**GitHub Documentation: Step-by-Step Guide**

**1. Creating a GitHub Account**

GitHub is a platform for hosting code repositories and collaborating on projects. The first step is to create an account.

**Steps to Create a GitHub Account:**

1. **Visit GitHub's Website**:
   * Open your browser and go to [github.com](https://github.com/).
2. **Sign Up for a GitHub Account**:
   * On the homepage, click on **Sign Up**.
   * Enter your **email address**, **password**, and **username**.
   * Choose whether you want to receive product updates.
   * Complete the verification by solving the puzzle.
3. **Finish Signing Up**:
   * GitHub will ask for additional details, such as how you plan to use GitHub (this is optional).
   * You may need to verify your email by clicking the link sent to your email inbox.
4. **Account Created**:
   * Now, you have access to your GitHub dashboard!

**2. Creating a GitHub Repository**

A **repository** is like a folder for your project. It stores your code and keeps track of all changes made to it.

**Steps to Create a Repository:**

1. **Go to Your GitHub Dashboard**:
   * Once logged in, click on the **Repositories** tab on the top-left of the page.
   * Click the green **New** button to create a new repository.
2. **Fill in Repository Details**:
   * **Repository Name**: Enter a name for your repository (e.g., MyFirstRepo).
   * **Description** (optional): Write a brief description of your project.
   * **Visibility**: Choose between a **Public** or **Private** repository:
     + Public: Anyone can see the repository, but only you (and collaborators) can make changes.
     + Private: Only you and people you invite can view or make changes.
3. **Initialize the Repository**:
   * Check the box for **Initialize this repository with a README**. This will create a README file where you can describe your project.
   * Optionally, you can also add a **.gitignore** file, which tells Git which files to ignore (like build files).
   * Select a **license** if your project needs one.
4. **Create the Repository**:
   * Click on the **Create repository** button, and your repository is ready.

**3. Adding Project Files to Your GitHub Repository**

Now that you've created a repository on GitHub, the next step is to add your project files from your local machine. You can do this via Git (the version control tool) using the command line. Here’s how to add your project files and push them to GitHub.

**Steps to Add Files to Your Repository:**

1. **Install Git (If Not Installed)**:
   * If you don’t have Git installed on your system, download it from [Git's official website](https://git-scm.com/) and follow the installation steps.
   * After installation, you can check if Git is installed by opening a terminal (Command Prompt or PowerShell) and typing:

|  |
| --- |
| git --version |

1. **Navigate to Your Project Folder**:
   * Open your terminal (Command Prompt, PowerShell, or Git Bash) and navigate to the folder where your project files are stored using the cd (change directory) command. For example:

|  |
| --- |
| cd D:\MyProject |

1. **Initialize the Git Repository**:
   * To start versioning your project, initialize Git in your project folder:

|  |
| --- |
| git init |

* + This creates a hidden .git folder where Git tracks changes to your project.

1. **Add the Remote GitHub Repository**:
   * Link your local project to the GitHub repository you created. Replace your-username and your-repo with your GitHub username and repository name:

|  |
| --- |
| git remote add origin https://github.com/your-username/yourrepo.git |

1. **Stage Your Project Files**:
   * Staging means selecting which files you want to track and include in your commit. To stage all files, use:

|  |
| --- |
| git add . |

1. **Commit Your Changes**:
   * A commit is like a snapshot of your project at a specific point in time. To commit your changes, use:

|  |
| --- |
| git commit -m "Initial commit" |

* + The -m flag lets you write a commit message, like "Initial commit," describing what changes were made.

1. **Push the Changes to GitHub**:
   * Finally, push your committed changes to the GitHub repository using:

|  |
| --- |
| git push -u origin master |

* + This will upload your project files to the master branch of your GitHub repository. If GitHub recommends using the main branch, replace master with main in the above command.

