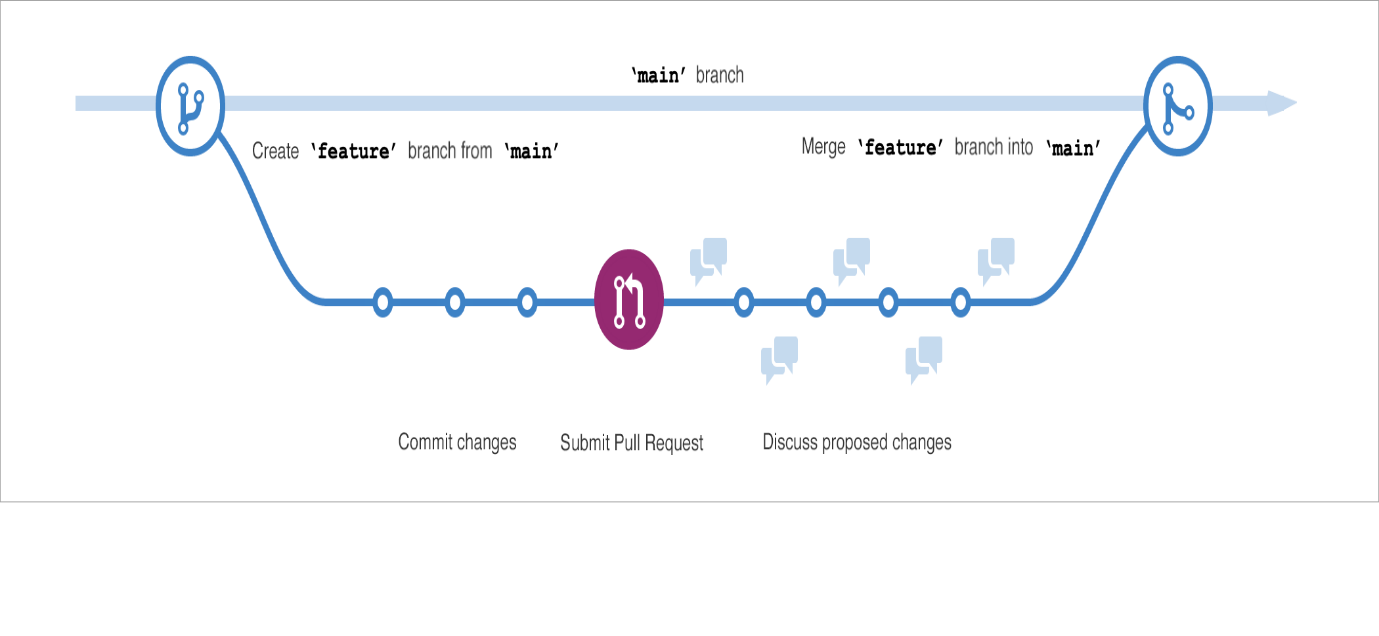
The team member submitting the **pull request** asks one or more reviewers to verify the code and approve the merge. These reviewers have the opportunity to comment on changes, add their own, or use the pull request itself for further discussion.

Once the changes have been approved (if required), the pull request's source branch (the compare branch) is merged into the base branch.



Now that we know of all the ingredients, let’s review the GitHub flow.

**The GitHub flow**



The GitHub flow can be defined as a lightweight workflow that allows for safe experimentation. You can test new ideas and collaboration with your team by using branching, pull requests, and merging.

Now that we know the basics of GitHub we can walk through the GitHub flow and its components.

1. Start by creating a branch so that the changes, features, and fixes you create don't affect the main branch.
2. Next, make your changes. We recommend deploying changes to your feature branch before merging into the main branch. Doing so ensures the changes are valid in a production environment.
3. Now, create a pull request to ask collaborators for feedback. Pull request review is so valuable that some repositories require an approving review before pull requests can be merged.
4. Then review and implement your feedback from your collaborators.
5. Once you feel great about your changes, it's time to get your pull request approved and merge it into the main branch.
6. Finally, you can delete your branch. Deleting your branch signals your work on the branch is complete and prevents you or others from accidentally using old branches.

**GitHub is a collaborative platform**

Collaboration is at the core of everything GitHub does. We went over repositories in the first unit of the module and learned that repositories help you organize your project and its files. In the last unit, we learned about pull requests, which is a way to keep track of changes made to your project.