1. **Verify the Upload**:
   * Go back to your repository on GitHub in your browser. You should now see your project files listed there.

**4. Understanding Key GitHub Terminology**

Before we move forward, it's important to understand the basic GitHub terminologies and how they work. These concepts will help you work more effectively with Git and GitHub, especially when collaborating on projects with others.

**a. Branches**

* A **branch** in Git is a separate line of development within a repository. It allows you to work on different features or changes without affecting the main codebase.
  + **Default Branch**: The default branch is typically named main or master. This is the main line of development.
  + **Feature Branches**: These are branches created to work on specific features or fixes. For example, you can create a branch named feature-xyz to add a new feature.
  + **Why Use Branches?**
    - Branches allow you to work on new features without affecting the main code.
    - They enable multiple people to work on different tasks simultaneously without conflicts.

**b. Pull Requests (PR)**

* A **Pull Request** is a way to request that changes in one branch be merged into another. Usually, this is used when a developer completes work in a feature branch and wants to merge it into the main branch.
  + **How Pull Requests Work**:
    1. You create a branch and make changes (e.g., new features or fixes).
    2. You push the branch to the repository and create a Pull Request (PR).
    3. The PR is reviewed by other team members.
    4. If everything is good, the PR is merged into the main branch, and the changes become part of the project.

**c. Actions**

* **GitHub Actions** allow you to automate tasks within your repository, such as testing, building, and deploying code. You can define workflows using configuration files, which are triggered by specific events (like a push or pull request).
  + **Common Use Cases for GitHub Actions**:
    - Automated testing: Run tests on your code whenever you push changes.
    - Continuous integration/continuous deployment (CI/CD): Automatically deploy code to a server after tests pass.

**d. Projects**

* **Projects** in GitHub are used for project management. It’s like a Kanban board where you can organize tasks, assign issues, and keep track of work progress.

**e. Issues**

* **Issues** are used to track bugs, tasks, or enhancements. Each issue can be assigned to team members, tagged with labels, and organized in a project.
  + **Why Use Issues?**
    - They provide a clear way to document bugs, improvements, or tasks.
    - Team members can collaborate by commenting on and resolving issues.

**f. Wiki**

* GitHub’s **Wiki** is a place where you can create and manage documentation for your project. It's useful for providing instructions, usage guides, and more.

**g. Insights**

* **Insights** offer analytics about your repository. You can view contributions, activity, and other statistics to understand how the project is progressing.

**h. Security**

* GitHub's **Security** tab helps you manage vulnerabilities in your code, secrets management, and more. You can use tools like Dependabot to identify security risks in your project.

**5. Connecting Visual Studio to GitHub (Brief Details)**

**Connecting Visual Studio to GitHub**

1. **Clone a GitHub Repository**:
   * In Visual Studio 2022, go to **Git** > **Clone Repository**.
   * Enter the URL of your GitHub repository and choose a folder to clone the project.
2. **Open a Local Git Repository**:
   * If you’ve already cloned the repository locally, you can open it by going to **File** > **Open** > **Open from Source Control**, and selecting the project folder.
3. **Authenticate with GitHub**:
   * When prompted, sign in to GitHub using your credentials or a personal access token.

**Key GitHub Operations in Visual Studio:**

1. **Stage Files**:
   * **Staging** means preparing your files for a commit. It’s like selecting the changes you want to track.
   * In Visual Studio, go to the **Git Changes** window. Here you can see all modified files.
   * Click the **+** icon next to each file or click **Stage All** to stage all the changes.
2. **Commit Changes**:
   * A **commit** is a snapshot of your changes.
   * After staging files, add a commit message in the **Git Changes** window (something like “Added new feature”).
   * Click **Commit All** to save the changes to your local repository.
3. **Push Changes**:
   * **Pushing** sends your local commits to GitHub.
   * After committing, click the **Push** button in the **Git Changes** window to upload the commits to the GitHub repository.
4. **Pull Changes**:
   * **Pulling** updates your local repository with changes from GitHub (like if someone else pushed changes).
   * Click **Pull** in the **Git Changes** window to fetch and integrate changes from GitHub into your local code.

**What are Staging, Committing, Pushing, and Pulling?**

* **Staging**: Selecting which changes to track. It allows you to choose what you want to include in your next commit.
* **Committing**: Saving a snapshot of your staged changes locally. This helps you keep track of your progress and changes.
* **Pushing**: Uploading your committed changes from your local machine to the remote GitHub repository, making them available to others.
* **Pulling**: Fetching changes from the remote GitHub repository to your local copy, ensuring you’re up-to-date with the latest code changes.

**6. Pushing Files and Tracking Progress**

Now that your local repository is connected to GitHub, you can push your files from Visual Studio to GitHub.

**Steps to Push Files to GitHub:**

1. **Stage Files**:
   * Make changes to your project in Visual Studio.
   * Go to the **Git Changes** window, select the files you want to include, and click the **+** icon to stage them.
2. **Commit Changes**:
   * Add a meaningful commit message, explaining the changes you made (e.g., "Added login feature").
   * Click **Commit All** to save the changes locally.
3. **Push Changes to GitHub**:
   * After committing, click **Push** to upload the changes to the GitHub repository.
   * You can verify the push by checking the repository on GitHub. Your changes will be visible there.

**7. Managing Branches in GitHub**

A **branch** is an independent line of development in Git, allowing you to work on features, bug fixes, or other changes without affecting the main codebase. Branches help developers collaborate and experiment without impacting the stability of the project.

**Understanding Branches**

1. **Main (or Master) Branch**:
   * This is the default branch in most repositories and typically contains the stable, production-ready code.
2. **Feature Branches**:
   * These are separate branches created to work on a specific feature or fix. Once the work is complete, you can merge the feature branch into the main branch.
   * Example: If you're working on a new login feature, you can create a branch named login-feature.

**Steps to Create a Branch in GitHub or Visual Studio**

**In GitHub:**

1. **Go to Your Repository**:
   * Open the repository in GitHub where you want to create a branch.
2. **Create a Branch**:
   * Click on the branch dropdown (it usually shows main by default).
   * Type the name of your new branch (e.g., feature-branch) and click **Create Branch**.

**In Visual Studio:**

1. **Create a Branch**:
   * Go to **Git** > **New Branch** in Visual Studio.
   * Name the branch (e.g., new-feature), and select the base branch from which you want to branch off.
   * Click **Create Branch**.
2. **Switch Between Branches**:
   * You can switch to a different branch by going to **Git** > **Manage Branches** and selecting the branch you want to work on.

**8. Pull Requests and Merging Branches**

Once you’ve completed the work on a feature branch, you can merge the branch back into the main branch through a **pull request (PR)**.

**What is a Pull Request?**

A **pull request** is a way of notifying others that you want to merge your changes from one branch into another. It allows for code review, discussion, and collaboration before the final merge.

**Steps to Create a Pull Request on GitHub:**

1. **Navigate to the Repository**:
   * Go to the **Pull Requests** tab in the repository.
2. **Create a New Pull Request**:
   * Click the **New Pull Request** button.
   * Select the branch you want to merge into the main branch.
3. **Review the Changes**:
   * GitHub will show the differences between the branches. You can review the changes to ensure they are correct.
4. **Submit the Pull Request**:
   * Add a title and description for your pull request.
   * Click **Create Pull Request**.

**9. Merging a Pull Request**

After the pull request has been reviewed, it can be merged into the main branch.

**Steps to Merge a Pull Request:**

1. **Review the Pull Request**:
   * Check the changes to ensure everything is correct.
   * If needed, request changes or leave comments.
2. **Merge the Pull Request**:
   * Once the pull request is approved, click the **Merge Pull Request** button.
   * Choose to either create a merge commit or squash the commits into one.
3. **Delete the Branch**:
   * After merging, you can delete the feature branch to keep the repository clean. This is optional but recommended for finished branches.