In this unit, we're learning about issues and discussions. These are two other pieces that contribute to the collaborative nature of the GitHub Enterprise Platform.

**Issues**

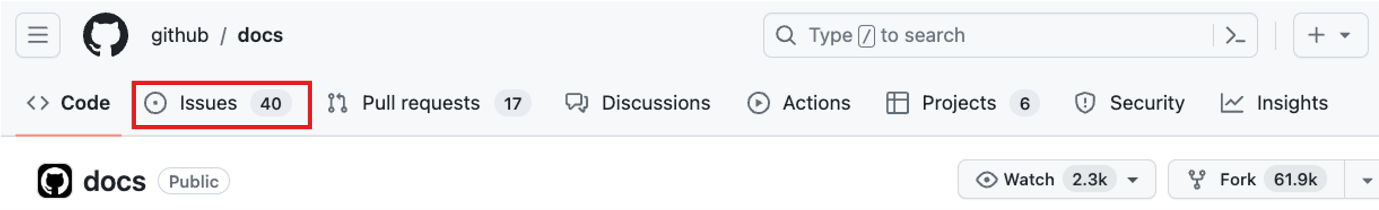
GitHub Issues were created to track ideas, feedback, tasks, or bugs for work on GitHub. Issues can be created in various ways, so you can choose the most convenient method for your workflow.

For this walkthrough, we'll go over how to create an issue from a repository. But issues can also be created from:

* An item in a task list.
* A note in a project.
* A comment in an issue or pull request.
* A specific line of code.
* A URL query.

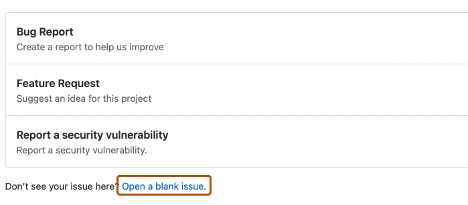
**Creating an issue from a repository**

1. On GitHub.com, navigate to the main page of the repository.
2. Under your repository name, select **Issues**.



1. Select **New issue**.
2. If your repository uses issue templates, next to the type of issue you'd like to open select **Get started**.

If the type of issue you'd like to open isn't included in the available options, select **Open a blank issue**. If not using templates, skip to Step 5.



1. In the **Add a title** field, enter a title for your issue.
2. In the **Add a description** field, type a description of your issue.
3. If you're a project maintainer, you can assign the issue to someone, add it to a project board, associate it with a milestone, or apply a label.
4. When you're finished, select **Submit new issue**.

Some conversations are more suitable for GitHub Discussions. You can use GitHub Discussions to ask and answer questions, share information, make announcements, and conduct or participate in conversations about a project.

In the next section, we’ll review Discussions and how to best utilize the feature.

**Discussions**

Discussions are for conversations that need to be accessible to everyone and aren't related to code. Discussions enable fluid, open conversation in a public forum.

In this section we go over:

* Enabling a discussion in your repository.
* Creating a new discussion and various discussion categories.

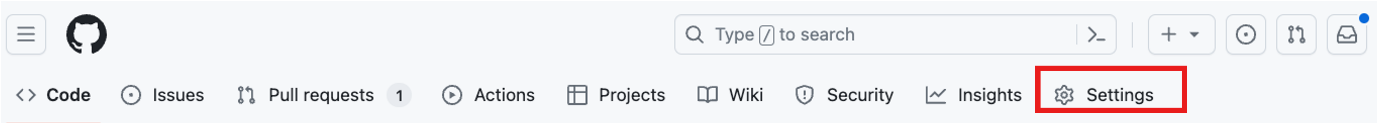
Let’s dive into enabling a discussion in your repository.

**Enabling a discussion in your repository**

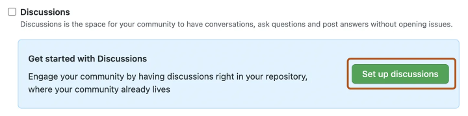
Repository owners and people with Write access can enable GitHub Discussions for a community on their public and private repositories. The visibility of a discussion is inherited from the repository the discussion is created in.

When you first enable GitHub Discussions, you're invited to configure a welcome post.