**10. GitHub Actions**

**GitHub Actions** is a tool to automate tasks like running tests, building code, and deploying applications whenever code is pushed or merged.

**How GitHub Actions Work:**

* **Workflows**: Automated scripts that define actions (e.g., run tests, build code) based on events (like pushing code).
* **Triggers**: You can trigger actions on events like push, pull request, or scheduled times.

**13. Collaborating with Others: Adding Users and Code Review**

**Adding Collaborators to a Repository**

If you want to collaborate with others on a GitHub repository, you can add them as collaborators or contributors. This allows them to access and contribute to the project.

**Steps to Add Collaborators**:

1. **Navigate to Your Repository**:
   * Go to your repository on GitHub.
2. **Access Repository Settings**:
   * Click on the **Settings** tab at the top of the repository page.
3. **Manage Access**:
   * In the left sidebar, click on **Collaborators and teams**.
4. **Add a Collaborator**:
   * Click the **Add people** button.
   * Enter the GitHub username or email of the person you want to add.
   * Select the role (e.g., Read, Write, Admin) and click **Add**.

**Code Review Process**

Code review is an essential practice to ensure the quality and correctness of code before it is merged into the main branch. This process involves reviewing changes made in a branch, discussing them, and approving or requesting changes.

**Steps to Review Code**:

1. **Create a Pull Request**:
   * When someone has finished their work on a branch, they should create a pull request (PR) to merge their branch into the main branch.
2. **Review the Pull Request**:
   * Go to the **Pull Requests** tab in your repository on GitHub.
   * Click on the pull request you want to review.
3. **Examine Changes**:
   * Review the code changes by looking at the **Files changed** tab. You can see what files were modified and the differences between the branches.
4. **Leave Comments**:
   * If you find issues or have suggestions, you can leave comments directly on specific lines of code or in the general discussion area.
5. **Approve or Request Changes**:
   * If the changes are satisfactory, you can approve the pull request.
   * If changes are needed, request changes and provide feedback for improvements.
6. **Merge the Pull Request**:
   * Once the pull request has been reviewed and approved, it can be merged into the main branch.
   * Click the **Merge pull request** button to complete the merge.
7. **Close the Pull Request**:
   * After merging, the pull request will be closed automatically. You can also manually close it if needed.

**14. Best Practices for Collaboration**

1. **Frequent Commits**: Encourage frequent commits to make it easier to review and understand changes.
2. **Clear Commit Messages**: Use clear and descriptive commit messages to explain the purpose of changes.
3. **Use Branches for Features**: Create separate branches for each feature or bug fix to keep the main branch stable.
4. **Communicate**: Use comments and discussions in pull requests to clarify any questions or issues.
5. **Automated Testing**: Implement automated testing (via GitHub Actions) to ensure code quality before merging.

**15. Limiting Access to a Specific Branch**

To restrict access so that collaborators can only work on a specific branch and do not have access to the main (or master) branch, you can use GitHub’s branch protection rules and repository permissions. Here's how to set it up:

**1. Set Up Branch Protection Rules**

Branch protection rules ensure that certain branches (like main or master) follow specific rules before changes can be merged. You can use these rules to prevent unauthorized changes to the protected branches.

**Steps to Set Up Branch Protection Rules**:

1. **Navigate to Your Repository**:
   * Go to your repository on GitHub.
2. **Access Repository Settings**:
   * Click on the **Settings** tab.
3. **Branch Protection Rules**:
   * In the left sidebar, click on **Branches**.
4. **Add Branch Protection Rule**:
   * Under the **Branch protection rules** section, click **Add rule**.
   * In the **Branch name pattern** field, enter the name of the branch you want to protect (e.g., main or master).
5. **Configure Rules**:
   * Check the options you want to enforce. For example, you might require pull requests before merging or require reviews before changes can be made.
   * Click **Create** or **Save changes** to apply the rule.

**2. Configure Repository Permissions**

You can manage repository permissions to restrict collaborators' access to specific branches.

**Steps to Configure Permissions**:

1. **Navigate to Your Repository**:
   * Go to your repository on GitHub.
2. **Access Repository Settings**:
   * Click on the **Settings** tab.
3. **Manage Access**:
   * In the left sidebar, click on **Collaborators and teams**.
4. **Add Collaborators**:
   * Click **Add people** to add collaborators.
   * Enter the GitHub username or email of the person you want to add.
   * Choose the role (e.g., Read, Write, Admin) that determines their access level.
5. **Restrict Branch Access**:
   * To restrict access to the main branch while allowing access to a specific branch, you need to manage permissions through GitHub Teams or organizations. This is more complex and usually involves creating a GitHub organization.

**For GitHub Teams**:

* + If your repository is part of a GitHub organization, you can create a team with specific branch access.
  + Go to **Organization settings** > **Teams**.
  + Create a team and assign them to the repository with specific branch permissions.

**For GitHub Organizations**:

* + Within an organization, you can set up branch permissions at a more granular level by creating protected branches and specifying which teams or individuals can push to them.

**Basic Git Commands**

1. **git init**
   * Initializes a new Git repository in the current directory. If the directory is already a Git repository, it reinitializes it.
2. **git clone [url]**
   * Creates a copy of a remote repository on your local machine.
3. **git add [file]**
   * Stages changes in the specified file for the next commit.
4. **git commit -m "[message]"**
   * Commits the staged changes with a descriptive message.
5. **git status**
   * Shows the status of changes in the working directory and staging area.
6. **git log**
   * Displays a list of commits in the current branch with details such as commit ID, author, date, and message.
7. **git diff**
   * Shows the differences between the working directory and the index (staged changes).
8. **git pull**
   * Fetches and merges changes from the remote repository into the current branch.
9. **git push**
   * Pushes commits from the local repository to the remote repository.

**Branching and Merging**

1. **git branch**
   * Lists all branches in the current repository.
2. **git branch [branch\_name]**
   * Creates a new branch with the specified name.
3. **git checkout [branch\_name]**
   * Switches to the specified branch.
4. **git checkout -b [branch\_name]**
   * Creates a new branch and switches to it.
5. **git merge [branch\_name]**
   * Merges changes from the specified branch into the current branch.
6. **git branch -d [branch\_name]**
   * Deletes the specified branch.

**Remote Repositories**

1. **git remote -v**
   * Lists the remote repositories linked to the local repository.
2. **git remote add [name] [url]**
   * Adds a new remote repository with the specified name and URL.
3. **git fetch [remote]**
   * Fetches updates from the remote repository but does not merge them.
4. **git pull [remote] [branch]**
   * Fetches and merges changes from the specified branch of the remote repository into the current branch.
5. **git push [remote] [branch]**
   * Pushes commits from the local branch to the specified branch of the remote repository.

**Staging and Unstaging**

1. **git add .**
   * Stages all changes in the current directory for the next commit.
2. **git reset [file]**
   * Unstages the specified file but keeps the changes in the working directory.
3. **git rm --cached [file]**
   * Removes the specified file from the index (staging area) but keeps it in the working directory.

**Commit History and Logs**

1. **git log --oneline**
   * Displays a simplified log with one line per commit.
2. **git log --graph**
   * Shows the commit history as a graph.
3. **git show [commit]**
   * Displays details about a specific commit.

**Configuration and Settings**

1. **git config --global user.name "[name]"**
   * Sets the name associated with your commits.
2. **git config --global user.email "[email]"**
   * Sets the email associated with your commits.
3. **git config --list**
   * Lists all Git configuration settings.