1. On GitHub.com, navigate to the main page of the repository.
2. Under your repository name, select **Settings**.



1. Scroll down to the **Features** section and under **Discussions**, select **Setup discussions**.



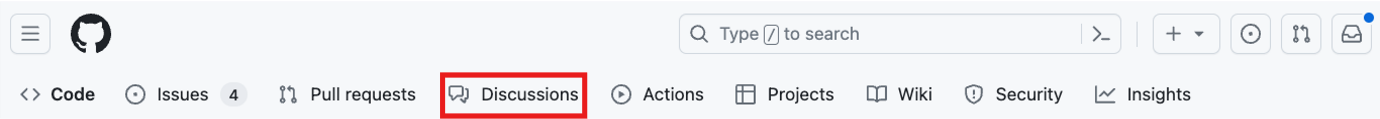
1. Under **Start a new discussion**, edit the template to align with the resources and tone you want to set for your community.
2. Select **Start discussion**.

You're now ready to create a new discussion.

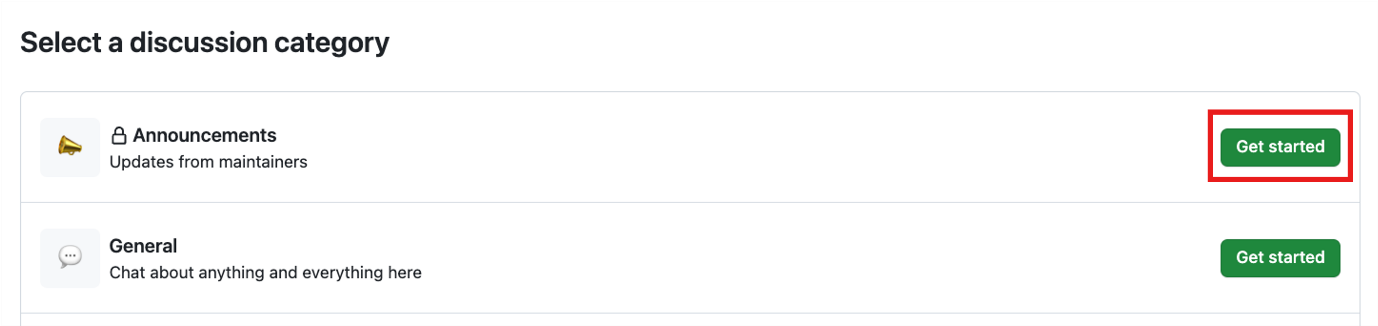
**Create a new discussion**

Any authenticated user who can view the repository can create a discussion in that repository. Similarly, since organization discussions are based on a source repository, any authenticated user who can view the source repository can create a discussion in that organization.

1. On GitHub.com, navigate to the main page of the repository or organization where you want to start a discussion.
2. Under your repository or organization name, select **Discussions**.



1. On the right side of the page, select **New discussion**.
2. Select a discussion category by selecting **Get started**. All discussions must be created in a category. For repository discussions, people with maintain or admin permissions to the repository define the categories for discussions in that repository.

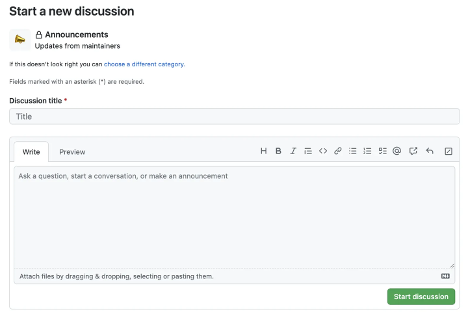


Each category must have a unique name, emoji pairing, and a detailed description stating its purpose. Categories help maintainers organize how conversations are filed. They're customizable to help distinguish categories that are Q&A or more open-ended conversations. The following table shows the default categories for discussions and their purpose.

Expand table

| **Category** | **Purpose** | **Format** |
| --- | --- | --- |
| 📣 Announcements | Updates and news from project maintainers | Announcement |
| #️⃣ General | Anything and everything relevant to the project | Open-ended discussion |
| 💡 Ideas | Ideas to change or improve the project | Open-ended discussion |
| 🗳️ Polls | Polls with multiple options for the community to vote for and discuss | Polls |
| 🙏 Q&A | Questions for the community to answer, with a question/answer format | Question and Answer |
| 🙌 Show and tell | Creations, experiments, or tests relevant to the project | Open-ended discussion |

1. Under **Discussion title** enter a title for your discussion, and under **Write** enter the body of your discussion.



1. Select **Start discussion**.

That covers a little about how GitHub inspires collaboration. Now let's move to how you can manage notifications, subscribe to threads, and get started with GitHub pages.